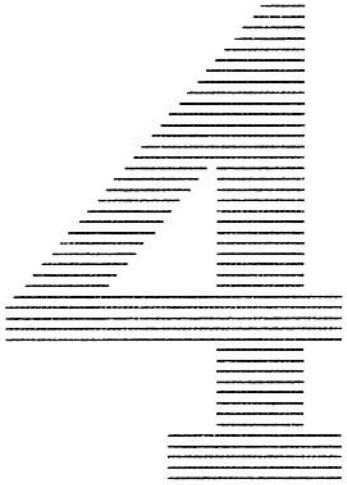
THE B NEWSLETTER

ISSN 0169-0191 CWI, Amsterdam Issue 4, September 1985

CONTENTS



A prototype of *B* is now available for the IBM PC and compatibles.
In this newsletter you will find a description of it and an order form.

THE B NEWSLETTER

Notes for Contributors

The newsletter is intended to provide information about *B* and to provide a forum for discussions. Therefore, you are encouraged to submit any articles you see fit.

Articles don't have to contain fully thought-out ideas, but may be yet undeveloped thoughts intended to stimulate discussion. The kinds of articles we have in mind are: interesting programs, either written or suggestions; unusual applications; letters, discussions on points of the language, proposed improvements, experience with the language, and so on.

If you are fortunate enough to be connected to a network with a gateway to UUCP net, you can submit articles and send mail to:

timo@mcvax.UUCP

Otherwise, articles and mail should be sent to

B Newsletter
Informatics / AA
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

New Unix Release

We have released a new version of the Mark 1 implementation of *B* for Unix systems. This version has all improvements mentioned in the article "Speeding Up the *B* Implementation" in the last newsletter. Also, you can easily change the key bindings of the editor operations; this allows you to adopt the editor to any terminal type.

If you have an older version of the Mark 1 implementation you can update it by sending us a 600 foot tape. If you don't know whether you have an old version, check the file *BugReport* in the directory *B/doc* on the tape you received, which says whether you have version Mark1A, Mark1B, or the new version Mark1C.

New Publications

There are two new publications since the last *B* newsletter. The publications mentioned in the previous newsletters are still available, as are the newsletters themselves.

Description of B,
Lambert Meertens and Steven Pemberton,
Informal definition of *B*, which can be used as a reference book, and as an introduction for people with ample programming experience. Published in SIGPLAN, Vol. **20**, No. 2, February 1985, pages 58 - 76. Previously published by the CWI as note CS-N8405.

The B *Programmer's Handbook*
Leo Geurts, Lambert Meertens, and Steven Pemberton, 80 pages.
A handbook containing a quick look at *B*, a guide to using the current implementations of *B*, and a description of *B*. Consists of three previously published documents, thoroughly revised and updated: *B* without Tears, A User's Guide to the *B* System, and Description of *B*. Published by CWI, ISBN 90.6196.295.1, price Dfl. 12.50.

CWI publications can be ordered from

Publications Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will be invoiced. The prices quoted exclude postage and packing, and for foreign orders there is an additional charge of Dfl. 7.50 to cover bank charges.

B for the IBM PC

A prototype version of *B* is now available for the IBM PC and compatibles. This version is functionally equivalent to the one currently distributed for UNIX systems. For ordering information, see the form at the end of this newsletter.

Features of the implementation

- The full *B* language as described in the Draft Proposal is implemented.
- The structured editor is used to enter and edit units, immediate commands, input to READ commands, and permanent targets.
- The editor suggests possible command continuations and closing brackets. It uses function and arrow keys to move the focus around, change its size, etc.. You can *undo* the last 20 key strokes. Text can be moved or duplicated within or between units and immediate commands. A sequence of keystrokes can be recorded, and played back later. You can recall the last command. (A more detailed description of the *B* editor can be found in the first newsletter.) By default all editing operations are bound to single keys: you can rebind the editing operations to other than the default keys, to suit your own taste, or to overcome deficiencies in your particular keyboard.
- Since different compatibles have different ways of addressing the screen, and not all screens have the same size, you can define an environment variable to reconfigure *B* for use with the "ANSI" screen driver, or for different screen widths and heights. There is a program supplied to help you decide what sort of screen you have.
- There are utility programs for such things as workspace recovery, and for listing the units in a workspace.

Small version

The full implementation needs at least 384 K bytes. There is a smaller version available, without the built-in editor, for those with only 256 K bytes. In this case commands must be typed in in full. To edit units the EDLIN editor is used.

Documentation

The documentation sent with the package includes "The *B* Programmer's Handbook" and a quick reference guide. The book describes the *B* language proper, the use of the system, the editor commands, and the use of the other uitilities. There are some example workspaces on the distribution disk.

Required system configuration

In order to run the *B* system you must have an IBM PC or compatible with

- at least 384 K bytes, or 256 K bytes for the small version (this includes space used for MS-DOS);
- MS-DOS version 2.x (we've only tested it with 2.0 and 2.11, but it may also run on MS-DOS 3.x or higher);
- one double-sided disk drive.

The system is expected to run on most IBM-PC compatible computers, but we cannot guarantee this, because we haven't tried them all. We know, however, that the system runs at least on the following compatibles: Olivetti M24, Apricot Portable, and Apricot F1. (If your copy runs fine on a machine which is not on the list, please tell us so we know we can extend the list. If your copy doesn't run, also tell us and we'll try to see what is the cause.)

Beware

THIS IS NOT YET A PRODUCTION VERSION!

The system is sometimes slow, and imposes severe limits on the maximum sizes of the targets and units in the work space. We *do* appreciate reports of bugs, but we don't promise we'll fix them: we'll do what we can.

Finally

The disk is not copy protected. You may make copies, and give them away, as long as you don't sell the copies, and as long as these same conditions are passed on to the people you give copies to. Fair enough?

Eh? *B* be 'ABC', see?

Lambert Meertens

In the last issue of the Newsletter, the question was raised which name to give to the definitive version of the *B* programming language, after revision. (See the article 'What is in the name of *B*?'.) We received several reactions, some just expressing (dis)like of one or more of the proposed alternatives, some offering new proposals. Among the new proposals were 'Bravo' (from the way of spelling the alphabet: 'Alpha Bravo Charlie ...'), 'Best', 'QUI' (sounds like 'CWI') and 'Lingo' (twice). One proficient name finder offered us no fewer than eleven suggestions, and her list suggested to us some more possibilities, such as 'Love'. (Why 'Banana', though? In honour of Steven Pemberton's predilection for this fruit?) Some of these proposals we discarded forthwith. Although we wholeheartedly agree with the sentiment expressed by 'Bravo' and 'Best', we feared that such a name might be perceived as a form of boasting that could have an adverse effect on reaching our objectives. Better to have a slightly more neutral name, and let the language speak for itself.

Although 'QUI' sounds like 'CWI' if 'CWI' is pronounced like 'QUI', no-one here pronounces it that way; we say 'Say Way Ee' or 'See Double You Eye', depending on whom we are addressing. As to 'Lingo', I have seen several languages tagged with that name—although I am not sure any of them are in active use, it seems better, after our previous experience, to avoid another potential conflict.

Considering all, and counting noses (including those of our own), we decided on 'ABC'. The merits of this name have already been expounded in the previous article and will not be repeated here. To avoid confusion, we shall keep referring to the present, unrevised, language as '*B*'.

Some people expressed the fear that 'ABC' might give legal problems, e.g., because of the broadcasting corporation, or because of a line of computers with that name. We are not really worried about that. In the first place, 'ABC' is just the name of a language, which by itself is not a tradeable commodity, and our understanding is that you can name such things as you please, just like you could name your daughter 'Ada' as long as you do not try to sell her. The 'ABC' of the broadcasting company is not a name by which they trade, either. Also, if you look in the yellow pages of the Manhattan telephone directory (and presumably of most other places), you will find that 'ABC' is one of the most common names given to businesses. Since these are obviously left alone, we don't see how we could be bothered, especially since there are no conflicting interests. There is even a second television company calling itself 'ABC'; British based I think.

As to the line of computers, I have a leaflet describing a line of computer terminals called 'ABC'. Maybe the two companies involved (both European) can fight this out between themselves before they descend on us. Also, there is a line of computers called 'Basic' (where have we heard that before?), and a new line of terminals called 'Elan', which, as a language, has been in use at least in the FRG and the Netherlands for a long time. By the way, we have a line of products in the Netherlands for fixing things around the house, called 'Rambo'. Not a bad name for *B* either. ('Rambo, the programming language that shows no mercy.')

A Program Example: Polynomials

Leo Geurts

A polynomial such as $7x^5 - 4x^4 + 3x - 1$ may be represented in *B* as a table with the powers as keys and the coefficients as associates:

```
{[0]: -1; [1]: 3; [4]: -4; [5]: 7}
```

This representation is convenient for most operations one wants to perform on polynomials. As *B* does not impose limits on the size of tables, nor on the magnitude of the numbers involved, this representation may be used for all polynomials of one variable with real coefficients. The simple package presented here consists of a section of functions such as addition, multiplication and evaluation, an input command, and a somewhat elaborate output section. The package is intended for use with exact coefficients.

The command `INPUT` facilitates the typing of polynomials one wants to work with. Entering the example polynomial above proceeds like this:

```
>>> INPUT a
coefficient: 7
power: 5
coefficient: -4
power: 4
coefficient: 3
power: 1
coefficient: -1
power: 0
coefficient: 0
>>>
```

The resulting value of `a` may be shown by simply writing it:

```
>>> WRITE a
{[0]: -1; [1]: 3; [4]: -4; [5]: 7}
```

or, for a fancier result, by using the special `PRINT` command:

```
>>> PRINT a
   5     4
7 X - 4 X + 3 X - 1
```

The functions supplied enable us to have the fourth power of `a` computed and stored:

```
>>> PUT a pow 4 IN a4
```

and have the second derivative printed of `a` subtracted from `a4`:

```
>>> PRINT 2 der (a4 minus a)
        18           17           16           15           14           13
912380 X  - 1876896 X  + 1439424 X  - 487424 X  + 1049280 X  - 1769880 X  +

           12           11           10          9          8         7         6
+ 1161888 X  - 329472 X  + 383064 X  - 526680 X + 285660 X - 65664 X + 47712 X -

          5         4        3        2
- 49896 X + 20520 X - 3580 X + 1212 X - 648 X + 108
```

Of course, instead of using INPUT, it is perfectly possible to type the table involved, as shown in this command to print the eleventh line of Pascal's triangle:

```
>>> PRINT {[1]: 1; [0]: 1} pow 10
 10       9       8       7       6       5       4       3       2
X  + 10 X + 45 X + 120 X + 210 X + 252 X + 210 X + 120 X + 45 X + 10 X + 1
```

The Package

```
YIELD a plus b: \                                          sum of two polynomials
   FOR q IN keys b:
      SELECT:
         q in keys a:
            PUT a[q]+b[q] IN a[q]
            IF a[q] = 0:
               DELETE a[q]
         ELSE:
            PUT b[q] IN a[q]
   RETURN a

YIELD a minus b: \                                     difference of two polynomials
   FOR q IN keys b:
      SELECT:
         q in keys a:
            PUT a[q]-b[q] IN a[q]
            IF a[q] = 0:
               DELETE a[q]
         ELSE:
            PUT -b[q] IN a[q]
   RETURN a

YIELD a times b: \                                       product of two polynomials
   PUT {} IN res
   FOR p IN keys a:
      FOR q IN keys b:
         PUT p+q IN s
         SELECT:
            s in keys res:
               PUT res[s]+a[p]*b[q] IN res[s]
               IF res[s] = 0: DELETE res[s]
            ELSE:
               PUT a[p]*b[q] IN res[s]
   RETURN res
```

```
YIELD a pow n: \                                          n-th power of polynomial a
   CHECK n > 0 AND n mod 1 = 0
   SELECT:
      n = 1:
         RETURN a
      n mod 2 = 0:
         PUT a pow (n/2) IN b
         RETURN b times b
      ELSE:
         RETURN (a pow (n-1)) times a

YIELD der a: \                                                    first derivative
   PUT {} IN res
   FOR p IN keys a:
      IF p <> 0: PUT p*a[p] IN res[p-1]
   RETURN res

YIELD n der a: \                                                  n-th derivative
   CHECK n >= 0 AND n mod 1 = 0
   IF n = 0: RETURN a
   RETURN (n-1) der der a

YIELD a at x: \                          value of a when its variable has value x
   PUT 0 IN s
   FOR p IN keys a: PUT s+a[p]*x**p IN s
   RETURN s

YIELD a zero (u, v): \                       a zero of polynomial a, u < zero <= v
   SHARE eps \ must already have been given a value
   CHECK eps > 0
   PUT a at u, a at v IN au, av
   CHECK sign au <> sign av
   IF au > av: PUT u, v IN v, u
   WHILE abs (u-v) > eps:
      PUT ~(u+v)/2 IN mid \ no need for exact arithmetic
      SELECT:
         a at mid < 0: PUT mid IN u
         ELSE: PUT mid IN v
   RETURN v

HOW'TO INPUT a: \                                                 read polynomial
   PUT {} IN a
   READ'COEFF
   WHILE coeff <> 0: \ coeff 0 indicates end of input
      READ'POWER
      PUT coeff IN a[p]
      READ'COEFF
READ'COEFF:
   WRITE "coefficient: "
   READ coeff EG 0
READ'POWER:
   WRITE "power: "
   READ p EG 0
   WHILE p < 0 OR p mod 1 <> 0:
      WRITE "Enter non-negative integer: "
      READ p EG 0
```

```
HOW'TO PRINT aa: \                                          display a polynomial
   SHARE lo, hi
   PUT "", "" IN lo, hi
   PUT aa IN a \ to prevent re-evaluation of possibly complex parameter aa
   SELECT:
      a = {}: LO "0"
      ELSE:
         FOR i IN {-#a..-1}: TERM \ handle terms in order of descending powers
   OUTPUT
TERM:
   PUT (-i) th'of keys a IN p
   SIGN
   COEFF
   XPOWER
SIGN:
   SELECT:
      a[p] < 0: LO "-"
      a[p] > 0:
         IF p < max keys a: LO "+"
COEFF:
   IF abs a[p] <> 1 OR p = 0: LO abs a[p]
XPOWER:
   IF p > 0: LO "X"
   IF p > 1: HI p

HOW'TO OUTPUT: \                                  (show output prepared by PRINT)
   SHARE hi, lo
   PUT 80 IN width
   PUT "", "" IN hi1, lo1
   WHILE #hi > width:
      PUT break IN end, begin
      PUT hi|end, hi@begin IN hi1, hi
      PUT lo|end, lo@begin IN lo1, lo
      WRITE hi1 /
      WRITE lo1 //
   WRITE hi /
   WRITE lo /
break: \ find place for one line to stop and for the next line to start
   PUT width IN s
   WHILE s > 0 AND s th'of lo not'in "+-":
      PUT s-1 IN s
   IF s > 0: RETURN s, s
   RETURN width, width+1

HOW'TO HI tt: \                                  (for the upper line of a display)
   SHARE lo, hi
   PUT "`tt`" IN t
   PUT lo^(" "^^#t), hi^t IN lo, hi

HOW'TO LO tt: \                                  (for the lower line of a display)
   SHARE lo, hi
   PUT "`tt`" IN t
   IF lo > "" AND lo@#lo <> " ":
      PUT " "^t IN t
   PUT lo^t, hi^(" "^^#t) IN lo, hi
```

(extremely) Simple Logic Programming in *B*

Tim Budd

A style of programming has recently become popular in which a desired objective is expressed in terms of "Horn clauses", which are a form of simple logical predicates. The task of the computer is then to determine if it is possible to satisfy the clauses. Such a style of expression is often called logic programming, although the name is somewhat misleading and "predicate checking" might be a more appropriate term.

In this note we will show, by means of a simple example, how problems formulated in this manner can sometimes be converted in a straightforward fashion into *B* programs.

For example, a traditional logic problem concerns a farmer, a wolf, a goat and some cabbage. It happens that all four are sitting on one bank of the Thames, and want to get to the other bank. Unfortunately, the small boat the farmer has can only carry himself and at most one of the other three passengers. Worse yet, if the farmer leaves the wolf alone with the goat then the wolf will eat the goat, and similarly the goat and the cabbage. (Apparently, the wolf has no taste for cabbage and is smart enough to avoid eating the farmer). Can the farmer safely carry his three passengers to the other bank? And if so what are the valid sequence of moves?

The formulation we will use is adapted from Kowalski[1]. Let us use the names south and north to represent the two sides of the Thames, and use a four position compound to represent the locations of the farmer, wolf, goat and cabbage, respectively. Let us assume that initially all four are all on the south side of the Thames. (For simplicities sake we will use numbers, such as 0 and 1, in the variables south and north, thus making the conversion from one to the other easier). We first define a predicate which is true if and only if the four protagonists are in a safe position; that is, one in which no damage can be done to any of them. From the specifications, we have that

> (either the farmer is with the goat, OR the goat is not with the wolf) AND
> (either the farmer is with the goat, OR the goat is not with the cabbage)

This can be simplified to

> either the farmer is with the goat, OR
> neither the wolf nor the cabbage is with the goat

Which can be translated directly into *B* as follows:

```
TEST safe (farmer, wolf, goat, cabbage):
   REPORT farmer = goat OR cabbage <> goat <> wolf
```

The harder part of the problem is then to determine whether the four passengers can be moved from one bank to the other. Again, we can formulate this in terms of a predicate. Let reachable(farmer, wolf, goat, cabbage) represent the set of states reachable from the initial positions, that is states for which there exists a sequence of moves starting from the initial position and such that every move results in a safe state. We can formulate information about reachable as follows:

reachable(south , south , south , south) is true
> This is our initial condition with all passengers on one side of the Thames.

reachable(farmer, wolf, goat, cabbage) is true if
> safe(opposite farmer, wolf, goat, cabbage) is true and
> reachable(opposite farmer, wolf, goat, cabbage) is true.

This relation corresponds to the farmer moving by himself from one bank to the other. The original state was reachable if it could be reached by the farmer moving himself, and the state he moved from was both safe and reachable. Here `opposite` is a unit which converts a value from one to zero or vice versa. It can be given as follows:

---

1. "Logic as a Computer Language", Robert Kowalski, in *Logic Programming*, edited by K. L. Clark and S. -A. Tärnlund, Academic Press 1982. While discussing the programming language Prolog, Kowalski merely provides the formulation, and does not give a solution in that language. In fact the naive Prolog solution fails for exactly the same reason as the naive *B* solution discussed in the text.

```
YIELD opposite position: RETURN 1-position
```

Three other predicates express the farmer moving one of the other passengers.

reachable(farmer, wolf, goat, cabbage) is true if
safe(opposite farmer, opposite wolf, goat, cabbage) is true and
reachable(opposite farmer, opposite wolf, goat, cabbage) is true.

reachable(farmer, wolf, goat, cabbage) is true if
safe(opposite farmer, wolf, opposite goat, cabbage) is true and
reachable(opposite farmer, wolf, opposite goat, cabbage) is true.

reachable(farmer, wolf, goat, cabbage) is true if
safe(opposite farmer, wolf, goat, opposite cabbage) is true and
reachable(opposite farmer, wolf, goat, opposite cabbage) is true.

The objective can then be expressed as "is reachable(north,north,north,north) true?" This can be determined by directly coding the information about reachable into a *B* TEST unit. The unit attempts to determine if a given position is reachable by checking (recursively) the legal moves from the initial position to the current position. Of course, if the current position is not safe we can say immediately that it is not reachable.

```
TEST reachable locations:
   SHARE south
   REPORT safe locations AND (initial'location OR next'move'reachable)
initial'location:
   REPORT locations = (south,south,south,south)
next'move'reachable:
   PUT locations IN (farmer, wolf, goat, cabbage)
   REPORT farmer'move OR wolf'move OR goat'move OR cabbage'move
farmer'move:
   REPORT reachable(opposite farmer, wolf, goat, cabbage)
wolf'move:
   REPORT reachable(opposite farmer, opposite wolf, goat, cabbage)
goat'move:
   REPORT reachable(opposite farmer, wolf, opposite goat, cabbage)
cabbage'move:
   REPORT reachable(opposite farmer, wolf, goat, opposite cabbage)
```

If one now types in this program and executes it, one would notice that it seems to take a very long time to execute. Inserting a WRITE statement in to print out the values being tested it becomes clear what is happening. The poor farmer is ferrying his goat back and forth from one bank to the other, and will do that indefinitely if we permit him. The problem is an inherent difficulty in this style of programming, and illustrates one of the pitfalls of the technique. We thus show how in many cases this difficulty can be overcome.

We can express our objective as a *goal* we desire to fulfill, in this case the goal is to show that reachable(north,north,north,north) can be satisfied. What the program is then doing is to repeatedly refine this into a series of sub-goals. For example our initial goal is refined into a sub-goal:

reachable(n, n, n, n)? ⟶ reachable(s, n, n, n)?

The safe condition here fails, and so the second alternative (moving the wolf along with the farmer) is tried and also fails. The third alternative, moving the goat, finally passes the safe test. Thus the goal reachable(south,north,south,north) is examined, and various subgoals are examined in an attempt to show it is satisfiable.

reachable(n, n, n, n)? ⟶ reachable(s, n, s, n)? ⟶ reachable(n, n, s, n)?

Because it is the series of goals that drive the program, this technique is sometimes referred to as *goal-directed* programming. The problem arises if along the way some goal appears twice in this chain:

... ⟶ A ⟶ ... ⟶ A

Here clearly we are going to get into a loop, since if attempting to satisfy A we must show that we can satisfy A, then to satisfy A we will eventually be required to show we can satisfy A, and so on. In this particular problem the appropriate solution is to FAIL on repeated goals, since if a solution exists a solution without loop must exist. (Unfortunately, this cannot be stated as a general principle, and each problem must be examined individually to determine the correct actions).

To implement this, we keep a list of the goals currently being attempted in a global list named `looking`. Initially the list is empty. As each new goal is attempted we first see if it is in the list and fail if so, otherwise we insert it into the list. Thus the refined solution is as follows (we have also added statements to write out the final solution).

```
TEST reachable locations:
   SHARE south, looking
   IF initial'location: SUCCEED
   IF locations in looking OR NOT safe locations: FAIL
   INSERT locations IN looking
   REPORT next'move'reachable
initial'location:
   REPORT locations = (south,south,south,south)
next'move'reachable:
   PUT locations IN farmer, wolf, goat, cabbage
   REPORT farmer'move OR wolf'move OR goat'move OR cabbage'move
farmer'move:
   IF reachable(opposite farmer, wolf, goat, cabbage):
      WRITE 'move farmer to other bank' /
      SUCCEED
   FAIL
wolf'move:
   IF reachable(opposite farmer, opposite wolf, goat, cabbage):
      WRITE 'move farmer and wolf to other bank' /
      SUCCEED
   FAIL
goat'move:
   IF reachable(opposite farmer, wolf, opposite goat, cabbage):
      WRITE 'move farmer and goat to other bank' /
      SUCCEED
   FAIL
cabbage'move:
   IF reachable(opposite farmer, wolf, goat, opposite cabbage):
      WRITE 'move farmer and cabbage to other bank' /
      SUCCEED
   FAIL
```

An important fact to note is that it is not necessary to remove locations from `looking`. Upon return from an invocation from the unit `reachable`, the global environment (including `looking`) is reset completely to the state it possessed prior to the unit being invoked. This is a two-edged sword, however, since in some cases one would like to modify the global environment and that then becomes more difficult.

Having defined the units, we can execute the program and produce the result.

```
>>> PUT 0,1 IN south,north
>>> PUT {} IN looking
>>> CHECK reachable(north, north, north, north)
move farmer and goat to the other bank
move farmer to the other bank
move farmer and cabbage to the other bank
move farmer and goat to the other bank
move farmer and wolf to the other bank
move farmer to the other bank
move farmer and goat to the other bank
```

~1 <> 1, A Nice Distinction?

Lambert Meertens

Numbers in *B* come in two kinds, 'exact' (rational) and 'approximate' (floating point). The distinction is made at run time. In most versions of BASIC all numbers are handled as approximate. We felt that this was undesirable, and that the user must be allowed control over 'exact' quantities, not subject to rounding errors. Most programming languages allowing this have a finite range of integers for the exact domain. For *B* we chose rational numbers, rather than only integers. This, and the absence of limitations on the size, are an additional service to the user. Not all computations can be performed exactly (think of `sin` etc.). We therefore also need approximate numbers.

On the other hand, we did not want to have two different (static) *types* of numbers in the type system. The user should have no need to worry about the distinction exact/approximate when it is not important. In particular, 'mixed arithmetic', adding exact and approximate numbers, must be allowed. It is only reasonable then to allow 'mixed numbers' to be put in the same target too, and it would be unreasonable to give a different semantic meaning to `PUT 0 IN x` dependent on whether all other assignments to `x` are exact, or some are approximate.

Because exact computations can be much more expensive than approximate ones, we decided that approximateness should propagate upwards in evaluating arithmetic expressions. This gives the user a simple way to specify approximateness throughout a complicated computation, as in

```
PUT ~1 IN s
FOR i IN {1..1000}:
   PUT s+1/i IN s.
```

This takes much longer if `~1` is replaced by `1`.

The approach taken has one fundamental problem. If `x` is approximate, `x-x` is not identical to `0`. For approximateness propagates, and the approximate result of the subtraction is therefore not an exact number.

For that reason, the test `x-x = 0` will fail if `x` has an approximate number as value. At least, that is the way things are defined in the *Draft Proposal* ('*DP*'). The relevant wording there is (1.2.1.b):

> The numbers are ordered according to their arithmetic magnitude, with some tie breaking rule (not further specified, but consistent) for exact and approximate numbers with the same magnitude.

The fact that the test `x-x = 0` can fail means that sometimes the user does have to worry. We chose this solution because we felt that the user should be careful anyway when comparing approximate numbers and has no business to expect exact answers.

Mark 0, the first implementation of *B*, did not stick to *DP* in this respect: the test `0 = ~0` did succeed. In a mixed comparison `x = y` (one exact and one approximate operand) the exact operand was converted to floating point (possibly entailing loss of precision) before performing the comparison. In other words, numbers could be *equal* but *not identical*. Comparison proceeded there as suggested by the following *B* code:

```
TEST x less'than y: \Mark 0
   SELECT:
      exact x AND exact y:
         REPORT x < y
      ELSE:
         REPORT ~x < ~y.
```

This means that if `x = y` succeeds, `x+z = y+z` can still fail. For example, although in Mark 0 we had `~1 = 1`, it was the case that `~1+10**-99 < 1+10**-99`. (In particular, `~1+10**-99 = ~1 = 1 < 1+10**-99`.) If we could guarantee equality in this and all similar cases, the distinction between exact and approximate would vanish! The approach of Mark 0 has one nasty consequence. It is possible that the test `x = y = z` succeeds, whereas `x = z` fails. It is not hard to imagine the paradoxical effects you can get if such values `x`, `y` and `z` are used as keys of a table—although, in actual programming, I don't remember having run across problems of this sort.

The Mark 1 version of the *B* system (the current version) does adhere to *DP*: `1` and `~1` are guaranteed to be different. In comparing the two, we find `~1 > 1`. Comparison in Mark 1 proceeds as suggested by the following *B* code:

```
TEST x less'than y: \Mark 1
   SELECT:
      exact x AND exact y:
         REPORT x < y
      ~x < ~y: SUCCEED
      ~x > ~y: FAIL
      ~x = ~y:
         REPORT exact x AND NOT exact y
```

Since Mark 1 became available, I have several times run into the obvious mistake of comparing `x = y` where I should have written `~x = ~y` or something like that. The approach of *DP* is bug prone. It is very surprising that the two tests `x > 0` and `-x > 0` can succeed simultaneously (since `-~0 = ~0`). This easily gives rise to infinite loops or other logic errors. For that reason, we are reviewing this decision, to see if we can do better in ABC.

Here are some pleasant properties that we would like to have:

(i) If `x = y`, then `x-y = 0`;
(ii) If `x = y`, then `x+z = y+z`;
(iii) If `x = y = z`, then `x = z`.

It is impossible to have (i) and (ii) together without dropping the distinction between exact and approximate. As we saw, Mark 0 had (i), but not (ii) and (iii), whereas *DP* and Mark 1 have (ii) and (iii), but not (i). However, it is also possible to have (i) and not lose (iii). So the question is which of the two, (i) and (ii), is the more important to keep.

Property (ii) is a special case of the 'axiom of referential transparency'. This 'axiom' is:

For all `x` *and* `y`, *and any function* `f`,
if `x = y`, *then* `f x = f y`.

What this really amounts to is the statement that 'equal' (tested by `=`) is the same as 'identical' (in so far as properties can be observed by functions). This is a desirable property indeed, no denying it. All I can say that from the years I have used the Mark 0 system, I cannot remember a single logic error in my programming caused by mistaken reliance on this property. Maybe there were a few, but there cannot have been too many. But since using Mark 1, errors because of falsely assuming property (i) to hold, have abounded.

For instance, take these commands that calculate the number of years necessary to pay off a mortgage of 28,000 at an interest of 8%, paying 2,300 per year:

```
PUT 1985, 28E3 IN y, m
WHILE m > 0:
   PUT 1.08 * m IN m
   PUT min {m; 2.3E3} IN dm
   PUT y + 1, m - dm IN y, m
WRITE y /
```

This reaches an infinite loop, with `m = ~0`. (As an exercise, try to find out where problems arise in Leo Geurts' polynomials program in this newsletter.)

Moreover, although property (ii) holds in Mark 1, the full axiom of referential transparency does not. Numbers may be equal, and yet not identical. This is the result of an undocumented feature:

```
>>> PUT 1, 1.00 IN x, y
>>> IF x = y: WRITE x, y /
1 1.00
>>> WRITE #'`x`', #'`y`' /
1 4
>>>
```

So the 'function' convert-to-text yields different results for otherwise equal values. This was put in on purpose, and we have no intention of taking it out.

So what is the new proposal? The idea is to do a conversion on one of the two operands in a mixed comparison `x = y`, but not from exact to approximate, as in the first implementation, but the other way around. For in all implementations of approximateness, the approximate numbers are, *mathematically*, a subset of the rational numbers, usually of the form `p/q` with `q` a power of 2.

So assume that we have a function `exactly` that returns an *exact* number, equal in arithmetic magnitude to the value of its operand. For two numbers, the order-test `x < y` is, under the new approach, equivalent to `exactly x < exactly y`. A definition that works in Mark 1 is:

```
YIELD exactly x:
   IF x <> ~x: \exact x
      RETURN x
   PUT round x IN r
   SELECT:
      x = ~r: RETURN r
      ELSE:
         RETURN r+0.5*exactly(2*(x-r))

TEST x less'than y: \New Proposal
   REPORT exactly x < exactly y.
```

Other order-signs work similarly. For compounds

with two fields, we have (as is the case already), that the order-test `(x1, x2) < (y1, y2)` is equivalent to `x1 < y1 OR (x1 = y1 AND x2 < y2)`. It is obvious how to generalize this to arbitrary compounds, and also to the comparison of lists and tables. In particular, we have, for example,

```
4#{~4} = 1;
{[1]: 2}[~1] = 2.
```

Sometimes it is important to be able to test for the exactness or approximateness of a number. Currently, a comparison like `x <> ~x` can be used to that purpose, but no longer so under the new proposal. Therefore, we would need a new proposition `exact` to test for exactness.

Both under the current and the proposed semantics, we have `{1; ~1} = {~1; 1}`. Since the entries in lists are ordered, the question is which of the two comes internally first. Under the current semantics, `1 < ~1`, so the exact 1 comes before the approximate 1 in the list (even though, written, we see `{1; 1}` appear). Under the proposal, the two possible lists with different internal orderings would be equal but not identical. There are two possible approaches here:

(i) Who cares about that ordering; after all the two possibilities are equal;

(ii) Define some (arbitrary) ordering for equal but not identical values, like we have now for the equality test, but only to be used for sorting list and table entries.

Personally, my preference is option (i). This is in fact the way in which `{1; 1.00}` is handled now in Mark 1.

It is probably the case that mixed comparisons will be slower under the proposed semantics. But mixed comparisons are relatively rare, and most cases will involve a comparison against the exact number `0`, and then `1`, which can easily be recognized as special cases that can be handled much faster. Also, I do not like the current implementation of mixed comparisons in Mark 1; even if it is decided to stick to `~1 <> 1`, then I still feel that the tie-breaking should apply only if the numbers involved have the same arithmetic magnitude, in other words, if `exactly x = exactly y`, and not as soon as `~x = ~y` (see the wording in *DP*). In that case, any advantage in terms of speed will be lost anyway.

To be honest, I should mention one more drawback of the proposal. The property that `x-y = 0` implies `x = y` is lost too: take `x = ~1` and `y = 1+10**-99`. In the subtraction, `y` is converted first to an approximate number, yielding `~1`.

Let me finish with a surprising fact that I discovered during this investigation. One more nice property is 'monotonicity': if `x < y`, then `x+z <= y+z`. This property does not hold under the proposal, and I thought initially that this was another drawback. However, it does not hold under *any* of the alternatives considered either.

HOW TO ORDER *B* for the IBM PC:

To order the prototype of the implementation of *B* for the IBM PC or compatibles, running under MS-DOS versions 2.0 (or higher), you should fill in the order form below, and send it together with a cheque or money order for Dfl. 100 or US $ 35 to:

B Group, PC distribution
Informatics / AA
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will then receive:

- a floppy with the binary;
- The *B* Programmer's Handbook;
- a *B* Quick Reference Card.

ORDER FORM

Please send me a copy of the prototype *B* system for the IBM PC, including documentation.
I enclose a cheque or international money order for Dfl 100 (or US $ 35) to cover materials and handling.

Name: .....

Firm/Institute: .....

Address: .....

.....

Country: .....

Telephone: .....

Network address: .....

Machine(s): ☐ IBM PC ☐ IBM XT ☐ Olivetti M24
☐ Apricot Portable ☐ Apricot F1 ☐ other: .....

Required media:

☐ 5¼" double sided, double density floppy disk
☐ 3½" double sided floppy disk

Signature and Date:

HOW TO ORDER *B* for UNIX:

To order the Mark 1 implementation of *B*, running on UNIX† systems, you should fill out the order form below, and two signed copies of the SOFTWARE AGREEMENT on the next page. Send it to:

B Group, Unix distribution
Informatics / AA
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will then receive:

- a tape with the sources (including an installation guide)
- the following documentation:
  - Description of *B*
  - A Users Guide to the *B* System
  - *B* Quick Reference Card
  - Manual Pages

Also, one of the two copies of the SOFTWARE AGREEMENT will be returned to you signed.

---

† UNIX is a Trademark of AT&T Bell Laboratories

ORDER FORM

Please send us the Mark 1 implementation of *B* for UNIX systems for the price of Dfl 100 (US $ 35) (to cover materials, postage and bank charges) for which we will be invoiced.

Name: .....

Firm/Institute: .....

Address: .....

.....

Country: .....

Telephone: .....

Internet network address: .....

Machine type: ☐ Vax ☐ Sun ☐ PDP ☐ other:

Operating System: ☐ 4.2 BSD ☐ Version 7 ☐ System V ☐ other:

Check required tape parameters:

density	☐ 800 bpi	☐ 1600 bpi
format	☐ Tar, blocksize 1	☐ Ansi D format
	☐ Tar, blocksize 20	☐ Ansi F format

(For other media and formats, please inquire.)

We include **two** copies, both **signed**, of the SOFTWARE AGREEMENT.

Signature and Date:

Please, fill out **both** copies below, and **sign** them.

SOFTWARE AGREEMENT

Effective as of ..... 198.., Stichting Mathematisch Centrum (SMC), having an office at 413 Kruislaan, 1098 SJ Amsterdam, and

(LICENSEE), having an office at

agree as follows:

SMC grants fee-free to LICENSEE a personal, non-transferable and non-exclusive right to use the computer programs and documentation relating to the Mark 1 implementation of *B* (LICENSED SOFTWARE).

LICENSEE agrees not to sell, lease or otherwise transfer or dispose of the LICENSED SOFTWARE in whole or in part.

SMC makes no warranties, express or implied. SMC shall not be held to any liability with respect to any claim by LICENSEE or a third party on account of, or arising from, the use or the inability to use LICENSED SOFTWARE.

Neither this agreement nor any rights hereunder, in whole or in part, shall be assignable or otherwise transferable.

signed by

For LICENSEE	For Stichting Mathematisch Centrum
Name:	Name: .....
Title:	Title: Director of Software Licensing
Signature and Date:	Signature and Date:

SOFTWARE AGREEMENT

Effective as of ..... 198.., Stichting Mathematisch Centrum (SMC), having an office at 413 Kruislaan, 1098 SJ Amsterdam, and

(LICENSEE), having an office at

agree as follows:

SMC grants fee-free to LICENSEE a personal, non-transferable and non-exclusive right to use the computer programs and documentation relating to the Mark 1 implementation of *B* (LICENSED SOFTWARE).

LICENSEE agrees not to sell, lease or otherwise transfer or dispose of the LICENSED SOFTWARE in whole or in part.

SMC makes no warranties, express or implied. SMC shall not be held to any liability with respect to any claim by LICENSEE or a third party on account of, or arising from, the use or the inability to use LICENSED SOFTWARE.

Neither this agreement nor any rights hereunder, in whole or in part, shall be assignable or otherwise transferable.

signed by

For LICENSEE	For Stichting Mathematisch Centrum
Name:	Name: .....
Title:	Title: Director of Software Licensing
Signature and Date:	Signature and Date: