stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 208/82 SEPTEMBER

L.J.M. GEURTS

AN OVERVIEW OF THE *B* PROGRAMMING LANGUAGE
OR
B WITHOUT TEARS

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, 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.).

1982 CR. Categories: D.3.2, D.3.3

AN OVERVIEW OF THE *B* PROGRAMMING LANGUAGE

OR

B WITHOUT TEARS†

by

L.J.M. Geurts

ABSTRACT
B is a powerful, easy-to-learn and easy-to-use interactive programming language, designed as a modern alternative to BASIC. This paper gives an informal introduction to the language, meant for those who are already familiar with one or two other programming languages.

KEYWORDS & PHRASES: programming languages, *B*

†This report will be submitted for publication elsewhere.

0. INTRODUCTION

B is a powerful, easy-to-learn and easy-to-use interactive programming language, intended for personal computing and designed as a modern alternative to BASIC. It supports data structuring and structured programming. What started as a beginners' programming language turns out to be a fascinating tool for novices and experts alike, even in the slow experimental implementation existing now.

B is not simply a language, it is a language embedded in a dedicated system containing a syntax-directed screen editor, 'file' maintenance functions, facilities for background processes, etc. Work on the *B* system is still in progress, but a definition of the language itself has already been published.* This definition is a proposal; suggestions for improvements are welcome. Since definitions are hard to read, this paper gives an informal overview of the language, meant for those who are already familiar with one or two other programming languages. (Remark: '*B*' is only a working title, not the definitive name.)

1. AN EXAMPLE

```
YIELD gcd (a, b):
    PUT abs a, abs b IN a, b
    CHECK a > 0 OR b > 0
    WHILE b > 0:
        PUT b, a mod b IN a, b
        \mod gives the remainder of division
    RETURN a
```

Once this piece of *B* (a function definition) has been entered, the command

```
WRITE gcd (21, 28)
```

writes 7 on the screen. The body of the YIELD-unit is straightforward:

- PUT has an expression to the left of IN, and a *target* to the right. As we shall see later, this multiple PUT in fact puts one *compound* expression in one *compound* target; in the same way, gcd is actually a monadic function with one compound parameter. PUTting IN a parameter of a function is allowed but does not affect the environment. It is impossible for a function (or, for that matter, any other expression) to alter its environment: it can only deliver a value.
- A CHECK keyword is followed by a test. If the test fails, an appropriate message is given and execution halts.
- A control command, such as a WHILE, controls the indented commands following the corresponding colon. Thus, the user need not bother about BEGIN ... END pairs or other delimiters for grouping commands. For the same reason, the whole body of the YIELD has been indented. If there is only one controlled command, the whole construction may be displayed on a single line:

```
IF a < 0: PUT -a IN a
```

- The line starting with \ is a comment. It could have been appended to the previous line.

In the *B* system a friendly editor takes care of the tedious aspects of typing the program, such as indentation and other layout matters, capitals, keywords, etc.

---

* Lambert Meertens, *Draft Proposal for the B Programming Language — Semi-Formal Definition*, Mathematical Centre, Amsterdam, 1981.

2. TYPES

The power of *B* is largely due to its carefully designed system of data types and associated operations. There are two basic types — *numbers* and *texts* —, and three structures creating new types from existing ones — *compounds*, *lists* and *tables*.

2.1. Numbers

Integers in *B* are conventional, except that there is no restriction on integer length. Furthermore, integers are just a special case of *exact*, i.e. rational, numbers (integral fractions). For example, `1.25` is an exact number. The monadic operators `*/` and `/*` yield the numerator and the denominator of an exact number. For this purpose, fractions are automatically reduced to lowest terms; so `1.25 = 125/100` is reduced to `5/4`. The following one-liner for the greatest common divisor of two exact positive numbers uses this property:

```
YIELD gcd (a, b): RETURN a / */(a/b)
```

In addition to exact numbers, there are *approximate* (floating point) numbers. The operator `~` converts to an approximate number: `~22/7` does not yield an exact, but an approximate number.

Other operations:

plain arithmetic	`+x, x+y, -x, x-y, x*y, x/y`
to the power	`x**y`
round upwards	`ceiling x`
round downwards	`floor x`
round to nearest integer	`round x`
round to `n` digits after decimal point	`n round x`
sign	`sign x`
absolute value	`abs x`
square root	`root x`
`n`th root	`n root x`
remainder (such that `sign (a mod n) = sign n`)	`a mod n`
natural logarithm	`log x`
logarithm to base `b`	`b log x`
exponential function	`exp x`
trigonometric functions	`sin x, cos x, tan x, atan x`

Mixed arithmetic is allowed, as in `2*sin(x-1)`.

2.2. Texts

`'Merry Christmas'` is an example of a text (*text* being a less esoteric term for *string*).

Operations:

join	`'now'^'here' = 'nowhere'`
repeat	`'-'^^5 = '-----'`
behead	`'nowhere'@3 = 'where'`
curtail	`'nowhere'|2 = 'no'`
number of characters	`#'nowhere' = 7`
number of occurrences	`'e'#'nowhere' = 2`
selection	`4 th'of 'nowhere' = 'h'`
scan the characters	`FOR c IN 'nowhere': PUT freq[c]+1 IN freq[c]`

The behead and curtail operations may be combined to yield any subtext. For example, to obtain the subtext starting at position 3 and having length 4, we may write:

```
'nowhere'@3|4 = 'wher'
```

Such subtexts may also be used as targets. For instance, after

```
PUT 'nowhere' IN word
PUT 'bless' IN word@3|4
```

`word` contains `'noblesse'`.

2.3. Compounds

Compounds are like records, except that they have no field names. They are useful for multiple PUTs, for parameters (as in `gcd` above), for multi-dimensional tables (`t[i, j]`), but also as values to be put in a table (instead of separate values put in separate tables under the same key). Examples:

```
PUT a, b, c IN b, c, a \cyclic permutation
PUT 'May', 22 IN anniversary
PUT 1813, anniversary IN date
PUT date IN year, (month, day)
```

2.4. Lists

Lists are like bags or multi-sets (sets with multiple occurrences allowed), e.g.:

```
{'eye'; 'nose'; 'eye'; 'mouth'}
```

All entries of a list must have the same type. To simplify most operations on lists, they are automatically kept sorted by the system. This is also their standard external representation. For example,

```
WRITE {'u'; 'e'; 'a'; 'y'; 'o'; 'i'}
```

gives

```
{'a'; 'e'; 'i'; 'o'; 'u'; 'y'}
```

and

```
FOR i IN {'ay'; 'i'; 'eye'; 'aye'}: WRITE #i
```

gives

```
2 3 3 1
```

This only works because an order is associated with each type, e.g.:

```
2.99 < 3.00
'az' < 'b'
('b', 'c') < ('z', 'a')
{} < {'a'; 'z'} < {'b'}
```

Instead of `{1; 2; 3; 4; 5; 6; 7}` the notation `{1..7}` may be used; similarly, `{'b'..'d'}` stands for `{'b'; 'c'; 'd'}`.

Operations:

initialize	`PUT {'wart'; 'eye'; 'nose'; 'eye'} IN face`
add entry to list	`INSERT 'mouth' IN face`
remove *one* instance (which must be present)	`REMOVE 'wart' FROM face`
number of entries	`#face = 4`
number of occurrences	`'eye'#face = 2`
scan	`FOR e IN face: INSERT e IN f2`
smallest (= first) element	`min {3; 2} = 2`
largest (= last) element	`max {3; 2} = 3`
selection	`2 th'of {1; 9; 8; 4} = 4`

Because lists are kept sorted there is no need for sorting programs. If a different order is required for the elements of a list, it is often useful to create a list of compounds. In the following example texts are wanted in length order:

```
PUT {} IN llist
FOR text IN list: INSERT #text, text IN llist
```

To output the texts in the desired order:

```
FOR l, text IN llist: WRITE text
```

(Note the use of a compound variable following FOR.)

2.5. Tables

Tables are like arrays, except that the keys (= indices) of a table may be of any type, and need not be consecutive:

```
PUT 1685, ('March', 21) IN birthday['Bach']
PUT 1756, ('January', 27) IN birthday['Mozart']
WRITE birthday
```

gives:

```
{['Bach']: (1685, ('March', 21)); ['Mozart']: (1756, ('January', 27))}
```

All keys of a table must have the same type. All associates (the stored values) must also have the same type, but, of course, the types of the keys and the associates need not be the same. Duplicate keys are not possible.

Operations:

initialize	`PUT {} IN count`
add entries	`FOR feature IN face: PUT 0 IN count[feature]`
modify entries	`FOR feature IN face:` `  PUT count[feature]+1 IN count[feature]`
length	`#count = 3`
number of occurrences (in the associates!)	`2#count = 1`
the list of keys	`keys count = {'eye'; 'mouth'; 'nose'}`
delete entry	`DELETE count['mouth']`
scan (the associates!)	`FOR i IN count: PUT s+i IN s`
smallest associate	`min count = 1`
largest associate	`max count = 2`

Note that the smallest or largest key of a table can be obtained using the keys operator:

```
min keys count = 'eye'
max keys count = 'nose'
```

3. USER-DEFINED COMMANDS AND FUNCTIONS

Just as a YIELD defines a new function, a HOW'TO unit defines a new command:

```
HOW'TO EMPTY s: PUT {} IN s
HOW'TO PUSH v ON s: PUT v IN s[#s+1]
YIELD top s: RETURN s[#s]
HOW'TO POP s: DELETE s[#s]
```

Once these unit definitions have been given, the commands

```
EMPTY numstack
PUSH 3 ON numstack
```

modify the table numstack in the specified way. So a HOW'TO is a procedure in sheep's clothing. Contrary to common practice in other programming languages, the general appearance of user-defined commands is the same as that of predefined commands such as PUT ... IN ... . (However, no control commands of the type WHILE ... : can be defined by the user.)

The above units may be called with parameters of any type, as long as no conflict occurs with the commands inside, e.g.:

```
EMPTY opstack
PUSH '+' ON opstack
POP opstack
```

but not

```
PUT 'ape' IN essence
POP essence
```

since essence[...] is meaningless.

In contrast to YIELDs, the execution of a command defined by a HOW'TO does change the environment when something is PUT IN a parameter: that is the very purpose of commands. Any changes to the environment by a YIELD are executed in a scratch-pad copy of the calling environment, which is thrown away upon termination of the YIELD. In accordance with this, parameter passing to a YIELD is by value, to a HOW'TO by name. Identifiers other than parameters of the unit indicate variables local to the unit. Since units may not be nested, it follows that there are only two scope levels: the global level and the local level.

If, in the example above, there is only a single stack for EMPTY, PUSH, etc. to work on, it is a nuisance to have to pass it as a parameter each time. The definitions above may then be changed to:

```
HOW'TO EMPTY:
    SHARE stack
    PUT {} IN stack
YIELD top:
    SHARE stack
    RETURN stack[#stack]
```

etc. SHARE stack indicates that the identifier stack in this unit will not introduce or use a variable local to the unit, but will refer to a global variable. (In the *Draft Proposal*, the keyword ALLOW is used instead of SHARE.) Other units wishing to refer to the same stack, should now also SHARE stack, since otherwise their use of the name stack would introduce a local variable at that point.

Parameters in both HOW'TOs and YIELDs require no parentheses (except to indicate priorities). The first top function is called thus:

```
PUT top opstack IN op
```

(but, of course, the programmer may write top (opstack) as well), and a call to the second, zeroadic one takes the form:

```
PUT top IN op
```

There are also dyadic functions, such as the predefined function mod.

Functions with more than two parameters have to be modeled as monadic or dyadic ones with the aid of compound parameters, e.g.,

```
YIELD (x, y, z) rotated (pitch, roll, yaw): ...
```

4. CONTROL COMMANDS

There are two selection commands. In addition to

```
IF a < 0: ...
```

(no ELSE allowed), *B* also has

```
SELECT:
    test1: ...
    test2: ...
    test3: ...
```

The first test to succeed determines the alternative to be executed. At least one test must succeed. Instead of the last test, the keyword ELSE may be used, which always succeeds:

```
SELECT:
    e < 0: INSERT e IN pos
    e > 0: INSERT e IN neg
    ELSE: PUT nzero+1 IN nzero
```

For loops, apart from FOR ... IN ..., there is also

```
WHILE test: ...
```

with the usual meaning.

For leaving a HOW'TO before its end, the command

```
QUIT
```

is used. It may occur anywhere inside a HOW'TO unit.

5. TESTS

An order is associated with all types, so <, <=, =, >=, > and <> (unequal) are defined for all values having the same type (but not for values of different types). Multiple comparisons are allowed:

```
WHILE min {x; y} < z < max {x; y}: PUT f z IN z
```

To discover if a list (or text or table) contains a certain element (or character or associate), the in test may be used:

```
SELECT:
    i in keys count: PUT count[i]+1 IN count[i]
    ELSE: PUT 1 IN count[i]
```

Tests may be combined with AND, OR and NOT:

```
IF i in keys t AND t[i] > 2: ...
IF NOT (c = 'a' OR c = 'b'): ...
```

Such combined tests are evaluated from left to right. In an AND combination, evaluation stops as soon as a failing test is encountered; in an OR combination it stops at the first succeeding test.

B also has versions of the mathematical quantifiers SOME, EACH and NO:

```
IF SOME d IN {2..n-1} HAS n mod d = 0: WRITE n, 'has factor', d
```

If the SOME test succeeds, it assigns to the target between SOME and IN the first value found to satisfy the test following HAS. Examples of NO and EACH are:

```
IF NO year, anniversary IN birthday HAS year = 1813:
    WRITE 'list is incomplete'
SELECT:
    EACH name IN keys birthday HAS name|1 = 'B':
        WRITE 'B-composers abound'
    ELSE: WRITE name
```

By combining the use of quantifiers with the keyword PARSING instead of with IN, powerful text recognition functions can be obtained:

```
IF SOME p, q, r PARSING text HAS q = ',': PUT p^' '^r IN text
```

The SOME test succeeds if text can be split into subtexts p, q and r (i.e., p^q^r = text) such that the test following HAS succeeds. This IF command substitutes a space for the first comma, if any, in text. The following command eliminates all bracketed parts of a text (assuming there are no nested brackets):

```
WHILE SOME a, l, b, r, c PARSING text HAS (l, r) = ('[', ']'):
    PUT a^c IN text
```

Just as a new function may be defined with YIELD, a new test may be defined with TEST:

```
TEST a subset b: REPORT EACH x IN a HAS x in b
```

In a TEST, REPORT is used instead of RETURN. To avoid REPORT 1=1 or REPORT 0=1, SUCCEED and FAIL are available. Thus, if zeroadic TESTs true and false are required, they could be defined by:

```
TEST true: SUCCEED
TEST false: FAIL
```

As in YIELDs, parameters are passed by value and the environment is not affected by changes made by a TEST.

6. REFINEMENTS

Hierarchical programming in *B* is aided by refinements. These are like light-weight procedures, but:

- refinements belong to a unit (i.e. HOW'TO, YIELD or TEST), and are written at the end of the unit;
- refinements have no parameters;
- refinements have no local names: the meaning of a name used in a refinement is determined by the context at the point where the refinement is called.

Refinements come in three kinds, analogous to HOW'TO, YIELD and TEST units. Some examples are:

```
HOW'TO SEPARATE red'vase FROM mixed'vase:
    FOR marble IN red'vase: IF NOT red: TO'MIXED
    FOR marble IN mixed'vase: IF red: TO'RED
red:
    PUT marble IN nr, color
    REPORT color = 'red'
TO'MIXED:
    REMOVE marble FROM red'vase
    INSERT marble IN mixed'vase
TO'RED:
    REMOVE marble FROM mixed'vase
    INSERT marble IN red'vase
```

Of course, refinements become more useful as programs become longer.

7. RANDOM

The command

```
DRAW r
```

assigns an arbitrary number to `r`, such that `0 <= r < 1`.

The command

```
CHOOSE cap FROM {'A'..'Z'}
```

assigns an arbitrary capital letter to `cap`. Likewise,

```
CHOOSE c FROM 'mississippi'
```

assigns an arbitrary letter from the text `'mississippi'` to `c`. In the same way, it is possible to CHOOSE an arbitrary associate FROM a table. The meaning of CHOOSE may be described in terms of DRAW in the following way:

```
HOW'TO CHOOSE item FROM collection:
    DRAW r
    PUT 1+floor(r*#collection) IN i
    PUT i th'of collection IN item
```

(This is another example of a unit capable of being used with parameters of different types — text, list or table —, as long as they are consistent with the operations inside.)

CHOOSE and DRAW are based on a pseudo-random sequence. To make experiments replicable, the sequence may be entered at a specified point by starting with a SET'RANDOM command, e.g. like this:

```
SET'RANDOM 'Today is my birthday.'
```

or with a call with any other expression of any type as a parameter.

8. INPUT/OUTPUT

Input from the user at the screen is read by a READ command, such as

```
READ size EG 0, 0
```

where the expression following EG specifies the type of the expression to be entered (here a compound with two numeric fields). This is the only spot in *B* where a declaration-like construction is needed to ensure full static type checking. The type of the expected expression may be given by e.g. `0` or `''`, or by any other expression. In the following example, the READ command demands the input to be of the same type as the elements of the list `legal'moves`:

```
READ move EG min legal'moves
```

READ prompts the user, who may then enter any expression of the desired type, using global variables and calls of YIELDs.

READ requires texts to be quoted, just as in *B* itself. For situations where this is a nuisance, a special version of the READ command is available:

```
READ line RAW
```

which interprets the input line as a text and assigns it to the variable `line`.

For output to the screen, a WRITE command is used. As shown above, WRITE can handle expressions of any type. Apart from the operations mentioned in 2.2, the following operations are useful for formatting purposes:

```
align left                (7*8)<<25 = '56                        '
align right               (7*8)>>25 = '                        56'
center                    (7*8)><25 = '            56             '
insert result of expression  '1K is `2**10` bytes' = '1K is 1024 bytes'
```

The dedicated *B* system now under development will contain facilities for screen manipulation and output to other devices. Some of these will certainly find their way into the language itself.

9. SAMPLE PROGRAM 1: MAKING AN INDEX

Let us assume we have the following problem. We have a book, represented as a table of lines, and we want to build an alphabetical index of the words occurring in the book, with references to the numbers of the lines where the words are to be found. We may or may not want capital letters included, etc., so let us say words have to be composed of characters from a given alphabet. We are not interested in words occurring in the book more than a given number of times. For this purpose we can write the following function:

```
YIELD index text:
    SHARE alphabet, max'occur
    PUT {}, {} IN ind, stoplist
    FOR nr IN keys text:
        PUT text[nr] IN line
        TREAT'LINE
    RETURN ind
TREAT'LINE:
    WHILE word'found:
        BOOK'WORD
        REMOVE'WORD
word'found: REPORT SOME head, word, rest PARSING line HAS word'isolated
word'isolated: REPORT head'garbage AND word'proper AND rest'trailer
head'garbage: REPORT NO c IN head HAS c in alphabet
word'proper: REPORT word < '' AND EACH c IN word HAS c in alphabet
rest'trailer: REPORT rest = '' OR rest|1 not'in alphabet
BOOK'WORD:
    IF word not'in stoplist:
        IF word not'in keys ind: PUT {} IN ind[word]
        SELECT:
            #ind[word] < max'occur: INSERT nr IN ind[word]
            ELSE: STOP'BOOKING
REMOVE'WORD: PUT rest IN line
STOP'BOOKING:
    DELETE ind[word]
    INSERT word IN stoplist
```

10. SAMPLE PROGRAM 2: A CRITICAL PATH PROBLEM

We have a project existing of jobs, each requiring a known time. The project may be represented by a directed acyclic graph, with jobs labeling its edges. A job *J* may only be started upon completion of its preceding jobs, i.e., those jobs ending in the node from which *J* starts. Otherwise, jobs can be performed in parallel. The problem is to find a schedule to finish the project in minimum time, and to determine for each job the earliest time and the latest time it may be started to achieve that minimum. If the slack between these two equals zero for a job, it is on the 'critical path'. If it is desirable to shorten the time for the whole project, the critical path tells us for which jobs we must first try to reduce the required times.

The following unit first sorts the jobs 'topologically', i.e., such that jobs preceding in the graph also precede in the sorting order. The early start times are then computed in one forward scan, and, dually, the late finish times backwards.

```
HOW'TO SCHEDULE:
    READ'NET
    TOP'SORT
    FORWARDS
    BACKWARDS
    OUTPUT
READ'NET:
    WRITE 'job, duration, from-node, to-node'
    READ jj, dd, ff, tt EG '', 0, 0, 0
    IF jj in {''; 'stop'}: QUIT
    INSERT jj, dd, ff, tt IN net
    READ'NET
TOP'SORT:
    PUT {}, net IN ts, n
    WHILE SOME item IN n HAS no'predecessor:
        PUT item IN ts[#ts+1]
        REMOVE item FROM n
    CHECK n = {} \otherwise cycle in net
no'predecessor:
    PUT item IN jj, dd, ff, tt
    REPORT NO j, d, f, t IN n HAS t = ff
FORWARDS:
    PUT {} IN es
    FOR j, d, f, t IN ts:
        IF f not'in keys es: PUT 0 IN es[f]
        IF t not'in keys es: PUT 0 IN es[t]
        PUT max {es[f]+d; es[t]} IN es[t]
BACKWARDS:
    PUT {}, max es IN lf, final
    WHILE ts > {}:
        PUT ts[max keys ts] IN jj, dd, ff, tt
        DELETE ts[max keys ts]
        IF tt not'in keys lf: PUT final IN lf[tt]
        IF ff not'in keys lf: PUT final IN lf[ff]
        PUT min {lf[tt]-d; lf[ff]} IN lf[ff]
OUTPUT:
    WRITE 'job       duration  early start  late start  slack' /
    FOR j, d, f, t IN net:
        WRITE j<<8, d>>6, es[f]>>10, (lf[t]-d)>>11, (lf[t]-d-es[f])>>9 /
```

ACKNOWLEDGEMENTS

I want to thank all those who commented on earlier versions of this paper, especially Jan Heering and Lambert Meertens.

36268