THE B NEWSLETTER

Mathematical Centre, Amsterdam

Issue 1, August 1983

CONTENTS

The purpose of this newsletter in general is to keep interested parties in touch with developments in the language and its implementation, and to provide a forum for discussions.

Clearly though, this first issue has an introductory role to play, and so you will find articles giving a taste of the language and its environment, giving details of the implementation, and telling you how you can find out more.

Future issues will contain more discussion articles, and you are encouraged to submit articles, ideas, or interesting B programs for inclusion.

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 the Unix network, then you can submit articles and send mail to *mcvax!leo.* Otherwise, articles and mail should be sent to

B Newsletter
Informatics Department
Mathematical Centre
POB 4079
1009 AB Amsterdam
The Netherlands

About This Issue

It is rare to find a programming language designed principally with ease of programming in mind, rather than ease for the implementor or squeezing the last drop of power from a computer. Yet in the very near future, when people will tend to have their own rather powerful personal computer, and won't have to share a computer, there will be more computing power than people will be able to use. Consequently there will be a need for programming languages that save time not for the computer, but for the programmer.

B is such a language and has been in development for some time, principally at the Mathematical Centre, Amsterdam. Now that some experience has been gained with the use of the language, and the portable implementation is approaching completion, is is time for us to report the state of play, and to try and convey to you our enthusiasm for the language.

THE B NEWSLETTER

Available Publications about B

A number of publications about B are currently available:

An Overview of the B Programming Language, or B without Tears,
Leo Geurts, 11 pages.

> This is the first place to go if you want to know more about B. It was published in SIGPLAN Notices Volume 17, number 12, December 1982, or is available from the Mathematical Centre, report IW 208/82, price Dfl. 3.30.

Draft Proposal for the B Programming Language,
Lambert Meertens, 88 pages.

> This book is a specification of the whole language, though rather technical for the casual reader. It also contains some thoughts on a B system. Published by the Mathematical Centre, ISBN 90 6196 238 2, price Dfl. 12.10. A part of it, the *Quick Reference* also appeared in the *Algol Bulletin* number 48, August 1982.

Making B Trees Work for B,
Timo Krijnen and Lambert Meertens, 13 pages.

> This describes a method of implementing the values of B. It is rather technical. Published by the Mathematical Centre, report IW 219/83, price Dfl. 3.30.

Incremental Polymorphic Type-Checking in B
Lambert Meertens, 11 pages.

> B allows you to use variables without having to declare them, and yet gives you all the safety that declarations would supply. This paper describes how this is achieved, but is *very* technical. Definitely not for the faint-hearted. Published in the conference record of the 10th ACM Principles of Programming Languages, pages 265-275, 1983, and also by the Mathematical Centre, report IW 214/82, price Dfl. 3.30.

On the Implementation of an Editor for the B Programming Language,
Frank van Harmelen, 18 pages.

> Gives details of a pilot implementation of the B dedicated editor. Published by the Mathematical Centre, report IW 220/83, price Dfl. 3.30.

Further publications are of course in preparation, and their availability will be announced in later issues of the newsletter.

Mathematical Centre publications can be ordered from

> Publications Department
> Mathematical Centre
> 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. 6 to cover bank charges.

A Short Introduction to the B Language

Leo Geurts

EXAMPLES

●**Prime Numbers.** The following program determines the prime numbers up to `n` by the classical sieve method, which first discards the multiples of 2 from the set of whole numbers, then the multiples of 3, the multiples of 5, and so on.

```
HOW'TO SIEVE'TO n:                          \name is SIEVE'TO
    PUT {2..n} IN numbers                   \set to be sieved
    WHILE numbers>{}:                       \repeat indented part
        PUT min numbers IN p                \smallest member of set
        REMOVE'MULTIPLES                    \refinement, see below
        WRITE p
REMOVE'MULTIPLES:
    PUT p IN multiple
    WHILE multiple<=n:
        IF multiple in numbers:             \present in set?
            REMOVE multiple FROM numbers
        PUT multiple+p IN multiple
```

●**Strings.** Strip leading blanks from a line.

```
WHILE line>'' AND line|1 = ' ':             \non-empty, first is blank
    PUT line@2 IN line                      \assign rest
```

●**Polynomials.** A polynomial can be represented by a table of its coefficients. For example,

```
{[n]: 1; [1]: -1; [0]: -1}
```

represents $x^n - x - 1$. The following function evaluates such a polynomial at a given point:

```
YIELD poly at x:                            \name is 'at'
    PUT 0 IN s
    FOR i IN keys poly:                     \keys gives list of indexes
        PUT s+poly[i]*x**i IN s             \** gives power
    RETURN s
```

●**Tables.** Here is a piece of program which, given a table `capital` storing names of capitals under names of countries, produces an inverse table `country`, which has capitals as its keys and stores countries. So, if initially `capital['Nederland']='Amsterdam'`, then after running the program we will have `country['Amsterdam']='Nederland'`.

```
PUT {} IN country                           \empty table (or list)
FOR c IN keys capital:                      \set of entries of table
    PUT c IN country[capital[c]]
```

•**Lists.** Given a table giving words and the frequency that those words appeared in a text, print the words in order of increasing frequency. The method works by inserting the frequency and the word, as a pair, into a list, and printing the list.

```
PUT {} IN list
FOR word IN keys freq:
      INSERT freq[word], word IN list
FOR pair IN list:                          \lists remain sorted
    WRITE pair /                           \/ gives new line
```

•**Garbage Collection.** A given table represents a group of people, and who can contact whom within that group. For instance, if `contacts['Mark'] = {'Kevin'; 'Bessy'}` then Mark can contact Kevin and Bessy (but not necessarily vice versa). The following function returns the sub-table of all people contactable from a given root person.

```
YIELD root reachable graph:                \parameters: root and graph
    PUT {root} IN accessible               \list of one element
    PUT accessible IN still'to'do
    WHILE still'to'do > {} :
        SELECT'NODE
        TREAT'NODE
    FOR n IN keys graph:
        IF n not'in accessible:
            DELETE graph[n]
    RETURN graph
SELECT'NODE:                               \refinement
    PUT min still'to'do IN node            \min is smallest element
    REMOVE node FROM still'to'do
    CHECK node in keys graph               \just to be sure
TREAT'NODE:
    FOR n IN graph[node]:
        IF n not'in accessible:
            INSERT n IN accessible
            INSERT n IN still'to'do
```

•**Palindromes.** Here is a procedure to see if a text is a palindrome, i.e. if it reads the same backwards, except for non-letters and capitals. It is not the shortest or quickest way, but it is a good demonstration of the use of refinements.

```
TEST palindromic sent:
    REDUCE                                 \to lower case letters
    REPORT sent = backwards
REDUCE:
    PUT '' IN s
    FOR c IN sent: PUT s^repr IN s         \^ joins texts
    PUT s IN sent
backwards:
    PUT '' IN b
    FOR c IN sent: PUT c^b IN b
    RETURN b
repr:
    SELECT:
        lower'case: RETURN c
        upper'case:
            RETURN rank th'of {'a'..'z'}   \lower case version
        ELSE: RETURN ''
lower'case: REPORT c in {'a'..'z'}
upper'case: REPORT c in {'A'..'Z'}
rank: RETURN #{'A'..c}                     \# gives number of elements
```

•**Prime Numbers Revisited.** The following program shows another way of calculating the prime numbers between 100 and 200:

```
FOR n IN {100..200}:
    IF NO div IN {2..n-1} HAS n mod div = 0:     \mod=remainder of division
        WRITE n
```

OVERVIEW

Most of the features of the language can be seen from the examples given, but here is a quick overview.

Types and Operations

There are 2 basic data types in *B* :

•numbers, with the usual sort of operations, and which are exact as long as only `+`, `-`, `*` and `/` are used;

•texts, with operators for concatenation and selection of subtexts: `PUT "John" IN name;`

and 3 ways of building data structures from smaller ones:

•compounds, which are records without field names, with no special operations (useful for multiple assignment, tables with multiple indexes, etc.): `PUT 0, 1 IN z;`

•lists, which are ordered collections of elements of any one type, with operations to insert and to remove an element: `PUT {1; 1; 2; 3; 5; 8} IN fib;`

•tables, which are arrays with keys of any one type and which stores values of any other one type: `PUT {["John"]: 4141; ["Mary"]: 3896} IN tel.`

There are no bounds on the length of any of these types. So, e.g., you will never see an error message 'number too big' or 'a list may only contain 256 items'.

Other Commands

For texts, lists and tables there are operators for scanning them (`FOR i IN fib: WRITE i /`), for seeing if a certain element is present (`IF 3 in fib: WRITE "yes"`), for getting the number of elements (`WRITE #fib`), etc. Furthermore, there are some commands that work for any type: `PUT .. IN ..` for assignment, `WRITE` and `READ`.

Units

There are three kinds of units (procedures) in *B* : `HOW'TO`, `YIELD` (for functions) and `TEST` (for conditions). Similarly, there are three kinds of refinements, which are like light-weight procedures, to support stepwise refinement.
Control commands include `WHILE`, `IF` (for 1 branch, no `ELSE`), `SELECT` (for more branches, including `ELSE`) and `QUIT` (for early termination of a `HOW'TO` or a refinement).
There are no declarations in *B* , but type checking is done to detect inconsistencies. So, if a unit has a line `PUT 'yes' IN answer`, then an error is signalled if you try to add `WRITE answer+1`. On the other hand, if you have

```
HOW'TO LIST series:                    \print items on separate lines
    FOR el IN series:
        WRITE el /
```

you may use a `LIST` command for a list of elements of any type, and also for a text or for a table of any type, since `FOR` and `WRITE` work for these too.

A Glimpse at the B Environment

Steven Pemberton

What is this water that the other fish tell me about, Mummy?

A Zen Story

Introduction

B as a programming language is small and easy to learn, yet powerful, and offers the facilities required for modern programming methods to structure both your program and your data. However, it is not enough to have just a language: when starting to use a computer system, much has to be learned — command language, editor, compiler or interpreter, details of the file system, etc — all with their own syntax, error messages and idiosyncrasies, and apparently to the new user, usually with no unifying design. This is the environment a programmer usually works in.

Part of the design of *B* is a *unified* environment, so that the user needs to learn as few new concepts and notations as possible.

The Command Language

The first example of the simplicity of B is that no special command language is necessary. A programmer will usually have a set of *workspaces* which are collections of *B* units, each workspace usually representing the units (or procedures if you like) for one program. The programmer may move amongst these workspaces at will. If the programmer has created a program that is a unit, called say `ADVENTURE`, then it may be started in the same way that you would call it from another unit, that is just by typing it as a command. Furthermore, there is no compilation phase: the program may be typed in, or edited, and run immediately. Similarly, all of the commands and other features of *B* are available to the user at the command level, giving the user the facilities of a calculator:

```
WRITE root 2
1.41421356237
```

Thus the user has no separate concepts of program and subroutine, nor of subroutine call, command, and program, but just of unit, and command.

Files

Similarly, at this outer level of 'immediate' commands as they are called, all variables created are permanent, in the sense that they remain not only while the programmer is working at the computer, but even after switching off, and returning later. Thus variables may be used instead of 'files' in the traditional sense, and so the user has no separate concepts of files and variables. This also means that there is no need for a separate mechanism for passing arguments to a program, since you just use the existing parameter passing mechanism for units. Since *B* variables are dynamic, and unrestricted in size, using them in place of files causes no difficulties. Quite the reverse in fact, since you now have the powerful data-types of *B* at your disposal, allowing random, and indeed associative, access to the contents. In fact traditional filestores are rather inconvenient, since they represent a machine detail, that there is a two-level store involved. To output to a file, you have to design an output format, unpack your data-structure into the file, and when you read the file back you have to parse the input, and re-create the data-structure. Our kind of user doesn't want with such details.

Reading Input

Another nice aspect of the use of variables for files is that when a program reaches a `READ` command and prompts the user for input, the system at this point is expecting the user to type *any* *B* expression of suitable type, so that for numeric input for example, `4` or `3+5` or `root 2` are all acceptable, as is using permanent variables like `a` or `a+b`. If you type in a value that is in any way inappropriate or incorrect, the system tells you, and re-prompts.

The Editor

One of the most important parts of the *B* environment is the intelligent editor. In fact the user never leaves the editor: whenever you are using the *B* system, you are doing so *via* the edi-

tor. This means that *whenever* you are typing, you have all the facilities of the editor at your disposal. The editor knows all about *B* of course, which helps a lot when composing programs. For a start, it can save you a lot of typing: when you are typing in a command, and you type a "p" as the first letter of the command, (upper or lower case), it guesses that you want a PUT command, and so displays on your screen

```
PUT □ IN □
```

(the underline shows where you currently are). If you did indeed want a PUT command, then you need only press the 'accept' key, and the cursor moves to the first of the two 'holes', and you can type in an expression and press accept again to move to the second hole. Similarly, the editor supplies matching brackets, so typing

```
P☞(
```

(where ☞ represents pressing the accept key) gives

```
PUT (■) IN □
```

(the filled in hole indicates that you are positioned at that hole). You get similar treatment with string quotes.

The editor also knows about your own units, so that if you didn't want a PUT command, but instead wanted to invoke a unit of your own, called say PRINT, then typing an "r" after typing the "p" gives you the following on the screen:

```
PRINT □
```

The hole shows you that it requires one parameter.

If you didn't want PRINT either, but you are typing in a new unit that uses another unit that you haven't yet written, say called PROCESS, then typing "o" after the "pr" will give you

```
PRO_
```

and you can carry on typing the rest of the characters. In fact you can always ignore all this guessing if you want: if you type all the characters of each command, without using the 'accept' facility, you will still get the right result.

Another feature of the editor's knowledge of *B* is that it knows where there must be indentation, and so supplies it for you: if you type in the first line of a FOR command, followed by an accept, it automatically positions the cursor at the right position, for example,

```
FOR i IN {1..10}:
    ■
```

which could have been typed as

```
F☞i☞{1..10☞
```

Another difference from usual editors is that the cursor, called the *focus* in the *B* system, can focus on large parts of text, such as a whole command. The focus is displayed by using some aspect of the terminal such as underlining, reverse video, or a different colour. As an example, consider the following commands, where the focus is on the first of them:

```
PUT max {x; y} IN a
WHILE a>b:
    PUT a-b IN a
WRITE a
```

If now the 'next' key is pressed, the focus moves to the next command:

```
PUT max {x; y} IN a
WHILE a>b:
        PUT a-b IN a
WRITE a
```

If now the 'narrow' key is pressed, the focus moves to the heading of the while command:

```
PUT max {x; y} IN a
WHILE a>b:
    PUT a-b IN a
WRITE a
```

Pressing 'narrow' focusses on WHILE, and then 'next' focusses on a>b. Inserting an open bracket here, supplies the matching close bracket:

```
PUT max {x; y} IN a
WHILE (a>b):
    PUT a-b IN a
WRITE a
```

But perhaps the most important aspect of the editor's knowledge is that it tells you about errors you make *as you make them.* This includes not only syntax errors, such as mis-matched brackets, but also rather more sophisticated errors, like 'type' errors. For instance, if you use a variable for one type of value, say a text, and later try to add 1 to it, the editor will complain as you type it in. Similarly the editor warns you if you try to use a variable that cannot yet have been given a value at that point in the program. Finally, you may also edit the contents of permanent variables.

The Screen

The system is organised in *documents,* such as a unit, or the output from a program, which the user may look at through *windows.* This is a technique of making many *logical* screens available to the user, using only one *physical* screen. Each window has a title line that describes its contents. You can look at a different document by editing this title line to the name of the document you want to see. Many windows may be visible on the screen at one time, and the user may delete these or move them at will. You move around the screen, to fix your attention on a particular window, by using a pointing device, such as a 'mouse', or a touch screen, a joystick, or direction keys, which ever is available on the terminal.

If there are too many documents to fit on the screen at one time, then some will not be visible. The user can always get a list of all documents currently around (though not necessarily visible). Of course, this list is itself placed in a window, which is made visible.

When a session is started, one document is created, the *command* document, and in this document a prompt is displayed. The user may then type commands into this document, which are then executed. Remember, though, that the user is typing in commands using the editor, and so it would be more correct to say that the user then *edits* the command document, and these commands get executed. This means that if a command gets incorrectly entered, it may be deleted or changed at will, and re-executed, so that the final effect is as if the erroneous command was never entered.

As an example, suppose the user had typed in the following commands:

```
PUT 2 IN a
PUT root a IN b
WRITE b
1.41421356237
```

and then goes back to the first command, and changes the `2` into a `10`, *and nothing else,* then the window displays:

```
PUT 10 IN a
PUT root a IN b
WRITE b
3.16227766017
```

In fact this ability to edit windows in order to change the state of the machine is a general facility. You may look at the contents of a permanent target through a window, and edit it to change the contents of that target (this is especially necessary when such a target represents a large file); when you display the list of available documents, deleting a line from that list deletes the corresponding document; you may display a list of units available in the current workspace (in a window of course), and delete an entry in that list to delete the corresponding unit; and so on. This mechanism can be likened to a thermostat, where you 'edit' the temperature reading to get the required heat. (It has been suggested that you should be able to edit the output from a program, and the system change the program for you, or possibly supply the corresponding input. This has not been implemented.)

The kinds of lists available are of workspaces, processes, units within a workspace, and permanent targets, as well as things like electronic mail, and network information, if such items exist.

Processes

The list of processes needs an explanation. Whenever the command document is being edited, and a program that writes output, or reads input, or both, is started, a document for its output is created, if necessary, and for its input similarly, and execution of this program continues apparently *in parallel* with other programs running. Thus many programs may be running simultaneously and you may switch between documents to supply input, or peruse output, at will. This explains the list for processes: it allows you to keep track of what is going on in the machine, which programs are still running, which finished, and which are waiting for input. Of course editing this list allows you to stop and restart processes if you need.

State of Play

Several of the features described here have already been tried in the pilot implementation now running at the Mathematical Centre, and a new implementation is currently in progress. Of course what has been presented is in some respects an ideal, and parts of the environment depend on hardware that may not be available at every site. In such cases a subset will have to be made available, but in the same spirit as the full environment.

Furthermore, this environment is only a first iteration, and clearly as experience with its implementation and use accumulates it will be revised.

Implementation Plans for *B*

Steven Pemberton

Introduction

The programming language *B* has been designed specifically for non-professional users, such as business-people wanting to use a small computer, hobbyists, and school children and students learning computing. Clearly, for the language to be available for this group of people it must be implemented on computers within their grasp. In particular, this means relatively cheap computers, particularly for individuals, but even for schools, who typically have limited funds for such acquisitions. However, it should be pointed out that *B* has never been intended for implementing on tiny computers like 8K 8 bit micro-computers. This would be 'designing for the past'.

The Current Implementation

There is a pilot implementation of *B* in operation at the Mathematical Centre, running on a VAX computer, and a PDP 11/45. This implementation was written in an extremely short time (one person in 6 weeks), and was written as a test of the language, its implementability and its usability.

Naturally enough, this implementation is not a production quality system — that is what pilot studies are about — and so it is large, tends to be slow (though not unusably so), and is not easily transportable to different computers.

Its slowness is principally due to two causes. Firstly, it is an interpreter, but an extremely literal one. Each time a command is executed, its source line is first re-parsed, character by character. Secondly, the values of *B,* which are for the large part dynamic, are implemented in a very quick and easy way, which makes for easy coding, but bulky and time consuming store usage.

On the other hand, the implementation is for the whole language. There are some restrictions, such as a limit on the size of numbers (on the PDP); the checks that are performed are mostly done when the program is run, and not before as they should be; and only a small part of the environment is implemented. But it is essentially complete. There is a pilot version of a dedicated editor, workspaces for different programs, and the so-called permanent environment, where the global variables of a program are kept even if you log out.

The Next Version

A new implementation of *B* is currently under construction which is planned to be released by late 1983.

Principally, the new version will be more portable. It is coded, like its predecessor, in the language *C,* which is available on a large number of machines, and it is coded to the standard of the language's defining document. Therefore any machine that is large enough, and has a *C* compiler, should be able to run the *B* system.

Secondly, the new version will be faster: the source text of programs will be pre-parsed, before execution, so that execution times will be reduced, and the data types of *B* will be implemented in a more efficient manner.

Thirdly, the new version will be more functional. Many of the checks will be done statically instead of at run-time (which incidentally should also give a time advantage), and more of the environment will be implemented, including a fuller version of the intelligent editor.

However, it is not clear at this stage how much smaller the new version will be. It is clear that there are many places where the pilot implementation's use of store can be improved, but on the other hand, the greater functionality of the new system will use some of this saving, and it is not yet clear how much. It seems though, that this new version will still require a large amount of store, probably of the order of 100K bytes, which will rule out many of the current smaller home computers. However, most of the new generation of micro-computers come in this size, for example the IBM micro, which currently costs between $3000 and $4000, and, as it is a new product, can be expected to get cheaper.

The Future

The version after the next one will mainly involve greater functionality. Some of the system will probably be re-coded in *B*, which will probably cause a reduction in size. It might be worthwhile re-coding the system in Pascal to make it available on more machines, and possibly take advantage of the wide-spread P-code systems, which also may make the system *runnable* on smaller machines, although at the expense of speed, since the P-code system allows procedures to be swapped in and out of main store in order to conserve space. However, it is unlikely that this system will be available before mid to late 1984, and by this time it may not be worth trying to squeeze the system onto very small machines since memory costs will be such a tiny proportion of any machine, and such small machines may by then be rapidly dying out.

Finally, there is a separate project starting in collaboration with Brighton Polytechnic, England with the aim of producing a *B* dedicated desk-top computer, but it is too early yet to make any predictions about the availability or cost of the final product.

Availability

The sources of the pilot implementation and some extra documentation are available on application to the Mathematical Centre for the cost of the media. The medium is magnetic tape at any of the usual densities. The preferred format is 'tar', but ASCII standard labelled tapes are also possible. It should be stressed that the pilot implementation runs on Unix version 7, and with a little effort on VAX/VMS with Eunice. Other machines and operating systems cannot be guaranteed. The address to write to is

B Group
Mathematical Centre
Computer Science Department
POB 4079
1009 AB Amsterdam
The Netherlands.

If you are interested in being informed when the portable implementation is available you should also write to this address.

Conclusion

In summary, there is currently an implementation of *B* available for VAX computers and PDP's with separate program and data spaces. The next implementation which will be ready by late 1983 will be transportable to different machines, as long as they have a *C* compiler, and are sufficiently large. Later versions will be more functional, but there is little chance of major reductions of size, such as below 64K bytes.