Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

L.G.L.T. Meertens, S. Pemberton

An implementation of the B programming language

Department of Computer Science Note CS-N8406 June

AN IMPLEMENTATION OF THE B PROGRAMMING LANGUAGE

L.G.L.T. MEERTENS, S. PEMBERTON

Centre for Mathematics and Computer Science, Amsterdam

B is a new programming language designed for personal computing. We describe some of the decisions taken in implementing the language, and the problems involved.

Note: *B* is a working title until the language is finally frozen. Then it will acquire its definitive name. The language is entirely unrelated to the predecessor of C.

A version of this paper will appear in the proceedings of the Washington USENIX Conference (January 1984).

1982 CR CATEGORIES: 69D44.

KEY WORDS & PHRASES: programming language implementation, programming environments, *B*.

Note CS-N8406

Centre for Mathematics and Computer Science

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The programming language B

B is a programming language being designed and implemented at the CWI. It was originally started in 1975 in an attempt to design a language for beginners as a suitable replacement for BASIC. While the emphasis of the project has in the intervening years shifted from "beginners" to "personal computing", the main design objectives have remained the same:

- simplicity;
- suitability for conversational use;
- availability of tools for structured programming.

The design of the language has proceeded iteratively, and the language as it now stands is the third iteration of this process. The first two iterations were the work of Lambert Meertens and Leo Geurts of the Mathematical Centre in 1975-6 and 1977-9, and could be described as both easy to learn and easy to implement. However, there are two sides to ease and simplicity. If something is easy to learn and define, it does not necessarily imply that it is also easy to use. BASIC is testimony to this: it is fine for tiny programs, but for serious work, it is like trying to cut your lawn with a pair of scissors.

The third iteration of *B*, designed in 1979-81 with the addition of Robert Dewar of New York University, adopts a new characteristic: it is still easy to learn, by having few constructs, but is now also easy to use, by having powerful constructs, without the sorts of restrictions that professional programmers are trained to put up with, but that a newcomer finds irritating, unreasonable or silly. Thus compared to most existing languages that supply you with a set of *primitive* tools, with which you can then build your more powerful tools, *B* does it the other way round by supplying you with *high-level* tools, which you may also use for primitive purposes if you wish.

One consequence of this approach is of course that the language is no longer so straightforward to implement. Another is that, although the language was designed with non-professionals in mind, it turns out to be of interest to professionals too: several people in our institute now use it in preference to other languages.

The sort of computers that we are aiming at are the very powerful personal computers just now appearing at the top end of the market. With such power at your disposal, you want a language that minimises *your* effort, not the computer's. It is not, and has never been, our intention to implement *B* on 8-bit micros. This would have been "designing for the past".

A taste of B

It is not the purpose of this paper to give a complete description of *B*, but the main features need to be described to explain the issues involved in the implementation. For further details of the language see reference [Geurts].

B is strongly typed, but variables do not have to be declared. There are two basic data types, numbers and texts, and three constructed types, compounds, lists, and tables. All types are unbounded. Thus numbers may have any magnitude, texts, lists and tables any length, within, of course, the confines of memory. There are no 'invisible' types like pointers; thus all values may be written.

Numbers are kept exact as long as possible. Thus as long as you use operations that yield exact results, there is no loss of accuracy. This includes division, so that `(1/3)*3` is *exactly* `1`. Clearly however, operations such as square root cannot in general return an exact answer, and in such cases approximate numbers are used, rounded to some length.

Texts are strings of characters. There are operations to join texts, trim them, select individual

characters (themselves texts of length one), and so forth.

Compounds are the equivalent of records, for instance in Pascal, but without field names.

Lists are ordered lists of elements. The elements must be all of one type, but otherwise may be of any type. Thus you may have lists of numbers or texts, but also of compounds, of other lists, and so on. Lists may contain duplicate entries (thus they are bags or multisets, rather than sets). There are operations to insert an element in a list, remove one, determine if an element is present, and so on.

Tables are generalised arrays, mapping elements of any one type to elements of any other one type. Again there are no restrictions on the types involved. You may insert entries in a table, modify or delete them, enquire if an entry is present, and so on. There is an operator, `keys`, that when applied to a table delivers the indexes of the table as a list.

Typical commands of *B* are assignment

```
PUT count+1 IN count

PUT a,b IN b,a

PUT {1..10} IN list
```

input/output

```
WRITE "list=", list

READ table EG {["John"]: 21}
```

data-structure modification

```
INSERT 10 IN list

DELETE table["Peter"]
```

control commands

```
IF value in list:
    WRITE value
    REMOVE value FROM list

WHILE answer not in {"yes"; "no"}:
    WRITE "Please answer with yes or no"/
    READ answer RAW

FOR value IN list:
    WRITE 2*value
```

And there are also means to define your own commands and functions.

An example

Here are the two major routines for a cross-reference generator. The first is used to save words and the list of line numbers that each word occurs at. The second is used to print out the resulting table. Note that in *B* indentation is used to indicate nesting, rather than using explicit BEGIN-END brackets.

```
HOW'TO SAVE word AT line'no IN xref:
    IF word not'in keys xref:          \if not yet in the table
        PUT {} IN xref[word]           \start entry with empty list
    INSERT line'no IN xref[word]       \insert in the list

HOW'TO OUTPUT xref:
    FOR word IN keys xref:             \treat each word in turn
        WRITE word<<20, ":"            \output it left-justified
        FOR no IN xref[word]:          \for each number in the list
            WRITE no>>4                \output right-justified
        WRITE /                        \output a newline
```

Implementations

The original *B* implementation was written in 1981. It was explicitly designed as a pilot system, to explore the language rather than produce a production system, and so the priority was on speed of programming rather than speed of execution. As a result, it was produced by one person in a mere 2 months, and while it was slower than is desirable, it was still usable, and several people used it in preference to other languages. It was written in C, but there was no attempt to make it operating system independent.

The second version, just completed, is aimed at wider use, and therefore speed and portability have become an issue, though the system has also become more functional in the rewrite. It is also written in C and was produced by first modularising the pilot system, and then systematically replacing modules, so that at all times we had a running *B* system. It was produced in a year by a group of four.

An important decision taken very early in the rewrite was to write the code as quickly and as simply as possible, without striving for code-speed before we had any information about where to optimise. We knew that such optimisations would only be needed in a very few places, and could easily be added later, by taking profiles of the system. In the event this decision paid off handsomely: first runs of the system showed that one single routine was unexpectedly consuming 90% of the run time. This was quickly rewritten optimally, giving a great increase in speed. We deemed it worthwhile to optimise a couple of other routines which from the profiles were clearly usually being called for very particular purposes, but this was more in the field of fine tuning.

A problem that we still have is with code size: a lesson that we are only now learning is that macros in C are extremely expensive. Certain very inoffensive looking pieces of code have been found to produce great welters of code, which on investigation have been due to the use of macros. Replacing the macros with routines usually reduces the code size significantly.

Modules

A *B* implementation can be broadly divided into four parts: Values, Parsing, Interpretation, and Environment. *B* has high-level data-types, and so some effort has to go into the values module, but once it is written the full semantic power of *B* types is available to all modules, which is no small advantage.

Values

All values in *B*, with the exception of compounds, are dynamic. In the pilot implementation values were implemented as pointers to contiguous stretches of store. This made for easy programming but slow speeds if the size of a value got large, with times $\Omega(n^2)$.

Copying of values was implemented using the scheme of Hibbard, Knueven and Leverett [Hibbard] as a basis, where each value has a reference count. Copying then consists merely of copying a pointer and updating reference counts. When a count reaches zero, the associated space can be returned to the free list. On the other hand, if a value is to be modified, such as by inserting a value in a list, if its reference count is greater than one then one level of the value must first be 'uniquified' by copying it to a fresh area of store. (Only one level need be copied because for instance if the list is a list of tables, the tables need only have their reference counts updated, since they are not changed themselves.) Already unique values may be modified *in situ.*

This scheme has one outstanding feature, that the cost of copying is independent of the size of the value. Therefore there is a size of value above which reference counting becomes cheaper than ordinary copying. This critical size is rather small, and since *B* values easily become large, it is advantageous. Furthermore, assignments are typically the most executed sort of statement in programs, and so choosing a method that favours copying is to one's advantage.

Since all values were implemented as contiguous areas of store, inserting or deleting parts of a value involved shuffling the rest of the value up or down to make room for the new elements or to take the place of the old. While this was only on one level of the value, its cost was still proportional to the length of the value.

The new implementation still uses the reference count scheme, but instead of contiguous areas of store now uses B trees (no relation) [Krijnen] to store the values. These are a form of balanced trees, and the cost of modifying an element is only O(log n). Instead of having to copy a whole level of the value on modification, now only a sub-section of the tree needs to be copied.

Additionally two essential optimisations for *B* were added: representing lists such as `{1..10}` only by their upper and lower bound; this is essential for cases like

```
FOR i IN {1..1000000}:
    PROCESS i
```

and representing the result of the `keys` operator in a special way to prevent copies being taken (essentially the reference count of the table is incremented, and the result marked as a keys value); this optimisation is essential for cases like:

```
IF k IN keys t:
    WRITE t[k]
```

The final change to the values module in the new implementation was in the numeric package. *B* has unbounded exact rational arithmetic, but for simplicity of implementation the pilot implementation used only pairs of real numbers to represent rational numbers, using the in-built floating point facilities of the machine. This was replaced in the rewrite by a proper unbounded-arithmetic package.

Use of values by the rest of the system.

As mentioned before, it is no small advantage to have the facilities of *B* values at your disposal in the rest of the system. It means for example that identifiers can be implemented using texts, with no problems about limiting the length of identifiers, and more interesting, that 'environments', which are the mapping of the identifiers of variables onto their contents, can be implemented as

tables mapping texts onto other *B* values. Furthermore, in certain places the semantics of *B* demand that such environments be copied (for instance to prevent side-effects when evaluating an expression); this is consequently a very cheap operation.

A notable feature of *B* is the so-called permanent environment. All global variables in a session survive logout, so that when you come back to your program later, all the variables have the same values as before. This obviates the need for files in *B*. The implementation of this was very simple. Since environments are just *B* tables, the permanent environment can just be written in the normal *B* way, using the equivalent of the `WRITE` command, but to a standard file. On re-starting, this can be read by the equivalent of the `READ` command. This has remained essentially the same in both versions, though there is now a plan to load permanent targets only when they are needed, to reduce start-up times for large permanent environments.

Parsing

In the pilot version the user's 'units' (procedures and functions) were stored on separate files in a directory. On running the *B* system all units in the current directory were loaded into a big buffer in main store, and represented quite literally as a stream of characters, with a special character to mark the end of each unit (this could cause problems for directories with many units, as start-up time then became rather long). Parsing then proceeded in a top-down fashion by identifying for the construct being parsed its 'skeleton' (such as `PUT ... IN ...` for an assignment; `{...}` for a list or table display) and then passing on the sub-strings for the inner constructs to be parsed. No attempt was made to produce a parse-tree or other internal representation; only the literal text form was used.

The new implementation uses the same scheme of storing units on individual files, but now only loads them on demand, i.e. when they get used the first time. Furthermore, although the same parsing scheme is used, an internal parse-tree *is* formed, and therefore only one text line at a time need be present in main store. The parse-tree has been designed to suit all purposes in the system (such as interpretation, and reconstructing the source from its internal representation) so that there need be only one canonical representation throughout the system. The representation is a fairly traditional abstract syntax tree; so, for example, a for-command representation has a node-type indicating it is a for command, and then has three sub-trees for the identifier, the expression, and the body of the for. Naturally, a parse-tree is represented using *B* values, by using nested compounds.

A feature of *B* relevant here is that expressions do not have a context-free grammar. In order to allow expressions like `sin x`, it is not always possible to decide if an expression like `f - 1` is a call of a function `f(-1)`, or the subtraction of `1` from a variable `f`. In the pilot implementation this could be ignored since expressions were parsed from context at the time of execution. We know of no case where this produced other than the effect intended, but it was possible in principal at least to construct an expression, for instance in a loop, which on the first iteration was parsed differently to the later iterations.

The new implementation takes the possibility of ambiguity into account, and the parser produces a special kind of node for expressions it cannot resolve. Just before executing a unit that contains such nodes, the ambiguity is resolved and the nodes replaced, or an error message is produced (which only happens when a unit uses a function that hasn't yet been defined).

Interpretation

In the pilot version parsing and interpretation were not separated. Thus there was a boolean variable that indicated whether the current command should be executed, and if so execution proceeded at the same time as parsing. This meant that during the execution of a `WHILE` for instance,

the body would be parsed and reparsed each time round the loop.

The new version uses a recursive-descent interpreter that traverses the parse-tree. There is a main routine that is called with any particular node, which then splits the node into its sub-fields and uses the node-type to index a table of routines and call the relevant routine with the right number of parameters. This routine can then execute the node, calling the main routine where necessary to execute the sub-nodes.

The B Environment

B is an interactive language, and consequently allows you to modify your units during a session. The pilot version did this by writing the unit out to its file, and then calling an editor (of your choice) as a subprocess to edit the unit. On return the unit was re-read, and re-parsed immediately (without execution) so that the user could have immediate feed-back about errors. It was then straightforward to re-enter the editor to fix the errors.

The new implementation now uses a dedicated *B* editor, which furthermore you use all the time and not just for editing units. The editor knows much about the syntax of *B*, and thus checks the syntax while you type, actually making impossible the standard sorts of mistyping such as unmatched brackets, and missing quotes.

Additionally it helps in reminding you with commands: 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 'tab' key, and the cursor moves to the first of the two 'holes', and you can type in an expression and press tab again to move to the second hole. Similarly, the editor supplies matching brackets, so typing P [tab] (gives

```
PUT (?) IN ?
```

You get similar treatment with text quotes.

If you didn't want a PUT command, but instead wanted to invoke an existing unit of your own, called say PRINT, with one parameter, then typing an "r" after typing the "p" replaces the suggestion and gives you the following on the screen:

```
PRINT ?
```

and you can use tab in the same way. Actually, you can ignore this guessing if you want: if you type all the characters of each command, without using the 'tab' 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 a newline, it automatically positions the cursor at the right position, for example,

```
FOR i IN {1..10}:
    ?
```

which could have been typed as F [tab] i [tab] {1..10 [newline].

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. The major advantage of

such a focus is that it makes the editor command set very small. You no longer need separate keys for moving over characters, words, lines, paragraphs etc., and separate keys for deleting each category, but only keys for adjusting the size and position of the focus, and one key for deleting. It additionally alleviates such traditional problems with structured editors of changing an `IF` into a `WHILE`.

Apart from the addition of this editor to the new system, it was also made possible to edit the contents of permanent variables as well as units. Since such variables replace the traditional use of files in *B*, and are typically large, this facility is very welcome.

Availability of the Implementation

The Mark 1 implementation runs under Unix. It currently runs on VAX 11/780, VAX 11/750, PERQ, Philips PMDS, Bleasdale, and other 68000 systems running Unix. There is a project underway to put it on an IBM PC.

The operating system interface is localised in three files (interface with signals and interrupts, interface with the file-store, and machine parameters such as word-length) and thus transporting the system to another sufficiently large machine should cause few problems.

There is a version that runs on PDP 11/45's and similar, but it has some restrictions (such as no unbounded arithmetic) and is slower.

The system is available at nominal cost in 'tar' format (preferably) or ANSI standard labelled tape format.

The future

The *B* group is now engaged on the next version. This will feature further efficiency improvements, such as speeding-up sequential access to a data-structure, by far the most common case, but will mainly involve functional improvements. First of all a full *B*-dedicated environment will be implemented, rather the current embryonic one. For instance, the editor will know about the semantics as well as the syntax of *B*, and thus many semantic checks will be performed before your unit is run. Additionally, there will be extensions such as graphics added to the system.

Conclusions

We are rapidly approaching the time when all personal computers will have the power of a VAX or greater. With power like that available, there will be less demand for languages that squeeze the last drop of power out of your processor. Our implementation project has shown that you can design a language with programmer ease as top priority and implement it with good performance.

F.P.Brookes advises in his excellent book [Brookes]: "Plan to throw one away; you will anyway". This sound advice has served us well. With relatively low initial cost we produced a prototype implementation that, while not perfect, allowed us to test our ideas, observe *B* programming practice, and note where optimisation was necessary and where not. With this information we could then proceed to a new version with relative ease.

References

[Brookes] F.P. Brookes, *The Mythical Man Month,* Addison Wesley, 1975.

[Geurts] L. Geurts, *An Overview of the* B *Programming Language, or* B *without Tears,* SIGPLAN Notices, December 1982.

[Hibbard] P.G. Hibbard, P. Knueven, B.W. Leverett, *A Stackless Run-time Implementation Scheme,* in Proc. 4th Int. Conf. on Design and Implementation of Algorithmic Languages, ed. R.B.K. Dewar, Courant Institute, New York, 1976.

[Krijnen] T. Krijnen and L. Meertens, *Making B Trees Work for* B, Report IW 219/83, Mathematical Centre, Amsterdam, 1983.