The *B* Programming Language and Environment

A new programming language and environment for personal computing designed at the CWI

by Steven Pemberton

New computers, old languages

It is a common observation that the latest personal computers are very powerful. Certainly more powerful and more capacious than many of the previous generation of 'large' computers. Thus, it is quite feasible that many personal computers will spend most of their time idle, not from lack of use, but from under-filled capacity. And this is not because of delusions of grandeur on the part of the purchaser: the central processor, the part that is responsible for much of the measure of speed in a computer, is but a tiny part of the cost of a modern computer; there is no economic (or other) advantage in using a slower processor.

It is therefore rather surprising to realise that most programming on personal computers is done with programming languages that are for the most part 15 to 20 years old, languages designed for computers of a previous generation (or before). In particular, whatever personal computer you buy, you can be sure that the one language that the manufacturer has supplied for you is BASIC, a language designed in the mid-sixties, and which has been described as "an adaptation to early and very marginal computer technology" [1]. Thus you have the strange situation of people programming the computers of the eighties with a language of the sixties, a language unable to take advantage of the increased capabilities of the newer machines.

Two main advantages of BASIC are that it is interactive, and that it is simple. Interactiveness is the ability to type in and run a program immediately without going through any intermediate stages like translating the program into machine code, and to correct a program and re-run it immediately. Strictly speaking, this is a property of an *implementation* of a language, and not of the language itself, since it is in principle possible to make an interactive implementation of any language, or a non-interactive version of BASIC. But notwithstanding, a language usually has features that orient it more or less towards interactive implementation, and BASIC is usually implemented interactively, and most other languages are not.

Simplicity is a property often claimed for a programming language or system, without the term being properly clarified. For there are two, in some ways conflicting, senses to the word. You can have definitional simplicity, where there are only a few concepts, and you can have what might be called psychological simplicity, where the concepts are closest to your needs. A

couple of examples.

- The idea of a Turing machine is specifically to give the simplest model of a computational machine. No one however would consider it simple to program for.
- Boolean algebra can be expressed using a single operator 'not and'. However, no one would consider the expression

  (q nand (q nand q)) nand ((p nand p) nand (q nand q))

  as simpler than the equivalent

  not (p or q)

  despite the larger number of concepts in the second.

The simplicity of BASIC is actually a definitional one. It is easy to implement, and has few concepts to be learnt, but once learnt, it only remains easy to use for very small programs. Beyond that it is like cutting your lawn with a pair of scissors.

In schools

BASIC is also the principle programming language taught in schools. Apart from the perceived simplicity, it is usually the case that schools simply could not afford machines large enough to run anything other than BASIC. That situation will change quickly enough with the coming generation of cheap large computers, but the risk is that schools will continue using BASIC from sheer momentum, and perceived 'investment' in the language (a common barrier to change outside schools too). Risk, because BASIC has little to recommend it for educational use (and I say this as someone who has had to teach students coming from schools where they have learnt to program with BASIC). Just as with teaching yourself to type with one finger, where if you then want to learn to type properly you must first get rid of your old habits, BASIC's paucity of structuring facilities means that much time has to be dedicated to learning ways of getting round its expressive poverty, and the student ends up learning bad habits that must only be unlearnt in order to progress to other languages.

Well, either that, or they learn nothing at all, for quoting from [1] again: "For the vast majority of students who learn to program in BASIC, learning to program means learning a few set pieces of programming from a textbook and devoting the rest of their time at the terminal to playing computer games."

B

B is a programming language being designed and implemented at the CWI, together with an integrated programming environment (it should be noted that '*B*' is just a working title; the system will gain its definitive name when the language is frozen.) It was originally started in 1975 in an attempt to design a language for beginners as a suitable replacement for BASIC. In the intervening years the emphasis of the project has shifted from "beginners" to "personal computing", but the main design objectives have remained the same:

- simplicity;
- suitability for interactive use;
- availability of constructs 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 CWI (then called the Mathematical Centre) in 1975-6 and 1977-9, and were more in the line of definitionally simple, being easy to learn and easy to implement.
In the third iteration of *B*, designed in 1979-81 with the addition of Robert Dewar of New York University, it became psychologically simple: 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.

However, *B* is not just a language, but a complete programming environment. Traditional computer use for programming involves not only learning the programming language, but also a whole host of sub-systems and their commands, often completely separate and non-cooperating. *B* on the other hand shows one face at all times to the user, and it is not necessary to learn anything outside the *B* system.

Simplicity

B has just two basic data types: texts and numbers, and three ways of combining other values: compounds, lists and tables.

Numbers have two surprises for the seasoned computer user: firstly, on the basis of the maxim of no restrictions, numbers may be as big as wanted (within, of course, the physical limits of the computer's available memory); you may just as easily calculate 10^{200} as 10^2. In fact a Dutch newspaper recently dedicated a whole page to printing the value $2^{132049}-1$ (the current largest prime), which had been calculated with the *B* program

```
WRITE 2**132049-1
```

which, though it took a while to run, nevertheless produced the final answer consisting of more than 37000 digits.
Secondly, as long as it is possible, numbers are always kept exact, even

fractional numbers. Thus, as long as you use exactness-preserving operations, such as addition, subtraction, multiplication, and even division, a number is calculated exactly. Operations such as taking the square root cannot of course produce an exact result in general, and so in this case an approximate number results, rounded to some length.

Mathematicians and computer scientists alike are often surprised by the following little program that uses the properties of arithmetic in *B* to calculate the digits of π by evaluating the continued fraction

$$\cfrac{4}{1+\cfrac{1}{3+\cfrac{4}{5+\cfrac{9}{\cdots+\cfrac{k^2}{(2k+1)+\cdots}}}}}$$

```
HOW'TO PI:
    WRITE "3."
    PUT 3, 0, 40, 4, 24, 0, 1 IN k, a, b, c, d, e, f
    WHILE 1 = 1:
        PUT k**2, 2*k+1, k+1 IN p, q, k
        PUT b, p*a+q*b, d, p*c+q*d IN a, b, c, d
        PUT f, floor(b/d) IN e, f
        WHILE e = f:
            WRITE e<<1
            PUT 10*(a-e*c), 10*(b-f*d) IN a, b
            PUT floor(a/c), floor(b/d) IN e, f
```

Texts are strings of printable characters. Unlike many other languages *B* has a full range of operations on texts, such as joining texts together, replicating them, taking sub-strings and so on. Just as with all types in *B*, there is no maximum limit imposed on the size of a text, nor does the size have to be declared in advance.

Compounds are the way of making tuples, or records as they are called in some other languages, for instance for making complex numbers:

```
PUT 0, 1 IN z.
```

Lists are sorted collections of elements, again unrestricted in size. The elements of a given list must all be of the same type, but otherwise, and this is another surprise for the experienced programmer, may be of any type. Thus you may have lists of texts, numbers, compounds, lists of other lists, and so on. Elements may be duplicated; thus a list is a multiset or bag. You can insert elements, delete elements, find out if an element is present, find the size of a list, and so on. Here is a program that uses lists of numbers and the sieve method to calculate primes.

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

The last type is the table. Tables are mappings from values of any one type onto values of any one other type, and as such are a generalisation of arrays in other programming languages. Standard programming languages only allow you to map integers (and sometimes a few other similar types) onto other types. It is one of the biggest surprises, bordering on disbelief, for experienced programmers, that you may use any type for the indexes of *B* arrays. Thus if you want mappings from texts to lists, or tables to numbers, or tables to other tables, all are possible.

As an example, consider representing a directed acyclic graph as a mapping from nodes to lists of nodes:

```
PUT {[0]: {3}; [3]: {7; 8}; [7]: {8}; [8]: {}} IN graph
```

You can write a test to see if two elements are connected as follows:

```
TEST a connected'to b:
    SHARE graph
    REPORT b in graph[a] OR indirectly'connected
indirectly'connected:
    REPORT SOME e IN graph[a] HAS e connected'to b
```

and then write

```
IF 0 connected'to 8: ...
```

Other examples of surprises for the seasoned programmer that the newcomer will find unremarkable are in the READ command. If a running program is to input a value from the user, the READ command is used. In traditional languages, you can only read numbers and characters, and furthermore only constants of these types. However, in *B*, any type of value may be read, and further, any expression may be typed as input. This includes the use of variables, functions and so on.

It may be remarked from the above examples that although the data types of *B* are unusual, the kind of commands, or statements, are rather familiar.

There are the usual input and output commands, the assignment command, if and while commands, and so on. In fact the only unusual feature is the refinement, such as `REMOVE'MULTIPLES` and `indirectly'connected` in the above examples. These explicitly support the idea of 'step-wise refinement', so often practised in programming, but so rarely supported by programming languages.

As you can see, *B* has a small set of rather powerful data types. This is in comparison with most other languages that supply you with a number of *low-level* tools, that you must then use to build your own high-level tools.

B does it just the other way round. You get high-level tools which you can use for low-level purposes if you wish. For instance:

- If the numbers you use in a program are all less than a certain limit, you don't have to do anything special in *B*. In other languages, if your numbers go higher than a certain limit, you must write your own numerical package.
- In traditional languages, if you wish to use sparse arrays, you must write a package to implement them using the non-sparse arrays in the language. In *B*, sparse arrays (i.e. tables) are the default, but you can use them in a non-sparse way without extra effort.
- Traditional languages sometimes supply a pointer type, which you can then use to create data space dynamically. In *B*, data space is automatically dynamic. Furthermore, if you study the use of pointer types in other languages, you will see that they are almost always used for sorting and searching purposes. *B* supplies these sorting and searching facilities as primitives. If you still need to use pointers, you can represent them using *B* tables, but with additional advantages, for instance that you can print tables while you cannot print pointers.

Another feature of the simplicity of *B* lies in its environment. Global variables 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 there is no need for extra file-handling facilities in the language. 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.

Compare the following two programs in *B* and Pascal for counting the number of characters in a text file. In *B*:

```
PUT 0 IN size
FOR line IN document:
    PUT size+#line IN size
WRITE size
```

In Pascal:

```
program count(document, output);
var document: text;
    c: char;
    size: integer;
begin
     reset(document);
     size := 0;
     while not eof(document)
     do begin
             while not eoln(document)
             do begin
                     read(document, c);
                     size := size + 1
                end;
             readln(document)
        end;
     write(size)
end.
```

This, I contend, speaks for itself. In fact, these two programs illustrate clearly how compact *B* programs turn out to be. It is my experience that *B* programs are around a quarter or a fifth of the size of their equivalent Pascal programs (this comparison includes a 1000 line Pascal program which resulted in a 200 line *B* program). The ratio against BASIC would be even further in *B*'s favour. This clearly has consequences for programmer efficiency, especially as programmer effort is proportional not to program length, but a *power* of program length. Brookes [2] reports that empirical studies show this power to be around 1.5. This would imply that *B* is something like an order of magnitude easier to use than traditional languages.

The other side of this efficiency coin is that, because of its higher level, the language is no longer so straightforward to implement, and because it is interpreted, a given program in *B* will not run as fast as an equivalent program written in a non-interactive language. However, we have already noted that the personal computers of the new generation are so powerful that they will spend a large proportion of their time idle. This trade-off of computer time against programmer time is more than reasonable in view of this excess computational capacity. Furthermore, there are other trade-offs involved when comparing non-interactive languages with interactive ones, such as the absence of a translation phase in an interactive language.

Comparing *B* with BASIC on this score is another matter. BASIC implementations tend to be slow anyway, yet many people are willing to accept this slowness in return for interactive access. For instance, Bentley reports [3] that BASIC on an (apparently large) personal computer he bought ran at 100 instructions per second. This is even slower than the first commercially

produced computers of the 1950's which ran at 700 instructions per second!

Of course, higher-level commands like those of *B* take more time individually, but on the other hand fewer have to be executed to do the same job and more work is done at the (faster) system level than with a lower-level language. The combined effect depends on the application: simple programs — which take little time anyway — will generally run slower, but more complicated tasks may well run faster than if coded in a lower-level language.
But still, even if a given program in *B* runs slower than acceptable (for instance in the case of a commercial application which must run as fast as possible on a slow micro-computer), the programmer efficiency of *B* still makes it a good choice for the prototyping phase of a project.

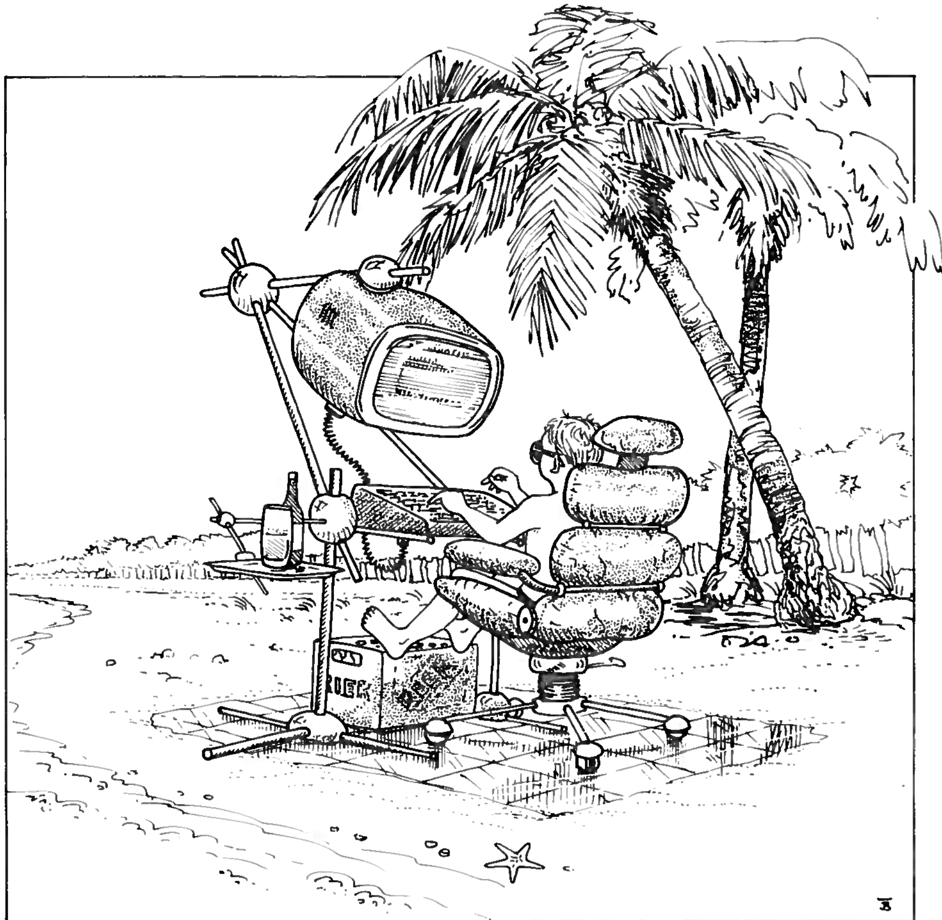
Interaction

Just as with BASIC, any command typed straight in at the terminal will be executed immediately. Thus you may use all the features of *B* as a sort of high-grade calculator:

```
WRITE root 2
1.41421356237
```

Furthermore, since user-written programs are called in exactly the same way as built-in *B* commands, much of the need for a separate command language so often found on computers disappears: as already pointed out, variables serve as files, and since programs are just the equivalent of subroutines in other languages, parameters can be passed to programs using the normal parameter passing mechanism of subroutines. In most systems, if parameters can be passed to a program at all, it is usually with a completely different mechanism.

One of the demands on an interactive language is that typing be minimised, since so much time is spent at the keyboard. One solution to this used by some interactive systems is to use abbreviated commands, but this generally results in very cryptic looking commands. *B* solves this by having a dedicated editor that knows much about the syntax and semantics of *B*. As an example, consider the above `WRITE` command. This is the second most used command in *B* (the first is `PUT`) and so when you type a `W` as first letter of a command, it is more than likely that you want a `WRITE` command. To this end, the moment you do type such a `W`, the system immediately suggests the rest of the command to you, by showing `WRITE` on the screen. If you do want a `WRITE` you can then press the 'accept' key and the system positions you so that you can type in the expression that you want to write. If you don't want a `WRITE`, but a `WHILE` say, then you just ignore the suggestion and type the next character, an `H`. The system then changes the suggestion to `WHILE`, and so on. This also works for the commands you write yourself (such as the `SIEVE'TO` unit defined earlier). The system also knows about things like matching brackets and supplies these for you. Thus certain typical sorts of typing error are just not possible in *B*.



The editor is also used in place of many functions that would normally be performed by a separate command language. For example, it is possible to edit the list of units (procedures and functions) that you have: if you delete an entry in the list using the editor, the corresponding unit disappears. Another feature of this is that you may edit the list of commands that you have typed in and executed: this then causes the changed commands to be re-executed as if you had typed the commands in in that way in the first place.

Another feature of the interactiveness of *B* is that declarations are not used. BASIC users usually perceive this as an advantage because it means less typing. Users of other languages, such as Pascal on the other hand, accept declarations on the grounds that they allow type inconsistencies and other similar errors to be detected before the program is run, therefore reducing the time taken to get a program correct.

B supplies the advantages of both, by inferring the types of variables from the way they are used (for instance, if you say `1+a`, `a` must be a number), and checking that all such uses are consistent. Furthermore, inconsistencies can be spotted as the command is typed in, increasing the interactive feel of the language.

Teaching

It is our feeling that *B* is well suited for teaching purposes. The availability of program and data structuring facilities, including support for step-wise refinement, means that students are less likely to adopt bad habits. More importantly, because of *B*'s high-level, a student can quickly get to a level of competence to produce useful working programs, rather than just trivial exercises.

B is currently being taught at a Dutch school in collaboration with the CWI to several classes of different school types. However, this has only recently started.

Implementation

Part of the effort at the CWI is to create an implementation of *B*. We have had a pilot implementation running for several years, and have now just finished a release version.

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.

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. Like the pilot system, it is written in the programming language 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.
This implementation runs on larger machines that run Unix* (with at least 128 Kbytes of main store) and is freely available for non-commercial use at the cost of the media.

One of the features of the implementation is the way values are implemented, based on the scheme of Hibbard, Knueven and Leverett [4]. Here, each value includes a count of how many copies of it there exist. When a value has to be copied, instead of copying the whole object, only a pointer to it is copied, and the associated counts are updated. When a count reaches zero, the value is disposed of.
When a value has to be modified, such as by inserting a value in a list, if its

* A trademark of AT&T Bell Laboratories

count is greater than one then the value must first be 'uniquified' by (really) copying it to a fresh area of store (actually only part of it is usually copied because, for instance, if the list is a list of tables, the tables need only have their counts updated, since they are not changed themselves.) Already unique values are 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 this method becomes cheaper than ordinary copying. This critical size is rather small, and since *B* values easily become large, it is advantageous. Furthermore, PUT commands are typically the most executed sort of command in programs, and so it makes sense to choose a method that favours them.

The implementation uses B trees (no relation) [5] to represent texts, lists, and tables. 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, only a sub-section of the tree needs to be copied.

We have been lucky to receive, through the generosity of IBM Netherlands, an IBM Personal Computer, and we are now busy transporting the implementation to it.

The Future

There will be one final polishing of the language before it is finally frozen, to clear up a few odd corners. However, most work on the system is now focusing on the environment, for instance to try and do for graphics and data-entry what up to now we have done for programming.

Further information

For more details of the language, refer to reference [6]. There is a *B* newsletter published at the CWI, with further details of the *B* environment. An annotated list of *B* publications is given in an appendix.

Conclusion

The time has come that personal computers have so much power that a new programming language is called for to take advantage of that power. *B* has been designed with just such an aim, to satisfy the needs of people who, while not being professional programmers, nevertheless need to use personal computers. Although the language was designed with these 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.

References

[1] Seymour A. Papert, *Computers and learning*, in M.L. Dertouzos (ed.), *The Computer Age*, MIT Press, 1979, 73-86.

[2] F.P. Brookes, *The Mythical Man Month*, Addison Wesley, 1975.
[3] Jon Bentley, *Programming Pearls*, Comm. ACM, **27** (1984) 3, 181-184.
[4] P.G. Hibbard, P. Knueven, B.W. Leverett, *A Stackless Run-time Implementation Scheme*, in R.B.K. Dewar (ed.), *Proc. 4th Int. Conf. on Design and Implementation of Algorithmic Languages*, Courant Institute, New York, 1976, 176-192.
[5] T. Krijnen & L. Meertens, *Making B Trees Work for* B, Report IW 219/83, Mathematical Centre, Amsterdam, 1983.
[6] Leo Geurts, *An Overview of the* B *Programming Language*, SIGPLAN Notices, **17** (1982) 12, 49-58.

Appendix: Available publications about B

A number of publications about *B* are currently available. Unless otherwise stated, all are published by the CWI; an order form can be found at the end of this newsletter.

An Overview of the B *Programming Language, or* B *without Tears,*
Leo Geurts, CWI report IW 208/82, 11 pages.

This is the first place to go if you want to know more about *B*. Also published in *SIGPLAN Notices* **17** (1982) 12, 49-58.

Draft Proposal for the B *Programming Language,*
Lambert Meertens, CWI, ISBN 90 6196 238 2, 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. A part of the book, the *Quick Reference*, also appeared in the *Algol Bulletin* 48 (August 1982).

Description of B,
Lambert Meertens & Steven Pemberton, CWI note CS-N8405, 38 pages.

This is the informal definition of *B* promised in the Draft Proposal. It aims to provide a reference book for the users of *B* that is more accessible than the somewhat formal Draft Proposal. While it is not a text book, it should also be useful to people who already have ample programming experience and want to learn *B*.

Computer Programming for Beginners — Introducing the B *Language (Part I),*
Leo Geurts, CWI note CS-N8402, 85 pages.

This is a text-book on programming for people who know nothing about computers or programming. It is self-contained and may be used in courses or for self-study. The focus is on designing and writing programs, as opposed to entering them in the computer, and so on. It introduces the language, and how to write small programs. Part 2, which will appear later this year, will treat the language, and programming, in greater depth.

A User's Guide to the B *System,*
Steven Pemberton, CWI note CS-N8404, 10 pages.
A brief introduction to using the current *B* implementation.

B *Quick Reference Card.*
A single card containing all the features of the language, the editor, and the implementation, for quick reference when using *B*.

An Implementation of the B *Programming Language,*
Lambert Meertens & Steven Pemberton, 8 pages.
This gives an overview of the implementation and some of the techniques used in it. Not published by the CWI. To appear in USENIX Washington Conference Proceedings (January 1984).

Making B Trees Work for B,
Timo Krijnen & Lambert Meertens, CWI report IW 219/83, 13 pages.
This describes a method of implementing the values of *B*. It is rather technical.

Incremental Polymorphic Type-Checking in B,
Lambert Meertens, CWI report IW 214/82, 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. Also published in *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1983, 265-275.

On the Design of an Editor for the B *Programming Language,*
Aad Nienhuis, CWI report IW 248/83, 16 pages.
Gives an overview of the design of a pilot version of the *B* dedicated editor.

On the Implementation of an Editor for the B *Programming Language,*
Frank van Harmelen, CWI report 220/83, 18 pages.
Gives details of a pilot implementation of the *B* dedicated editor.

Towards a Specification of the B *Programming Environment,*
Jeroen van de Graaf, CWI report CS-R8408, 23 pages.
This report contains an informal description and a tentative specification of the environment.

The B *Newsletter,*
This is produced to keep interested parties in touch with developments in the language and its implementation, and to provide a forum for discussions. Issues 1 and 2 have already appeared.