THE B NEWSLETTER

CWI, Amsterdam

Issue 2, June 1984

IMPORTANT NOTICE

If you want a copy of the Mark 1 *B* Implementation, there is an application form on the back two pages. This applies equally to everyone who has written to us asking for a copy.

CONTENTS

About This Issue

Amongst the articles in this issue are comparisons of *B* with Pascal and Basic. While we feel that it should not really be us who make such comparisons, we were specifically asked for these brief overviews, and we thought it worthwhile to include them in the newsletter. The same applies to the brief description of the highlights of *B*.

The Mark 1 Implementation

There is now an implementation of *B* available, for machines with at least 256k bytes of main store running under Unix. It is a full implementation of the language, with a small environment that includes a *B* dedicated editor front-end to the interpreter, variables that survive after logging out, and independently editable program units (procedures and functions).

The system is written in C, and while it currently runs under Unix, operating system dependencies are localised in a pair of files, and so modifications to allow it to run on a similar operating system are not difficult to make. Machine dependencies are localised in one file.

There is a trimmed down version that runs on larger PDP/11's (with split I and D spaces) that also comes with the release.

The editor uses the Termcap data-base to avoid dependencies on particular terminal types.

THE B NEWSLETTER

Plans for the Near Future

There will shortly appear a "Rationale for *B*" that will describe the reasons behind the features of *B*, the alternatives rejected, possible improvements, and so on. A list of seriously considered changes will be published in a future *B* newsletter. This will initiate the process of putting the final polish to the *B* language, and people will then be invited to submit proposals for changes to the language.

Thanks to the generosity of IBM Netherlands, we have been lucky enough to receive an IBM PC. Work is just beginning therefore on transporting the current implementation to it.

Work will also continue on optimising the implementation, including speed improvements as well as code-size reductions.

More function will be added to the implementation in the form of static checks, and an improved environment. In this phase the static type check and the static content check will be added, though they will only be done within units, and not between them. The environment will be extended to take it closer to what was described in the first newsletter, with a better screen manager, proper support for workspaces, and an editable command document. To this end the report by Jeroen van de Graaf will be followed.

There is a current project teaching *B* in a Dutch school to classes of several types, and this project will continue, and will hopefully broaden. Part 2 of the beginners' course will appear, as will a translation of Part 1 in Dutch.

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!timo*. Otherwise, articles and mail should be sent to

B Newsletter
Computer Science Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

From MC to CWI

On September 1, 1983, our institute changed its name into CWI — Centrum voor Wiskunde en Informatica, that is, Centre for Mathematics and Computer Science.

The name of the foundation remains unchanged: Stichting Mathematisch Centrum.

Examples of *B*

Steven Pemberton

Our experience with introducing *B* has shown that people don't appreciate the simplicity of *B* simply by having all its features enumerated, but rather by seeing example programs. Consequently here are a few interesting programs that demonstrate some of the features of *B*.

Numbers

Here is a simple guessing game.

```
HOW'TO GUESS:
    CHOOSE number FROM {0..99}
    WRITE 'Guess my number from 0 to 99: '
    READ guess EG 0
    WHILE guess <> number:
        SELECT:
            guess < number: WRITE 'Too low,  try again: '
            guess > number: WRITE 'Too high, try again: '
        READ guess EG 0
    WRITE 'Correct'

GUESS
Guess my number from 0 to 99: 50
Too high, try again: 25
Too high, try again: 15
Too low,  try again: 20
Too high, try again: 17
Too low,  try again: 19
Correct
```

• This next program, due to Lambert Meertens, prints the value of *pi* to a large number of places. It works by evaluating the continued fraction

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

It depends on the unbounded exact arithmetic of *B*, since the targets `a`, `b`, `c`, and `d` get very large indeed (for instance, after printing 80 digits of *pi*, all four values are larger than 10^{200}.)

```
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
```

Rather than let the program run to completion, we shall only print a little of its output here:

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899
```

Texts

A rather frivolous example of using texts (more serious ones follow) is this one that appears to answer questions with 'yes' or 'no' with some consistency. In fact all it does is output 'yes' if the number of n's in the question is even, and 'no' otherwise.

```
HOW'TO ORACLE:
    INPUT
    WHILE line <> '':
        SELECT:
            ('n'#line) mod 2 = 0: WRITE 'Yes' /
            ELSE: WRITE 'No' /
        INPUT
INPUT:
    READ line RAW

ORACLE
Are you unhappy?
No
So you are happy.
Yes
Do you know what time it is?
No
So you're stupid then?
No
But this program is.
Yes
```

Lists

Here is the complement of the guessing game program given above: the computer guesses the number. It works by keeping the list of numbers that the answer must lie in (initially `{0..99}`). Then as each guess is high or low, the range is restricted accordingly (for instance, if 50 is too low, the list becomes `{51..99}`). If the list ever becomes empty, then the player must have given the wrong answer at some stage.

```
HOW'TO PLAY:
    WRITE 'Think of a number from 0 to 99, and press [return]: '
    READ return RAW
    PUT {0..99} IN possible
    TRY
    WHILE reply <> 'y' AND possible <> {}: TRY
    IF possible = {}: WRITE 'Cheat!'
TRY:
    CHOOSE guess FROM possible
    WRITE 'Is it `guess`? '
    READ reply RAW
    PUT reply|1 IN reply
    SELECT:
        reply = 'y': WRITE 'Good' /
        reply = 'h': PUT {min possible..guess-1} IN possible
        reply = 'l': PUT {guess+1..max possible} IN possible
        ELSE: WRITE 'Possible answers are y(es), h(igh), l(ow)' /
```

```
PLAY
Think of a number from 0 to 99, and press [return]:
Is it 17? no
Possible answers are y(es), h(igh), l(ow)
Is it 33? l
Is it 38? l
Is it 53? h
Is it 45? l
Is it 50? y
Good

PLAY
Think of a number from 0 to 99, and press [return]:
Is it 93? l
Is it 96? h
Is it 94? h
Cheat!
```

• This next program solves the Towers of Hanoi problem by simulating parallel processes. Each disc is a separate process that repeatedly moves after a fixed interval. Each disc is put on a list of scheduled processes, with the time it is due to move at, its piece number (the smallest is numbered 1, the next largest 2, and so on), which tower it will next move to, the direction it is moving in (+1 or −1), and the time between its moves. Then the earliest process is removed from the list, the information printed and the process rescheduled.

```
HOW'TO HANOI n:
    INITIALISE
    FOR i IN {1..(2**n)-1}:
        SELECT'PROCESS
        WRITE 'Move piece', piece, ' from', from, ' to', to /
        RE'SCHEDULE
INITIALISE:
    PUT {} IN process
    FOR i IN {1..n}:
        INSERT 2**(i-1), i, 1, (-1)**(i+n), 2**i IN process
SELECT'PROCESS:
    PUT min process IN time, piece, from, direction, wait
    REMOVE min process FROM process
RE'SCHEDULE:
    INSERT time+wait, piece, to, direction, wait IN process
to:
    RETURN ((from+direction-1) mod 3) + 1

HANOI 3
Move piece 1 from 1 to 2
Move piece 2 from 1 to 3
Move piece 1 from 2 to 3
Move piece 3 from 1 to 2
Move piece 1 from 3 to 1
Move piece 2 from 3 to 2
Move piece 1 from 1 to 2
```

Tables

At first sight to the experienced programmer, the absence of a pointer type in *B* suggests that certain types of program cannot be written. In fact, thanks to tables, they can, and with certain advantages not least of which is the ease of printing a table.

To show the use of tables in this way, we will produce a small program to convert simple arithmetic expressions into trees, a traditionally typical example for pointers.

A tree can be represented in *B* as a table of nodes. It is relatively unimportant what the keys of such a table are, and in this example we will use numbers. Each node then consists of an indication of what kind of node it is (here we will use texts), and a table of numbers. These numbers point to the sub-trees of this node. For example, the tree for `a*2+b*2` can be represented as the table

```
{ [0]: ('a', {});
  [1]: ('2', {});
  [2]: ('*', {[1]: 0; [2]: 1});
  [3]: ('b', {});
  [4]: ('2', {});
  [5]: ('*', {[1]: 3; [2]: 4});
  [6]: ('+', {[1]: 2; [2]: 5})
}
```

In this case, instead of tables of sub-nodes, lists could be used since the associates are in increasing order anyway. However, if the nodes were identified by texts, say, then tables would have to be used.

The text of the expression to be compiled is passed over as parameter:

```
HOW'TO COMPILE expression:
    SHARE line, tree
    PUT {}, expression IN tree, line
    EXPRESSION x
    DISPLAY tree

HOW'TO DISPLAY tree:
    FOR i IN keys tree:
        WRITE i, ': '
        PUT tree[i] IN type, sub'nodes
        WRITE type
        FOR n IN sub'nodes:
            WRITE n
        WRITE /
```

The expression is parsed using recursive-descent: `EXPRESSION` calls `TERM` to parse a sub-expression; `next'char` returns the first character of the expression and `SKIP'CHAR` trims off the first character from the expression. `GENERATE` adds a node to the tree.

```
HOW'TO EXPRESSION x:
    TERM x
    WHILE next'char in {'+'; '-'}:
        PUT next'char IN op
        SKIP'CHAR
        TERM y
        GENERATE (op, {[1]: x; [2]: y}) GIVING x

HOW'TO GENERATE c GIVING p:
    SHARE tree
    PUT #tree, c IN p, tree[#tree]

YIELD next'char:
    SHARE line
    IF line = '': RETURN ' '
    RETURN line|1
```

```
HOW'TO SKIP'CHAR:
    SHARE line
    PUT line@2 IN line
```

TERM is very similar to EXPRESSION. OPERAND recognises a single letter or digit as operand to a sub-expression.

```
HOW'TO TERM x:
    OPERAND x
    WHILE next'char in {'*'; '/'}:
        PUT next'char IN op
        SKIP'CHAR
        OPERAND y
        GENERATE (op, {[1]: x; [2]: y}) GIVING x

HOW'TO OPERAND x:
    SELECT:
        next'char in {'a'..'z'} OR next'char in {'0'..'9'}:
            GENERATE (next'char, {}) GIVING x
        ELSE:
            WRITE 'Error at:', next'char /
            PUT -1 IN x
    SKIP'CHAR

COMPILE 'a*2+b*2'
0 : a
1 : 2
2 : * 0 1
3 : b
4 : 2
5 : * 3 4
6 : + 2 5
```

Tables and lists

An amusing program is this one from Leo Geurts for processing sentences in a natural language. It takes a number of sentences in some language, supplied by the user, and analyses them by saving all triples of characters that occur in the sentences. Thus the sentence "The cat is fat." contains the triples "The", "he ", "e c", " ca", "cat", "at ", "t i", and so on, up to "fat" and "at." These are saved in a table called followers that maps pairs of characters that occur onto the list of characters that may follow that pair. Thus in the above example, the table will contain, amongst others,

```
followers['a', 't'] = {'.'; ' '}
```

because of the two triples "at " and "at.". The program then starts generating new sentences at random based only on these triples. What is surprising is that it often generates real words that were not in the input.

ANALYSE saves each triple in a given line.

```
HOW'TO ANALYSE line:
    SHARE starters, followers, enders
    CHECK #line > 2
    PUT 1 th'of line, 2 th'of line IN c1, c2
    INSERT c1, c2 IN starters
    INSERT line@#line IN enders
    FOR c3 IN line@3:
        IF (c1, c2) not'in keys followers:
            PUT {} IN followers[c1, c2]
        INSERT c3 IN followers[c1, c2]
        PUT c2, c3 IN c1, c2
```

GENERATE generates a new sentence. It chooses the next character to output at random from the list of followers.

```
HOW'TO GENERATE:
    SHARE starters, followers, enders
    CHOOSE c1, c2 FROM starters
    WRITE c1, c2
    WHILE c2 not'in enders:
        CHOOSE c3 FROM followers[c1, c2]
        WRITE c3
        PUT c2, c3 IN c1, c2
    WRITE /

HOW'TO START:
    SHARE starters, followers, enders
    PUT {}, {}, {} IN followers, starters, enders
    READ line RAW
    WHILE line <> '':
        ANALYSE line
        READ line RAW
    FOR i IN {1..3}:
        FOR j IN {1..4}:
            GENERATE
        WRITE /
```

```
START
Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went,
That lamb was sure to go.

That lamb,
And as snow,
And as fleece as was wherywhite thad everywhite that lamb,
Its snow,

Mary hat Mary hat lite was fleece as snow,
Its fle where to go.
And every was sure to go.
Marywhittle whittleece lamb,

Mary had everywhery hat lamb,
Its sure as sure thad evere a little lamb,
Thad every went,
Mary hat lamb was snow,
```

Tables and texts

Here is a cross-reference generator. It takes a table of texts (the standard way in *B* of representing a traditional text-file) and produces from it a new table, mapping the words that occurred in the texts to their 'line numbers'.

The major routine is the one that saves a word in the cross-reference table.

```
HOW'TO SAVE word AT line:
    SHARE xtab
    IF word not'in keys xtab:
        PUT {} IN xtab[word]
    INSERT line IN xtab[word]
```

The cross-reference is made by taking each line in turn, and saving each word in that line in the table `xtab`

```
HOW'TO XREF text:
    SHARE xtab
    PUT {} IN xtab
    FOR line IN keys text:
        TREAT'LINE
    OUTPUT xtab
TREAT'LINE:
    FOR word IN words text[line]:
        SAVE word AT line

HOW'TO OUTPUT xtab:
    FOR word IN keys xtab:
        WRITE word<<10
        FOR line IN xtab[word]:
            WRITE line>>4, ' '
        WRITE /
```

The words in a line are generated as a list. A word has been isolated if its first character is alphabetic and nothing follows it or the first character following is not alphabetic.

```
YIELD words line:
    PUT {} IN list
    WHILE SOME head, word, tail PARSING line HAS word'isolated:
        INSERT word IN list
        PUT tail IN line
    RETURN list
word'isolated:
    REPORT word > '' AND alphabetic word|1 AND tail'not'alphabetic
tail'not'alphabetic:
    REPORT tail = '' OR NOT alphabetic tail|1

TEST alphabetic char:
    REPORT char in {'a'..'z'} OR char in {'A'..'Z'}


FOR n in keys text: WRITE n, text[n] /
1 Now is the time
2 for all good men
3 to come to the aid
4 of the party
```

```
XREF text
Now             1
aid             3
all             2
come            3
for             2
good            2
is              1
men             2
of              4
party           4
the             1       3       4
time            1
to              3       3
```

• Sometimes it is very useful to be able to invert a table. Consider a telephone directory mapping names to telephone numbers:

```
WRITE directory
{['Ed']: 4130; ['Han']: 4145; ['Jan']: 4130; ['Jo']: 4145; ['Leo']: 4141}
```

Clearly, some numbers have more than one subscriber, so the inverse mapping will have to be from numbers to lists of names:

```
YIELD inverse t:
    PUT {} IN t'
    FOR k IN keys t:
        IF t[k] not'in keys t': PUT {} IN t'[t[k]]
        INSERT k IN t'[t[k]]
    RETURN t'

WRITE inverse directory
{[4130]: {'Ed'; 'Jan'}; [4141]: {'Leo'}; [4145]: {'Han'; 'Jo'}}
```

The Highlights of *B*

Leo Geurts
Steven Pemberton

B is a system fully concentrating on ease for non-experienced users. In designing it we have looked for attractive features in existing systems, and combined some of those with our own ideas. Although it is difficult to show how easy it is to use *B* by just giving its individual features, here is a list of the good points of *B*.

The language proper

Types. *B* offers all the advantages of strong typing found in languages such as Pascal. Two nice data types of *B* are the *list* (a bag or collection of items of one type) and the *table* (an array with indexes of any one type and stored values of any one type). Both these types are nicely related via the function `keys`, which delivers the list of indexes of a table. Unlike the types of Pascal, *B*'s lists and tables are fully dynamic. The number of types in *B* is small: 5, and they have been chosen in such a way that they may easily be combined to simulate any other type.

No declarations. Unlike most typed languages, *B* has no declarations, the types being inferred from context.

No limits. Apart from sheer exhaustion of memory, *B* does not allow limits to be imposed by the implementation. So identifiers may have any length, numbers may have any magnitude, a list may contain any number of items, and so on.

Refinements. To support top-down programming *B* has refinements, which behave like parameterless light-weight procedures.

Nesting by indentation. Indentation is used to indicate nesting. This obviates constructs like **begin** ... **end** and **do** ... **od**, and allows a better view of program texts. It also prevents confusion due to contradiction between indentation and keywords.

The environment

No files. Because global variables are permanent in *B* and since values such as tables may be extended at will, there is no need for an extra file concept and the bother of special file handling commands.

One face. The design of the environment is based on the philosophy that one consistent face is shown to the user at all times. This means that all parts of the environment should know about the language and about each other. The user is always speaking to the syntax-directed editor, also when entering input to a program. All the time the editor guards against syntactic errors, and signals many kinds of semantic errors as soon as they are typed in, rather than saving up these reactions until after some final analysis. The kind of feel that the environment gives the user is that of a conversation, not that of a bureaucracy.

You get what you expect. The orientation towards non-expert users may be seen from the fact that many details are organized in a way that may be unusual to the experienced programmer, but which reflects the expectations of a newcomer. The precise arithmetic and nesting governed by indentation were already mentioned. Other examples from the language proper are notations such as

```
3 root 2    (rather than:  2**(1/3)),
sin pi      (rather than:  sin (pi)),
0 < p < 1   (not:  0 < p AND p < 1),
IF nrs > {} AND max nrs > 5:...
(rather than:
IF nrs > {}:
    IF max nrs > 5:...).
```

One way this approach is reflected in the design of the environment is the fact that any document the user changes with the editor and sees changed on the screen is really changed itself, rather than a working copy of it. One step further, when the user edits the document which lists the *units* (procedures) available and deletes one of them, then that unit itself will also be discarded.

Generalized undo-mechanism. By editing the document which lists the commands executed so far, the user may undo any undesired effect of erroneous commands.

A Comparison of Basic and *B*

Leo Geurts

Steven Pemberton

1. Good points of Basic

Good properties of most versions of Basic are:

1. Basic may be used in an interactive way: after being corrected, a program may be re-executed immediately, and input may be supplied at the moment the program asks for it.
2. Basic is simple, in the sense that one only needs to learn a little to know the whole language.
3. Variables need not be declared.
4. It is easy to write a Basic program for a small problem. Such a program may then quickly be executed, and, if necessary, be changed and re-executed.
5. In most cases Basic is embedded in a standard miniature operating system, suitable for file manipulation, editing and program execution.

2. Shortcomings of Basic

During its existence, some shortcomings of Basic have become apparent:

1. The Basic editor is based on typewriter-terminals and, consequently, does not exploit the extra possibilities of a visual display screen.
2. Numbers and strings are the only types in Basic and there is no way to build structured types using numbers and strings, except for a restricted form of array.
3. The most important control command is GOTO, plus a FOR construct.
4. In conclusion, although Basic was designed as a language that was easy to learn, programming in it turns out to be more difficult than in more modern languages, because Basic programs for all but the smallest problems are too complex for a reasonable overview.

3. *B*

B combines the advantages of Basic with improved aspects of modern programming language design:

1. *B* has a number of data structures which can easily be learnt, and which enable the user to manipulate large groups of data as a whole.
2. *B* has a variety of control structures, including procedures and refinements (i.e. parameterless local procedures, very suitable for stepwise refinement).
3. *B* has full type checking, warning against many errors while the program is being typed in. Even so, variables need not be declared in *B*.
4. For *B*, a powerful programming environment is in preparation, which will enable programmers to have an overview of the programs and data they are working with, and to manipulate them as if they were objects lying on a desk.
5. An important part of the programming environment of *B* will be the editor, which not only detects errors, but also formats the program for the user. It will make full use of the facilities of visual displays.
6. In conclusion, *B* is reasonably quick to learn, and is very easy to use, both for small programs and for rather large ones, especially because the user may write a program using the same kind of terms and concepts as used while thinking about the program.

A Comparison of Pascal and *B*

Steven Pemberton

Pascal was principally designed as a teaching language. *B* is principally a language for non-professional users. This means that the two languages have a similar, if not identical, audience.

1. Good points of Pascal

1. Pascal supports modern programming practices by allowing you to create your program and data in a structured way.
2. Pascal is a relatively simple language, with a relatively small set of program structures to learn.
3. Many of the errors a programmer makes are caught before the program runs by using declarations and so-called strong typing to check for inconsistencies in the program.
4. Many aspects of a program may be expressed in a problem-oriented way rather than a machine-oriented way.
5. Pascal was designed so that many of its types can be combined to build new types.

2. Shortcomings of Pascal

However, Pascal still has some shortcomings:

1. Despite its simplicity, Pascal has many data-types (between 11 and 13, depending on your point of view). This means there is much to learn, and it can be confusing for a beginning programmer.
2. Most of Pascal's data-types are fixed-size, which is often inconvenient when writing a program, and there are only very limited facilities for string handling.
3. Pascal's dynamic data type (pointer) is very low level, and consequently hard to use and error-prone.
4. Pascal is consistent in many of its rules. However there are many irritating little inconsistencies, and a few larger ones, justified not from a programming point of view, but from an implementation view.
5. The syntax of Pascal is very fussy, and much trouble can be caused by mis-placed characters.
6. It has only very limited input and output facilities.
7. Pascal was not designed for interactive use, and consequently is hard to use in such a way.
8. Because of the one-pass nature of Pascal compilers, you are forced to write your programs in the reverse order to how you compose them.

3. *B*

B combines the advantages of Pascal with the following improvements:

1. As well as allowing program- and data-structuring, *B* has refinements which support the practice of stepwise refinement.
2. All data-types in *B* are dynamic, in the sense that their size is neither predetermined nor fixed. The programmer does not have to take care of allocating space for dynamic values.
3. *B* has only a small number of types (five) which may be combined in any way to produce new data-types. There are full string-handling facilities.
4. *B* was designed with interactive implementations in mind, and includes an effective programming environment.
5. *B* has all the advantages of strong typing, for discovering errors in advance, but without the disadvantages of declarations.
6. In conclusion, *B* is small, powerful, and reasonably quick to learn. Furthermore the programmer is not troubled by restrictions due to limitations imposed for non-algorithmic reasons.

New Publications about *B*

Since the appearance of the first *B* newsletter a number of new publications about *B* have appeared or are about to appear. The publications mentioned in the first newsletter are all still available, as is the newsletter itself.

Computer Programming for Beginners — Introducing the B *Language — Part 1,*
Leo Geurts, 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, and not on 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. Published by CWI, note CS-N8402, price Dfl. 11.90.

Description of B,
Lambert Meertens and Steven Pemberton,
38 pages.

> This is the informal definition of *B* promised in the Draft Proposal. The aim is 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*. Published by CWI, note CS-N8405, price Dfl. 6.00.

An Implementation of the B *Programming Language,*
Lambert Meertens and Steven Pemberton,
8 pages.

> This gives an overview of the implementation and some of the techniques used in it.
> Published in USENIX Washington Conference Proceedings (January 1984), to appear.

A User's Guide to the B *System,*
Steven Pemberton, 10 pages.

> A brief introduction to using the current *B* implementation. Published by CWI, note CS-N8404, price Dfl. 3.70.

B *Quick Reference Card.*

> A single card including all the features of the language, the editor, and the implementation, for quick reference when using *B*. Available from CWI.

The B *Programming Language and Environment*
Steven Pemberton, 12 pages.

> Gives a description of *B* along with some background to it, and some justification for its existence, arguments about simplicity, interactiveness, programmer productivity, and talks about its suitability for use in schools. Published in CWI Newsletter, Vol 1, No 3. Available free from CWI.

On the Design of an Editor for the B *Programming Language,*
Aad Nienhuis, 16 pages.

> Gives an overview of the design of the first approximation of the *B* dedicated editor. Published by CWI, report IW 248/83, price Dfl. 3.70.

Towards a Specification of the B *Programming Environment,*
Jeroen van de Graaf, 23 pages.

> This report contains an informal description and a tentative specification of the environment. Published by CWI, report CS-R8408, price Dfl. 3.70.

CWI publications can be ordered from

> Publications Department
> CWI
> POB 4079
> 1009 AB Amsterdam
> The Netherlands

You will be invoiced. The prices quoted exclude postage and packing, and for foreign orders there is an additional charge of Dfl. 6.50 to cover bank charges.

HOW TO ORDER MARK1:

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

B Implementation
Computer Science Department
CWI
POB 4079
1009 AB Amsterdam
The Netherlands

You will then receive:

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

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

---

† UNIX is a Trademark of AT&T Bell Laboratories

--

--

ORDER FORM

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

Name: .....

Title: .....

Firm/Institute: .....

Address: .....

.....

Telephone: .....

USENET network address: .....

Check required tape parameters:

density	☐ 800 bpi	☐ 1600 bpi
blocking factor	☐ 1	☐ 20
format	☐ Ansi labelled	☐ Tar

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

Signature and Date:

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

SOFTWARE AGREEMENT

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

(LICENSEE), having an office at

agree as follows:

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

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

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

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

signed by

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

SOFTWARE AGREEMENT

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

(LICENSEE), having an office at

agree as follows:

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

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

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

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

signed by

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