Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S. Pemberton

A user's guide to the B system

Department of Computer Science Note CS-N8404 June

A USER'S GUIDE TO THE *B* SYSTEM

S. PEMBERTON
Centre for Mathematics and Computer Science, Amsterdam

B is a new interactive programming language being developed at the CWI. This report gives a brief introduction to using the current *B* implementation. It does not teach about the language, for which you should refer elsewhere.

1982 CR CATEGORIES: 69D49.
KEY WORDS & PHRASES: programming languages, *B*, programming environments.

Note CS-N8404
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Introduction

This guide is a brief introduction to using the current *B* system. It does not attempt to teach you about the language, for which you should refer elsewhere.

Furthermore, certain details such as how to start the system up, and which keys on your terminal correspond to the keys mentioned, depend on your local installation, and are not mentioned in this guide. Keys are just referred to here by their name, such as $\boxed{\text{accept}}$. Refer to the "Quick Reference" which should have been modified for your system, for these details.

Some older *B* systems don't let you make changes to immediate commands, nor give you suggestions to immediate commands, but only in units.

Starting up

The first response you should get from the *B* system when you start it up is a prompt that looks like this:

```
>>> ?
```

The underlined question mark is the indication from the *B* system that it is expecting input from you. (In fact, it depends on the sort of terminal you have whether it is underlined, displayed in reverse video, or what. In any case it is displayed in some special way, and we shall use underlining in this guide.) When it follows the three arrows >>>, called the *command prompt*, it is expecting you to type in a command. The question mark is called a *hole* and indicates that something should be filled in; the underline is called the *focus* and shows where you are currently working.

You can fill this hole by typing in a `WRITE` command for instance: you type a `W` (which you don't have to type in upper-case: the system knows that it may only be upper-case here), and you immediately see:

```
>>> W?RITE ?
```

This extra stuff to the right of the focus is a *suggestion*. Most times that you type a `W` as the first letter of a command, it is because you want a `WRITE`. Therefore the editor suggests this, with an additional hole for the expression that you want to write. If you do want a `WRITE` (as in this case) you may press the $\boxed{\text{accept}}$ key to accept the suggestion — the editor then moves to the first unfilled hole in the command, which in this case is the only one, and you get:

```
>>> WRITE ?
```

You can now type an expression and press the $\boxed{\text{newline}}$ key. The system evaluates the expression, prints the result, and then gives you a new command prompt. Here are a few examples of `WRITE` commands:

```
>>> WRITE 2+2
4
>>> WRITE root 2
1.414213562373095
>>> WRITE pi
3.141592653589793
```

If you make a mistake while typing and spot it before you type $\boxed{\text{newline}}$, an easy way to correct it is to use the $\boxed{\text{back}}$ key. Pressing $\boxed{\text{back}}$ takes you back to the situation before you typed the last key (*exactly* the situation, as you will see clearly after a little use). If you type $\boxed{\text{back}}$ twice, you will be taken back to the situation as it was two keys ago, and so forth. You can regard the $\boxed{\text{back}}$ key as a way of travelling back in time.

Thus, if you meant to type `WRITE pi`, but instead typed `WRITE po`, you will see this:

```
>>> WRITE po?
```

Now pressing $\boxed{\text{back}}$ will give you

```
>>> WRITE p?
```

Now you can type the `i` and the $\boxed{\text{newline}}$. Currently you can only go back a maximum of 100 keystrokes, and in any case only as far as the command prompt (and thus not back to previous commands). You will see other ways to correct mistakes shortly.

If you make a mistake so that the result is illegal *B*, but don't notice before you press $\boxed{\text{newline}}$ you will get an error message:

```
>>> WRITE root 3**2+4**2
*** There's something I don't understand in your command
    WRITE root 3**2+4**2
                 ^
*** The problem is: priorities? use ( and ) to resolve
```

The problem here is that the system doesn't know if you want to apply `root` to `3` or `3**2` or `3**2+4**2` and you should use brackets to show which.

When you type an open bracket, the system automatically supplies the matching closing bracket for you:

```
>>> WRITE root(?)
```

You now type in the expression

```
>>> WRITE root(3**2+4**2?)
```

You may now type $\boxed{\text{newline}}$ ($\boxed{\text{accept}}$ will take you over the closing bracket, but it is not necessary to do this):

```
>>> WRITE root(3**2+4**2)
5
```

You can write any legal *B* value:

```
>>> WRITE {1..10}
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> WRITE 'Hello! '^^3
Hello! Hello! Hello!
```

Just as with brackets, the system automatically supplies the closing brace `}`, and the closing quote `'`. In the latter case, where you want to type something after the closing quote you may either use $\boxed{\text{accept}}$ or type the quote yourself, in order to position after it.

Commands typed as a response to the command prompt are called 'immediate' commands, since they are executed immediately. Another example is the `PUT` command. Just as with `WRITE`, when you type the first letter of the command, the system provides a suggestion:

```
>>> P?UT ? IN ?
```

Again, you use $\boxed{\text{accept}}$ to go to the first hole:

```
>>> PUT ? IN ?
```

Here you type an expression,

```
>>> PUT root 2? IN ?
```

followed by another $\boxed{\text{accept}}$ to take you to the second hole:

```
>>> PUT root 2 IN ?
```

where you can type a target, followed by $\boxed{\text{newline}}$:

```
>>> PUT root 2 IN a
>>> PUT root 3 IN b
>>> WRITE a
1.4142135623730951
>>> WRITE b
1.732050807568877
>>> WRITE a,b
1.4142135623730951 1.732050807568877
>>> WRITE a*a, b*b
2 3
```

The targets that you create in this way, through immediate commands, are called ‘permanent targets’, because if you stop using the system, and log out, and come back later and start using the system again you will find that the targets are still there, with the same values as before.

You can find out which targets exist by typing two equals signs after the prompt:

```
>>> ==
a b
>>> PUT 'hello', {1..10} IN message, list
>>> ==
a b list message
>>> WRITE list
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> WRITE message
hello
```

To get rid of targets you no longer want, use the DELETE command:

```
>>> DELETE a, b
>>> ==
list message
>>> WRITE a
*** Can't cope with problem in your command
    WRITE a
*** The problem is: a has not yet received a value
```

As you can see, after the DELETE command both a and b have ceased to exist.

Other Immediate Commands

In fact, almost any *B* command can be used as an immediate command; the only exceptions are the commands used to terminate TESTs and YIELDs, namely RETURN, REPORT, SUCCEED and FAIL.

```
>>> WRITE list
{1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
>>> INSERT 5 IN list
>>> REMOVE 6 FROM list
>>> WRITE list
{1; 2; 3; 4; 5; 5; 7; 8; 9; 10}
>>> CHOOSE number FROM list
>>> WRITE number
9
>>> CHECK 5 in list
>>> CHECK 6 in list
*** Your check failed in your command
    CHECK 6 in list
>>> FOR i IN list: WRITE 10*i
10 20 30 40 50 50 70 80 90 100
>>> ==
list message number
```

Note that a CHECK command that succeeds doesn't print any message. Also note carefully that the target i used in the FOR command doesn't exist afterwards.

When you want to type a WHILE command, and you type the initial W, the suggestion you get is of course W?RITE ?. Here you must type an H, and the system then suggests

```
>>> WH?ILE ?:
```

Now you can press accept to go to the hole, type the test, and press newline :

```
>>> WHILE list <> {}:
        ?
```

Because the *B* system knows that the commands of a WHILE must be indented, it indents for you automatically. You may now type in the commands to be part of the WHILE, and each time the system indents you to the right place. After the last command you just need to type an extra newline , and the system undoes the indentation one level, and executes the WHILE. (If there is only one simple command to be repeated, it may be on the same line as the WHILE, but doesn't have to be).

```
>>> WHILE list <> {}:
        CHOOSE number FROM list
        REMOVE number FROM list
        WRITE number
2 8 4 7 9 5 3 1 5 10
>>> WRITE list
{}
```

Finishing a B Session

The one command that has a different meaning when you type it after a prompt is QUIT, which just terminates the *B* session. When you type the Q you will get a suggestion as usual.

```
>>> Q?UIT
```

Here there are no holes, but you must still press accept to accept the suggestion, before pressing newline .

Making your own Commands

You can create your own commands by typing in a how-to unit that defines what your command means. Suppose you type in response to the *B* prompt:

```
>>> HOW'TO GREET:
        WRITE 'Hello'
```

The system gives a suggestion for `HOW'TO`, and supplies indentation for you, just as with `WHILE`, and you finish by pressing $\boxed{\text{newline}}$ until you get the command prompt again. Well, now you've defined your first command, and can execute it by typing its name after the prompt. You will notice that after typing the `G` you will get a suggestion for it:

```
>>> G?REET
```

Just as with `QUIT`, there are no holes, but you must $\boxed{\text{accept}}$ the suggestion.

```
>>> GREET?
```

Now press $\boxed{\text{newline}}$ and your command gets executed, and you get the prompt again:

```
>>> GREET
Hello
>>>
```

You may use your own commands just like any in-built command:

```
>>> FOR i IN {1..10}: GREET
HelloHelloHelloHelloHelloHelloHelloHelloHelloHello
```

Correcting errors

Apart from $\boxed{\text{back}}$, another way of correcting errors is to correct a whole line. Here you use the ability to move the focus about. Earlier the focus was a single character, just the hole. However, the focus may be more than one character: it may be several characters, a whole command, or even several commands.

Two of the eight keys for moving the focus are $\boxed{\text{up}}$ and $\boxed{\text{down}}$.

The $\boxed{\text{up}}$ key moves the focus up to the previous line so that it includes the whole line. So if you have the following situation:

```
>>> FOR i IN {1..3}:
        WRITE /
        GREET
        WRITE /?
```

then typing $\boxed{\text{up}}$ gives you

```
>>> FOR i IN {1..3}:
        WRITE /
        GREET
        WRITE /
```

Here the hole in the last line has disappeared (because the line is legal *B*) and the focus has moved up to the whole of the preceding line. You may press $\boxed{\text{up}}$ several time to go up several lines. So, pressing $\boxed{\text{up}}$ again gives:

```
>>> FOR i IN {1..3}:
        WRITE /
        GREET
        WRITE /
```

The $\boxed{\text{down}}$ key does the same but in the other direction.

Now the point of all this is, that if you have a line in a command or unit that you want to change, you can move the focus to it, and press $\boxed{\text{delete}}$ to get rid of it. This leaves a hole in its place so that you can type a replacement line:

```
>>> FOR i IN {1..3}:
        ?
        GREET
        WRITE /
```

If you don't want to replace the line, but completely delete it, then pressing $\boxed{\text{delete}}$ again deletes the hole too:

```
>>> FOR i IN {1..3}:
        GREET
        WRITE /
```

Changing Existing Units

Just as you can type two equals signs to find out what permanent targets you have, you can type two colons to find out what units you have in the current workspace. This gives you a list of the first line of each unit in the workspace:

```
>>> ::
HOW'TO GREET:
```

If you want to change any of these units, you can type a colon followed by the name of the unit you want:

```
>>> :GREET
```

(If the unit you want to change is the last unit you typed in or changed in this session, or the last unit that you got an error message about, then you don't even need to type its name: the system remembers the name of the unit, so all you need to do is type a single colon.)

What happens now is that the whole unit is displayed (or as much as will fit on the screen if it is big) with the focus on the line you were last at in the unit. You can now use all the focus moving keys to position to the lines you want to change, and change them. When you have finished, you use the $\boxed{\text{exit}}$ key.

Errors in Units

If when you type in or change a unit, the result has an error in it, you will get an error message from the *B* system. This may happen when you press $\boxed{\text{exit}}$, or when you run the unit, depending on the sort of error it is. For instance, in this unit, the parameter is `x`, but `n` is used instead:

```
>>> HOW'TO SQUARE x:
        WRITE n*n
>>> SQUARE 4
*** Can't cope with problem in line 2 of your unit SQUARE
    WRITE n*n
*** The problem is: n has not yet received a value
```

When you get such a message, it is very easy to make the necessary correction: since the unit you want to change is the last one that you got an error message for, you only need to type a colon after the prompt:

```
>>> :
HOW'TO SQUARE x:
    WRITE n*n
```

As you see, you are positioned at the line that gave the error message. Now you can either press $\boxed{\text{delete}}$ and retype the whole line, or use the other focus-moving keys to focus on only the part that is in error.

Other Focus Moving Keys

Apart from $\boxed{\text{up}}$ and $\boxed{\text{down}}$ there are six other keys for moving the focus, two for making the focus smaller: $\boxed{\text{first}}$ and $\boxed{\text{last}}$, two for enlarging it: $\boxed{\text{widen}}$ and $\boxed{\text{extend}}$, and two for moving it sideways: $\boxed{\text{previous}}$ and $\boxed{\text{next}}$.

In this case we want to make the focus smaller, by going to the last part of the line. Thus, we can press $\boxed{\text{last}}$ and we see:

```
HOW'TO SQUARE x:
    WRITE n*n
```

Pressing $\boxed{\text{delete}}$ deletes the part in the focus, leaving only a hole:

```
HOW'TO SQUARE x:
    WRITE ?
```

Now we can type the correct expression and then press $\boxed{\text{exit}}$:

```
HOW'TO SQUARE x:
    WRITE x*x
```

Suppose now you have the following unit to solve quadratic equations (if you don't know what these are, it doesn't matter):

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /

*** There's something I don't understand in line 3 of your unit SOLVE
    WRITE (-b+x)/2*a, (-b-x)/2*a /
                   ^
*** The problem is: priorities? use ( and ) to resolve
```

The system doesn't know whether to divide by 2 or 2*a, so you must insert brackets.

```
>>> :
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

Now we want to narrow down to 2*a. Pressing $\boxed{\text{last}}$ gives

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

and then $\boxed{\text{first}}$ gives

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

and another $\boxed{\text{first}}$ gives

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

Now $\boxed{\text{last}}$ gives us

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

The key $\boxed{\text{extend}}$ tries whenever possible to extend the focus to the right. But it sometimes can't, if what is to the right isn't associated with what is in the focus. This is the case here: the comma is associated with the *whole* expression (-b+x)/2*a, and so $\boxed{\text{extend}}$ extends to the left instead:

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

and once more pressing $\boxed{\text{extend}}$:

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/2*a, (-b-x)/2*a /
```

Now we have the expression we want, and typing an opening bracket encloses the whole expression:

```
HOW'TO SOLVE a X2 b X c:
    PUT root (b**2-4*a*c) IN x
    WRITE (-b+x)/(?2*a), (-b-x)/2*a /
```

Now you can use the focus moves to do the same to the second 2*a.

The focus move keys are very easy to use. After a little practice you will find they come very naturally.

Copying

It is often the case that you want to duplicate a piece of program. If the focus is on something other than a hole, the $\boxed{\text{copy}}$ key copies whatever is in the focus to what is called the *copy buffer* and lets you know that there is something in the copy buffer by displaying the words `[copy buffer]` at the bottom of the screen, along with the first bit of what is stored.

If, however, the focus is on a hole, then the $\boxed{\text{copy}}$ key copies the contents of the buffer into that hole. The `[copy buffer]` message then disappears, though actually the contents of the buffer remain, so you can continue to use it. However, if the contents of the buffer may not be copied there, you just get a bleep; for instance you can't copy a `WRITE` command to a place where an expression must be.

You can use the $\boxed{\text{copy}}$ key for moving things too: focus on what you want, press $\boxed{\text{copy}}$, press $\boxed{\text{delete}}$ twice to delete it and the hole that gets left after the first delete, move to where you want, make a hole, and press $\boxed{\text{copy}}$ again.

Making a hole is straightforward. A $\boxed{\text{newline}}$ always makes a hole on a new blank line after the line that the focus was positioned on. An $\boxed{\text{accept}}$ always takes you to the first hole on a line, or if there is no hole on the line, then it makes one at the end of the line. Finally, pressing $\boxed{\text{first}}$ or $\boxed{\text{last}}$ enough times makes one, $\boxed{\text{first}}$ in front of the focus, $\boxed{\text{last}}$ after it.

You can use the $\boxed{\text{copy}}$ key for copying between different units too. Even if you log out, and come back later, and start using *B* again, the copy buffer is kept.

Additionally, if the `[copy buffer]` message isn't being displayed, that is, if the copy buffer is empty, and you $\boxed{\text{delete}}$ something, whatever you delete is put into the copy buffer; similarly if the copy buffer is empty, each immediate command is stored in the copy buffer. Thus if you mistype an immediate command, you can use $\boxed{\text{copy}}$ to bring it back, and edit it.

Renaming and deleting units

If you change the name of a unit, or the adicity of a `YIELD` or `TEST` (such as making a dyadic `YIELD` into a monadic one) by changing its heading, then you get the new unit *and* the old one. So, renaming `GREET` into `HELLO`, and then giving a `::` command would give:

```
>>> ::
HOW'TO GREET:
HOW'TO HELLO:
HOW'TO SOLVE a X2 b X c:
HOW'TO SQUARE x:
```

If you delete the *whole* of a unit (by pressing $\boxed{\text{widen}}$ until the focus is on the whole unit, and then pressing $\boxed{\text{delete}}$) the unit disappears. Thus deleting `GREET` will give you:

```
>>> ::
HOW'TO HELLO:
HOW'TO SOLVE a X2 b X c:
HOW'TO SQUARE x:
```

Changing targets

You may also use the editor for changing permanent targets. Just as you use a single colon for editing units, a single equals followed by the name of a target will display the contents of the target, and let you make changes in the usual way. In fact, you may replace the contents by any *expression*. When you press $\boxed{\text{exit}}$, the expression is evaluated, and if all is ok the value is put in the target.

Workspaces

A workspace is a collection of units plus a permanent environment. You can have several workspaces: to create a new one you just start *B* up in a new directory in the filestore.

To move units and targets between workspaces, just use the $\boxed{\text{copy}}$ key: start *B* up in the one workspace, save whatever you want to move in the copy buffer, exit *B*, move to the destination workspace, re-enter *B*, and copy the buffer back.

Record and play

Sometimes you need to repeat a sequence of keystrokes several times. For instance, if you want to rename a target in a unit, you have to do it once for each occurrence of the target. An easy way to do this is to *record* a sequence of keystrokes. If you press $\boxed{\text{record}}$, the message `[recording]` appears at the bottom of the screen, and any keys that you type thereafter are processed normally and recorded at the same time, until you press $\boxed{\text{record}}$ again. Then pressing $\boxed{\text{play}}$ plays those recorded keystrokes back.

So, for instance, focus on the target you want to rename, press $\boxed{\text{record}}$, press $\boxed{\text{delete}}$, type the new name, and press $\boxed{\text{record}}$ again. Then focus on the next occurrence of the target, and press $\boxed{\text{play}}$.

Keystrokes that cause an error during recording are not recorded.

Redisplaying the screen

Sometimes the screen can get messed up (for instance, if your terminal gets accidentally unplugged). If this happens, or you don't believe what you see on the screen, you can always get confirmation by pressing $\boxed{\text{look}}$. This causes the screen to be redisplayed.

Interrupting a running command

If a command is executing, and you want to stop it, pressing $\boxed{\text{interrupt}}$ aborts the command and gives you a prompt again.

Incomplete units

If, when typing in or correcting a command or unit, you press $\boxed{\text{exit}}$ and there are still unfilled holes, the system tells you so, and you must fill or delete them. If you want to exit leaving the holes, to fill them later, use the $\boxed{\text{interrupt}}$ key. The system won't let you run an incomplete unit.

Getting help

Pressing the $\boxed{\text{help}}$ key gives you a quick summary of all the keys. Refer to the "Quick Reference" card for a brief reminder of the features of the *B* language, and the "Description of *B*" for more detailed explanations.

Running *B* in the background

If your computer system lets you, you can also run *B* non-interactively. In this case *B* commands are read from the standard input, and executed. You may only use normal *B* commands in this case: no focus moves, or editing, and no `::` or `==` commands.