stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 248/83 DECEMBER

A. NIENHUIS

ON THE DESIGN OF AN EDITOR FOR THE $\mathcal{B}$ PROGRAMMING LANGUAGE

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11 February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B99

1982 CR. Categories: D.2.3., D.3.4.

On the design of an editor for the *B* programming language

by

Aad Nienhuis

ABSTRACT

Language-dedicated editors use language-specific knowledge about syntax and (static) semantics of programming languages. They form a much more powerful tool for creating and modifying programs than conventional text editors.

This report describes such an editor, specially designed for editing programs written in the language *B*.

KEY WORDS & PHRASES: Programming environments, syntax-directed editors, *B*.

1. INTRODUCTION.

Since 1975 the *B group*[1] has been working on the design of a programming language, provisionally named *B*, suited for the non-professional programmer, although the professional will also find features in *B* which make it worth using. In contrast to many existing languages, *B* is a simple language that supports structured-programming. Design objectives for *B*, as found in [Geur76] are:

- simplicity,
- suitability for conversational use,
- inclusion of structured-programming tools.

From the start of the design, the *B* group did not envisage *B* as an isolated programming language embedded in some general operating system, but as part of an environment dedicated to the *B* programmer. Such an environment, the *B system*, consists of tools to assist the programmer with the process of creating, editing, running and updating *B* programs. One of the tools of the system[2] is an editor specially designed for editing *B* programs.

About ten years ago, the user of an editor created text by just typing it in and corrections were made by giving commands to replace, delete or insert text on a specific line (*line editor*). Without the updated representation of the text on a terminal screen, it was difficult for the user to keep track of which line she was working on.

To cope with this difficulty several solutions were found, such as more complex pointing mechanisms (like pattern matching) and printing (corrected) lines on the terminal screen. The next step was to use the terminal screen in a more natural way: allowing the user to make corrections in the representation of the text on the screen directly (*screen editor*), instead of giving commands which should make the corrections somewhere in the text.

One of the early screen editors is the one described by Irons and Djorup in [Iron72]. This editor conceives of text as organized in a quarter plane extending indefinitely in width and length. Several function buttons are used for moving through the text and for inserting and deleting lines or characters. To insert text, the user moves to the particular place and types the text. The environment is *modeless*; there is no need for the user to switch to a command-mode (for giving commands) or a text-mode (for inserting text).

More recent screen editors are Z [Wood81] and EMACS [Stal81], both using the same starting-points as Irons and Djorup, but having facilities that aid in program creation. The Z editor knows about program elements such as quoted strings, blank-separated words, matching tokens (such as begin/end) and indentation. The user can search for the end of a unit or skip over matching expressions, etc. EMACS is a highly extensible editor: every key is considered a command, the user can bind any command to any key and write her own commands. For example, one can bind a command to the close-parenthesis which, every time the user types it, moves the cursor briefly to the matching open-parenthesis. (The keys for the printing characters are normally bound to commands to insert these characters in the text.) It is possible to tailor EMACS to a specific programming language. The first pilot implementation of the *B* editor was such a tailored EMACS editor [Grun82].

The editor XS-1 [Burk80] abandoned the idea of editing plain text. It sees text as a hierarchical structure of data (*structure editor*). The structure can be seen as a tree with nodes representing subsets of data. The user can move up, down, left and right in the tree and delete or insert nodes. Although such editors are dedicated to specific types of texts (with an imposed hierarchical structure), they take no advantage of the syntax rules of the text the user is editing. If the user edits a program, which usually has a well defined syntax, the editor can help the user in several ways:

---

[1] CWI, Computer Science Department, POB 4079, 1009 AB Amsterdam, The Netherlands.

[2] We shall use system, editor, etc. for *B* system, *B* editor, etc. in cases where there is no danger of confusion.

- taking care of the generation of pieces of text that the user otherwise would have to type in,
- signaling violations against the structure.

The Cornell Program Synthesizer [Teit81] is such a *syntax-directed editor*. A collection of *templates* (program constructs) is generated from the syntax of a programming language. Such a template exists for all but the simplest statements. For example, the template for a conditional statement in PL/CS is:

```
IF (condition)
      THEN statement
      ELSE statement
```

The places to be filled are called *placeholders*. To construct a program, the user fills the placeholder `object` with templates and *phrases* (arbitrary sequences of typed symbols). There are commands to generate, delete, move, etc. templates. Templates cannot be modified, and each placeholder designates the syntactic class of permissible insertions. So after every operation, the editor guarantees a syntactically correct program.

In the next sections we describe a syntax-directed editor for the language *B*. (Certain aspects of the editor can also be found in the editors mentioned above.) The design objectives for the editor are discussed in section 2 and the more concrete consequences in sections 3 and 4. Section 5 gives an impression of the *B* system.

More about editors can be found in [Meyr82a] (a tutorial describing interactive editors) and in [Meyr82b] (a survey of particular editors). A first impression of the language *B* can be obtained from [Geur82]; a specification of the whole language and some thoughts on the *B* system from [Meer81]. An overall picture of the language and the system, in Dutch, can be found in [Geur83].

A pilot implementation of the *B* editor is described in [Harm83] and is running on a VAX 11/780 under UNIX[1)].

2. ABOUT THE EDITOR DESIGN.

The typical user of languages such as *B*, does not have the time nor the ambition to learn a complicated language [Geur76]. Such a user needs a **simple** editor to construct her programs. This simplicity should not be reflected by the lack of (powerful) commands, but by a simple *conceptual model*, the abstract framework on which the editor is based [Meyr82a].

Each user who gets acquainted with an editor forms a model of the editor. This (individual) model is based on experience with the use of the editor, the documentation, the structure of the editor commands, etc. If the conceptual model of the editor is complicated, the user has more problems to form an **accurate** individual model. The more other aspects of the editor (user interface, documentation) support the ideas on which the editor is based, the more it helps the user. So our design objectives for the editor are:

- a simple conceptual model,
- a simple user interface,
- power.

[1)] UNIX is a trademark of Bell-Laboratories.

2.1. Simple conceptual model.

The editor is designed to be used with a terminal as input and output device. It is basically a screen-editor: an updated version of the program is represented on the screen; corrections are made directly on this representation.

So most of the time editing is a local action. Somewhere in the program the user adds or deletes characters. The exact place is marked by the cursor, a means to select a specific character position on the terminal screen. On this place the next character will appear when the user strikes a key. The cursor is not very suitable to select objects other than characters. For example, to select a word, one usually brings the cursor on the first character of the word. But now, it is not deducible from the screen which meaning the cursor has: selecting a word or the first character of this word. To select any object, we use a *focus* (of attention), which selects a specific **area** on the screen. The focus is made visible for the user by displaying the text within the focus in another manner (e.g. underlining, inverse video) than the surrounded text.

Edit commands, such as DELETE and INSERT, work on a specific object. To delete a line or word, the object on which the command works has to be that line or word. To define an object as domain of an edit command, the user has to bring the focus on the object. *Pointing commands* are available to bring the focus on the desired object. The combining of edit and pointing commands to form one edit operation can also be found in several other editors. In the *B* editor, however, these classes of commands are strictly separated and are presented as independent commands. Without the chance of changing her program by accident, the user can first bring the focus on the object she wants to delete, check it by the representation of the focus on the screen, and then complete the edit operation.

2.2. Simple user interface.

During the design of an editor, the set of commands tends to become very large (VI [Joy83] has around 70 commands, and EMACS even more than 250). It is tempting to collect all the nice commands of other editors and implement them in the new one. However, a large set of commands is difficult for the user to manage: it is almost impossible to remember all the commands. We have very carefully selected which commands are absolutely necessary to work comfortably with the editor. This has resulted in a small set of commands.

Because of the modeless environment, commands have to be distinguishable from text to be inserted. A common practice is to use the CONTROL key to give other keys a special meaning. It is preferable to implement, at least the most commonly used commands, with (properly labeled) function keys: the user sees at a glance which commands she can use.

The terminal screen is used to give maximum feedback to the user: every key-stroke will have some visible result on the screen; the user always sees what is going on. It also means that the editor is very helpful: in almost every situation any key-stroke results in some (useful) action of the editor (for example, if the user ignores the suggestion mechanism and just types her program, the editor does not protest). In other situations, the editor gives an understandable message (for example, if the user strikes the CONTROL key and another key which has no specific meaning).

The editor is very robust, in the sense that the user cannot easily damage a program irrevocably. We have achieved this by two means:

- edit commands only bear on the area which is currently selected by the focus; the surrounding (not selected) text is never affected by the command,
- an *undo mechanism* will reverse former actions of the editor.

2.3. Power.

Every programming language has (syntax) rules, which describe what a valid program is in that language. A program is usually composed from a limited set of language constructs. Feeding the editor with knowledge about the syntax rules gives the possibility of assisting the user with the construction of a program. If a user starts to type a construct, it is possible to deduce in an early stage which construct she is typing. As soon as the editor recognizes this construct, it suggests the completion of the construct (*suggestion mechanism*). Another aspect of the knowledge with syntax rules is the possibility to signal violations against those rules. The editor thus gives immediate feedback to the user about the correctness of her program.

The division of the set of commands in pointing and edit commands gives the user a powerful mechanism to perform edit actions: a large set of structures can be edited by a small set of edit commands.

3. DESCRIPTION OF THE EDITOR.

3.1. Focus.

It is assumed that there is some way to impose a tree structure on a program. Such a tree consists of objects, which have (hierarchical) relations with each other. Consider a particular object in the tree; it usually has a *parent* and one or more *children*. Children of a parent are called the *siblings* of each other. The focus is on such an object in the tree. The user needs commands to bring the focus on other objects in the tree. There are four directions for moving the focus: up (to a parent), down (to a child), left and right (to a sibling). Because of the effect on the screen these movements are called WIDEN, NARROW, PREVIOUS and NEXT respectively.

The choice of how to structure a program has a great influence on the ultimate appearance of the focus movements to the user. Objects and structure should be chosen in such way that an object is a contiguous piece of text and that moving through the tree can be expressed by actions like: going to a bigger/smaller object, or going to just after/before the object. (Though the focus movements are heavily based on the tree structure, we do not require the user to think in the same manner.)

The representation of the tree contains redundant information. For example the command

```
PUT letter IN box
```

contains the redundant object `IN`. Its function is to separate the parameters `letter` and `box` and to increase the readability of the command. This holds in the same way for the (redundant) blanks in the command. The user cannot change these objects (to preserve the integrity of the *B* program). So the user is also forbidden to focus on such objects. It is, however, possible that the object lies within the focus (for example, if the focus is on the command `PUT letter IN box`).

3.2. Suggestion mechanism.

Consider the following conversation (the focus is represented by underlining the objects):

editor:	□
user:	W
editor:	WHILE □ : □
user:	R
editor:	WRITE □
user:	I
editor:	WRITE □

The conversation above is based on the following criteria:

- As soon as the user types a character, the editor gives a suggestion suitable for that particular place.
- As long as the user types characters which do not conflict with the suggestion, the editor insists on its suggestion.
- When the user leaves the word on which the suggestion was based, she thereby consolidates the suggestion. (This criterion expresses the special status of a suggestion.)

There are several ways to leave the word on which the suggestion was based (and to consolidate the suggestion). The most simple one is to type the characters of the word. After a blank the editor consolidates the suggestion and moves the focus to the next character or hole. If the user strikes the ACCEPT key, she consolidates the suggestion and the focus is brought on the first hole to the right. The user can also use the various pointing commands to move the focus. For example, to reach the hole in the example above, the user can type:

editor:	WRITE □
user:	TE BLANK
or:	ACCEPT
or:	WIDEN NEXT
or:	NEXT NEXT
editor:	WRITE □

There is no need to restrict the suggestion mechanism to newly created lines:

editor:	WRITE □
user:	DELETE
editor:	WHILE □ : □

If the holes in a construct are already filled, the suggestion mechanism follows the same conventions as DELETE [4.6].

3.3. Insertion mechanism.

There have to be conventions about the place where the characters will appear and about what happens with the text the current focus is on, when the user types characters. The behavior of the editor depends on the sort of object the focus is on. We distinguish three kinds of objects:

- a hole,
- a character,
- other objects.

A *hole* is an empty node in the tree, some place to be filled in. There are two kinds of holes: holes on places where some text is obligatory and holes on places where it is optional. An optional hole will vanish when the focus is not on it any more (so there is at most one of them). If the focus is on a hole and the user types a character, that character will fill up the hole, a new hole is created just

to the right of the old one and the focus is now on the new hole.

If the focus is on a character and the user types a new one, then the character will be replaced by the new one and the focus will now be on the character just to its right. If there is no character there, a hole is created and the focus is brought on it.

If the focus points at another object and the user types some character it will not be inserted: the editor gives some appropriate message. We decided to protect the user in this case against accidental destruction of large objects by a single key-stroke.

The above mechanism is a compromise between automatic overstrike of existing objects (i.e., if the focus is on a character) and the need to give an explicit delete command to overstrike objects (i.e., if the focus is on other objects). Another possibility we have considered is to allow characters to be inserted only in a hole. To correct a simple typing error will then be usually more complicated (deletion of the wrong character; creation of a hole; typing the new character).

3.4. Pretty printing.

Pretty printing a piece of a *B* program takes into account the syntax restrictions, the habits of many program writers and our own individual thoughts on what a pretty program is. Because of the complex (and not fully stated) definition of a pretty program, the only way to guarantee a reasonable pretty printer is to parameterise it: every construct in *B* points to a description of its layout. The description can be similar to the primitive language as found in [Medi82] in which one can express actions like newline, tab, spaces, etc. When we get a strange feeling about some of the layout decisions while experimenting with the editor, we can easily change the description and try again.

Another problem arises when a program has to be represented for which the size of the screen is too small. The program has to be *condensed.* (If the focus is on the whole program, representing just a piece of it is not acceptable.) There are several strategies to condense a text:

- deleting deeply nested commands,
- representing just the first and last command(s) of a block,
- replacing blocks by a (user-supplied) text,
- eliding pieces of a command.

The question is whether condensation requires any explicit user-control. The Cornell Program Synthesizer [Arch82] demands a user-supplied comment to represent a program construct. The user has a condense/expand command to control the condensation. Mikelsons [Mike81] describes a condensation algorithm, which does not need user involvement. Because of the intensive communication between the user and the editor, it is usually known what part of the text is of interest to the user. This knowledge can be used to make a choice between the several condensation strategies.

We prefer the last solution for representing text on the screen (it simplifies the user interface), but a pilot implementation should teach us about the properties of such an automatic system.

4. COMMANDS.

In this section we illustrate the editor commands on the following piece of *B*:

```
HOW'TO MAIL letter:
    SHARE box
    PUT letter IN box
    SEND
```

The structure of the text is stated in the rules:

how-to-unit:	header AND body
header:	SEVERAL keywords OR parameters
body:	SEVERAL commands
command:	SEVERAL keywords OR parameters
keyword:	SEVERAL characters
parameter:	SEVERAL characters

4.1. WIDEN.

WIDEN brings the focus on the parent of the object the current focus is on. For example, if the current focus is on the keyword PUT, a WIDEN will focus on the whole command.

```
editor:     PUT letter IN box
user:       WIDEN
editor:     PUT letter IN box
```

There are circumstances in which the representation of the focus would not change. In such situations the focus is brought on the successive first parent whose representation differs from the present object (to ensure a visible result of the key-stroke). For example, if the current focus is on the **keyword** SEND, after a WIDEN the focus is not on the parent which is the **command** SEND, but on the body of the how-to-unit.

```
editor:     HOW'TO MAIL letter:
                SHARE box
                PUT letter IN box
                SEND
user:       WIDEN
editor:     HOW'TO MAIL letter:
                SHARE box
                PUT letter IN box
                SEND
```

4.2. NARROW.

NARROW brings the focus on the leftmost child of the current focus. For example, if the current focus is on the command PUT letter IN box, a NARROW will focus on PUT.

```
editor:     PUT letter IN box
user:       NARROW
editor:     PUT letter IN box
```

If the focus is on several siblings, the focus is brought on the leftmost sibling of that group.

```
editor:     PUT letter IN box
user:       NARROW
editor:     PUT letter IN box
```

If the focus is now on a object that has just one child, it is not deducible from the screen whether the focus is on the object, or on its child. In those cases the focus is brought on the child.

4.3. NEXT.

NEXT brings the focus on the nearest right sibling of the object the current focus is on. For example, if the focus is on the keyword PUT, a NEXT will focus on the parameter `letter`.

```
editor:     PUT letter IN box
user:       NEXT
editor:     PUT letter IN box
```

If there is no right sibling, the first parent is found that has a right sibling and the focus is brought on that sibling. For example, if the focus is on the character `x` in `box`, the first parent that has a right sibling is the command `PUT letter IN box`, and the focus is brought on its sibling: the command `SEND`.

```
editor:     PUT letter IN box
            SEND
user:       NEXT
editor:     PUT letter IN box
            SEND
```

4.4. PREVIOUS.

PREVIOUS behaves similarly to NEXT, bringing the focus to the left, instead of to the right.

4.5. EXTEND.

If the focus is on an object with many siblings, it would be clumsy if the focus could not be brought on several of such objects (for example, to copy a group of commands). We allow to focus on several adjacent siblings of the object. We need some mechanism to extend the focus. There are two obvious candidates:

- A FIX key and an UNFIX key: when pressing the FIX key the focus will grow with every focus movement until the UNFIX key is pressed.
- An EXTEND-LEFT and EXTEND-RIGHT key: when pressing either key the focus will enclose the nearest (left or right) sibling of the object the current focus is on.

To avoid an extra mode (fixed or unfixed) we chose the EXTEND-LEFT/-RIGHT keys. We do not permit focusing on objects other than a sequence of adjacent siblings. For example,

```
editor:     PUT letter IN box
user:       EXTEND-LEFT
no move:    PUT letter IN box
user:       EXTEND-RIGHT
editor:     PUT letter IN box
```

4.6. DELETE.

If the user strikes the DELETE key the object the focus is on will vanish (and will be saved in a buffer). If the object was optional, the focus is brought on the sibling of the object. If it was a non-optional object or it has no sibling, a hole is created and the focus is brought on that hole.

```
editor:     PUT letter IN box
user:       DELETE
editor:     PUT □ IN box
```

If the user deletes an object, leaving a syntactical incorrect construct, a hole is created for that object. When the user fills the hole with another object, the editor tries to fix the construct. For example, if the user deletes the (first) keyword from a command, all the keywords are replaced by holes, and the parameters of the command are saved. If the user fills the first hole by another command, as many holes as possible are filled by the keywords of the command (possibly leaving too many parameters).

editor:	PUT letter IN box
user:	DELETE
editor:	□ letter □ box
user:	I
editor:	INSERT letter IN box

4.7. ADD.

When the user wants to add some text, she needs some means of creating a hole. The behavior of the editor depends on whether the focus is on a hole or not. If the focus is on a non-empty object, ADD creates a hole as right sibling of the object, and the focus is brought on it. In certain cases it is not deducible from the screen what kind of object the focus is on. For example, if the focus is on the keyword SEND an ADD brings the focus on a hole next to the keyword. A subsequent ADD brings the focus on a hole under the keyword, to allow the user to insert a new command. Thus, if the focus is on a hole that has no right sibling, ADD behaves as if the focus is on the **parent** of the hole.

editor:	SEND
user:	ADD
editor:	SEND □
user:	ADD
editor:	SEND □

4.8. INSERT.

INSERT behaves similarly, but instead of a hole to the right of the focus, a hole to the left will be created.

4.9. COPY.

The COPY key moves objects to and from a special buffer. If the focus is on a non-empty object and the user strikes the COPY key, then that object will be copied to the buffer. If the focus is on a hole, then the object in the buffer will be moved to that hole. In [5.2] we will discuss some means of editing the buffer and of putting more than one object in it.

4.10. UNDO.

UNDO brings the state of the editor, the screen, etc. back in the state before the last key-stroke was given. Subsequent UNDO's will undo former key-strokes. (The UNDO itself is not counted as key-stroke.)

4.11. ACCEPT.

ACCEPT accepts a suggestion of the editor. The focus is brought on the first hole.

5. CONCLUSIONS.

5.1. The *B* editor.

We have tried to design an editor that behaves as a typical user would expect. Conflicts arose when in certain situations we tended to forbid specific commands (to preserve a simple conceptual model), but also wanted to serve the user. In these conflicts we counted the last aspect heavily. For example, if a user strikes WIDEN when the focus is on a how-to-unit (the root of the tree) there should follow some expected action, like focusing on a list of all the how-to-units.

However, the typical user does not exist. There will always be some diversity between the sort of users and their expectations. So we have also taken more or less arbitrary decisions. In such cases we have built in the possibility of easily changing those parts of the editor.

The suggestion mechanism proves to be a very valuable mechanism. The user can use it without needing to learn a specific set of commands. The possibility of ignoring the suggestions lets her choose how much help she wants. The mechanism can be extended to suggest possible completions of (user defined) variables and how-to-units. It may also be valuable to structure expressions in such a way that the editor can suggest open and close parentheses.

The focus movements are not optimal. For example, the use of NEXT gives in certain situations big changes in the size of the focus. It may be possible to improve it by changing the (tree) structure of a *B* program. Another possibility is to change the definitions of the commands in such a way that they preserves the size of the focus in such situations.

The present deletion mechanism is a combination of two strategies:

- in any case a hole is created after the user strikes the DELETE key (it avoids accidental deletion of several adjacent objects and facilitates the insertion of new objects),
- a hole is only created after an explicit request. If the user strikes DELETE an implicit NEXT brings the focus on another object (which facilitates successive deletion of objects).

If it appears to be confusing for the user to work with the combination of the strategies, we will have to choose one of them.

However, all these improvements should be carefully considered after we have had more experience with the use of the editor in the pilot implementations.

5.2. The *B* system.

Both the editor and the *B* interpreter are tools to modify objects (programs and variables). They both have their own command language to give the user the possibility of expressing the actions on the objects. The structure of the command language gives an indication of the expected use of the tool, more than the kind of objects does.

The editor is based on intensive communication with the user; it should be used in an interactive environment, where the user gets after each command immediate information upon which the next commands should based. The editor is not designed to behave as a programming language: there are no structures to group or repeat commands. The *B* interpreter, on the other hand is based on the algorithmic aspect of modifying variables. The user has at her disposal constructs to express an algorithm in an easy and structured way.

Both tools can handle most of the tasks the user directs to the system. For example, the interface between the system and the outside world can be implemented with variables which have a special

meaning. Those variables can easily be used in programs to react upon or to control the events outside the system (like a variable `temperature` that has the value of the temperature of the room and a variable `heating` that controls the capacity of some heater, by putting a value in it).

Other tasks within the system are more suited to be handled by the editor. Consider a screen divided into several *windows*, where every window has its own focus and its own meaning. Some of the windows can be used to edit programs, others can show a list of running processes. If a user focuses on such a process and strikes the DELETE key, the process will be killed. Narrowing the focus will give more information in the window about the process. Another possibility is to give a window the meaning of the edit buffer. If a user edits a program and strikes the COPY key, she can edit the buffer or collect objects from other programs, before she copies the objects back to her program.

It is possible to use the editor as a front-end for many existing tools in other systems [Fras80]. The advantages are clear: the user does not have to master several command languages; editor commands can be bound in a natural way to actions that control the *B* system.

Acknowledgements.

The design of the editor is based on a lot of ideas and suggestions of the members of the *B* group. The discussions with and the support of Hans van Vliet were indispensable to adapt those ideas in the ultimate design. I would also like to thank Paul Klint and Lambert Meertens for their support and suggestions while writing the different versions of this report and Steven Pemberton for his help in reaching a readable result.

REFERENCES.

[Arch82]: Archer Jr, J., and R. Conway, "Display Condensation of Program Text", *IEEE Transactions on Software Engineering, volume SE-8, number 5, pages 526-529,* September 1982.

[Burk80]: Burkhart, H., and J. Nievergelt, "Structure-Oriented Editors", Institut für Informatik, ETH, Zürich, 1980.

[Fras80]: Fraser, C.W., "A Generalized Text Editor", *Communications of the ACM, volume 23, number 3, pages 154-158,* March 1980.

[Geur76]: Geurts, L.J.M., and L.G.L.T. Meertens, "Designing a beginners programming language", *in: New Directions in Programming Languages 1975 (S.A. Schuman, editor), pages 1-18,* Rocquencourt, 1976.

[Geur82]: Geurts, L.J.M., "An Overview of the *B* Programming Language or *B* without Tears", *SIGPLAN Notices, volume 17, number 12, pages 49-58,* December 1982.

[Geur83]: Geurts, L.J.M., "Ontwerp van een Programmeeromgeving voor een Personal Computer", *in: Colloquium programmeeromgevingen (J. Heering and P. Klint, editors), pages 39-51,* Mathematisch Centrum, Amsterdam, 1983.

[Grun82]: Grune, D., "bx - *B* interpreter with Emacs editor", *internal document,* Mathematisch Centrum, Amsterdam, 1982.

[Harm83]: Harmelen, F.A.H. van, "On the implementation of an editor for the *B* programming language", Mathematisch Centrum, Amsterdam, 1983.

[Iron72]: Irons, E.T., and F.M. Djorup, "A CRT Editing System", *Communications of the ACM, volume 15, number 1, pages 16-20,* January 1972.

[Joy83]: Joy, W., "An Introduction to Display Editing with Vi", (revised by M. Horton), *UNIX Programmer's Manual, volume 2c, document 41,* August 1983.

[Medi82]: Medina-Mora, R., "Syntax-Directed Editing: Towards Integrated Programming Environments", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, March 1982.

[Meer81]: Meertens, L.G.L.T., "Draft Proposal for the *B* Programming Language - Semi-Formal Definition", Mathematisch Centrum, Amsterdam, 1981.

[Meyr82a]: Meyrowitz, N., and A. van Dam, "Interactive Editing Systems: Part I", *Computing Surveys, volume 14, number 3, pages 321-352,* September 1982.

[Meyr82b]: Meyrowitz, N., and A. van Dam, "Interactive Editing Systems: Part II", *Computing Surveys, volume 14, number 3, pages 353-415,* September 1982.

[Mike81]: Mikelsons, M., "Prettyprinting in an interactive Programming Environment", *in: Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, SIGPLAN Notices, volume 16, number 6, pages 108-116,* June 1981.

[Stal81]: Stallman, R.M., "EMACS The Extensible, Customizable Self-Documenting Display Editor", *in: Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, SIGPLAN Notices, volume 16, number 6, pages 147-156,* June 1981.

[Teit81]: Teitelbaum, T., and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM, volume 24, number 9, pages 563-573,* September 1981.

[Wood81]: Wood, S.R., "Z - The 95% Program Editor", *in: Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, SIGPLAN Notices, volume 16, number 6, pages 1-7,* June 1981.

APPENDIX.

In this appendix we describe the commands from section 4 in the language *B*. The descriptions are based on the following 'primitive' commands, predicates and functions.

commands:

`BRING focus ON object`	brings the focus on the object.
`COPY'BUFFER'TO'HOLE`	copies the contents of the buffer to the hole.
`COPY'FOCUS'TO'BUFFER`	copies the object the focus is on to the buffer.
`CREATE'LEFT'SIBLING`	creates a hole as left sibling of the object the current focus is on.
`CREATE'RIGHT'SIBLING`	creates a hole as right sibling of the object the current focus is on.
`DEL object`	deletes the object.
`SOME'OTHER'ACTION`	performs some not-specified action.

predicates:

`focus'has'child`	succeeds if the object the focus is on has a child.
`focus'has'left'sibling`	succeeds if the object the focus is on has a left sibling.
`focus'has'parent`	succeeds if the object the focus is on has a parent.
`focus'has'right'sibling`	succeeds if the object the focus is on has a right sibling.
`focus'is'multiple`	succeeds if the focus is on more than one object.
`focus'is'optional`	succeeds if the object the focus is on is optional.
`focus'on'hole`	succeeds if the focus is on a hole.

functions:

`left'child'of'focus`	returns the left child of the object the focus is on.
`left'of'focus`	returns the leftmost object within the focus.
`left'sibling'of'focus`	returns the left sibling of the object the focus is on.
`object'in'focus`	returns the object the focus is on.
`object1 with object2`	returns the concatenation of the objects.
`parent'of'focus`	returns the parent of the object the focus is on.
`right'child'of'focus`	returns the right child of the object the focus is on.
`right'sibling'of'focus`	returns the right sibling of object the focus is on.

editor commands:

```
HOW'TO UNDO:
    \ The last key-stroke is undone

HOW'TO ACCEPT:
    \ The suggestion is accepted
```

```
HOW'TO WIDEN:
    SHARE focus
    WHILE focus'has'no'sibling AND focus'has'parent:
        BRING focus ON parent'of'focus
    SELECT:
        focus'has'parent:
            BRING focus ON parent'of'focus
        ELSE:
            SOME'OTHER'ACTION
focus'has'no'sibling:
    REPORT NOT (focus'has'left'sibling OR focus'has'right'sibling)

HOW'TO NARROW:
    SHARE focus
    SELECT:
        focus'is'multiple:
            BRING focus ON left'of'focus
        focus'has'child:
            BRING focus ON left'child'of'focus
            WHILE focus'has'no'sibling AND focus'has'child:
                BRING focus ON left'child'of'focus
        ELSE:
            SOME'OTHER'ACTION
focus'has'no'sibling:
    REPORT NOT (focus'has'left'sibling OR focus'has'right'sibling)

HOW'TO NEXT:
    SHARE focus
    WHILE focus'has'no'right'sibling AND focus'has'parent:
        BRING focus ON parent'of'focus
    SELECT:
        focus'has'right'sibling:
            BRING focus ON right'sibling'of'focus
        ELSE:
            SOME'OTHER'ACTION
focus'has'no'right'sibling:
    REPORT NOT focus'has'right'sibling
```

```
HOW'TO PREVIOUS:
    SHARE focus
    WHILE focus'has'no'left'sibling AND focus'has'parent:
        BRING focus ON parent'of'focus
    SELECT:
        focus'has'left'sibling:
            BRING focus ON left'sibling'of'focus
        ELSE:
            SOME'OTHER'ACTION
focus'has'no'left'sibling:
    REPORT NOT focus'has'left'sibling

HOW'TO EXTEND'LEFT:
    SHARE focus
    SELECT:
        focus'has'left'sibling:
            BRING focus ON left'sibling'of'focus with object'in'focus
        ELSE:
            SOME'OTHER'ACTION

HOW'TO EXTEND'RIGHT:
    SHARE focus
    SELECT:
        focus'has'right'sibling:
            BRING focus ON object'in'focus with right'sibling'of'focus
        ELSE:
            SOME'OTHER'ACTION

HOW'TO DELETE:
    SHARE focus
    SELECT:
        focus'on'hole:
            SOME'OTHER'ACTION
        focus'is'not'optional OR focus'has'no'sibling:
            CREATE'RIGHT'SIBLING
            BRING focus ON right'sibling'of'focus
            DEL left'sibling'of'focus
        focus'has'right'sibling:
            BRING focus ON right'sibling'of'focus
            DEL left'sibling'of'focus
        focus'has'left'sibling:
            BRING focus ON left'sibling'of'focus
            DEL right'sibling'of'focus
focus'is'not'optional:
    REPORT NOT focus'is'optional
focus'has'no'sibling:
    REPORT NOT (focus'has'left'sibling OR focus'has'right'sibling)
```

```
HOW'TO ADD:
    SHARE focus
    IF focus'on'hole:
        IF focus'has'no'right'sibling AND focus'has'parent:
            SELECT:
                focus'is'optional:
                    BRING focus ON parent'of'focus
                    DEL right'child'of'focus
                ELSE:
                    BRING focus ON parent'of'focus
            CREATE'RIGHT'SIBLING
            BRING focus ON right'sibling'of'focus
            QUIT
        SOME'OTHER'ACTION
        QUIT
    CREATE'RIGHT'SIBLING
    BRING focus ON right'sibling'of'focus
focus'has'no'right'sibling:
    REPORT NOT focus'has'right'sibling

HOW'TO INSERT:
    SHARE focus
    IF focus'on'hole:
        IF focus'has'no'left'sibling AND focus'has'parent:
            SELECT:
                focus'is'optional:
                    BRING focus ON parent'of'focus
                    DEL left'child'of'focus
                ELSE:
                    BRING focus ON parent'of'focus
            CREATE'LEFT'SIBLING
            BRING focus ON left'sibling'of'focus
            QUIT
        SOME'OTHER'ACTION
        QUIT
    CREATE'LEFT'SIBLING
    BRING focus ON left'sibling'of'focus
focus'has'no'left'sibling:
    REPORT NOT focus'has'left'sibling

HOW'TO COPY:
    SHARE focus
    SELECT:
        focus'on'hole:
            COPY'BUFFER'TO'HOLE
        ELSE:
            COPY'FOCUS'TO'BUFFER
```

69D23
69D44