stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 220/83 FEBRUARI

F.A.H. VAN HARMELEN

ON THE IMPLEMENTATION OF AN EDITOR FOR THE *B* PROGRAMMING LANGUAGE

kruislaan 413 1098 SJ amsterdam

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

The Mathematical Centre, founded 11th 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.).

1982 CR. Categories: D2.3, D2.6, D2.2

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

by

Frank A.H. van Harmelen (*)

ABSTRACT

As a first step towards an integrated environment for the *B* programming language a pilot implementation of a syntax-directed editor has been made. This report describes the (technical) design and implementation of the editor. Improvements on the present implementation are discussed.

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

(*) This work was done while the author was at the Mathematical Centre.
Author's current address is: Facultaire Vakgroep Informatica, Universiteit van Amsterdam, Plantage Muidergracht 6, 1018 TV Amsterdam.

1. INTRODUCTION

Since 1975 the design of a programming language for beginners has been formulated at the Mathematical Centre in Amsterdam [GEUR76]. This language was meant to be used on personal computers. From the start a dedicated environment was envisaged as well. The present report deals with one of the key components of such an environment, viz. a syntax-directed editor. The first design, called *B*0, ultimately resulted in the definition of the programming language *B* (still a provisional name) [MEER81]. A more palatable treatment is given in [GEUR82]. A pilot implementation under UNIX (*) and running on a VAX 11/780 is available.
Important criteria in the design of the language were simplicity and suitability for conversational use. While simplicity is paramount for the language proper, the latter criterion plays a key role in the design of the environment as well:

- The system ought to follow the "utterances" of the user closely, and react immediately whenever appropriate, rather than keeping its reaction till the final moment of the analysis when the user is done.
- It ought to display one "face" to the user, rather than a variety of faces of subsystems on different levels, such as an editor, a file system, a compiler, each with its own conventions and reactions and hardly aware of each other' s existence.
- It should not leave the user uncertain about whose turn it is and should prompt whenever a reaction is required.

In [GEUR83] the following requirements for the *B*-environment are formulated:

- Because the ultimate goal of the *B*-project is a computer which has *B* as its basic (!) language, the language and the environment will have to be inseparable: the environment is completely dedicated to programming in *B*.
- The interfaces of various system functions must differ as little as possible. Functions that can be used in a certain mode of the system must work in a comparable way in other modes, whenever this is reasonable.
- The *B*-language is the command-language of the *B*-environment. If, for some reason, some system functions would not be assimilated in the language they should have a radically different format.
- Just as in the language, only a small number of constructs will be allowed: the power of the environment should come from the coherence of the building blocks, not from the sheer number of them.

In such an environment, a general editor plays an important role. The editor is the instrument that allows the user to create various objects, change them, put them aside, and bring them back into attention again.

Because of the tentative nature of many of the ideas it was generally felt that an implementation of part of the environment was needed first. With this implementation we could test the decisions that were already made, and try various solutions to problems that were as yet undecided.
The language editor is a part of the editor that can easily be isolated from the other parts of the environment, therefore it was decided to start with a pilot-implementation of it.

The main objective of this report is to describe the (technical) design and implementation of this pilot implementation (sections 3 and 4). Section 2 gives a short overview of the user-interface of the editor. A comprehensive discussion of it is given in [NIEN83]. The present implementation is an intermediate result. The code itself can be improved considerably, and the editor can be extended with additional features. Sections 5 and 6 elaborate on this.

(*) Unix is a trademark of Bell-Laboratories

2. INFORMAL DESCRIPTION OF THE EDITOR

Instead of the usual cursor, which traditionally points to a one-character position on the screen, the *B*-editor uses a so-called focus: the focus is that part of the text which is currently the area of attention; it will be displayed in a special way (e.g. inverse video or a different type-font or color).
It is assumed that there is some "natural" way to impose a syntactical structure on a text. For a given text, this results in some parse-tree. The focus always contains a smaller or larger syntactical unit from this parse-tree. The user can move the focus around in the text by calling one of a few functions. In the next sections these functions will be called 'special editor functions'. Roughly speaking their definition is as follows:

widen	Move one level up in the tree.
narrow	Move one level down in the tree.
next	Move the focus to the right brother node of the current focus.
previous	Move the focus to the left brother node of the current focus.
extend-left(right)	Extend the focus with the left (right) brother node of the current focus.

When a user is typing characters, these characters will normally overstrike the text that was already in the document. To add new text to a document, the user has to call one of the functions 'add' or 'insert'. These functions create (before or after the current focus, respectively) something that is called a 'hole'. Text that is typed while the focus is on such a hole is added to the document. A 'delete' function exists for deleting the text that is currently in the focus.

Because the editor knows about the general structure of the objects that are being edited, it can assist the user in various ways. One way of assisting the user is to display the edited document in a suitable layout.
Another way of assisting is to take over actions from the user. As soon as the editor has an idea of what the user might mean, it tries to finish what the user was doing: for instance, in a *B*-program, after the user has typed a 'W' the editor might suggest: 'WHILE .. :'. If this is what the user meant, she may confirm the suggestion and continue with the condition-part. If she meant to type 'WRITE' instead, she just types 'R', after which the editor tries a second guess: 'WRITE ..'. Numerous variations on this theme are possible.

3. DESIGN

The previous section provides us with an informal description of the functions the editor should perform. The next stage in the development of the system is to design the architecture of it. We shall separate the presentation of our design into two steps. First we shall state the goals we want to achieve with our design. Second we shall present our design in a rather informal way. For each module a description will be given of the functions it performs and the interfaces to other modules.

3.1. Design objectives

At the time the system was designed, we already knew that several properties of the system would be subject to heavy changes. We therefore wanted to design our system in such a way that parts of it can be changed easily without conflicting with other parts. With this "design for change" we aim at the following goals:

Flexibility of the special functions

We want to be able to change our decisions about the behaviour of these functions without being worried by the consequences this will have for other parts of the program.

Separation of internal and external representation

We want to be able to independently change the way in which the edited objects are stored internally and the way in which they are presented to the user.

Syntax independence

As explained in [NIEN83], it may very well be possible that the editor will not only be used to edit *B*-programs. In the future, when the complete *B*-environment comes to life, the editor will also be used to operate on process-lists, complex data structures, session-records etc. [GEUR83]. These objects will obey syntaxes that are different from that of the *B*-language (although the syntaxes will have some properties in common). Furthermore the *B*-language may change in small ways. It should be reasonably easy to adjust the editor to different syntaxes.

Device independence

When it becomes an integral part of the *B*-system the editor will be distributed to various people and places. In all these places different terminals are installed, with different I/O-capabilities. It should be reasonably easy to adjust the editor to a new device.

3.2. Informal description of the design

In order to understand the design of the system the reader should keep in mind fig. (I). The basic pattern in the picture is a cycle: starting from the terminal it goes from input-module to parse-tree, virtual-screen, screen-handler back to the terminal-screen. The syntax-description/suggestion-handling part is hooked onto this cycle.

The parse-tree module

Internally, the system needs a representation of the object that is to be edited. The object will be stored in a parse-tree. The parse-tree module provides an abstract data-type for this tree, and thereby hides its representation. The special editor functions are implemented in this module as well.
The parse-tree module contains routines to provide for a linearized representation of the contents of the tree. This is the information that eventually has to appear on the screen. But at this stage of processing we don't want to be concerned with problems like maximum line length, screen height etc. To postpone all these problems to a later stage the output of the parse-tree module is written onto something called the virtual screen.

The virtual screen

The virtual screen behaves like an idealized screen: it has an unbounded number of lines, each of unbounded length. The virtual screen module provides functions for writing on the screen and reading from it; again the details of the data structure are hidden.

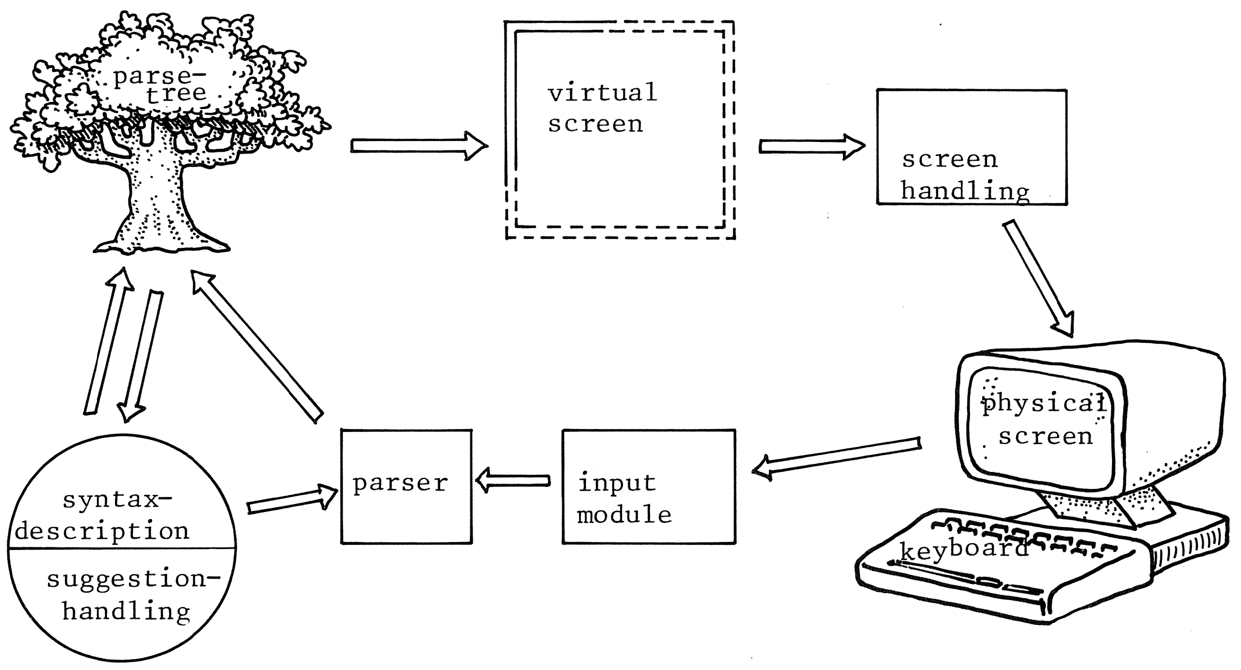

fig. (I)

The screen-handling module

The next module is again one step closer to the terminal. This module translates the information on the virtual screen to the ultimate representation on the terminal. It contains a set of data structures, each representing a "mini-screen", called a window. The task of the screen-module is threefold:

1) Translating the information on the virtual screen to information on the windows. This process also involves line-truncating and fitting large texts in small windows.
2) Organizing the windows on the screen, which includes decisions about where to put windows, and about which windows are to be shown when the screen becomes too crowded, etc.
3) Updating the screen. The module has to know about the capabilities (or lack thereof) of the terminal, and how to use them in order to update the screen in an efficient way.

The input module

When the user is thus provided with new, updated information about the state of the objects she is editing, she probably wants to perform some operations on it. These operations are entered via the keyboard, or with a device like a mouse or a touchpen. The input-module interprets these commands. A command is either a call of a special function, or it is some character that is supposed to become part of the edited object, or it is erroneous in which case an error message should be generated.

The parser module

Before the input character can really become part of the edited object, the system has to check that this action doesn't change the object into something that is not syntactically correct. This job is done by the parser module. It starts to determine the syntactic context of the current focus (some subtree of the parse tree). Then it checks if the input character is correct in the current context. If so, appropriate actions are taken to update the internal structure of the edited object.

The syntax-description and suggestion-handling modules

Some of the modules discussed up to now depend on the syntax of the edited object: the parse tree is a representation of an object that obeys this syntax and the parser calculates the current context in terms of the syntax. Rather than building knowledge about the syntax in the various modules, we decided to incorporate this in a separate module.
On top of this the parser-module takes "appropriate actions" to update the internal data structure. These actions will be syntax-dependent too, and they are provided by the suggestion-handling module.
It is not clear to us whether this module is so intimately connected with the syntax-description module that is has to be replaced with every new syntax description, or whether it only uses limited information from the syntax-description module, in which case the suggestion-handling module would really be a separate module like all the others. For the moment we will treat the suggestion-handling module as a separate module, but we are aware that practice might change this view.

This concludes our tour along the various parts of the system. Now let us go back to the design goals and check to which degree our design meets its goals.

Flexibility of the special functions

These functions together form a rather independent layer in the parse-tree module. Changing these functions is confined to that layer, though this can only be properly assessed after the discussion in section 4.1 .

Separation of internal and external representation

The internal representation is determined by the parse-tree module, while the external presentation is determined by the screen-handler. These two communicate with each other by means of the virtual screen. This provides an excellent separation of concerns. As long as we don't change their interface, viz. the virtual screen, we may change either of them without conflicting with the other.

Syntax independence

The syntax-dependent information in the system is entirely concentrated in the syntax-description module. Replacing the syntax means replacing this module.

Device independence

The only parts of the editor that know about devices are the input-module and, more importantly, the screen-handler. When we want our system to run on a new type of terminal we have to adjust these modules.

4. THE IMPLEMENTATION

The discussion on the implementation will be quite detailed. The main reason for this is that this section is meant to be a technical guide for everyone who wants to take a look into the source code of the system. As will be discussed in sections 5 and 6 a lot of changes to the system can already be foreseen. The programmer who is going to make these changes will want to read this section.

The section is divided into 7 paragraphs, each of which is concerned with one module. We will explain the general organization of each module and the meaning of its routines, discuss data structures and algorithms, explain, and where necessary criticize, choices made in the current implementation. In some places alternative solutions will be discussed.

The editor is entirely implemented in C [KERN78]. The main reasons for choosing C as the implementation language were dictated by the *B*-project: the *B*-interpreter is written in C, *B* is developed in an environment where C is a strongly supported language (a good interface with the operating system, a debugger, a dedicated editor, a prettyprinter, standard software packages), and people in the *B*-project think of C as "an almost tolerable language".

Before starting to examine the modules, a few words about the notation employed may be helpful. The terms 'procedure', 'routine' and 'function' are used interchangeably. Variable names are written in italics: *result.* Procedure names in the text are indicated by parentheses, as in *narrow().*

4.1. The parse-tree module

As mentioned in secton 3.2, the main goal of this module is to provide a data structure in which we can store the edited object. The module consists of three layers: The lowest layer provides an abstract data type to represent the tree. A second layer provides higher level routines to manipulate the tree. These routines are defined in terms of the lower layer. In the third layer of the module the special functions of the editor are defined in terms of the routines provided by the two other layers. The key to understanding this module is a good understanding of the key data structures. The general layout of the tree is as follows: to store the tree in fig. (I) we build something like fig. (II).

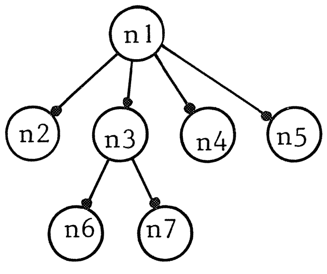

fig. (I)

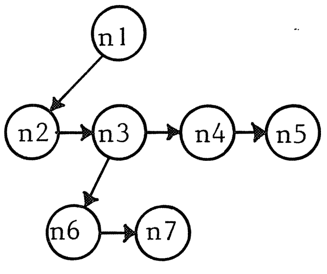

fig. (II)

This means that each node is connected with at most two other nodes: its left-most son and its right brother. By not connecting a father to each of his sons we circumvent the problems that arise with the variable number of pointers that a node would contain in such a construction.
In our construction, only pointers in a downward and rightward direction occur. But often we want to make our way through the tree in an opposite direction: from right to left, or from son to father. Various solutions exist for this problem. We considered the Schorr-Waite algorithm [SCHO67], but this does not work in our situation: the algorithm presupposes a "current position" in the tree from where the next step is to be made in some direction. But while using our parse-tree various places in the tree might each correspond to a "current position". For instance, if we have multiple windows

viewing the same tree, each window has its own focus which will be one of the current positions. This causes the algorithm to break down. Several alternatives exist, such as a stack algorithm, or an algorithm which uses a doubly linked tree. We chose the latter, so for each arrow in fig. (II), we also store an arrow in its opposite direction. The tree now looks like fig. (III).

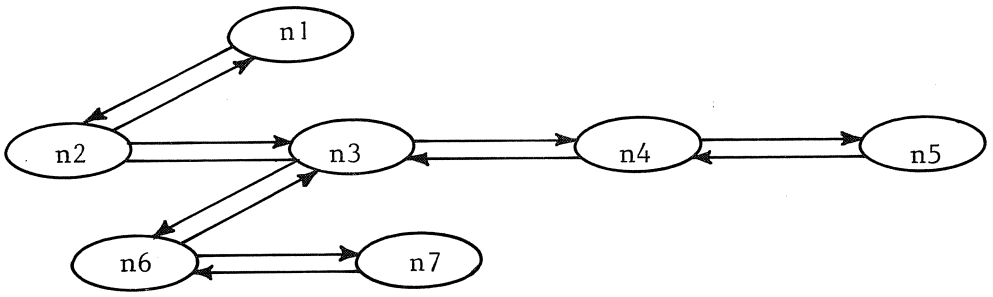

fig. (III)

This implementation uses a lot of extra storage, but it gives ease of programming and readability in turn.

The actual data type which represents a node is a structured data type called NODE. We use the struct-facility of C, which is something like the record in Pascal.
The fields of a NODE come in two kinds: bookkeeping information (the fields TYPE, FLAGS and INDEX) and links to the rest of the tree (the fields PRENODE, BROTHER and SON). The TYPE field is used to denote the meaning of the NODE in terms of the syntax. For instance, if the syntax is that of *B*-programs, the node-type could have values like COMPOUND, HOW TO-UNIT, etc. Note that these values are fetched from the syntax-description module; the parse-tree module itself is syntax-independent. The FLAGS field is used to assign certain properties to a NODE. We will see examples of this later on. The INDEX field is an index in a table that contains pretty-printing information. This table will be discussed below. (See also section 4.6.)
The BROTHER field is pointing to the right brother of the NODE. If the NODE doesn't have a right brother its value is NULLNODE. The PRENODE field is used for double linking: it contains the counterpart of the BROTHER- or SON-pointer. We distinguish the two possibilities by using the procedure *father_relation()* from the syntax-module.

We use the convention that all information in the tree is located in the leaves. This means that a NODE contains either a character-string (if it is a leaf) or it has a son (if it is not a leaf), but not both. The union-construction of C is used to combine these two possibilities; the CONTENTS field contains either a pointer to a son of the NODE or a pointer to a character-string. We remember which possibility is used by setting the flags POINTER or WORD in the FLAGS field.
A NODE is empty if it has neither a son nor a word in its CONTENTS field. This property is detected by the routine *is_empty_node()*. An empty node will be represented in a special way when the tree is printed.

The routines that manipulate the tree or return attributes of some node are *node_type()*, *right_brother()*, *left_brother()*, *left_son()*, *right_son()*, *father()*, *word()*, *flags()*, *assign_son()*, *assign_right_brother()*, *assign_left_brother()*, *set_on_flags()*, *set_off_flags()*, *is_empty_node()*, *newnode()* and *assign_word()*. These are all straightforward and will not be discussed in detail. Notice that these routines completely hide implementation details of the type NODE. If one wants to change the data structure one only has to change the body of these routines.

The routines and data-types discussed so far form the abstract data type this module provides. They form the lowest layer of this module. A few higher level routines are *add_subtree()*, *insert_subtree()*, *replace_subtree()*, and *delete_subtree()*.

The only routine of the standard tree-handling package that remains to be discussed is *read_tree()*. This routine (or better: the routine *read_tree_body()* which does all the work) visits the nodes of the tree and prints their contents (if any). To do this job properly, it needs prettyprinting information and instructions about where to add extra syntactic elements. Obviously, this information is syntax dependent, so it has to be provided by the syntax-description module. For this purpose the syntax-description module provides a routine *layout()* that takes an INDEX as its argument and returns prettyprinting instructions. They tell the read-routine in which order the nodes are to be visited and where to insert extra syntactic elements. The instructions are to be interpreted as follows:

```
<str>    print <str>.
N        print a newline.
+        increase indentation level.
-        decrease indentation level.
*        print N tabs, where N is the indentation level.
b        remove the last character that was printed.
$        PUT succeed IN flag
         SELECT:
             the current node has a son which is the NULLNODE:
                 PUT fail IN flag
             the current node has a son:
                 process this son
             the current node contains character-information:
                 print the current node
             ELSE:
                 print the empty-node representation
#        PUT succeed IN flag
         SELECT:
             the current node has a right brother which is not the NULLNODE:
                 process this right brother
             ELSE:
                 PUT fail IN flag
%        process the father of the current node.
[..]     WHILE flag <> fail:
             repeat the instructions between '[ and '].
.        end of instruction string.
```

The routine *read_tree_body()* now proceeds as follows. First, the layout instruction is fetched. This string can be interpreted as a program in the mini-language described above. The variable *layoutpattern* is used as a program counter to step from instruction to instruction. (The calls concerning the focus will be discussed later.)
The mini-language in this routine was borrowed from [MEDI82].

Read_tree_body() embodies the design principle that the external representation should be easily modifiable. If we want a different layout for our output we just rewrite the layout instructions in the syntax-description module.

Up till now only standard tree operations have been discussed. For the special functions of the editor we need a representation of the focus, functions to manipulate it and functions that perform

the user calls ADD, INSERT and DELETE.
The definition of the focus in [NIEN83] implies that the elements in the focus have to be adjacent sons of the same father. This means that, in terms of the parse-tree, the focus can be described by two nodes, its leftmost node and its rightmost node, called *leftfocus* and *rightfocus* respectively.
N.B.: When a node is part of the focus, the complete subtree below it is also part of the focus. For instance, if in the tree of fig. (I) of this section *leftfocus* is n3 and *rightfocus* is n4, the complete focus consists of {n3, n6, n7, n4}.
The routines that provide access to the focus are *set_focus()*, *left_focus()* and *right_focus()*.
The functions that the editor provides to the outside world to handle the focus (viz. *narrow()*, *widen()*, *previous()*, *next()*, *extend_left()* and *extend_right()*) are almost literal C-translations of the algorithms stated in [NIEN83]. This also applies to the functions *add()*, *insert()* and *delete()*.
Note that these functions are implemented as separate subroutines; they do not call each other and only make use of the standard tree-handling facilities. This provides the flexibility of the special functions as mentioned in section 3.1 .

Obviously, the focus has to be somehow represented in the output of *read_tree()*. The routine *read_tree_body()* takes care of this. Each time a node is visited the routine checks if it is part of the focus. If so, the generated output is marked in a special way by the the routine *mask_char()*. In this way other modules are able to recognize these parts of the representation.

4.2. The virtual screen

In section 3.2 it was stated that this module should behave like an idealized screen: a quarter-plane extending indefinitely in width and length with the top, leftmost character being the origin.
The basic element in the implementation of this concept is what we call a VS_LINE (= virtual screen line). These VS_LINE' s are chained together in a doubly-linked list to form the virtual screen. The first field of a VS_LINE contains the information that is written on the line. Conceptually this is a string. However, the virual screen only contains linearized representations of parts of the parse-tree. This means that the edited object would be stored twice; once in the parse-tree and once on the virtual screen. To prevent this we just incorporate in a VS_LINE a pointer to the part of the tree whose representation is on that line. If somebody from outside the module calls for the contents of a VS_LINE, it is generated on the spot by calling the routine *read_tree()* in the parse-tree module. Notice that this decision is invisible from outside the virtual-screen module. The routine *read_line()* returns the string that is written in a VS_LINE just as if it were always available.

The routine *vs_construct()* is an initialization routine. It generates a virtual screen whose contents correspond to the tree pointed to by the argument.

4.3. The screen handling module

This module translates the information on the virtual screen to the ultimate representation on the terminal screen of the user. The current version of this module is very rudimentary. As will be discussed below, a major part of the task of this module is not performed properly.
The module contains an intermediate data structure, the frame. A frame is to be seen as an internal representation of a mini-screen with its header line. The frame structure is defined in terms of a data structure provided by the CURSES software package [ARNO80]. This data structure is called window. A window is a rectangular area that can be filled with text. A frame consists of an overall_window which is divided into a header_window and an inner_window: the header_window contains the title of the frame and the inner_window is the body of the frame that contains the text.

As explained in section 3.2 the screen-handling module has three tasks:

Translating the information on the virtual screen to information on the frames

This task is performed by a set of routines that operate on the frame-structure: *createframe()* generates a new frame; this frame can be filled with text using *write_on frame()*. While writing this frame it may turn out that lines are too long, or that there are more lines than can be written on the frame. Lines that are too long will have to be truncated in a meaningfull way. The current implementation does not recognize this problem and uses the default solution provided by CURSES: it just starts writing on the next line when the current line is full.
It may also happen that a text has more lines than can be fit on a frame. This problem isn't handled properly either. Lines that don't fit on the frame are just not printed. Possible solutions will be discussed in chapter 6.

Organizing the frames on the screen

When the various frames are filled with text the module has to decide how to place them on the screen: where should the frames be placed, and which frames should be shown when the screen becomes too crowded? The current implementation does not handle these problems. A possible solution is discussed in chapter 6.

Updating the terminal screen

The internal representation that results from the previous stages still has to be printed on the terminal screen. What we need is an algorithm to update the screen efficiently, and knowledge about the capabilities of the terminal in order to perform this updating.
These are both provided by the CURSES-package. The routine *refresh()* compares the state of the frames with the information currently on the screen and calculates a set of instructions to update the screen. To do this it needs knowledge about the capabilities of the terminal: does the terminal have an addressable cursor? Can it insert/delete lines? Can it write in inverse video? Does it have scrolling capabilities? What are the timing conditions of the terminal, etc. Furthermore it has to know how to activate these capabilities: Which escape sequences have to be sent to the terminal to delete a line, etc.
For this purpose the CURSES-package uses a terminal database, called TERMCAP [JOY81]. This database provides a language in which capabilities of a terminal can be described. The CURSES-package fetches the name of the terminal type it runs on from the operating-system, looks it up in the TERMCAP database, adjusts its algorithm and sends the correct sequences to the terminal.
Obiously this is a very powerfull mechanism: if we want the editor to run on a new kind of terminal we only have to add its description to the database. It however means that this basic layer is very much UNIX-dependent.

4.4. The input module

The smallest module in the system is the input module. Its task is to translate input from the user to some device-independent internal code. Correct input comes in two ways: it is either a call of a special function like ADD or NEXT, or it is a character that is supposed to become part of the edited object. It is the task of the input module to distinguish between these two kinds of input. On incorrect input the module should generate some error message.
Character input will probably come from an ordinary terminal keyboard. It can be handled using the standard input calls. Special function calls on the other hand can be implemented in many different ways, depending on the facilities provided by the actual terminal device. They might be implemented as keyboard character strikes preceded by an escape sequence. More likely, they will be implemented using special function keys on the keyboard. But the character sequence that results from such a special function key differs from terminal to terminal.

The routine *interpret_char()* translates these sequences to a terminal independent internal code. The current implementation can be adapted to different terminals by changing this routine. In the future this module might be organized in the same way as the screen-handling module, i.e. using a terminal description database.

4.5. The parser module

The internal tree structure will have to be adjusted according to the input from the user. This task obviously depends on the current area of attention. For instance, in a *B*-program, changing a HOW TO-keyword and changing the name of variable will have different effects.
The parser starts by determining this "current area of attention" in terms of the syntax. This is done by the routine *level()*. Starting at the focus, the path leading to the root of the parse-tree is determined. While climbing to the root of the tree the routine concatenates the node-type of the current level with the node-types of all the levels visited so far. The number that is returned is a representation of the path from root to focus. The parser checks whether the input character is meaningfull with respect to this path, updates the data-structure and calls for actions from the suggestion-handling module.

N.B.: The current implementation of the parser is an ad hoc solution. The syntax of *B* is simply built into the parser. A better solution would be to incorporate the necessary knowledge into the syntax-description module, for instance by means of attribute-grammars.

4.6. The syntax-description module

One of the design goals in section 3 was the possibility to adapt the editor to a new syntax with minimal effort. This is achieved by concentrating the knowledge about the syntax in the syntax-description module.
Each time we want to add a syntax to the set of syntaxes, an additional version of this module has to be supplied. Therefore it is important that it be simple so that it can be easily updated by somebody who does not have intimate knowledge of the other parts of the system.

The module consists of three parts: a set of constants, a set of routines and a table containing prettyprinting information.
The constants make up the various types that can be assigned to NODE's in the parse-tree. For a given syntax, the only constraint on the values of these constants is that they are all different and not equal to zero.

The routines in this module must be supplied for each new syntax. These routines must have unique names. A special routine is *init_.. syntax()*, where '..' stands for something like an acronym of the name of the syntax. (In the current version the syntax for *B*-programs is used and unique names for all routines were made by prefixing them with '*B*'.) The sytem uses standard names for all these functions. The initialization routine binds the standard names to the syntax-dependent routines. In this way the editor can switch to a new syntax at run time. By calling one of the routines *init_.. syntax()* a specific binding is established, and thereby the syntax described by these functios becomes the current one. This is a very powerfull mechanism which gives the editor as many faces as there are syntax descriptions available.
The following functions are to be supplied in a syntax-description:

- *line_node(node)*: returns TRUE if the screen-representation of the argument occupies exactly one line. Otherwise it returns FALSE.
- *increase_indent_type(type)*: returns TRUE if printing a node with TYPE-field *type* causes the indentation to be increased by one.
- *decrease_indent_type(type)* is the counterpart of *increase_indent_type()*.

● *compulsory_node(node)*: returns TRUE if the presence of *node* in the parse-tree is mandatory.
● *rbr_allowed(node)*: returns TRUE if *node* can have a right brother in the tree.
● *lbr_allowed(node)* is the counterpart of *rbr_allowed()*.
● *rbr_type(node)* returns the type of a right brother, if a right brother is possible.
● *lbr_type(node)* is the counterpart of *rbr_type()*.
● *father_relation(node1, node2)*: returns TRUE if *node1* can be the father of *node2*.
● *init_ .. syntax()* establishes the link between the syntax-dependent functions and the standard names that are used by the system.
● *init_layout_table()*: In the description of the routine *read_tree()* (section 4.1) it was stated that the syntax-description module should provide node-type dependent prettyprinting information. For this purpose, the syntax description module maintains a table where, for each syntax, prettyprinting information is stored. This table is called the *layout_table*. The function *layout()* accepts an index in this table as its argument and returns the corresponding prettyprinting code. The function *extend_layout_table()* adds a new pair (index, prettyprinting code) to the table. When a syntax is initiated this *layout_table* has to be set up. This job is done by the routine *init_layout_table()*.

4.7. The suggestion-handling module

In [NIEN83] it is stated that the editor should suggest particular texts to the user as soon as it has an idea of what the user might want. The suggestion-handling module provides these suggestions. It is clear that these suggestions depend on the syntax that is currently in use: in a *B*-program a 'P' might yield 'PUT .. IN ..' as a suggestion. While editing a process list the same 'P' might very well stand for 'PROCESS NUMBER ..'.

The input to the module is a string from the parser module. This string is used to generate an appropriate suggestion. With 'suggestion', we mean a parse-tree-like structure corresponding to the text that is being suggested. For instance in the former case above, the delivered suggestion would be something like fig. (I).

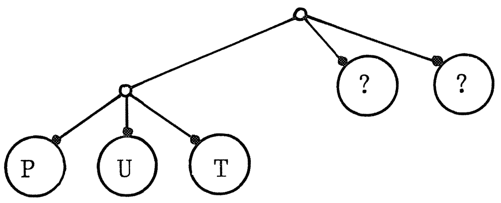

fig. (I)

The module performs its job in two stages. First, the input-string is matched against a set of language-elements. In the *B*-language this set would contain, among others, keywords like PUT, WHILE and IF, user-defined commands, names of variables that are currently in use, etc. These words are stored in a table called *string_table*. An element of this table consists of three fields: the first field is the language element, the second field is its associated prettyprinting information (or at least an index in the table that contains this information), and the last field is a pointer to a routine that knows about the suggestion tree.
In the second stage, the suggestion-tree is generated. When the input-string matches one of the strings in the string_table, the associated routine is called and its result is returned as the output of the module. If the input-string matches none of the string in the table a default value is delivered.

The matching is done using a first-fit policy. By a clever ordering of the language elements, often used constructs will be suggested first. Also, the *string_table* may be adjusted dynamically, in order to improve the behaviour of the editor.

5. IMPROVEMENTS ON THE IMPLEMENTATION

In this section the current implementation of the editor will be assessed. We shall not try to quantify the merits (or lack thereof) of the implementation, but only informally discuss its major shortcomings. Our purpose is to warn future users against weaknesses of the system and to show parts of the system that need attention in the near future.
The discussion will be guided by the following rather vague criteria: robustness, portability and efficiency.

5.1. Robustness

We will distinguish three kinds of robustness:

- Robustness concerning the user interface. How susceptible is the system to errors or inconsistent behaviour of the user.
- Robustness concerning the system-interface. How susceptible is the program to errors made by the system it runs on. What happens when this system breaks down.
- Internal robustness. How susceptible is the program to internal inconsistencies and programming errors.

User interface

In various situations, actions can be undertaken by the user that are meaningless, or simply wrong. The current version of the system detects this and responds by doing nothing. This is obviously not correct. The user should be informed about her mistakes. Unfortunately, the detection of inconsistent and erroneous user actions is spread throughout the system.
A different situation is that of a user doing something she actually didn' t want to do. Undo-facilities to prevent damage still have to be implemented. Various possibilities are described in [NIEN83].

Machine interface

It is unacceptable that a major or minor machine crash results in loss of work done by the user. The editor should prevent this. Various methods are available:

1) Maintaining backup copies of the world as it was before the edit session began. This also provides a (primitive) means of protecting a user against damage done by her own actions.
2) Checkpointing: every so many key-strokes the complete state of the world is saved in a backup copy.
3) Maintaining a key-stroke history. This history can be used to "replay" the lost edit-session.

None of these techniques is as yet employed by the current implementation. When something goes wrong, simply everything is lost.

Internal robustness

The current version of the program does not check for internal inconsistencies. Functions that are only called from within the system do not check for wrong parameter values, inconsistent constructions, etc. What we need is the following: conceptually, every routine should check for error conditions. If they occur, special routines should be called that let the errors bubble upwards to the level where they can be handled. Small errors that are easy to repair should be handled by the system itself. Serious, irreparable errors should evoke routines that halt the system, inform the user and make a sensible dump for debugging purposes.

5.2. Portability

By portability we mean the ability of the program to run in a different environment. This means a different computer, a different operating system, a different compiler, linker, loader etc. We will not discuss the effects of different I/O-devices, since this topic has already been handled in section 4.3 . While developing the program, care has been taken to make no system calls. An exception to this is the screen updating package as used by the screen handling module. This package will only work under the UNIX-operating system. The implementation uses only parts of C that are guaranteed by the language definition in [KERN78]. This is guaranteed by requiring that the program passes the lint program-checker without causing warnings [JOHN78].

5.3. Efficiency

In this section, a few of the major inefficiencies of the current implementation will be mentioned. The discussion does not pretend to be complete. Eventually, some benchmarking of the system might be necessary to find additional troublespots.

Tree data structure

As discussed in section 4.1, the implementation of the tree data structure is rather inefficient; the tree is doubly-linked. An alternative would be a stack algorithm, in which each variable pointing into the tree maintains a history of the nodes it visited while traversing the tree. Removing on of the links would save us 4 of the 16 bytes that a NODE uses on the VAX 11/780. Such an optimization can be made locally in the parse-tree module.

Tree storage

In the current implementation, every character that is part of the edited object is stored in a single leaf of the tree. This means that for a character occupying 1 byte on the VAX 11/780, we use a NODE of 16 bytes. But explicit storage of each character in a single node is not always necessary. The complete tree structure is only needed when we have to calculate focus movements at character-level or when we have to make changes to the leafs of the tree. These actions only occur in the subtree that is currently in focus. In the other parts of the tree, we can very well store a set of characters in one node. For instance, instead of storing 'PUT aap IN noot' in the tree of fig. (I) we can store it in the tree of fig. (II).

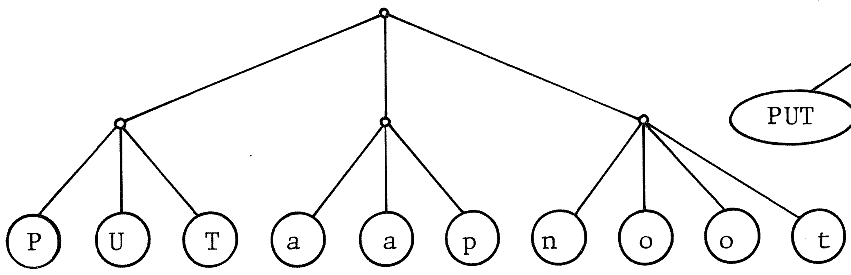

fig. (I)

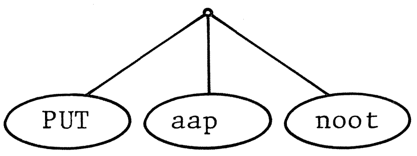

fig. (II)

In such an implementation the tree would normally be in configuration (II) and only when necessary a subtree would be "unpacked" to configuration (I). Since each new level in a tree consumes about (N-1) times the space of all the higher levels together, where N is the average number of sons per node, the optimized tree will use about 1/N'th of the space currently in use.

String allocation

Various routines generate character strings to store their intermediate results or their final output. The obvious example is the routine *read_tree()*. While executing this routine, new output is added to the result string by using one of the routines *addchar()* or *string_cat()*. Each time one of these routines is called the entire string is reallocated: to create a final result of length N we use a total number of 0.5*N*(N+1) memory cells. This number can be considerably reduced by allocating strings in blocks of, say, 10 characters. In our application such an algorithm would use on the average one third of the storage that is currently being used.

Updating the virtual screen

In the current implementation, no means exist to update the virtual screen, except for rebuilding it completely. As a consequence, the virtual screen is completely built from scratch each time the editor has to update the terminal screen of the user (which is after each key-stroke). Obviously, this leads to an enormous amount of extra work. What we need is a set of routines which compare the current state of the virtual screen with the current state of the parse tree, and subsequently compute an efficient way to let the virtual screen correspond to the tree again.

Updating windows

What is said about updating the virtual screen is equally true for updating a window. Currently only routines exist that read from and write on a window. A "screen-updating algorithm" for windows is needed.

6. EXTENSIONS TO THE CURRENT IMPLEMENTATION

In this section we mention extensions that need to be made to the current implementation in the near future. This enumeration is not meant to be complete. Numerous other features might, and probably will, be implemented.

Line truncating

In the process of translating text from the virtual screen to a window, long lines may have to be truncated. We want these truncations to occur at specific places in the text. For instance, in

```
IF a>0 AND b>0:
```

truncation just before one of the '>'s looks awkward, while truncation just before or after 'AND' seems quite acceptable. Because these breakpoints depend on the structure of the text, they have to be determined in the parse-tree module. But there we don't know where truncation will be necessary. A solution to this problem would be to mark the various places where line-truncation would be acceptable while making a linear representation of the tree in the routine *read_tree()*. These places can be defined in the layout-information strings in the syntax-module. In the screen-module we can look for these marked places and perform truncation if necessary.

Text condensing

The problems that arise when not all the lines fit on the window are more difficult to handle. A solution would be to only print the parse tree down to a certain level. Lower levels could then be represented by a short-hand expression. For instance, the tree in fig. (I) would not be printed as

```
WHILE a>0:
    PUT a-1 IN a
    WRITE a
```

but as

```
WHILE a>0:
    <body>
```

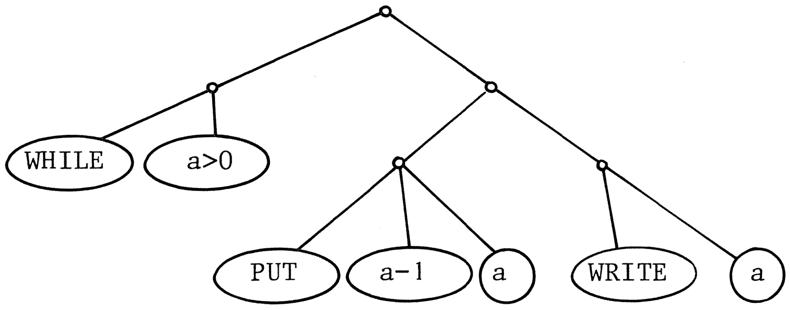

fig. (I)

Another possibility, used in the Cornell Program Synthesizer [TEIT81] is to replace parts of the program by comment-lines provided by the user. It is interesting to note that this solution has resemblance with the refinement facility of the *B*-language. Practical experience is needed to make further decisions about this problem.

Screen handling

As mentioned in section 4.3 two problems arise: where should windows be placed on the screen, and, if the screen becomes too crowded, which windows should be shown. A possible solution would be to associate a default screen position to each window and a priority which determines if it is important enough to overlap other windows. The user should be able to adjust the values for positions (moving the windows around on the screen) and priorities (putting other windows on top).

Integrating the *B* system and editor

As discussed earlier the editor internally stores *B*-programs as a parse-tree. The compiler does something likewise. In a future implementation these two tree-structures could very well be integrated, so that compiler and editor (and in the future other tools of the environment) share the same data structure. As mentioned in [NIEN83] there is no reason to use the editor for constructing *B*-programs, and not for editing the commands that are to be executed by the *B*-system immediately. Somebody who types in a FOR-loop to be executed immediately should be able to handle it in the same way in which she handles *B*-programs: to move the focus around, to add, to delete, etc, until the loop is ready to be executed.

Nice features

Numerous features can be thought of that will be useful to implement as integrated parts of the editor. We will only mention a few:
History list. An (editable) list of previously given commands is available. This list can be used to

replay parts of the session.
Delete stack. Objects that are deleted are not just thrown away, but are stored somewhere by the editor, for instance in the form of a stack, the most recently deleted object being the top of the stack. These stack elements can be fetched by the user to use them somewhere else in the program.
Editing plain text. Facilities to edit plain text will have to be provided. These include knowledge about words, sentences and paragraphs, spelling-correction, etc.
Copy facilities. It should be possible to copy the contents of the focus to a destination somewhere else in the same, or possibly another, window.
Search facilities. It should be possible for a user to describe a context in a certain way (for instance with a regular expression), and to have the system mark all the places that satisfy the description.
On-line help facilities. The user should always be able to ask for help. The answers she will get might vary from just a listing of all possible commands to a sophisticated database that gives information depending on the current state of the system.
Profiling. It might be possible for a user to make up a profile in which she tells the editor which behaviour she prefers most. This profile might contain information about wordy or short error messages, preferred layout of the screen, extensive or shorthand help facilities, etc.

7. CONCLUSION

We described the design of a pilot implementation for a syntax-directed editor for *B*. With this design we aimed at flexibility of various features of the editor:
● In order to be able to easily adjust both the external and internal representation an interface (a virtual screen) was designed to separate the two. The internal representation is implemented as an abstract data type. The external representation is determined by a set of adjustable, syntax-dependent "mini-programs". A database is used to describe terminal-dependent information.
● The syntax of the edited objects can be replaced. This syntax is described by a set of replaceable routines. The editor can switch to another syntax (another set of syntax-description routines) at run time.

The current implementation is an intermediate result; the program is both incomplete and inefficient. Because of the "separation of concerns" paradigm employed in the design of the editor, we are confident that a number of improvements can be made without to great an effort. Also, future users will have to assess the functionality of the user interface. We believe that the present framework easily allows for experimentation in this.

Acknowledgements

I am very grateful to Hans van Vliet. His ideas, his countless critical remarks and his helpful comments were a great support.

References

[ARNO80]: Arnold, C.R.C., "Screen updating and cursor movement optimization: A library package", Dep. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif., Sept. 1980.

[GEUR76]: Geurts, L.J.M., and Meertens, L.G.L.T., "Designing a beginners programming language", New Directions in Programming Languages 1975, pp. 1-18, (S.A. Schuman ed.), Rocquencourt, 1976.

[GEUR82]: Geurts, L.J.M., "The *B* programming language, or *B* without tears", SIGPLAN Notices, **17**, (12), (Dec. 1982), pp. 49-58.

[GEUR83]: Geurts, L.J.M., "Ontwerp van een programmeeromgeving voor een personal computer", in: Heering, J. and Klint, P. (ed), Syllabus colloquium programmeeromgevingen, To appear, Mathematisch Centrum, 1983.

[JOHN78]: Johnson, S.C., "Lint, a C Program Checker", Comp. Science Tech. Rep. No. 65, Bell Laboratories, Murray Hill, New Yersey (1978).

[JOY81]: Joy, W. and Horton, M., "TERMCAP", in: UNIX programmer's manual, 7th ed., Berkeley Release 4.1, Dep. of Electrical Engineering and Computer Science, Univ. of California, Berkeley Calif., June 1981.

[KERN78]: Kernighan, B.W. and Ritchie, D.M., "The C programming language", New Jersey: Prentice Hall 1978.

[MEDI82]: Medina-Mora, R., "Syntax-Directed Editing: Towards Intergrated 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, 1981.

[NIEN83]: Nienhuis, A.J.C., "On the design of an editor for the *B* programming language", Masters Thesis, Univ. Amsterdam, 1983, to appear.

[SCHO67]: Schorr, H. and Waite, W., "An efficient machine-independent procedure for garbage collection in various list structures", Comm. ACM, **10**, (8), (Aug. '67), pp. 501-506.

[TEIT81]: Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-directed Programming Environment", Comm. ACM, **24**, (9), (Sept. 1981), pp. 563-573.

69 D23
69 D26
69 D22