Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.A.M. van de Graaf

Towards a specification of the *B* programming environment

Department of Computer Science Report CS-R8408 May

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

TOWARDS A SPECIFICATION OF THE B PROGRAMMING ENVIRONMENT

JEROEN VAN DE GRAAF
Centre for Mathematics and Computer Science, Amsterdam

A primary aim in the design of a dedicated environment for the B programming language has been to make the environment easy to use and to learn. We tried to achieve this by tuning the features to the expected use, by integrating the language and the environment and by allowing only a limited number of powerful, coherent concepts which have in varying context a similar meaning. This report contains an informal description and a tentative specification of the environment.

1980 MATHEMATICS SUBJECT CLASSIFICATION: 68B20.
1982 CR CATEGORIES: D.2.6, D.3.4.
KEY WORDS & PHRASES: programming environments, personal computing, B.

Report CS-R8408
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. Introduction

In the design of the programming language *B* (a provisional name), intended for personal computing, the ease of use has always been put first. And right from the start it has been the intention not only to design a programming language, but to embed *B* in a programming environment fully dedicated to *B*.

This report is the first formalization of the ideas on the *B* environment present among the members of the *B* group. It describes and specifies the elements of the environment and their semantics. An explicit form or syntax is not always given. A structured editor and a reasonable display algorithm are assumed to exist. Problems that might arise in the implementation of these ideas are fully disregarded, as has been the case in the design of the language too. Since the *B* environment is still in its development phase, some questions are still open and some solutions are rather arbitrary. A prototype of the environment will decide these issues.

This report presupposes some knowledge of the language *B*. An informal introduction to *B* can be found in [1] or in [2]. The latter is written in Dutch and gives some preliminary views on the environment as well. A formal treatment of *B* is given in the *Draft proposal for the* B *programming language*, which we will refer to as DP.

In section 2 we make some general observations about traditional environments (operating systems). Section 3 is an informal introduction to the *B* environment. Section 4 treats the design objectives and principles. Section 5 contains some preliminaries to the specification, and section 6 contains the specification proper. Since these last two sections are intended to be self-contained, some repetition is unavoidable. The reader who is not interested in too much detail might prefer to skip the last section.

This report is a result of many discussions among the members of the *B* group. I would like to thank Steven Pemberton and Guido van Rossum for proofreading and correcting an earlier version, and Lambert Meertens for all his remarks, suggestions and patience.

2. Programming environments in general

An *environment* can be defined as a set of tools for preparing and maintaining documents. From now on we reserve the word *document* for the "things" an environment acts on. A specific sub-class of documents are *programs*, and the development of correct and executable programs is an important task of a *programming* environment. We can distinguish the following global functions in a programming environment:

— displaying documents
— editing documents
— managing the screen and its division in windows
— controlling the existence of documents
— history and undo
— process management
— searching texts
— tracing and debugging.

Conventional systems are usually equipped with one (or several) different system programs for each function, e.g. with an editor, an interpreter or compiler, a file system etc. Though this is understandable for reasons of modularity, it has led to many different modes, each with its own syntax and conventions, all apparently ignorant of each other's existence. In the *B* environment the

functions assist each other. For instance, the editor has some knowledge of the *B* grammar and tries to give suggestions.

Furthermore we see in conventional operating systems a rather artificial distinction between the job control language, files (which are permanent) and programs on the one hand, and the programming language, local variables (which exist only temporarily) and procedures on the other. In [3] (in Dutch) Jan Heering gives a detailed analysis of this phenomenon. One design objective for the *B* environment is to integrate the job control language and *B*, which means that functions for the environment are either added to the language, or have a radically different form.

3. An informal walk through the B environment

We reserve the word *documents* for those objects of the *B* environment whose contents can be inspected and manipulated by the user. The *B* editor has a focus, a region of attention, and edit actions affect either the contents or the position of the focus.

From the *B* language proper we inherit the following two types of documents: the *targets* (the formal term for variables in *B*) and the *units* (the equivalent of procedures and functions.

It is clear that an interpreter needs a part of (virtual) computer memory where the names and contents of the documents can be stored, otherwise the result of a computation in one line is lost in the next. This virtual memory is called the *workspace* or *permanent environment*. All documents in the permanent environment remain in existence, even when the computer is switched off. This makes a file system superfluous, and the traditional distinction between files and variables disappears. Because the user often has several projects she is working on, she can create several, different workspaces. During execution a unit creates its own, *local environment* which is different from the permanent environment. A unit can only change the contents of a permanent target if this target is a HOW'TO unit parameter or a SHAREd target.

The user can look at documents by having them displayed on the screen. The system always shows a document as it *is*, not some copy of it. This implies that if a process changes the contents of a document while it is displayed on the screen, the screen representation is changed immediately. If the user wants to change the contents of a displayed document, she can do so by editing it. This means that the user changes the screen representation of the document, and the system adjusts the contents accordingly.

For instance, suppose that the target `shopping'list` is displayed:

```
shopping'list = {'bread'; 'milk'}
```

Now the user can change the contents of `shopping'list` by editing the right hand side of the equation, e.g. she can add `; 'tea'` after `'milk'` She can also rename the document by editing the left hand side or delete `shopping'list` by widening the focus to the entire line and pressing DELETE.

We prefer to use a `<name> = <contents>` representation of documents rather than a window displaying just the contents with the name of the document in a foot line. In this way changing the name of a target is much easier. Note that in a unit the name is contained in its first line.

More formally we can say that there exists a *system invariant*, which states that the screen representation of a document should be equal to its contents. If this invariant is disturbed by changes on either side, the system adjusts the other side so that the equality holds again. Invariants play a central role in the philosophy of the *B* environment.

Each workspace has an INDEX'OF'DOCUMENTS, which gives a list of the targets and units present in the workspace. In our example this might look like:

```
phi = 0.6180339887498948
shopping'list = {'bread'; 'milk'}
YIELD strip sentence:  ...
```

The INDEX'OF'DOCUMENTS can be regarded as a long distance view of all the documents present in the workspace; documents occupying more than one line are compressed. Editing a line of the INDEX'OF'DOCUMENTS affects the underlying document. For instance all the edit actions on `shopping'list` shown above, are also permitted here. Again an invariant is involved: the contents of the INDEX'OF'DOCUMENTS should be equal to the one line representations of the objects in the workspace. And again a disturbance of the invariant must be restored by the system.

The way to select a document in order to display it is done by pressing the VISIT key. This is interpreted as: give me the document whose name is in the focus. So the user selects the desired document with the focus positioning functions and presses VISIT. Especially if the current document is an index this offers a convenient, menu-like way of selecting documents.

The system keeps a stack of the visited documents. By means of the EXIT key the user leaves the active document and returns to the previous active document; repetition is allowed. REVISIT is the reverse of EXIT, so these two functions permit the user to jump between two or more documents.

If the name of the desired document is nowhere on the screen, the user can press SPECIFY'DOCUMENT'TO'BE'VISITED, type the name of the desired document and press VISIT.

Another important document is the SESSION'RECORD, one for each workspace. In response to the prompt, `>>>`, the user types her commands (here shown in italics) on this document, for instance:

```
...
>>> PUT 'call jack' IN forgotten
```

Now the *B* interpreter will make sure that the contents of `forgotten` becomes `'call jack'`. A new prompt appears when the interpreter has completed the execution of the previous command. Conceptually we can think of the SESSION'RECORD as a ladder: the *B* commands are the spaces between the rungs and the *B* system is on some rung, waiting for the next command to be typed. Here we also have an invariant: the state of the system should be equal to the result of all the commands written on the SESSION'RECORD. The user gives a command by editing the SESSION'RECORD, and execution of such a command is nothing else than restoring the invariant.

This invariant can be used intensively for correcting and undoing erroneous commands. For instance, if the user continues her session with

```
...
>>> PUT 'call jack' IN forgotten
>>> WRITE forgoten
```

the system will signal an error (unless another target `forgoten` had already been defined). Now the user positions her focus back to the offending *B* command and inserts a `t`:

```
...
>>> PUT 'call jack' IN forgotten
>>> WRITE forgotten
```

When she moves the focus to the next line the corrected command is re-executed and the error message is wiped from the screen. Instead of correcting the `WRITE` command, the user can also type an entirely different command, thus undoing the previous command.

With the UNDO key the previous key stroke is undone; this is an extension of the well-known backspace key. UNDOs can be repeated, and they can be undone with REDOs.

For a certain category of mistakes the *B* system supplies a PROTEST-OVERRIDE mechanism. For instance, if the user tries to leave an incomplete unit, the system protests, displays a message and positions the focus on the missing part. Now if the user presses OVERRIDE she is allowed to leave the unit. The unit is labelled as incorrect, which means that its use is prohibited. The only way to unlock the unit is to correct it by editing.

Now we have seen the most important documents, functions and features of the *B* environment. In the rest of this section we give a small, imaginary session. Again the user input is shown in italics, and the keys pressed in small capitals.

```
SPECIFY'DOCUMENT'TO'BE'VISITED                The user specifies the INDEX'OF'WORKSPACES
===                                           (=== is an abbreviation)
VISIT                                         and visits it.

CENTRAL                                       The system displays the required
MEMOS                                         document. There are four workspaces.
NUMBER'THEORY
POEMS
WORD'GAMES                                    The user creates a new workspace
                                              by adding its name.
VISIT                                         She visits it at once.

>>>                                           She arrives in the SESSION'RECORD.
                                              (If the workspace existed already, she would
                                              arrive in the workspace as she left it.)
>>> HOW'TO IS'PALINDROMIC word:               The user creates a new HOW'TO unit to
                                              tell if a word is a palindrome or not

HOW'TO IS'PALINDROMIC word:                   Another window, displaying the HOW'TO,
    SELECT:                                   becomes active and in this window the user types
        word = reversed'word:                 the HOW'TO text.
            WRITE word, ' is a palindrome' /
        ELSE:
            WRITE word, ' is not a palindrome'
reversed'word:
    PUT '' IN rw
    FOR l IN wrd
        PUT l^rw IN rw
    RETURN rw

EXIT                                          The user wants to return to the SESSION'RECORD,
*** error                                     but apparently an error is discovered.
*** your target wrd is undefined

    {original text}
    FOR l IN wrd:                             The focus is positioned on the error and
    {original text}                           the user can correct it easily.

EXIT
```

```
>>> IS'PALINDROMIC 'mmmmmm'
mmmmmm is a palindrome
>>> IS'PALINDROMIC 'ftft'
ftft is not a palindrome
>>> PUT 7*11*13 IN n
>>> IS'PALINDROMIC n
*** error:
*** your parameter is not a text

>>> IS'PALINDROMIC '`n`'
1001 is a palindrome
>>> IS'PALINDROMIC 'Ada'
Ada is not a palindrome
>>>
```

The user starts testing.

The user realizes that she has to convert `n` to text; this is done with backquotes inside quotes.

The user now understands that her `HOW'TO` does not cover such instances. She remembers her `YIELD strip sentence`, which is located in the central workspace, and which converts upper-case letters to lower case and strips the punctuation marks.
(The text of this `YIELD` can be found in section (6.1.2).)
Importing this `YIELD` is done by adding its name to the INDEX'OF'DOCUMENTS prefixed with a asterisk.

```
SPECIFY'DOCUMENT'TO'BE'VISITED
==
VISIT

 n = 1001
 IS'PALINDROMIC word
*YIELD strip sentence: ...

EXIT

...
>>> WRITE strip 'Ada'
ada
>>> IS'PALINDROMIC strip 'A man, a plan, a canal, Panama !!'
amanaplanacanalpanama is a palindrome
>>>
```

The user returns to the SESSION'RECORD.

4. Design philosophy

Since *B* is intended for beginning and non-specialist users, the ease of use has been the most important goal in the design. We did not keep in mind questions like "Can we implement that?" or "Will it be fast?", but "Can we keep it simple?" and "Does the system react as the naive user would expect?".

The most important condition for simplicity is: *it must be easy for the user to form an intuitive image of how the system works*. In theory it should be unnecessary to consult a manual.

From this basic condition we derive a few others:

— Suppose that a particular action in a certain context has a certain, well defined effect. If this action has a reasonable, meaningful interpretation in a slightly different context, then it must be possible to perform the action in this context as well. We call this the *fair expectation rule.*

— The strength of the system must lie in the coherence of the functions, not in their number. Traditional systems often have dozens of functions, sometimes with two or three functions performing almost the same task. We have already mentioned this in a previous section.

— The existence of sophisticated facilities must not impede the beginning user.

From these conditions emerges the desire to design the environment such that as few new concepts as possible are added; if possible the system functions are embedded in the already existing structures. This is carried out:

— by adding environment functions to *B* (thus integrating the job control and programming language); and

— by allowing the user to edit system documents.

If neither of these possibilities work, a new meta-key (or key combination) is added (like VISIT). Let us treat the alternatives mentioned above in more detail.

Some functions lend themselves well to being added to *B*. Such a function must fulfill the requirement that its syntax and semantics fit in the philosophy of the *B* language. An example is the `DIRECT'TO <document>:` command. With this command, output of the command-suite following `DIRECT'TO` is written on the specified document, instead of on the default document SESSION'RECORD.

The idea of editing system documents is due to Fraser [4]. *System documents* are documents that describe parts of the state of the system. Editing system documents really affects the state of the system. We saw this already in the example of the INDEX'OF'VARIABLES. The strength of this mechanism is shown by the fact that in this example functions like create-document, rename-document or delete-document become superfluous. In the SESSION'RECORD example special history and undo functions are no longer necessary.

The easiest way to describe the semantics of edit actions on system documents is by means of *invariants*. As we have seen already, invariants are equations that hold when the system is at rest and which should be restored if they are disturbed. The direction in which an invariant is restored is very important. The direction is from the changed side to the other side. (If it were the other way around each user action would be meaningless since it would be undone.) This is called the *invariant principle.*

A simpler, user view on invariants is the following. One side of the equation is the internal state of the system, the other a description of this state written on a (system) document. If a process changes the system state then the description is changed accordingly, and —the other way around— if the user alters the description then the system state is altered.

The *principle of locality* says that a document name (tag or keyword) is bound tightest to the nearest environment/workspace. We distinguish in descending order of nearness the local environment (=inside a unit), the current workspace and the central workspace. See also DP 2.2. and 4.0.2.

Another general rule is that the system only visits another document when the user has pressed VISIT. Note however that the keywords `HOW'TO`, `YIELD` and `TEST` induce an implicit

VISIT.

5. Preliminaries to the description of the environment

5.0. Assumptions

Before we start with the description of the environment we will state the abstractions and assumptions we make. We concentrate on the semantics of the elements of the environment; the exact form and syntax are sometimes vague, sometimes even omitted.

An important assumption is that there exists a reasonable mapping of the environment on the screen. The practical fact that screens have a limited size is a nuisance for which solutions are suggested in section 5.2. and 5.3. We also assume the existence of a syntax-directed editor.

We will not specify screen and window manipulation functions, since their semantics depend heavily on the terminal and input devices (mouse, joy stick etc.) available, and this is beyond the scope of this paper. The mapping of the environment to the screen and suitable window and screen manipulation functions still need careful study.

For a searching mechanism no good solution has yet been found, and the details of a trace and debug facility still have to be worked out. In section 5.5. and 5.6. we give an outline of our ideas on these topics.

5.1. Objects and documents

The objects are the "variables" of the *B* system. The set of objects is divided into two subsets:

1) The set of objects whose contents can be made visible on the screen and which have a name in the system. We will call these objects *documents*.

2) The set of *abstract objects*. These are just virtual notions which are needed to define the state of the system but do not exist for the user (though they might exist in the implementation of the environment).

Every document has :

— a *name* by which it is recognised in the system,
— a *type* indicating its syntax,
— a *content*,
— one (or two) *focus*(ses),
— a *workspace* to which the document belongs,
— an *involved-in-process label*, indicating whether the document is involved in a process,
— an *is-correct label*, indicating whether the document is syntactically correct or complete, and
— a *remark field*, where remarks concerning that particular document are stored (e.g date of creation, permission, dependency on other documents etc.).

Abstract objects only have a content.

The *state of the system* is defined by the state of all objects present in the system. The *state of*

an object is defined by the contents of every entry mentioned above, except the *remark field.*

5.2. Displaying and editing documents in general

After pressing the VISIT key the specified document becomes the active document. The active document is always visible on the screen. Since the system can deduce the syntax from the type of the document, the displayer adjusts to this syntax and shows the document in a convenient, visually attractive way.

Contrary to other systems, displaying and editing are not separate functions. Instead the B-system has an invariant (6.2.2) that states:

contents-of(<doc>) ≡ representation-on-the-screen-of(<doc>)

So if a document is visible and its contents are changed, then the invariant is disturbed since the equality does not hold any longer. But now the system changes the screen representation of the document accordingly, and the invariant is restored. It also works the other way around: if the user changes the representation of the document with an edit command, the contents of the document change.

Clearly, at the end of an edit session no special save-action is necessary; a VISIT to another document terminates the edit action. With EXIT a VISIT is undone and the previous active document becomes active. In its turn EXIT can be undone with REVISIT.

In the *B* system each document that can be visited may also be edited, except during a process. In that case each document involved in the process can be inspected, but not edited. Another rule is that the only documents whose contents the *B* interpreter may alter, are targets. To allow the interpreter to alter the contents of system documents was considered too confusing and too complicated.

Because many documents have a tree-like structure rather than a sequential structure, the editor is equipped with a *focus* instead of a cursor. The focus, the region of attention, indicates a part of the tree. Edit commands affect either the position or the contents of the focus. The editor assists the user by giving suggestions; this is possible because the syntax dictates often only a limited number of possibilities. E.g. when an opening parenthesis is typed, the editor supplies the matching parenthesis. For more details on the editor and the suggestion mechanism see DP (appendix), [5] and [2].

The focus is associated with the document, not with the window or with the screen. Therefore, when a document disappears from the screen and is visited again later, the focus is still on the same place.

The editor provides a COPY facility. When COPY is pressed, the text in the focus is duplicated into the copy-buffer. When COPY is pressed again, the text in the buffer is inserted in the hole on which the focus stands. This mechanism is extended: if there is still text in the buffer and the user visits another document, the buffer is imported to the other document, i.e. there is only one, global, copy buffer.

5.3. The screen and windows

Of course the screen is too small to display the entire environment. So that the user can see at least more than one document at a time, the screen is divided into windows, each displaying (a part of) a document. If a document is too big to fit in the window, parts of the document that are not in the focus are represented with an ellipsis. But if possible the contents of the focus are always displayed entirely. If the uncompressed representation of the part in the focus is larger than the

window, an ellipsis will compress the actual text. If the user narrows the focus and moves it to the ellipsis, it expands to show what was written originally. We assume that a reasonable text compression algorithm is available; in [6] such an algorithm is given.

It is convenient to be allowed to look at two (or more) different places in the same document. This is possible by means of a function which splits the active window into two identical copies. In this way we see two windows displaying the same document. Just one of the windows is active, and focus movements affect that particular window only. Contents modifications are also visible in the non-active window, since a window shows a document as it *is*. Note that VISIT cannot distinguish between the two windows, so a special facility (e.g. pointing with the mouse) is needed. The two windows are displayed until one window "falls off" the screen because it is overwritten by another window. If later the user revisits this document the focus will be as it was in the window that was on the screen last. The focus in the twin window is lost forever.

Perhaps there are functions needed to control the number and size of the windows visible on the screen. A window is addressable through the name of the document that is displayed in it, but functions like "make the window where my cursor/mouse is now the current window" are also necessary.

5.4. Process handling and parallelism

In the *B* system there are no special process-handling commands. Each process is started by typing a *B* command in the SESSION'RECORD, and can be killed by deleting that command. It can also be stopped by pressing the INTERRUPT key.

Concurrent processes are allowed only if the semantics is such that the final result is equal to the result of sequential execution of the processes. From this general principle we derive some specific conditions:

— A process computes with the contents the targets and units had at the moment the process was started. Modifications to documents occurring in a running process, do not affect the final result of the process.

— All SHAREd targets and all parameters of a HOW'TO unit that might be altered by the process (e.g. because they appear in a READ or DELETE, or as a target in a PUT, INSERT, or REMOVE command) are labelled as *involved-in-a-process*, and as a consequence locked for editing or for modifications by other processes. Just looking (a VISIT without editing) is permitted though.

— A new process may only start if all the documents it wants to use are not locked.

So the user can type several commands in the SESSION'RECORD, and the system decides according to the rules above if these commands are run simultaneously. Observe that the user can switch to another workspace by means of the SPECIFY'DOCUMENT'TO'BE'VISITED function, and that processes in distinct workspaces cannot lock each other.

The *B* system provides an output redirection mechanism by means of the DIRECT'TO command (see (6.3.9)). We have abandoned a pipe mechanism (the output of one process serving as input for the next) because *B* offers alternatives to this: first the user can exploit permanent targets to supply the result of one process to the next, and second, she can use the COPY facility of the edi-

tor to move the output of one command to the input of another.

5.5. Flagging and searching

It is not yet clear whether searching is a system function or an edit function or a combination. It must be possible to look in every document for the occurrence of a specified text; if this is the case the text is flagged. If the document that contains this text appears in an index, the name of that document is flagged in that index as well.

We have the following model in mind: the search area is displayed entirely (eventually reduced), and every spot where the specified text is found is clearly flagged (e.g. displayed brighter or in a different colour). The user visits these spots, does the appropriate modifications and proceeds to the next.

There is also a facility needed to combine searching with an edit command, for instance to change each occurrence of a specified text into another.

5.6. Tracing and debugging

There is a special trace-mode available. When this mode is set, the focus always indicates the statement where the next execution step will take place. If the user presses DOWN this command is executed, if she presses a key like UP or UNDO the previous command is undone. She can also manipulate the size of the focus and thus control how precisely she wants to follow the execution. She can also use VISIT to check whether units work as they should. At the same moment she can look at several targets and their values in another window.

6. Specification of objects

Below, all objects of the *B* environment are specified. Most specification entries should be clear, but perhaps two need an explanation:

— *visitable:* either *yes* or *no*, tells whether the object is a document or an abstract object. This entry thus indicates whether this object is visitable and editable (these are equivalent).

— *edit semantics:* indicates which consequences an edit action on this document can have on other documents.

If we refer to a specification entry we use the -of-notation, e.g. visitable-of(<target>) = *yes*.

We use the *B* language to describe algorithms. We have not attempted to write efficient or complete algorithms; they only serve to give a rough idea of what should be done.

6.1. Objects

6.1.1. <TARGET>

informal description: These objects are the ordinary *B* targets

visitable: yes

name: valid *B* tag

type: number, text, compound, list, table

contents: the *B* value

focus: edit-focus

display examples:

```
phi = 0.6180339887498948
shopping'list = {'bread'; 'milk'}
days'of'week = {[1]: 'Sunday'; [2]: 'Monday'; [3]: 'Tuesday'; [4]:
        'Wednesday'; [5]: 'Thursday'; [6]: 'Friday'; [7]: 'Saturday'}
```

edit semantics:

The user cannot change the name of a document into a name that already exists in the workspace.

See also: WORKSPACE (6.1.8).

remarks: see also: DP (1.2).

6.1.2. <UNIT>

informal description:

These objects are the ordinary HOW'TO-, YIELD- and TEST-units of *B*.

visitable: yes

name:

If HOW'TO-unit, name is a valid *B* keyword; if YIELD- or TEST-unit then name is a valid *B* tag. Note that the name of the unit is contained in its first line.

type: HOW'TO, YIELD or TEST

contents: the text of the unit

focus

During editing there is an edit-focus, under the control of the user. When the unit is executed there is an execution-focus, indicating the command that is currently executed. This execution-focus is usually controlled by the system, but in trace/debug-mode also by the user.

display example:

```
YIELD strip sentence:
    PUT '' IN res
    FOR l IN sentence:
        PUT res^simplify'letter IN res
    RETURN res
simplify'letter:
    SELECT:
        l in {'a'..'z'}: RETURN l
        l in {'0'..'9'}: RETURN l
        l in {'A'..'Z'}: RETURN rank th'of {'a'..'z'}
        ELSE: RETURN ''
rank: RETURN #{'A'..l}
```

edit semantics:

Invariant 6.3.6. must be preserved, so if the name is changed the first line must be changed too, and the other way around. And if the name is changed, all the unit calls must be changed as well.

It is not allowed to have homonymous units in one workspace, except for two functions differing in adicity (but not zeroadic and monadic); see WORKSPACE (6.1.8).

remarks:

See also: DP (4), esp. (4.1.1.b), (4.2.1.d) and (4.3.1.d) for the names.

6.1.3. SESSION'RECORD(<WORKSPACE>)

informal description:

The SESSION'RECORD is the document on which the whole session is written. In other words: it is the protocol of a session. The SESSION'RECORD is composed of a COMMAND'RECORD (for the immediate *B* commands), an INPUT'RECORD (for supplying the input to a READ command), an OUTPUT'RECORD (where the output is printed) and an ERROR'MESSAGE'RECORD. The SESSION'RECORD "synchronizes" these four records; it always knows which record is in turn. We call this synchronization process *interlacing*. Each workspace has its own SESSION'RECORD.

Since edit actions on targets and units destroy the invariant of the SESSION'RECORD (invariant (6.2.2)), such edit sessions are written on the SESSION'RECORD of the workspace to which the document belongs. When this happens, the *B* commands preceding the edit session can no longer be edited, because it is not clear whether the old or the new version of the edited document must be used. When the user wants to re-execute a sequence of *B* commands she can use the COPY facility to duplicate these commands to a place below the execution-focus.

Below the current *B* command means the *B* command where the execution-focus is.

visitable: yes

name: `session'record` or an abbreviation

type: RECORD

value: the text of the session

focus:

There are two focusses: an edit-focus, indicating the place where the user is editing, and an execution-focus which is placed on the particular B-command that has not been executed yet. These two focusses are often identical, but especially if the user types faster then the computer can execute, the edit-focus is ahead of the execution-focus. The focus of the SESSION'RECORD defines the focusses of the records of which it is composed, see (6.1.4) - (6.1.7).

display example:

```
  ...
<edit action; letter = 'z'>
>>> WRITE #{'a'..letter}
26
>>> READ letter EG 'a'
'm'
>>> PUT letter^^10 IN delicious
>>> WRITE delicious
mmmmmmmmmm
>>> _
```

(the underscore (_) indicates the focus position)

edit semantics:

See also: invariant (6.3.2).

RETURN:

informal description:

The current *B* command is offered to the *B* interpreter.

definition:

```
HOW'TO EXECUTE current B command:
    SELECT:
        current B command contains static error:
            GIVE'ERROR'MESSAGE
            MOVE focus TO error
        current B command starts with HOW'TO, YIELD or TEST:
            VISIT unit
        current B command involves locked target:
            GIVE'LOCK'MESSAGE
            MOVE focus TO name of locked document
        current B command is statically correct:
            INTERPRET current B command
        ELSE: \\
```

example:

If in the previous display example UP is pressed once, the WRITE command is undone and the following is displayed:

```
  ...
<edit action; letter = 'z'>
>>> #{'a'..letter}
26
>>> READ letter EG 'a'
'm'
>>> PUT letter^^10 IN delicious
>>> WRITE delicious
mmmmmmmmmm
```

(the undone lines, here in italics, are displayed in a different colour or with half intensity)

remarks:

If the user types faster than the system can compute, the system supposes that these characters are commands. If the user wants to supply input for a READ in advance she must precede each line of her input with a special key. She can also wait until the input prompt appears.

The system executes the commands concurrently if the targets do not conflict. If there is a conflict which prohibits the simultaneous execution of these commands, the commands are executed consecutively in the order in which they are written on the SESSION'RECORD.

See also: process handling and parallelism (5.4).

6.1.4. COMMAND'RECORD (<WORKSPACE>)

informal description:

Here all the immediate *B* commands are written.

visitable: no

type: RECORD

contents: list of *B* commands

focus: an execution-focus, indicating the next command to be executed

example:

```
...
WRITE #{'a'..letter}
READ letter EG 'a'
PUT letter^^10 IN delicious
WRITE delicious
```

remarks: see also: SESSION'RECORD (6.1.3).

6.1.5. INPUT'RECORD (<WORKSPACE>)

informal description:

Here all input supplied to READ commands is written.

visitable: no

type: RECORD

contents: text for the READ commands

focus: a read-focus, indicating the place where the next input will be read

example:

```
'm'
```

remarks: see also: SESSION'RECORD (6.1.3).

6.1.6. OUTPUT'RECORD(<WORKSPACE>)

informal description: Here all the output resulting from WRITE commands is written.

visitable: no

type: RECORD

contents: text of the WRITE commands

focus: a write-focus; indicating the place where output will be written

example:

```
26
mmmmmmmmmm
```

remarks: see also: SESSION'RECORD (6.1.3), the `DIRECT'TO` command (6.3.9).

6.1.7 ERROR'MESSAGE'RECORD(<WORKSPACE>)

informal description:

Here the messages emerging from runtime errors are written.

visitable: no

type: RECORD

contents: list of error messages

focus: ---

remarks:

See also: SESSION'RECORD (6.1.3).

If the user corrects a mistake by editing the erroneous command, the error message is erased.

6.1.8 <WORKSPACE>

informal description:

A workspace is a collection of documents belonging together. Since it is not a document itself a workspace is not visitable in *the strict sense*. But if we switch to a workspace we can say we VISIT (go to) it. In fact we change the value of the system document CURRENT'WORKSPACE, which is done via the VISIT function.

visitable: no (see above)

name: <WORKSPACE>; only keywords are allowed

type: WORKSPACE

contents: { <obj> | workspace-of <obj> = <WORKSPACE> }

remarks:

See also: CENTRAL'WORKSPACE (6.1.9), INDEX'OF'DOCUMENTS (6.1.10).

An advantage of splitting the total environment into workspaces is that the user is relieved of the responsibility of inventing unique names for each document; it is allowed to have the same name for two documents in different workspaces (the principle of locality).

B allows the existence of a zeroadic function and a target; but in the interpreter a target prevails over a unit. Combinations of either a zeroadic or monadic target with a dyadic target for a `YIELD-` or `TEST-`unit are also permitted (e.g. `root x` and `n root x`).

In the case of a SPECIFY'DOCUMENT'TO'BE'VISITED function the user can type a context to solve ambiguity problems, e.g. she can type `root 256` or `4 root 256`. If the system cannot determine from the context which one is meant, a temporary window appears displaying a one line representation of each alternative. With the focus positioning commands the user selects the desired document.

6.1.9. CENTRAL'WORKSPACE

informal description:

In the CENTRAL'WORKSPACE the user defines units she wants to be able to use in several workspaces. For instance, she can define her own additional `HOW'TO` commands. A unit of the CENTRAL'WORKSPACE is imported to a lower workspace by mentioning its name in the INDEX'OF'DOCUMENTS, preceded by a special character. Targets cannot be imported.

The semantics of importing is as if a local copy of the unit is present in the lower workspace. This means in particular that for SHARE d targets the local version is taken.

An imported unit can only be changed in the CENTRAL'WORKSPACE. If the user tries to do this in another workspace a error message or a warning about implicit workspace changes is given.

other entries: see <WORKSPACE> (6.1.8).

6.1.10. CURRENT'WORKSPACE

informal description:

At any moment one workspace is the current, active workspace.

visitable: no

contents: the current, active workspace

remarks: see also: <WORKSPACE> (6.1.8).

6.1.11 INDEX'OF'DOCUMENTS(<WORKSPACE>)

informal description:

Each workspace has a list of all the documents that exist in that workspace. This list consists of two parts: the targets and the units. For each document only one line is displayed: targets are compressed to one line, and of units only the first line is shown.

visitable: yes

name: `documents[<workspace>]`, perhaps an abbreviation like `::` or `==` .

type: INDEX

contents: {one-line-representation-of(<doc>) | workspace-of(<doc>) = <WORKSPACE>}

focus: ...

display example:

```
phi = 0.6180339887498948
days'of'week = {[1]: 'Sunday'; [2]: 'Monday'; [3]: 'Tuesday'; ...}
YIELD strip sentence: ...
```

If the required number of lines is greater than the number of lines on the screen, the displayer uses an ellipsis again to represent several units or targets.

edit semantics:

See invariant (6.2.4).

It is allowed to use the COPY facility of the editor in order to duplicate the contents of one or several documents from one workspace to another. So the metaphor that the INDEX'OF'DOCUMENTS is a "long distance view" of the workspace extends to the COPY function as well.

remarks: see also: <WORKSPACE> (6.1.8).

6.1.12 INDEX'OF'WORKSPACES

informal description:

This document contains a list of all the workspaces. It can be regarded as the highest in the hierarchy.

visitable: yes

name: WORKSPACES and/or an abbreviation like ::: or ===

type: INDEX

contents: { name-of (<obj>) | type-of (<obj>) = WORKSPACE }

focus: ...

display example:

```
CENTRAL
MEMOS
NUMBER'THEORY
POEMS
```

edit semantics: see invariant (6.2.5).

remarks: ---

6.1.13 ACTIVE'DOCUMENT

informal description:

This is the system document which indicates what the current, active document is on which the editor acts and which is always displayed on the screen. If possible the name of the active document, i.e. the contents of the object ACTIVE'DOCUMENT, is always displayed in a small window.

visitable: no, see also DOCUMENT'TO'BE'VISITED (6.1.14)

contents: the current, active system document

edit semantics: ---

remarks: see also: DOCUMENT'TO'BE'VISITED (6.1.14) and SPECIFY'DOCUMENT'TO'BE'VISITED (6.3.7)

6.1.14 DOCUMENT'TO'BE'VISITED

informal description:

When SPECIFY'DOCUMENT'TO'BE'VISITED is pressed, the window displaying the name of the active document becomes active. By editing the user specifies in this window the name of the document she wants to visit. During this edit action the contents of ACTIVE'DOCUMENT is DOCUMENT'TO'BE'VISITED, and when the session is terminated the contents of ACTIVE'DOCUMENT becomes the contents of SPECIFY'DOCUMENT'TO'BE'VISITED.

visitable: yes, through the SPECIFY'DOCUMENT'TO'BE'VISITED function.

contents: the desired document

edit semantics: ---

remarks: see also: ACTIVE'DOCUMENT (6.1.13) and SPECIFY'DOCUMENT'TO'BE'VISITED (6.3.7).

6.1.15 LINEPRINTER

informal description:

Every text written on this document by an edit action, a DIRECT'TO command or a HARD'COPY key press is printed. LINEPRINTER is a global document, implicitly imported in each workspace.

visitable: yes

name: printer

type: target

contents: every text to be printed

focus:

Besides an edit-focus there exists an execution focus which indicates what has been printed already and what not. Editing above the execution-focus is prohibited.

display example: ---

edit semantics: ---

remarks: see also: HARD'COPY (6.3.8), DIRECT'TO (6.3.9)

6.2. Invariants

Invariants are equalities that hold when the system is at rest. If an invariant is disturbed one side of the equation has been changed. *The direction in which an invariant is restored is always from the changed side to the other side of the equation.* The equality sign ≡ below must be read as "should be equal to".

State invariants

The overall system invariant is:

(6.2.1) actual state of the system ≡ semantical result of all commands and edit actions, taking into account the order in which they occurred, starting from the initial state

(For the definition of the state of the system see section 5.1.)

A consequence of (6.2.1) is, that if the system is at rest (e.g. if it is in its initial state) and the user types a command, the system must execute the command in order to restore the invariant.

Loosely speaking we can call the right hand side of (6.2.1) the desired state as it is described by the user, so we obtain

(6.2.1a) actual state ≡ desired state

And if the user changes the description of the desired state, the actual state has to change accordingly.

Invariant (6.2.1) can be narrowed to one workspace:

(6.2.2) actual state of <WORKSPACE> ≡ effects of commands and edit actions written on SESSION'RECORD (<WORKSPACE>), up to the command where the focus stands

This works because the workspaces do not affect each other; only actions on documents of the CENTRAL'WORKSPACE which are imported, affect other workspaces.

Edit-display invariant

(6.2.3) contents-of(<doc>) ≡ representation-on-screen-of(<doc>)

So if a document is displayed and its contents changes, then the invariant is disturbed since the equality does not hold any longer. But now the system changes the screen representation of the document accordingly, and the invariant is restored. It also works the other way around: if the user changes the representation of a document by an edit command, the contents of the document must be changed too.

Note that evaluating the equation from left to right is what we used to call displaying, and from right to left editing.

Indexes

The contents of most system documents is defined through the existence and state of other documents. Examples are the indexes:

(6.2.4) {one-line-representation-of(<doc>) | workspace-of(<doc>) = <WORKSPACE>}≡
≡ contents-of(INDEX'OF'DOCUMENTS(<WORKSPACE>))

(6.2.5) {name-of(<obj>) | type-of(<obj>) = WORKSPACE}≡
≡ contents-of(INDEX'OF'WORKSPACES)

Unit name invariant

The name of a unit is contained in its first line:

(6.2.6) name-of(<unit>) is-contained-in first-line-of(<unit>)

Restoration of the invariants.

At first sight it might seem best if each invariant is restored immediately after each key press. But for instance in the case of the SESSION'RECORD it would be burdensome if the system tried to interpret a *B* command while the user is still editing it: an inexhaustible sequence of error messages would appear and disappear. A similar argument holds for the INDEX'OF'DOCUMENTS.

In most documents we distinguish "meaningful entities", a certain amount of information belonging to each other. Inside such an entity updating the invariant after each character is useless. Instead the system waits until the user presses VISIT or moves her focus from the entity, thereby indicating that she is finished with editing the entity. At that moment the invariant is restored in one blow. In the SESSION'RECORD the meaningful entity is a *B* command or an input line for READ, and in the INDEX'OF'DOCUMENTS and INDEX'OF'WORKSPACES one name is.

6.3 Functions

6.3.1. UNDO

informal description:

This function means: undo previous key stroke as if it were never pressed. It incorporates the traditional backspace, but undoes also the effect of every other key, e.g. a focus positioning command or a VISIT. Repeatedly pressing UNDO is interpreted as going back in time. Perhaps there will be a limitation to the number of permitted UNDOs. UNDO's can be replayed with REDO. But if the user is somewhere back in time and presses a key other than REDO, then everything that was undone is lost forever.

definition

```
SELECT:
    key = UNDO :
        POP input'stack TO last'key
        UNDISPLAY last'key
        PUSH last'key ON undone'stack
    key = REDO :
        POP undone'stack TO last'key
        PUSH last'key ON input'stack
        DISPLAY last'key
    ELSE: \ key is printable
        PUSH key ON input'stack
        DISPLAY key
        EMPTY undone'stack
```

6.3.2. REDO

informal description:

REDO is the reverse (the undo) of UNDO; see further UNDO.

6.3.3. OVERRIDE

informal description

When the user makes a mistake, for instance trying to leave an incomplete or syntactical incorrect unit, the system protests and gives a warning. The user can ignore such a warning by pressing OVERRIDE. This means: I know I am wrong but I don't have time to correct my mistake now. OVERRIDE does not solve the problem, it only postpones it. The document concerned cannot be used in any *B* command until the error is corrected. Only a DELETE is allowed.

6.3.4. VISIT

informal description

The VISIT function is used for selecting documents. It is interpreted as "make the document whose name is in the focus the current document". VISIT is independent of the active document, but searches the specified document only in the active workspace. In the indexes VISIT offers a menu-like way of selecting a document: by pressing UP and DOWN (or similar edit functions) the desired document is chosen, and then the VISIT key is pressed.

Visited documents are pushed on a special stack. VISIT key presses can be undone with the EXIT function. EXIT means: visit the previous active document. EXITs can be repeated. The reverse of EXIT is REVISIT. Thus we can say that EXIT = undo-visit, and REVISIT = redo-visit.

definition

```
HOW'TO VISIT specified'document:
    IF contents of system'current'document is incorrect or incomplete:
    \\ can we safely leave the system'current'document?
        PROTEST
        IF overridden:
            LOCK system'current'document
            SWITCH'TO specified'document
```

```
    QUIT
IF specified'document is involved in a process:
\\ documents involved in a process may be displayed,
\\ but not edited on
    LABEL noneditable
    SWITCH'TO specified'document
    QUIT
IF system'current'document in units'of [current'workspace]:
\\ because of the locality of a unit
    SELECT:
        specified'document in refinements [system'current'document]:
            MOVE focus TO refinement
            QUIT
        specified'document in local'targets [system'current'document]:
            ERROR
            \\ the contents of local targets is  undefined during editing
            \\ except during trace/debug mode
            QUIT
        ELSE: \\ go on with the general cases
SELECT:
    specified'document = index'of'workspaces:
        \\ to index'of'workspaces
        SWITCH'TO specified'document
    specified'document in index'of'workspaces:
        \\ specified'document is a workspace
        PUT specified'document IN current'workspace
        SWITCH'TO current'document'of[current'workspace]
    specified'document in index'of'documents[current'workspace]:
        \\ specified'document is unit or target in the current workspace
        SWITCH'TO specified'document
    specified'document in index'of'documents[central'workspace]:
        \\ specified'document is unit or target in the current workspace
        SWITCH'TO specified'document
    specified'document is predefined unit:
        SWITCH'TO manual [specified'document]
        \\ gives an explanation
    ELSE: \\ specified'document is unknown
        ERROR
```

HOW'TO SWITCH'TO new'document:
PUSH new'document ON visit'stack
PUT new'document IN system'current'document :>

6.3.5. EXIT

informal description

EXIT undoes an VISIT, see further VISIT (6.3.4).

6.3.6. REVISIT

informal description

REVISIT undoes an EXIT, see further VISIT.

6.3.7. SPECIFY'DOCUMENT'TO'BE'VISITED

informal description

The user can use this function at any time. If she does so, a window displaying the DOCUMENT'TO'BE'VISITED appears temporarily on the screen. Here the user can specify the name of the document she wants to visit. When she is finished with editing the user presses VISIT, and the specified document becomes the active document.

definition

```
HOW'TO SPECIFY'DOCUMENT'TO'BE'VISITED:
    PUT document'to'be'visited IN active'document
    \\ VISITs to this document are not pushed on the stack
```

6.3.8. HARD'COPY

informal description:

When this key is pressed, the text in the focus is appended on the document `printer`. Ellipses are expanded to the original full text in the hard copy. What happens if the active document is an index is not yet clear.

6.3.9 DIRECT'TO

informal description:

The `DIRECT'TO` command serves as the system output redirection mechanism. Every output resulting from a WRITE command in the command-suite following the `DIRECT'TO` is send to the specified document, instead of to the SESSION'RECORD. This document must be a table of text, and must already be defined. Every time a newline symbol is encountered, a new key and associate pair is generated. Preceding associates are freely available. A particular instance is the `printer`: every character written on this document is printed out on paper.

example

```
...
>>> PUT {} IN output'doc
>>> DIRECT'TO output'doc:
>>>    WRITE 'Ciao !!!' /
>>> \\ Now output'doc = {[1]:'Ciao !!!'}
```

6.3.10. INTERRUPT

informal description

INTERRUPT stops the execution of the current command (process) and acts as if a runtime

error was encountered. With UNDO the process is resumed.

References

[DP] Meertens, L.G.L.T., "Draft Proposal for the *B* programming language - Semi-Formal Definition", Mathematisch Centrum, Amsterdam, 1981.

[1] Geurts, L.J.M., "An overview of the *B* programming language or *B* without tears", *in: SIGPLAN notices volume 17, number 12, pages 49-58,* December 1982.

[2] 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.

[3] Heering, J.,"Taaldefinities als kern voor een programmeeromgeving", *in: Colloquium programmeeromgevingen (J.Heering and P.Klint, editors), pages 39-51,* Mathematisch Centrum, Amsterdam, 1983.

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

[5] Nienhuis, A.J.C., "On the design of an editor for the *B* programming language", Mathematisch Centrum, Amsterdam, 1983.

[6] 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 147-146,* June 1981.