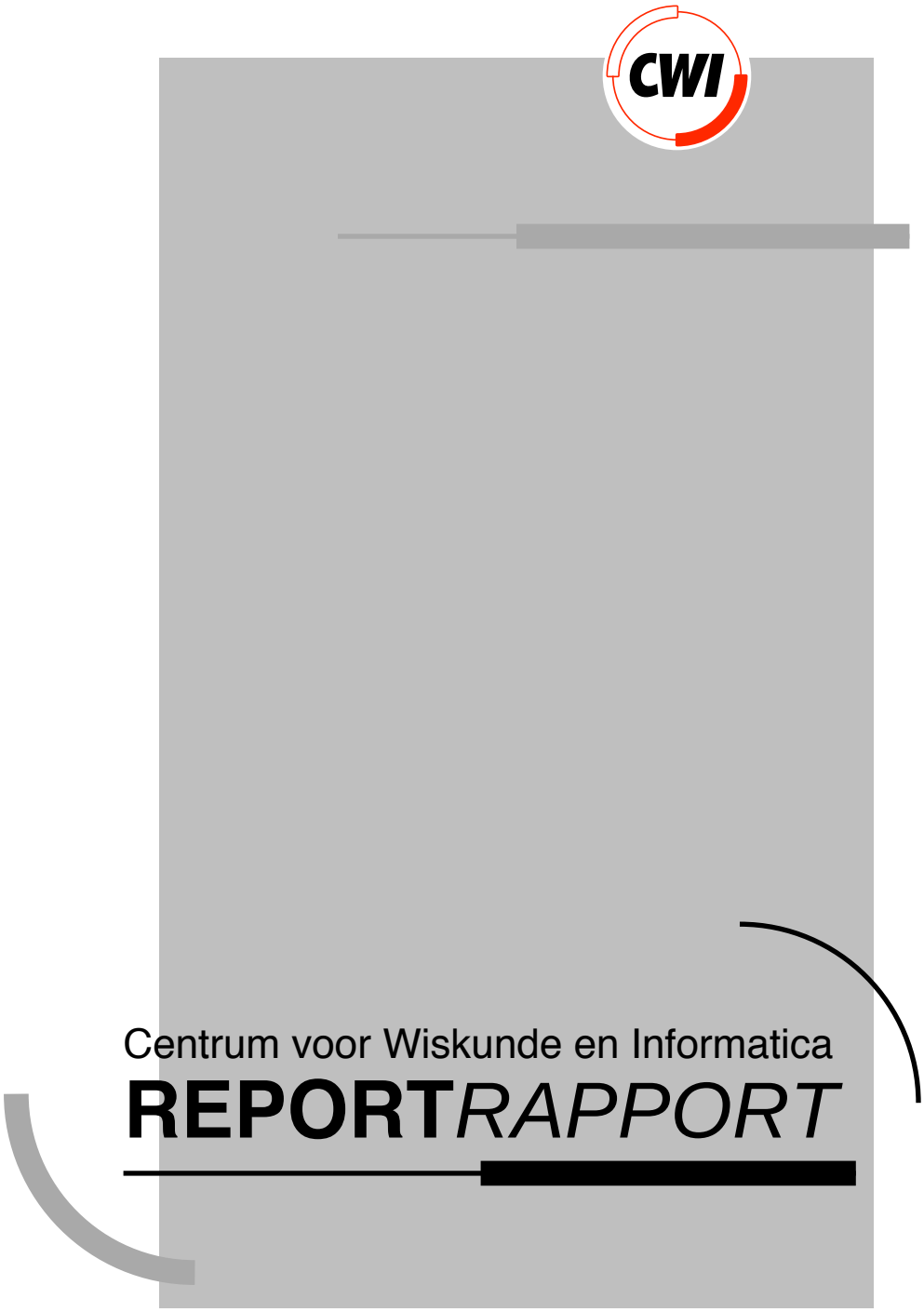


Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

The ABC structure editor. Structure-based editing for the ABC programming environment

L.G.L.T. Meertens, S. Pemberton, G. van Rossum

Computer Science/Department of Algorithmics and Architecture

CS-R9256 1992

The ABC Structure Editor

Structure-based Editing for the ABC Programming Environment

Lambert Meertens, Steven Pemberton and Guido van Rossum

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
Email: Lambert.Meertens@cwi.nl, Steven.Pemberton@cwi.nl, Guido.van.Rossum@cwi.nl

Abstract

ABC is an interactive programming language where both ease of learning and ease of use stood high amongst its principle design aims. The language is embedded in a dedicated environment that includes a structure-based editor. To fit in with the design aims, the editor had to be easy to learn, demanding a small command set, and easy to use, demanding a powerful command set and strong support for the user in composing programs, without enforcing a computer-science understanding of issues of syntax and the like.

Some novel design rules have led to an interesting editor, where the user may enter and edit text either structurally or non-structurally, without having to use different “modes”.

1991 Mathematics Subject Classification: 68N15, 68Q50.

1991 CR Categories: D19, D.2.2, D.2.6, H.1.2, H.5.2.

Keywords and Phrases: programming environments, human factors, user interfaces, editing, structure editing.

1 Introduction

This article concerns a dedicated structure editor that forms part of the ABC programming environment. Several novel ideas have been applied in its design. While it is not the case that all design aspects of the ABC editor generalise beyond the context to which it is dedicated, many do, and even those that do not still offer points of interest.

ABC is a programming language and environment designed and implemented at the CWI [6, 12]. The principal design aims were to provide a structured, interactive, and above all simple language for beginning programmers, as a good alternative to BASIC, in which "interactive" means that the language is embedded in a dedicated environment [5].

From the inception of the project we have planned for applying an iterative method to the design of the language. Indeed, since the first version of ABC (which was originally called *B*), the language has been redesigned three more times, based on experiences with each prior version [8, 9]. The current version of the language [6] is the fourth, and while remaining true to its original aims, has matured into a language that is a useful and powerful tool for beginners and experts alike.

An aim of the project was that ABC should be a complete programming environment [16] offering a single face to the user, where it should not be necessary to learn a separate command language, editor, file system, compiler, and programming language, just in order to program.

To achieve this goal, the design of the ABC system was reduced to two elements: the language, and a dedicated editor. ABC is used as both programming language and command language, and for tasks where the ABC language is unsuitable, the editor is used.

The ABC language is not a topic of this paper, but some of its syntactic characteristics are relevant to some discussion points. The following is a typical piece of ABC code:

```
PUT 0 IN count
FOR di, dj IN neighbours:
   IF (i+di, j+dj) in keys c:
      PUT count+c[i+di, j+dj] IN count
SELECT:
   count = 3 OR count+c[i, j] = 3:
      PUT 1 IN n[i, j]
   ELSE:
      PUT 0 IN n[i, j]
```

The commands of ABC have a skeleton of keywords (like "PUT … IN …") alternating with expressions. Layout is significant. In particular, each command starts on a new line, and grouping is indicated by indentation. So the `FOR` command governs the next two lines, up to but not including the equi-indented `SELECT` command. Note that this is not just automatic prettyprinting; in ABC the layout is not redundant, unlike in a prettyprinted Pascal or C program.

2 General design aims

The design considerations for the ABC editor, described in the next section, must be understood in the light of the general design aims for the complete ABC environment, including the language.

Of the three design objectives: structured, interactive, and simple, we concentrate here on the last one. The first was aimed primarily at the language *per se*, and the second is too obvious a requirement in relation to editor design — which is the topic of this paper — to warrant a separate discussion.

Our objective of simplicity refers both to the ease of learning and to the ease of use. While these two are often perceived as conflicting aims in user-interface design, necessitating separate forms of support for novices or casual users and for more seasoned or regular users, we felt that ease of use is also of great importance for beginning users, and that these two aspects of simplicity are not irreconcilable when approached in an integrated way. Although this has not always been easy, in retrospect we feel that we have been successful in combining these aims, if only because the insistence upon both aspects forced us to reconsider solutions that would otherwise have been deemed acceptable, to challenge set approaches, and to identify the fundamental aspects of the issues involved. Thus, solutions have been found that would otherwise never have surfaced. Other evidence that the needs of a spectrum of users can be accommodated by a single design is reported on in [7].

The design of the language has largely preceded that of the rest of the environment. An unusual approach, at least compared with the approach taken in the design of most other languages, was used for the ABC language: a specific (although not detailed) *user model* and the user's (intended) *mental model* were the driving force for the design. It is, perhaps, also unusual that we consciously chose a user model that is somewhat unrealistic in its extremeness. More precisely, in the design of the language the user model was that of a naive, pristine, beginning computer user with only positive expectations of computers. This ideotypical target user is the very antithesis of the computer sophisticate, who has grown, if not callous, then at least used and resigned to the many often quite arbitrary limitations and peculiarities of current computer systems. In terms of this user model our aim can concisely (but simplistically) be formulated as the wish not to spoil the user's expectations by confrontation with the "facts of life" of computerdom, but offering a shock-free, ideal, environment. So, for example, rather than basing the conceptual model for the language on some machine-oriented implementation model (the design approach consciously chosen for, e.g., Pascal, ALGOL 68, and C, and apparently also at least implicitly for most other common languages), we aimed vigorously at finding those concepts that were most appropriate for the user's task from a human-oriented point of view.

To help ourselves in enforcing this approach, consideration of implementation issues was anathema during a design phase. Only after the completion of a design iteration was the question of how to implement the design efficiently allowed to be addressed, and could new implementation methods be researched. (Actually, this stern attitude was only taken from the third design iteration on, which started in 1980.)

This approach has paid off: it has resulted in a language that can be learned *in its entirety* in a couple of hours, and yet offers an order of magnitude improvement in programming time over traditional languages like C, Pascal or BASIC. Quite naturally then, we adopted the same approach for the design of the environment.

To assist, specifically, in the difficult design aim of integrating the two aspects of simplicity, several basic rules were formulated to guide the design, and to measure particular decisions against. The set of design rules we used was not fixed once-and-for-all in advance; rather, it evolved during the design process from our attempts to identify the commonalities in the rationales for various design decisions (although a surprisingly large subset can already be seen in the discussion of the design objectives in [5]). The explicit formulation of the principles that appeared to guide earlier decisions was also

helpful in increasing the overall consistency of the design. While these rules were not intended as hard-and-fast prerequisites of any design decision, they proved to be extremely useful rules-of-thumb.

Of the rules formulated, we mention here only the most important two:

- **Economy-of-Tools Rule:**
  The number of concepts (functions, features etc.) is small, but the concepts themselves are powerful and on the appropriate, task-oriented, abstraction level.
- **Fair-Expectation Rule:**
  If a concept may be lawfully used in context X, and the same concept is (conceptually) applicable in context Y, then it may be lawfully used in context Y, with the expected meaning.

It is apparent that these two rules are of a different nature. The application of the first one requires an understanding of the task domain. The way it was used was as follows. For “candidate” concepts we asked the question for what (higher-level) tasks they would be useful, and next what the most appropriate *basic* concept(s) were for addressing these specific tasks. While it may not be obvious that this helps to keep the set of “tools” small, it actually does, in particular in combination with the next design rule, whose application is fairly straightforward. Other design rules, not elaborated upon here, concern, for example, uniformity and incremental learnability.

A particular general design requirement for the ABC environment was that it should be “modeless”. While it is inescapable that the history of the keys struck up to a given point has a bearing on the interpretation of further key strokes, so that it is not feasible (nor desirable) to make the system entirely “stateless”, the user of the system should never have to deal with different modes of operation. In particular, this implies that the user never “leaves” the editor: *all* user input to the system is effected through the ABC editor, whether it be while entering or modifying a program, giving commands to be immediately executed, or providing input to a running program.

3 Design considerations for the editor

The principal position of the ABC editor in the whole system increases the importance of a good design. This, in turn, is related to the role played by the editor within the system.

In the first publication concerning this project [5], we find the following statements: “The B editor should already perform the parsing and detect most syntactical errors. […] If the editor knows the syntax, this also gives perspectives for simplifying editing commands.” This was written as early as 1975, when “CRT editors”, as visual screen editors were then called, were still somewhat of a novelty. At that time we certainly did not have more than a vague notion of the possibilities. Still, it was clear that it would be of importance that various syntactic constructions could be locally recognised. A grammatical framework formalising this requirement [4] was employed for the language syntax.

A preliminary design of the ABC editor was described in [9]. By that time, 1981, syntax-directed editing was a widely discussed idea, and it was (to us) an obvious decision that the ABC editor would be syntax-oriented. As indicated above, the main advantage over an “unstructured” text-based editor, is that the opportunity for making syntax errors is dramatically reduced, which is clearly of particular importance to beginning users who are still in a learning phase. A secondary advantage is (at least potentially) the easier use of the editor, for example for people who are poor typists.

To fit in with the basic aims of ABC, the editor had to be simple to learn and yet easy to use. A problem with several structure-based editors for programming languages is that they force a top-down, template approach to entering text, corresponding to a pre-order traversal of the abstract syntax tree [15]. This was not acceptable in the ABC context. In the first place, it demands some computer-science level knowledge of syntax. Although this could, conceivably, be taught, we felt that this would be an unacceptable hurdle to learning to use the editor. Secondly, it inhibits a natural mode of entering text, in particular being very awkward for formulas and expressions, and can make certain changes difficult [17]. Had we felt that pre-order tree traversal is, from the user's point of view, a good linearisation order for entering text — which we emphatically did not think — we would simply have chosen it as the textual linearisation order (giving so-called *Polish notation*).

The usual solution to the awkwardness of making modifications while respecting the structure of the abstract syntax tree is a *hybrid* approach, in which the editor supports both structure-based and unstructured operations. In most hybrid systems, the user can switch between two edit modes: structured and unstructured (see, e.g., [1], [2] and [10]). These two modes are radically different, and in the unstructured mode all advantages of the structured approach are lost. Another approach is to let the choice between the two modes be determined automatically (i.e., outside the user's control) by the grammatical type of the portion of text; below a certain level the tree nodes consist of unstructured text. The hybrid approach, in both forms, violates the requirement of modelessness.

Important design aims consistent with the target user and the overall design rules were of course that the total set of edit operations should be small, and that the execution of operations should be allowed whenever (to the user) conceptually meaningful. The second aim goes against the grain of the structure approach in that it is not very likely that a user's concept of meaningful changes is as restricted as that of a strictly structure-based editor. The first aim forbids us to consider the way out of having two sets of mixable operations, like "change non-structurally" next to "change structurally". (And even if the size of the operation set were not a problem, that approach runs into serious problems.)

Still, we did not want to give up the advantages of the structured approach. The desire then was that the editor should, in a *single* mode, permit the user to work with equal ease in a structure-based way as a non-structured, textual, way. A particular consequence was a requirement that came to be called "the blind typist's rule": the user should be able to enter a program, or a program fragment, just by typing it character for character from left to right and get, in a more laborious way, exactly the same result as the typist who took advantage of the editor's structured support. Somewhat more generally, we aimed at a set of operations for the editor that, although having a structure-based meaning, would also support a not particularly structured approach of the editing task.

Since our imagined typical user was someone with little or no knowledge of computers, we tried where possible to base the elements of the conceptual model on real-world examples, rather than computer-science examples. A good example of this is the fact that rather than editing a copy of a document in a buffer, that periodically and at the end has to be saved, the user is always considered to be editing the document itself. The one real advantage of the copy-in-a-buffer model is that a user can always abandon the changes and revert to the original copy. This is offset in the ABC system by providing a (conceptually) unbounded Undo operation, where each Undo undoes the effect of exactly one keystroke (other than Undo itself), whether it changed the document, moved the cursor, or whatever. This form of Undo has the added advantage that users are encouraged to try actions out to see their effect; if some action does not do what was wanted, it can always be undone.

4 The basic “grammar’’ of the edit operations

Editing consists mainly of a sequence of tasks of the form “perform <action> on <selection>”, where <action> may for example denote insertion, deletion or replacement, and the <selection> may be a character, word, line, or some other textual entity.

In several editors the basic operations provided correspond directly to such tasks. Ideally, for an operation set defined “orthogonally”, if there are M kinds of actions and N kinds of selections, the operation set consists then of M x N operations. In editors using this approach (e.g. both the modeless *emacs* and the modeful *vi*, to mention just two common tools in the UNIX environment), the design is usually not purely orthogonal. Next to what we consider as unalloyed quirks, there are usually some specialised operations combining a specific action always with a specific selection (e.g., for *vi,* there is an operation to capitalise the next character, an action not available for any other kind of selection). In addition, there are operations for moving the cursor without performing an action on the contents on the document.

In editors with this approach, the “syntax” of the operations typically obeys the verb object order: first the action is specified, and then the selection. To specify the selection, the user has to indicate the entity (like character, word, or line), typically with an optional count, as in “three lines”, and a direction (forward or backward from the cursor position). Apart from problems with the size of the set of operations and the complexity of the resulting syntax, a problem here is the lack of easy visual confirmation that the selection specified is the one intended. For example, in the operation “perform deletion on the next five words”, the user never gets to see the selection, but only the result after the operation. Since it is easy to make mistakes in the specification of a selection, this is an annoying ergonomic problem.

The approach described only makes sense if the operations are viewed as a “linguistic” channel through which the user gives commands to an edit serf who executes them; at least for people with a natural-language background with the verb object order (like in English) there is something “natural’’ to the approach. (It is possibly less natural to users whose mother tongue has an object verb order, such as Japanese.) We contend that even “modeless” editors in this class are not truly modeless from the user’s conceptual point of view: next to the master serf mode of editing, there is a direct-manipulation mode of editing, namely when text is entered. The fact that this may be forced to fit formally within the “perform <action> on <selection>” master serf mode, namely by describing the entering of the character ‘A’, say, as “perform insertion-of-‘A’ on position-in-front-of-the-cursor”, is irrelevant.

From the direct-manipulation viewpoint, it is more natural to decompose the edit tasks into: “make <selection>”; “perform <action>” (namely on the current selection). Operations for moving the cursor are then subsumed by the selection operations. This was the first design decision taken. At the time we did not have experience with editors using such a decomposition; now this is of course a common approach. Next to the better uniformity of the mode of editing, we saw several further advantages. The first is that it becomes *impossible* to introduce unorthogonalities in the design process, since the factorability implied by the notion of orthogonality is built-in into the approach. Secondly, the factorability extends immediately to learning to understand the editor: the notion of a “syntax’’ for the edit operations either disappears, or whatever syntax remains becomes less complex. A final important issue is the possibility of visually confirming the selection. (The last point is in fact not only a nice possibility; with the direct-manipulation approach it becomes mandatory.)

An additional advantage, not to the users but to the designers, is of course also that the task of designing the editor has become factored into two fairly independent subtasks: designing the selection operations, and designing the action operations.

5 The design of the selection operations

5.1 The allowed selections

The “current selection” in a document will often be called the “focus”. In the presentation on the screen, the focus is highlighted.

In a purely unstructured text editor, a selection may consist typically of any portion between two positions in the text. In a strict-structure editor for textually presented objects, there is a (not specifically visually presented) deep tree structure, the syntax tree, and selections may consist only of a complete subtree of the full tree. The selection operations are then operations to “navigate” through the tree.

As noted before, making changes in a strict-structure editor may be awkward. There are, at least potentially, some further problems specific to the structure approach. A rather general problem here is that a given language does not have a single unique grammar. The structure of the syntax tree may therefore be different from the user’s mental view of the structure of the text. For example, here are two different fragments of BNF grammar that define the same “‘language”, namely a non-empty comma-separated list of “items”:

```
<item list> ::= <item>| <item list>, <item>
<item list> ::= <item>| <item>, <item list>
```

They correspond to different ways of imposing a tree structure on item lists, and would make a good deal of difference in the behaviour of a structure editor. In a language-defining report the choice between these two forms is neutral. Given that a choice between the two must be made — we argue below that both are bad — most users would prefer the second version.

Suppose that the second is chosen, and consider the task to bring the focus on the item `“i3”` in the list `“i1, i2, i3, i4”`, starting from a situation in which `“i2”` is selected. A sequence of tree-navigation operations is needed like: *select-parent; select-right-child; select-left-child*. To revert back to the focus on `“i2”`, the sequence would be: *select-parent; select-parent; select-left-child*. The asymmetry revealed by the implementations of these tasks, conceptually each other’s mirror image, makes it clear that this is unlikely to correspond to the mental structure a user will have. It would therefore be an obstacle in using (and learning to use) the editor, until the user has become so proficient that the sequences have been “chunked” into conceptually basic operations, in user interface terminology, when they have become committed to *muscle memory*. The other choice has of course the same problem.

Finally, in a strict-structure editor, it is impossible to focus on a comma from an item list, because it is not part of the (abstract) syntax tree — only of the more concrete parse tree. From the strict-structure point of view this makes perfect sense: there is no structurally sound operation that can be performed on such a comma. Supposing that there is also a notion of “item sequences”, being a semicolon-separated sequence of items, the change of the item list “i1, `i2`” into the item sequence “i1; `i2`” (without re-typing the items, which might be complicated expressions) is a major structural change, and accordingly awkward for the user to implement. The complexity of the following sketch of a possible implementation is not at all exaggerated.

- First, find a “‘harmless” place in which a construction that may contain an item sequence is allowed.
- Enter the template for that construction, and refine it till there is an item sequence with two empty slots for items.

- Bring the focus on the first item of the item list, make a copy in the copy buffer, focus on the first slot, and paste.
- Similarly for the second item.
- Focus on the whole item sequence and copy.
- Focus on the item list and replace it.
- Delete the construction that was created in the second step.

There are real-world examples that provide a metaphor for this. Given a sauce pan with gravy, and the task to replace the contents of the sauce pan with the gravy after straining, an extra temporary container is needed one way or another. The existence of a real-world example does not make it any less awkward, however.

We return now to the wish to "coalesce" structure-based and non-structured editing into a single mode. The key issue here is which selections are possible. If the ABC editor were to allow any selection between two text positions, it would be virtually undistinguishable from a structure-less editor. If only complete subtrees could be selections, it would be too awkward to use: there are too many useful potential selections that do not correspond to a complete subtree. On the other hand, there are also many selections which are very unlikely that a user would want to make. For example, in

```
IF (i+di, j+dj) in keys c:
   PUT count+c[i+di, j+dj] IN count
```

it is unlikely that a user would want to select the portion "`) in keys c: PU`". Considering which kinds of selections not corresponding to a full subtree were useful and therefore likely to be desirable from the user's point of view, we identified the following points. First, it is necessary to consider the parse tree rather than the abstract tree, in the sense that the skeletal parts of the various constructions (like the "`PUT`" and "`IN`" of a "`PUT`" command) should be explicitly represented in the tree. Otherwise, many useful selections (like the comma above, in order to change it into a semicolon) are impossible. In one respect, however, we take a more abstract view. In our trees, parents are not allowed to have a single child; if this is the case, the two nodes involved are fused into a single node. Secondly, it is then possible to represent *any* potential selection uniquely by the smallest collection of nodes in the tree such that the selection corresponds precisely to the "leaves" of the subtrees descending from these nodes. In strikingly many cases, these node collections turned out to be a (contiguous) segment of sibling nodes, i.e., nodes with the same parent node. Note that this includes all "empty selections", which correspond to an empty segment of siblings. Notable exceptions were, for example, a selection like "`i2, i3`" within the list "`i1, i2, i3, i4`", and "`a+n*`" in the formula "`a+n*p[i]`".

We turned this into a principle by postulating that the parse tree should be such that the exceptions would no longer be exceptions, but be covered as well by the sibling-segment criterion. This can be accomplished by a mild and in fact fairly common extension to the original BNF grammars. In a production rule of a strict BNF grammar, the right-hand side consists of a sequence of alternatives (separated by the choice operator "|"), each of which consists of a sequence of terminal or non-terminal symbols (where the concatenation operator is, traditionally, not explicitly represented by a symbol, but denoted by juxtaposition). Now choice and concatenation are two of the operators allowed in so-called regular expressions, but in addition a repetition operator (usually denoted by a postfix "*") is allowed, and, moreover, subexpressions may be parenthesised as in standard mathematical notation, so that the choice operator may occur within the operand of a concatenation. The mild extension referred to is now to allow the operations of regular expressions in their full generality in the right-hand side of a production rule. This does not change the formal expressive power of the BNF-type of grammar (they can still describe

precisely the context-free languages), but makes them somewhat more convenient as a formalism when defining a language. Notationally it is a true extension: each strict BNF grammar is allowed in the extended formalism, since its right-hand side has the form of a (particular kind of) regular expression. With this extension, the grammar rule for item-lists can be expressed as

```
<item list> ::= <item> ( , <item> )*
```

We use this extension now for another purpose, namely to redefine the notion of parse tree in the desired direction. (In fact, a redefinition is needed in any case, since the original definition depends on the restricted form for the right-hand sides.) Although there still is an asymmetry in the grammar rule above, it must not find its way into the tree. It is possible to give a precise formal definition of the new parse trees, but the idea is so obvious that we only sketch it. Let *N* be a nonterminal symbol. What possible children can a node labelled with *N* have in a parse tree? If, for a moment, we consider the nonterminal symbols in a right-hand side as terminal symbols, then for the original, strict BNF grammars, the possible sequences formed by the labels of the children of an *N*-node in the parse tree are exactly the sentences of the regular language described by (the regular expression which is) the right-hand side. Here, we consider a leaf node representing a terminal symbol (terminal in the grammar itself) to be labelled with that symbol. This way of viewing "traditional" parse trees gives us the definition for the extended BNF grammars: take, as the children of an *N*-node — and therefore all on the same level — again a sentence produced by the regular expression in the right-hand side. Now a potential selection like `"i2, i3"` consists of a segment of sibling nodes, and is therefore an allowed selection. For formulas, it is fairly easy to give a regular expression, ignoring the "structural information" provided by the priorities of the operators.

Of course, with this approach it is still, and even more so, the case that a given language does not have a unique grammar. Choosing the grammar has now become an even more important part of the editor design process. The first attempt was revised several times, mainly to incorporate some useful selection initially overlooked.

5.2 The selection operations

If a pointing device like a mouse is present, the ABC editor allows its use for quick positioning. Clicking at a position selects the largest structural entity starting at that position. Furthermore, the arrow keys, if any, can be used, always resulting in an empty selection.

Among the further selection operations we have "traditional" ones for structured tree-navigation. In the following, "moving to" or "selecting" a node (or a collection of nodes) is used as an abbreviation for "bringing the focus on the selection formed by the text covered by the (sub)trees descending from the node(s)". For "moving up" in the tree, there is an operation (**widen**). (The operation names have been chosen to correspond to the "visual" effect in the textual representation, rather than some abstract tree view, in which moreover the convention for what is called "up" and what is "down" is quite arbitrary. The notation used reflects the intention that such operations correspond — at least conceptually — to a single keystroke. On keyboards without function keys it is, unfortunately, necessary to map most operations to key combinations. Which key or keys are used to effect the (**widen**) operation and other operations may be customised by the user, taking account of what is convenient given the characteristics and peculiarities of the keyboard. On, e.g., the Macintosh implementation of ABC, the operations can also be selected from a menu. From now on, we will refer to operations as if they were keys on the keyboard. To narrow (move down), a possibility in conventional BNF grammars is to have as many operations as there are possible children. If there are more than just a few children, however, this

becomes inconvenient. With the extended-BNF approach there is no bound on the number of children. Instead, just two narrowing operations are provided: (**first**) and (**last**). To move to other children than the first or last one, the operations (**previous**) and (**next**) allow moving from a node to an adjacent sibling.

Thus far, the selection operations described can only be used to make selections that are also allowed selections in a strict-structure view (except for the possibility of selecting "skeletal" nodes). To enable the selection of a sibling segment with more than one node, the operation (**extend**) has been provided. Its meaning is: extend the current selection (being a segment of siblings) with the next sibling to the right. With these five selection operators, any allowed selection can be made. However, we also need an (automatic) "normalising step" for the view in terms of nodes. For according to the meanings given it could happen (by (**first**) followed by a suitable number of (**extend**)s) that the selected sibling segment would come to consist of all children of some node. Actually, the selection is not a collection of nodes (although it is represented internally thus in our implementation, but that is besides the issue), but a "portion" of the document. The same portion is the selection corresponding to the common parent of the siblings, and only the latter gives the unique representation mentioned in the previous subsection, since it is a singleton collection of nodes and therefore strictly smaller. As expected, there is no need to explain any of this to the users; it already conforms naturally to their mental model.

It would have been more uniform to provide a mirror version of (**extend**) as well, since the other operations also come in symmetrical pairs. However, the mirror version turned out to be in low demand, mainly due to the extension in meaning described in the next two paragraphs. So, as we wanted to keep the operation set small, it was removed.

In designing the functionality of any non-trivial system, the design process proceeds, starting from a vague sketch, through various refinement stages until the desired level of detail and precision is reached. In particular, various "features" (here: the editor operations) will initially be defined in terms of their behaviour for a "typical" situation. In the refinement process, the question will arise what meaning (possibly including "user error") to ascribe to them in an "atypical" situation that is not catered for by the original, loose definition. For example, what is the meaning of (**next**) if there is no next sibling because the focus is on the last child? In the design of the ABC language we used the *Fair-Expectation Rule* to answer such questions (though we do not mean to suggest that this design rule can be consulted like an oracle.) The question we asked is: given the *conceptual* high-level meaning of the feature, what might the intention of a user invoking it (within the bounds of fair reason) have been? If the answer was sufficiently unambiguous, we made sure that the meaning in the particular situation would cover that intention. Now in the design of a *programming* language, this rule has to be used very carefully. In that context, something has to be very unambiguous before it is "sufficiently" unambiguous. Before extending the meaning to previously undefined cases, the intended meaning has to be clear beyond all reasonable doubt. (To make the scope not entirely vacuous, we have to ascribe, for the purpose of wielding this rule, to our model user not any specific knowledge but a fair amount of clear-headedness and intelligence.) For an interactive editor, "reasonably likely" is a sufficient level of unambiguity. Unlike the programming-language case, the user always can see the effect immediately, so any misunderstandings (between the designers with their idea of "the" user and the actual user) can immediately be remedied. All operations can be undone with one keystroke, and typing errors are common enough to make any (mainly initial) "ill effects" of possible misunderstandings disappear in the statistical noise. Further, even if the meaning ascribed by us is not exactly that hoped-for by the user, there is a very fair chance that the effect is at least a step towards the goal the user wants to achieve.

Although the Fair-Expectation Rule, as a design principle, is of a general nature, we mention it here because the possibility and desire to extend an initial meaning manifested itself mainly with the selection operations. Except for (extend) for which this is already defined, we must answer for all cases what the meaning is of an operation when the focus is on a segment of two or more siblings. For (widen) the meaning is: focus on the common parent node of the siblings. This extends in a uniform way the meaning for a single-node focus. The meaning of this operation is in fact only a "user error" (resulting in a beep) if the focus is on the root node (thus selecting the whole document). Note that the "normalisation step" mentioned above is important for the meaning of this operation; in fact, this operation and (extend) are the only ones in the operation set of the ABC editor for which normalisation can make a difference. This is partly by design: for "guessing" extended meanings for the other operations, a useful trick was to consider their meaning in terms of an unnormalised representation. For the operation (first) on a multi-sibling focus, the meaning is: focus on the first of these siblings (and similarly for (last) of course). If the focus is on a (singleton, childless) leaf node, the operation focusses on the empty selection just preceding that node. In a situation in which there is no next sibling (next) is handled as if preceded by the least number of (widen)s needed to give the focus a right sibling. Thus, if there is anything to the right of the focus, the operation will move the focus to the right, with the left edge at the position of the previous right edge. Finally, (extend) when there are no further siblings to the right, will start taking in siblings to the left. Thus, the mirror effect of the very common operation sequence (first) (extend) (extend) … is obtained by (last) (extend) (extend) …

By "popular demand" two more selection operations have been added, namely (downline) and its mirror image (upline). The ABC language is easy enough to learn so that in a one-day course the attendants can be taught the language and get some hands-on experience with it. Unfortunately, editors are not easy to teach "in class", even not if as simple as the ABC editor. The interactive learning is indispensible, and takes some time. For the very first phase, it is easier for the novices to make changes on a line-oriented basis, simply retyping whole lines to make a correction. It is important to note here that ABC program lines tend to be short, and that the grammar is such that a line is always a valid selection. These two operations then select the whole next or previous line; they can be used as a complete navigation-cum-selection toolkit.

6 The design of the actions

Some basic editing tasks are to delete a selection, to replace its contents by another text, and to insert (add) new text. Of these three, strictly speaking only *replace* is needed: deletion can be modelled as replacing the selection to be deleted by empty text, and insertion as replacing an empty selection by new text.

This would, however, require a separate user action to signal the completion of the replacement action (as in "make <selection>"; "begin replace"; "enter <text>"; "end replace"); otherwise, deletion could not be expressed. This would mean that "end replace" would take a dual conceptual role: meaning *delete* if issued immediately after making a selection, and a *no-op*, annoying if required, otherwise. Better, then, to have an explicit (delete) operation.

It is clear now that insertion and replacement can be expressed (given deletion and the selection operations) in terms of each other. This can be done in a reasonably convenient way, as will be shown below, so only one of the two has to be provided as a primitive. For both possible choices it is not necessary to provide an explicit key-bound operation. The default meaning of the user entering text could be "insert <text> in front of current selection", at the same time narrowing the focus to an empty selection following that text. (This is required to make this work in a reasonable way if the action is repeated.)

This option we call *auto-insert*; it is the default in the *emacs* editor. Another possible default meaning is "replace current selection by <text>", again narrowing the focus as before. This we call *auto-replace*; it is the default in, e.g., most word-processors for the Macintosh.

Clearly only one of the two can be adopted. If insertion tasks dominate, auto-insert is more convenient, whereas auto-replace is preferable if replacement is the more common tasks. The following table shows the implementation of the two tasks for each of the two options:

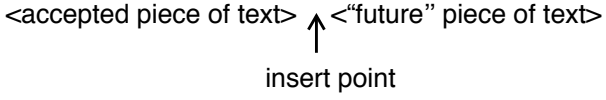
	if *auto-insert*:	if *auto-replace*:
to insert `<text>`:	`<text>`	(first) * `<text>`
to replace `<text>`:	(delete) `<text>`	`<text>`

The "*" following (first) means that the operation has to be repeated until the focus is an empty selection.

For the ABC editor we chose the auto-insert option. The rationale is as follows. It is hard to say which task type, insertion or replacement, is the more frequent one. In the "life cycle" of a document, initially insertion is (obviously) more common, but after some time replacements may start to dominate. Whether this actually happens may depend on the user's style of working. Some users happily throw a program together that they know to be full of errors, confident that they will be able to debug it. Other users may carefully develop their programs, and have little need of later changes. Although ABC does not enforce a style of working, we cared more for providing support for the careful user than the happy-go-lucky one; in fact, providing support for a structured approach to programming was the first objective of the whole project, and the ABC environment does not contain any "debugging tools". Accordingly, we chose the option that makes the composition task easier, namely auto-insert, rather than the one making modifications easier.

We did not perform controlled experiments to validate this decision. The difference in convenience, if any, is probably small, so that a rather large group of subjects would be needed to provide analysable data. Also, as indicated above, the best choice may be (high-level) task-dependent. One further advantage of auto-insert should be noted, namely that the now composite task "replace <text>" has a fixed implementation, which is therefore easily stored in "muscle memory". For auto-replace, the insertion task becomes composite, but requires a variable, context-dependent number of (first) operations. (In another project in which we needed a structure-based editor we copied many parts of the design of the ABC editor. However, since from our task analysis it was clear that replacement would be the dominant task there, we chose auto-replace as the default option for that editor, but providing an extra selection operation to select, in one go, an insertion point in front of the focus.)

While text is entered, the editor performs a continual incremental syntax check, on a character-by-character basis. The algorithms used are fairly complex, and a detailed exposition would go beyond the scope of this paper. Basically, in the entity under construction, the general situation can be sketched as



This situation is “acceptable” if there is some piece of text, such that if it is inserted between the accepted and the future pieces, the whole becomes a valid ABC diction (in the given context), and a character is accepted at the insert point if adding it to the already accepted piece results again in an acceptable situation. (This description is in two respects a simplification of the actual situation, as will be explained in the next section.) The main problem here was to decide which structural part of the text among several possibilities will be considered to be “the” entity under construction, and to give formal, algorithmically expressible, criteria for that choice. Our reasoning here was similar to that related above for the choice of the allowed selections, but the resulting criteria are not easy to formulate without the support of some formal-language theory. One of the criteria — actually a consequence of one of the main criteria — is however easily explained. Various pairs of “brackets” can occur in ABC expressions: “(…)”, “[…]” and “{ … }”. In a complete program they are always pairwise matched, with nesting allowed. We required then that brackets would also be matched in an acceptable *incomplete* program, and in any acceptable incomplete structural entity. This limits the scope of choice for the entity to be considered in a very effective way. Next to the “visible” brackets mentioned below, indentation also provides implicit bracket pairs: increase-indentation and decrease-indentation, and the requirement extends to these as well. The astute reader will have noted that the criterion implies that it is impossible to enter a single bracket, since that would turn an acceptable situation and therefore with matched brackets into one with ill-matched brackets and therefore unacceptable. This is correct: brackets must always be entered in pairs. This is described in the next section.

Given the notion of acceptability, not all attempted deletions can be allowed. A selection may only be deleted if the resulting situation is acceptable. Thus, it is not possible to delete one bracket of a bracket pair. In general, it is impossible to create, using the ABC editor, a bracket mismatch.

Next to entering text from the keyboard, it must be possible to copy or move an existing piece of text. The (**copy**) operation, when performed on a non-empty selection, makes a copy of it into an internal copy buffer. To paste such a copy into an insert point, we could have provided a separate paste operation, but instead we decided to let copy do double duty: if invoked on an empty selection, it functions as a paste operation. This is possible only by virtue of the preceding choice for auto-insert. Although this choice does away with one operation, it is debatable if this is truly a simplification of the kind aimed at by the *Economy-of-Tools Rule*. A (not perfectly matching) metaphor from the real world is provided by the case of a single push-button serving for the two tasks *light on* and *light off*. Not considering power failures and burnt-out bulbs, whatever the situation, only one of the two makes sense, so they can be mapped onto the same physical operation. In this example a unifying task description is “change lighting status”; such a unification is not so easily constructed for the double-duty (**copy**) operation.

An attempted paste will succeed when the resulting situation is acceptable; an equivalent description (in view of the blind typist’s rule) is: it will succeed when the text could have been entered (without causing error messages) from the keyboard.

There are a few more operations not yet mentioned, the most important one of which will be described in the next section. The other ones are mentioned here for the sake of completeness (and also to show that the entire operation set of the editor is indeed small). Next to (**undo**) mentioned before, there is (**redo**) also “unbounded “, which undoes the effect of an Undo operation. Together, they allow the user to “travel through time” in two directions. The operations (**record**) (a “toggle”) and (**playback**) allow the user to store a task consisting of a sequence of operations and repeat it several times. This is only useful for experienced users. Finally, there are some “meta-operations” that are not edit operations in the true sense, so that we just mention them: one for refreshing the screen if it gets corrupted (e.g. by transmission faults), one for “help”, and one for exiting

the document being edited, upon which some static-semantics checks are performed that do not make much sense for an incomplete document, such as a check for uninitialised variables.

7 Structured support for entering text

An orthogonal design decision to the choice of edit actions is the manner that text is to be entered. With the same reasoning as for the structured selection operations, we had two main requirements. We wanted more than just “flat” text entering; in particular, we wanted the editor to have some sort of template facility, so that it could offer help in the entering of syntactically-correct text, and reduce the amount of typing necessary for what is a somewhat wordy language. On the other hand, as pointed out earlier, we still required the option of “flat” text entering (referred to above as the blind typist’s rule), so that the user is not forced to use any template mechanism. Above all, these two options had to be available in the same mode. It followed that “flat” text entering had to drive the template selection.

Once we had seen this consequence of the requirements, the direction in which to proceed was clear. The solution used was to let the user type in text, and the editor would then guess ahead based on what had already been typed, supplying “suggestions” for what was to come, which the user may then accept, or choose to ignore and carry on typing in the same way. These suggestions are then templates, with holes where other pieces of text still have to be filled in.

A text-driven approach to template selection was also implemented in the Poe editor [3]. An important difference with our solution is that in Poe the level of control is formed by the tokens (like “`IF`” or “`WHILE`”), whereas our mechanism is driven at the level of the characters. Another difference is that Poe allows only strict-structure selections.

Here is an illustration of the suggestion mechanism in action. When the user is using the editor, the holes are represented on the screen by a question mark. In view of our target users, we did not feel tempted to represent the hole by a cryptic indication of the grammatical type of construct expected; our experience is moreover that users who happen to be computer-scientists also do not perceive the absence of that information as a lack. So the first thing the user sees is such a hole:

```
?
```

Typing a `W` at this point causes the editor to give a suggestion template. Since only a command is acceptable at the current position, and since `WRITE` is the most common command beginning with a W, the editor “guesses” that, and displays:

```
W?RITE ?
```

The second hole shows that WRITE has one parameter that must be filled in. Since ABC has a keyword structure, and the keyword skeleton of commands is always with capital letters, we extended the guessing principle in the sense that a lower-case w is also accepted and gives the same effect.

To accept the suggestion, the user can use the **accept** operation, which then positions the focus on the parameter hole:

```
WRITE ?
```

If the user types an open bracket at this position, the editor suggests the closing bracket:

```
WRITE (?)
```

Similarly, if the user had typed a string quote, the editor would have suggested the matching closing quote:

```
WRITE "?"
```

This explains at the same time the question, left unresolved in the previous section, how the acceptability of the situation is maintained if the user types a bracket. If the user next types `Hello!`, the editor displays now:

```
WRITE "Hello!?"
```

Now the user can accept the (suggested) closing quote (which means that the operation **accept** skips over the quote), but the blind typist's rule demands that the user can equally "flatly" type the quote, thus having typed altogether the sequence of characters `"Hello!"`. This shows that the editor's rule for determining the acceptability of a character has to be somewhat more complicated than the version stated in the previous section.

The `PUT` command is ABC's assignment command, and has two parameters. Typing a `P` at the initial position then shows:

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

and **accept** brings the focus on the first parameter:

```
PUT ? IN ?
```

Typing `0` followed by **accept** then gives:

```
PUT 0 IN ?
```

where the name of the variable may be typed, followed by a newline. The syntax of the second parameter does not allow it to start with a digit, and accordingly a digit at that position is refused by the editor.

If the user does not accept a suggestion but continues typing, the editor always matches the suggestion incrementally to what has been typed so far. So having typed a `W`:

```
W?RITE ?
```

if the user then types an `H`, the editor "guesses" a `WHILE` command, and the suggestion changes to:

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

which shows that `WHILE` has two parts that must be filled in: a condition and a command suite.

It is this process of incrementally matching the suggestion to what has been typed so far that allows flat text to be typed: having typed a `W` and then an `R`, the suggestion still remains for `WRITE`:

```
WR?ITE ?
```

Indentation is significant in ABC, and is used to show grouping of commands, rather than using BEGIN-END or similar. If the user has accepted the suggestion for `WHILE`, and then types `a > 0` followed by a newline, the editor automatically supplies the indentation:

```
WHILE a > 0:
    ?
```

Now the user may type any number of commands here, each followed by a newline, and the editor will continue to indent at the same level. An extra newline will take the indentation out one level:

```
WHILE a > 0:
```

```
        PUT a-b IN a
        WRITE a
    ?
```

The treatment of indentation may be viewed as an exception to the blind typist's rule: the indentation itself cannot be explicitly typed in, but end-of-indentation has to be explicitly indicated. The convenience of this treatment of indentation is, however, so large that we have not considered giving it up. A unifying description for the effect of typing newline is: "go to the nearest level of indentation left of (the starting point of) the present focus". It turns out that this is perceived as quite natural by surprisingly many users; it would seem that for them the "go left" aspect of newline conceptually dominates the "go down" aspect. In fact, we have on a few occasions seen "rank tyro" users, while being taught the editor's use hands-on under guidance, spontaneously, without having been told, hitting newline twice to decrease the indentation level; when asked how they knew or guessed this, their reaction was invariably a surprised '"how else?". An aditional benefit over an explicit indent operation is the ability to stutter, especially since newline is a large key and already firmly in muscle memory.

Suggestions work not only for the built-in commands of the language, but also for commands defined by the user (which have the same style of keyword structure as the built-in commands). This is required of course from the point of view of fair expectation for the user, but in fact had an unanticipated beneficial effect, namely that users were not frightened of using long descriptive names for their commands, since they generally needed only a few key strokes to use them:

```
G?ET LIMITS FOR ? AND ?
```

The suggestion mechanism works best if the language being typed is close to LL(1), since then the editor can guess ahead with more probability of being correct. But even where there are several possibilities, the editor can still be of help. One technique that we used was the following: if after each keystroke there is still a number of possible suggestions that match what has been typed, select another suggestion to the one just used, so that the suggestions get run through in a cyclic fashion. For instance, to type the command `REPORT` (which occurs very infrequently) you get the following. Type an `R`:

```
R?ETURN ?
```

an `E`:

```
RE?AD ? EG ?
```

and a `P`:

```
REP?ORT ?
```

Initially we considered the possibility of providing a form of adaptiveness, in the sense that suggestions would be based on the frequency with which the user had used the constructs being suggested. We decided, however, that the advantage for the user of being able, after having obtained some experience, to predict the editor's behaviour, and being able to type "head down" if required while still taking advantage of the suggestions using memory-muscle chunks like `W` (**accept**) for WRITE, and `WH` (**accept**) for `WHILE`, was preferable to saving probably no more than a fractional number of keystrokes per command on the average.

8 Experience and future plans

Like the language, the editor has been designed iteratively. The incremental suggestion mechanism was already described in the first, preliminary, design of the editor [9], but the navigation operations given there are strict-structure ones. An initial pilot study was

programmed on top of the programmable editor Emacs by Dick Grune. After that there have been two major iterations. The first one [11] resulted in a prototype that already implemented all the major ideas, including in particular the extend operation. It was written in 1984 by Guido van Rossum, and we were so satisfied by its functionality (and speed) that it was released for public use. For the second iteration an important concern was formed by the modifications made necessary by the last iteration in the language design. Apart from this, the changes in the functionality were minor ones. The present version of the ABC editor was realised by Timo Krijnen, working from Van Rossum's prototype.

The editor is used by a wide range of users, from school children to professionals, and we are satisfied that the basic design is a good one; the editor is easy to learn (with a good "learning curve") and offers a lot of much appreciated support in the construction of ABC programs.

When the suggestion mechanism was invented, we were worried about the possibility that the ever-changing suggestions would be distracting to novice users. This has not turned out to be a problem.

The current implementation of the module of the ABC system for executing programs has grown, by evolution, from a prototype written in 1981. Although the received wisdom in software engineering is that building upon a prototype is an ill-advised approach, we have never regretted it. For the editor we followed the same approach, but here, alas, we feel with the power of hindsight that we took the wrong decision. A major problem with the current version is that it is still too *ad hoc*. Ideally the whole editor should be parametrised with a syntax of the language and no more. Unfortunately, much ABC-specific knowledge is hard-wired into the code. Worse, several design decisions are "spread out" over the code, making it hard to ensure uniformity of its behaviour, and to experiment with different policies. Although the general principles appear to be fine, there remain many small issues for which the editor could do with a further polish.

A major policy change that we would like to experiment with is to let the focus moves relate more to the abstract syntax tree rather than the concrete tree. For instance, to get from:

```
PUT a+b IN a
```

to:

```
PUT a+b IN a
```

the user has to type **first** (taking the focus to `PUT`) and then **next**. However, in practice, the user almost always wants to work on the expression, and almost never on the keyword `PUT`. It would be far more useful if the **first** operation took the user to the first abstract sub-node, in this case the expression, and similarly, if **next** took the user to the next abstract-tree sibling. Some way of reaching the skeletal parts must still be provided, and the arrow keys could serve for that.

Another problem with the current version is that while it lets the user edit in both structured and unstructured ways, there are certain unstructured changes it refuses (on the grounds that the result would be "unacceptable") which are nevertheless useful and conceptually reasonable. In some cases the problem could be remedied by modifying the extended BNF grammar, but in most cases it is the notion of acceptability that is still too restrictive. The presently implemented notion was to some extent obtained by (insufficiently) exhaustive consideration of all possible kinds of situations, instead of being derived fully from formulated principles.

However, rather than go on to another iteration of the ABC editor, we are now working on the design of a generic structure editor, also for graphical and mixed-media documents, and allowing hypertextual facilities and "active" documents and elements within

documents. For the techniques to describe document structure and presentation we are leaning heavily on those developed for the Grif editor [14], but the way the user interacts with the new editor will be based on the principles learned from our experience with the design of the ABC editor, but generalised [13].

9 References

[1] Rolf Bahlke and Gregor Snelting, The PSG system: from formal language definitions to interactive programming environments, *ACM Transactions on Programming Languages and Systems* **8**, 4, October 1986, 547—576.

[2] P. Borras *et al.*, CENTAUR: the system, in *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (P. Henderson, ed.), joint issue: *Soft. Eng. Notes* **13**, 5, November 1988, and *SIGPLAN Notices* **24**, 2, February 1989, 14—24.

[3] C. N. Fischer *et al.*, The Poe language-based editor project, in *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, joint issue: *SIGPLAN Notices* **19**, 5, and *Soft. Eng. Notes* **9**, 3, May 1984, 42—48.

[4] Leo Geurts and Lambert Meertens, Keyword Grammars, in *Implementation and Design of Algorithmic Languages* (J. André and J.-P. Banatre, eds) 1—12. IRIA, Rocquencourt, 1978.

[5] Leo Geurts and Lambert Meertens, Designing a beginners' programming language, in *New Directions in Algorithmic Languages 1975* (S. A. Schuman, ed.), 125—138. IRIA, Rocquencourt, 1976.

[6] Leo Geurts, Lambert Meertens and Steven Pemberton, *The ABC Programmer's Handbook.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-000027-2.

[7] Merle P. Martin and William L. Fuerst, Using computer knowledge in the design of interactive systems, *International Journal of Man-Machine Studies* **26**, 1987, 333—342.

[8] Lambert Meertens, Issues in the design of a beginners' programming language, in *Algorithmic Languages* (J.W. de Bakker and J. C. van Vliet, eds), 167-148. North-Holland Publ. Co., Amsterdam, 1981.

[9] Lambert Meertens, *Draft Proposal for the B Programming Language*, Mathematical Centre, Amsterdam, 1981. ISBN 90 6196 238 2.

[10] J. R. Morgan and D. J. Moore, Techniques for improving language-based editors, in *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, joint issue: *SIGPLAN Notices* **19**, 5, and *Soft. Eng. Notes* **9**, 3, May 1984, 21—29.

[11] Aad Nienhuis, *On the Design of an Editor for the B Programming Language*, Report IW 248/83, Mathematical Centre, Amsterdam, 1983.

[12] Steven Pemberton, An alternative simple language and environment for PCs, *IEEE Software* **4**, 1, January 1987, 56—64.

[13] Steven Pemberton, The Views Application Environment, Report CS-R9257, CWI, Amsterdam, December 1992.

[14] Vincent Quint and Irène Vatton, Grif, an interactive system for structured document manipulation, in *Text Processing and Document Manipulation* (J.C. van Vliet, ed.) 200—213. Cambridge University Press, 1986.

[15] T. Teitelbaum and T. Reps, The Cornell Program Synthesizer: a syntax-directed programming environment, *Communications of the ACM* **24**, 9, September 1981, 563—573.

[16] Jeroen van de Graaf, *Towards a Specification of the B Programming Environment*, Report CS-R840, CWI, Amsterdam, 1984.

[17] R.C. Waters, Program editors should not abandon text oriented commands, *SIGPLAN Notices* **17**, 7, July 1982, 39—46.