Edgar H. Sibley
Panel Editor

Lara, a text editor developed for the Lilith workstation, exemplifies the principles underlying modern text-editor design: a high degree of interactivity, an internal data structure that mirrors currently displayed text, and extensive use of bitmap controlled displays and facilities.

CONCEPTS OF THE TEXT EDITOR LARA

J. GUTKNECHT

Text editing exemplifies the type of computer application that exploits a computer's nonnumerical capabilities and characteristics, specifically, its capability to store and display data, handle abstract and complex data structures, and react immediately on command. Indeed, a steadily increasing degree of interactivity has characterized the development of new computer applications over the years. In fact, the degree of interactivity of a system in large part determines the way it is used and even its underlying architecture: It was essentially the desired increase in interactivity compared to conventional batch processing that led to the design of terminal-based real-time processing systems.

Text editing, and editing in general, can be seen as a further milestone in the development of increased interactivity and real-time processing. Editing processes are typically composed of a great number of elementary actions whereby the computer assumes the role of a *tool* rather than that of a machine. Looked at from this point of view, reaction delays appear unnatural, and single-user or personal computers best comply with the requirement of minimizing them.

It is not surprising, therefore, that the architects of personal computers (or *workstations*, as they are more appropriately called) have adapted expertly to the demands of text-editing applications. In fact, modern workstations, with their mouse pointing devices, high-resolution displays, and printers, are ideally suited for editing applications. Despite (or should we say because of) their conceptual simplicity, these devices have fundamentally influenced the discipline of interactive editing, in particular, the kind of command specification and variety of possible forms of representation occurring in a document. Put another way, the new ingredients in the modern workstation have inundated software designers with an abundance of possible applications and thus imposed on them the burden of applying clear and consistent design concepts. Our most important insight, in this regard, was that successful use of an interactive text editor necessitates a certain basic appreciation of its underlying concepts—particularly its data structure and command hierarchy.

In the case of Lara, the goal was to develop a versatile text editor for the workstation Lilith [6]. The Lara text editor that emerged in 1984 is a successor to the Andra system [2]—the outcome of a project initiated in 1981 by the Institut für Informatik of ETH Zurich and inspired by Xerox's Bravo editor [4].

Andra was strongly influenced by Xerox's Tioga editor [5], which was derived from Bravo and introduced the notions of *style* and *text node*. Andra adapted both these notions and also their shortcomings. Specifically, since the style concept based document "looks" (e.g., fonts, line widths) on a style file, interchanging docu-

ments with different style files could produce bizarre results. The difficulty with the text node in Andra was that it led to an arbitrary tree structure where each node had an arbitrary set of attributes that sometimes contradicted those of other nodes. Moreover, there was some difficulty in locating and identifying nodes containing certain style attributes when they were needed at a later date.

In contrast to Andra, Lara documents are self-contained in the sense that they do not depend on a style file. Instead of applying style elements, Lara introduces a new method of formatting text that consists of copying attributes from one place on the display to another, be it in the same document or another. Sample documents can be created to achieve format consistency within a group of documents.

Furthermore, by restricting the tree structure of a document to four universal levels that are each associated with a distinct set of formatting attributes, Lara provides a close connection between displayed text and internal data structure. This means that the internal data structure is completely determined by the currently displayed text: It is not explicitly dependent on the history of the editing process.

It is these characterizations that distinguish Lara from other similar new-generation text editors, for example, Bravo and MacWrite [3]. Although Lara is exclusively a text editor, most of the design concepts incorporated in it are equally applicable to other kinds of interactive systems. In this paper, we examine first the *static* elements of the Lara text editor (i.e., its data structure), then the *dynamic* or text-editing aspects, and finally its *program* structure.

DATA STRUCTURE

Although at first glance the structure of a text may seem uninteresting and trivial, the very opposite is true. Disregarding for a moment the aspect of representation, text can be seen as a sequence of *characters* that are manipulable by two canonical operations: *delete* and *insert*. Text can be inserted either from the same or a different text file, or via the keyboard. Although in general both insert and delete operations involve shifting a part of the original character sequence, we circumvent the actual moving of data in computer memory (following a technique introduced in the Bravo system) by introducing a more sophisticated and abstract structure to describe the text.

In Lara, a contiguous subsequence of a text file is known as a *piece*. A *piece descriptor* is then a record indicating the file, starting position, and length of the respective subsequence. The key idea is to describe a text about to undergo an editing process by a *chain* of piece descriptors. Initially, the chain consists of a single element. The effect of the delete and insert operations on a chain is illustrated in Figure 1 (p. 944).

In this way, the editing operations affect a suitably dynamic data structure that describes the current state of the text rather than the text itself; the actual text remains untouched during the editing process. As a result, the text file need not be read into memory at the beginning. Instead, a new text file is normally explicitly created upon termination of the editing session.

Structural Units

Although a text is defined as a sequence of characters, it actually has a much richer structure whose intrinsic hierarchy is described by the notions of *chapter*, *paragraph*, *sentence*, *word*, and *character*. On the other hand, from the formatter's point of view, a text appears as a sequence of *pages*, where each page consists of a sequence of *lines* and each line consists of a sequence of characters.

Since a text is intimately connected with the form of its *representation*, it is certainly desirable that the representation reflect the inherent structure of the text. In fact, it is possible to establish natural correspondences between intrinsic and formatter notions, that is, between chapter and page and paragraph and line. More precisely, in Lara we require that each chapter be represented as a (context-free) sequence of pages and each paragraph as a (context-free) sequence of lines; that is, we require each chapter to start on a new page and each paragraph to start on a new line.

In Lara, document, chapter, paragraph, and character are the *structural units* that comprise a document. In extended Backus normal form (EBNF) notation,

```
Document  = {Chapter}.
Chapter   = {Paragraph}.
Paragraph = {Character}.
```

Formatting Attributes

Each structural unit consists of a sequence of structural units of the next lower *level*, and each *structural unit type* is connected with a set of *formatting attributes* that are applicable to instances of that type. The corresponding class decomposition of a set of typical attributes is given in Table I (p. 945). This link between formatting attributes and structural units is of paramount importance in Lara. Notice that the domain of an attribute naturally extends to sequences of units of its basic type and henceforth to units of any higher level. Note further that the notions of sentence and word do not have formatter-oriented counterparts such as page and line and are therefore unconnected to specific formatting attributes: They do not serve as explicit objects to be handled by the Lara editor, but are, nonetheless, implicit objects influencing line breaking and line adjusting.

The *format* of a text is defined as the assignment of values to the formatting attributes (also referred to as looks) of its structural units. The format specifies the details of representation and is thus an integral part of the text itself. Consequently, the data structure describing the text must be extended to incorporate formatting details. In fact, a tree hierarchy of descriptors of its structural units reflects the current state of a Lara document, as shown in Figure 2 (p. 945). Each descriptor describes either a single structural unit or a *group* of structural units: A group is a sequence of structural units of any level with the same looks.

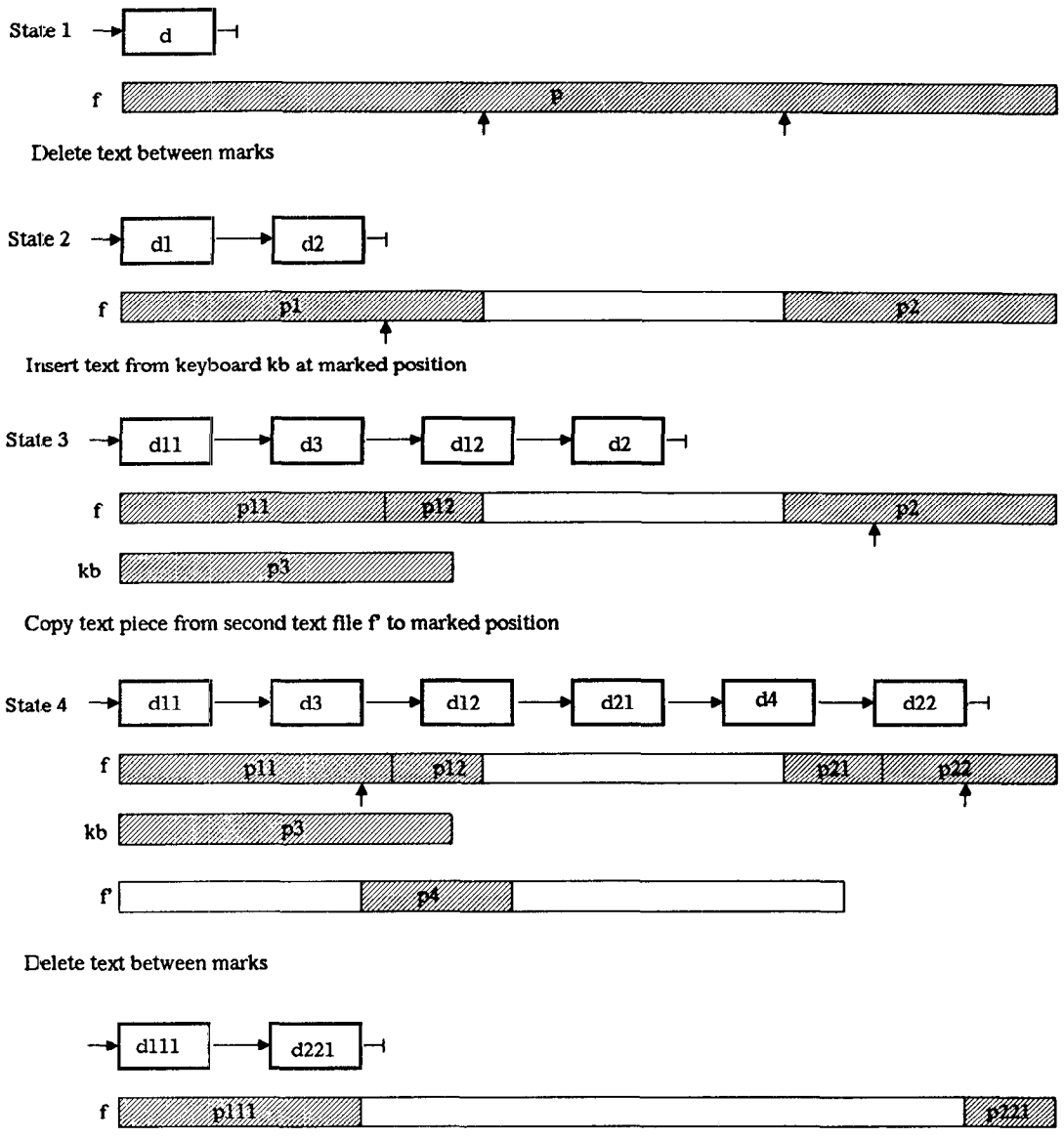


FIGURE 1. Sample Editing Sequence on Text File *f* and Development of the Piece Chain

Formatting Operations

Formatting operations can be divided into two categories: The first consists of operations that *create* new and *cancel* existing structural units, while the second includes *assignments* of new attribute values or combinations of values to structural units. Exactly as contents-oriented textual operations like insert and delete influence the structure of the piece chain, formatting operations affect the format tree. While formatting operations of the first category change the global structure of the tree, second-category actions lead to a modification of individual descriptors. However, the depth of the tree—corresponding to the number of different levels—remains constant.

The overall data structure of Lara presents itself as the sum of two independent parts: a piece chain and a format tree. Formatting operations clearly have no effect on the piece chain, but on the other hand, textual operations may involve the format tree (see Figure 3) (p. 946).

Document Files

The final status of the dynamic data structure that is maintained during the editing process must ultimately be recorded on a *file*. Each Lara document file is self-contained in the sense that the values of all formatting attributes occurring within the document are included in the *format header* of the document file. Apart from

TABLE I. Structural Units and Their Respective Attributes

Level	Structural unit type	Attribute
4	Document	Document type (unformatted ASCII, formatted)
3	Chapter	Page header Top margin Bottom margin Vertical formatting (page adjusting) Vertical tabulator positions
2	Paragraph	Left margin Right margin Horizontal formatting (line adjusting) Horizontal tabulator positions Line lead Paragraph lead First-line indentation
1	Character	Font (family, type, size) Underlining Capitalizing Spacing Vertical offset

font files, which are referenced by name, a Lara document file does not refer to any other information such as styles.

The syntax of Lara document files reflects the "preorder" representation of the corresponding format tree:

```
TextDocument   = FormatHeader Text.
FormatHeader   = DocumentDescriptor
                 {ChapterPart} ChapterPart.
ChapterPart    = ChapterDescriptor
                 {ParagraphPart} ParagraphPart.
ParagraphPart  = ParagraphDescriptor
                 {CharacterDescriptor}
                 CharacterDescriptor.
Text           = {Chapter} Chapter.
Chapter        = {[VTAB]Paragraph}
                 [VTAB]Paragraph.
Paragraph      = {[HTAB]Character} EOL.
```

where

```
VTAB = vertical tabulator (13C).
HTAB = horizontal tabulator (11C).
EOL  = end of line (36C).
```

In the Lara data structure, we see epitomized the inclusion of formatting details in the interactive editing process and the concomitant transition from constant to variable formats that characterizes recent progress in computerized text processing. The inclusion of interactive formatting operations requires a sophisticated data structure to describe the current global state of a text. The conventional method used by "off-line" formatters, namely, to *encode* formatting information directly into the character stream, is obviously inappropriate.

EDITING

The convenience a computer represents as an editing tool is related in great measure to the convenience of its input and output devices. Recently, bitmap controlled high-resolution displays and printers have opened up new horizons so that, instead of displaying individual objects (e.g., characters or lines), these devices simply reproduce a two-dimensional array of bits

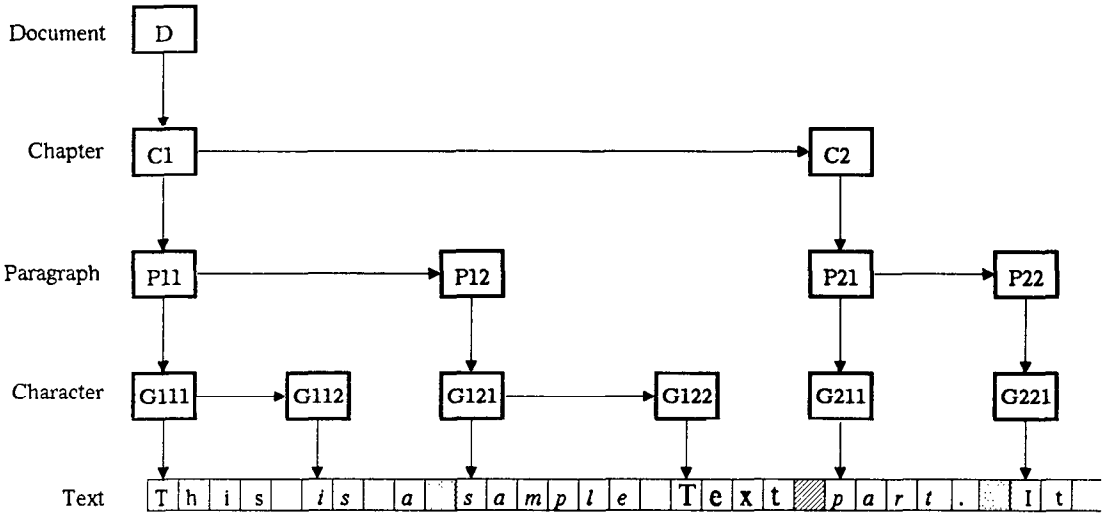


FIGURE 2. Format Tree

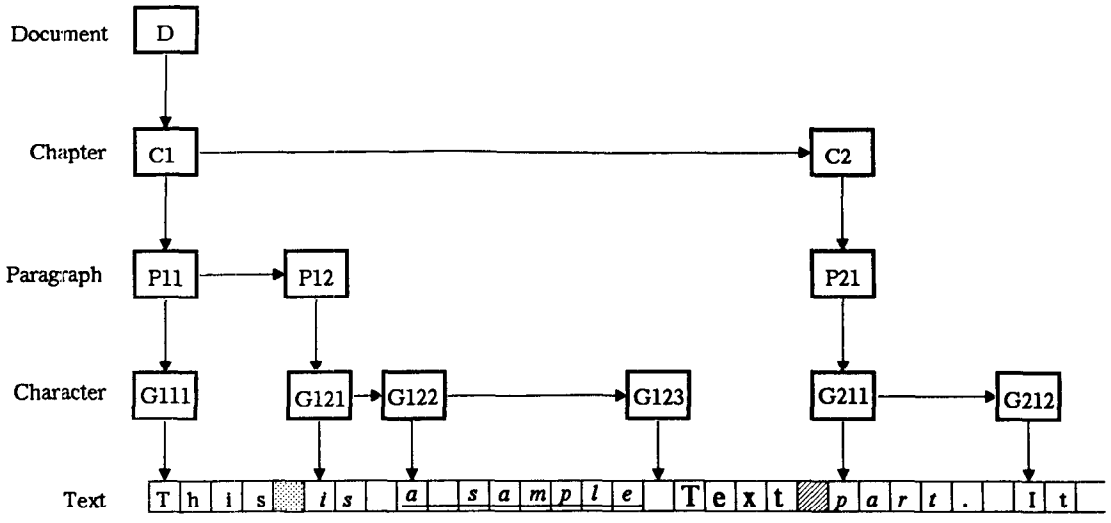


FIGURE 3. Effects of Formatting Operations on the Format Tree Shown in Figure 2

on the output medium, where each bit that is set corresponds to a black dot.

As a result of this technique, the display processor need no longer be aware of the *kind* of displayed data. Assuming the bitmap is properly loaded with *raster data*, it is perfectly possible to display simultaneously any combination of graphic and textual data, with characters appearing in different styles at precisely precalculated positions. A second consequence of the bitmap technique, or, more appropriately, the unification of display and print techniques, is the intimate relationship between displayed and printed data—a not insignificant matter. This relationship constitutes a prerequisite for obeying the recommendable rule "what you see is what you get."

Lara makes extensive use of these bitmap controlled displays and facilities. Each document is displayed within its own *window*—a rectangular and titled frame. In each window, one or more possibly noncontiguous portions of a document can be displayed in one or more *subwindows*. If more than one document is open at the same time, the different windows may even overlap each other. A typical window layout on the display is given in Figure 4.

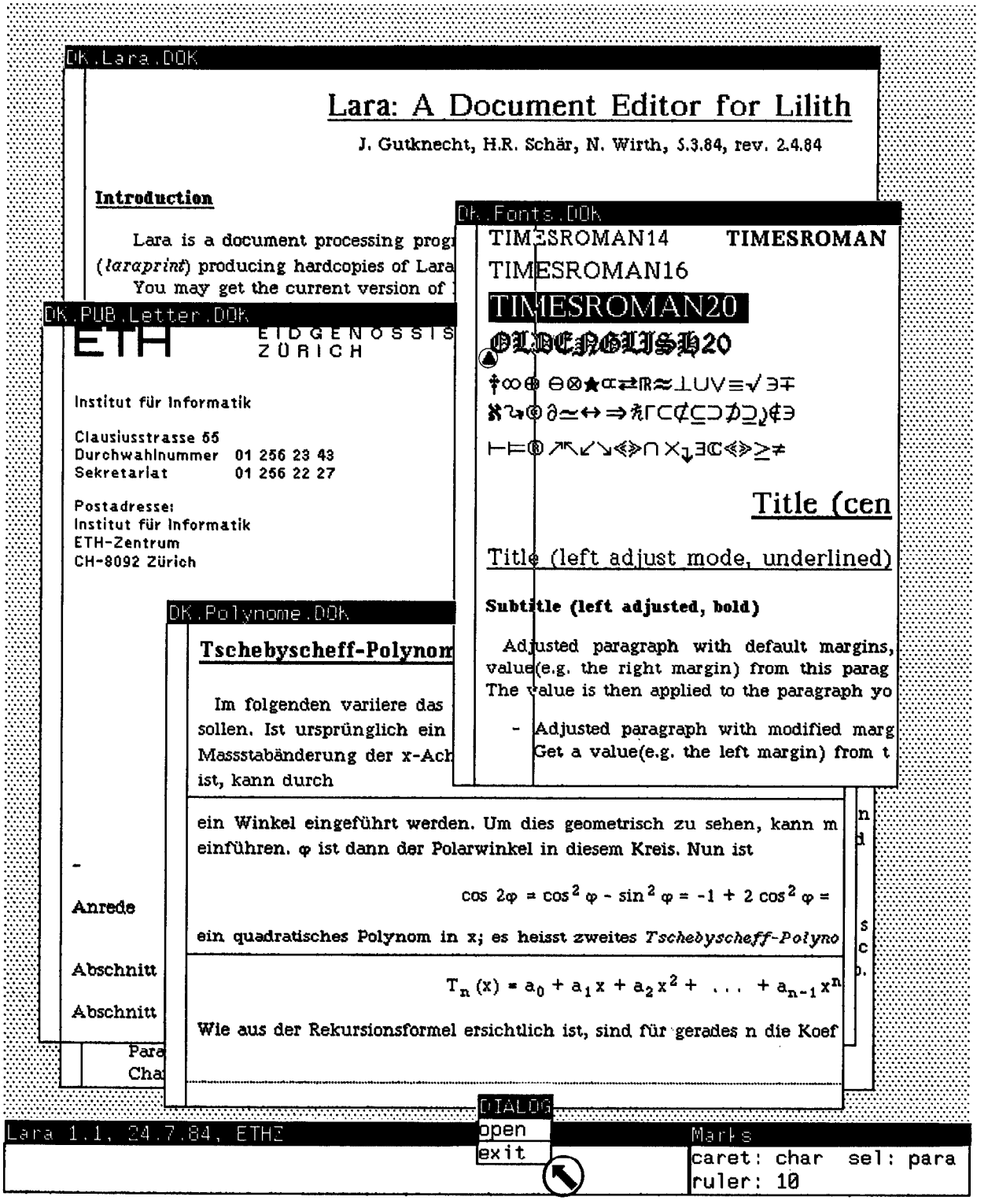
A second pillar of the modern text editor is the pointing device known as the *mouse*. The movement of the mouse on the desk is translated into a corresponding movement of the cursor on the display. The Lilith mouse or Lara mouse is equipped with three keys that are defined in terms of their main use: the P (pointing key), M (menu key), and S (selecting key). The agility of the mouse enables it to play a crucial role in the specification of actions. In combination with the *menu technique*, the mouse allows for convenient command input: If the M-key is depressed, the Lara *command menu* (Figure 5, p. 948) appears at the cursor where the eyes generally rest. A selected command is carried out as soon as the respective key is released.

Lara Commands

Proceeding with the same definitional style used in the previous section, we call the contents-oriented commands presented in Figure 5 *textual commands*. As with all Lara commands, they assume their operands to be previously defined. There are two types of operands: the *selection* and the *caret*. (A list of the textual commands and their operands is given in Table II, p. 948.)

The selection indicates a currently selected structural unit or contiguous sequence of structural units. The caret *separates* two structural units within the text. The selected text portion is marked by inverse video, and the caret is a small triangle (see Figure 4). To make a selection, the S-key must be clicked at the desired location. The number of clicks determines the structural level: Selecting a paragraph requires a double click, and selecting a chapter a triple click. If the S-key is kept depressed while dragging the mouse, a whole sequence of corresponding units can be selected. An analogous procedure applies to the caret when pressing the P- instead of the S-key. A single click of the P-key places the caret between two characters, a double click places it between two paragraphs, a triple between two chapters, and so on.

The Copy and Move commands are *generic* in the sense that their exact effect depends on (the structural type of) their operands. All attribute values of the selected text units up to the level of the destination (i.e., the level of separated structural units) are transferred. Attribute values of higher levels are inherited by the inserted text from the environment. If, for example, the Copy command is given in the case of a selected paragraph, and the caret is placed in the interior of that



Note: The actual screen color in Lara is black and white. In Figures 4–10, the color has been added for emphasis—*Ed.*

FIGURE 4. Typical Layout of Lara Document Windows on the Display

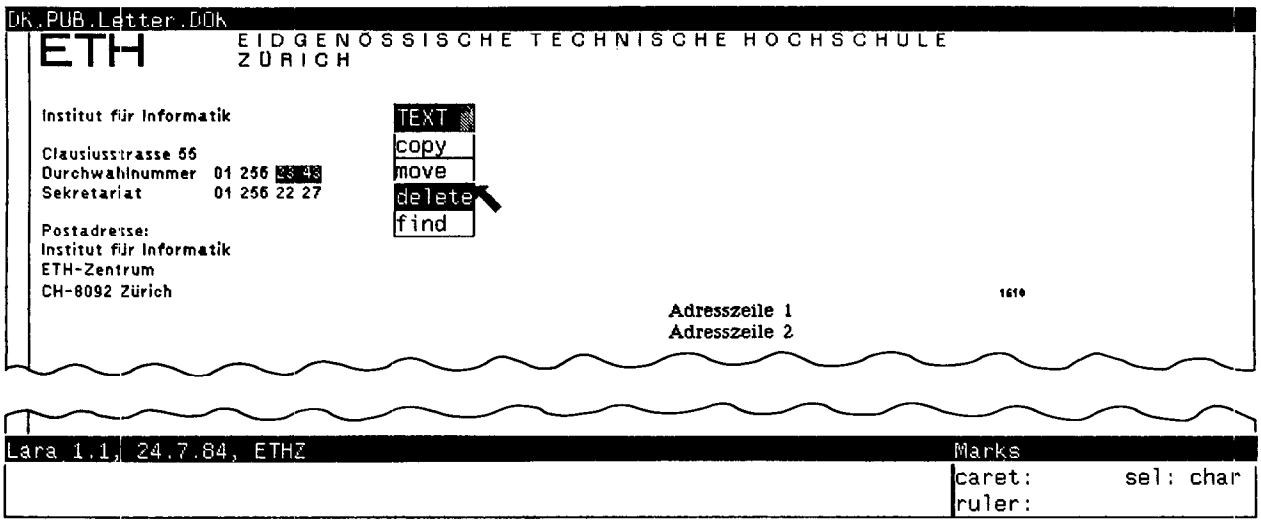


FIGURE 5. The Lara Command Menu

paragraph, the font, underlining, and vertical offset attributes are transferred, while the mode of adjusting, the left margin, and the line width are inherited from the environment. However, if the caret separates two paragraphs, all attribute values are transferred. Figures 6a–d (pp. 950–951) illustrate these respective cases.

If the selection does not overlap with structural units of a higher level (compared to its own level), the Delete command is a *local* operation: It does not impact the remaining text. When the selection does overlap, however, it may have *global* side effects. In fact, if partial structural units remain after deleting the selected text, the units are merged automatically (see Figure 7, p. 952).

So far, we have ignored the most creative action, namely the input of new text via the keyboard. This action is closely related to the Copy operation. In fact, we view it as a variant of the latter, where the selection is replaced by the next character arriving from the keyboard and supplied with the *caret attributes.* Character attribute values are assigned to the caret as follows: If the caret is located at the beginning of a paragraph, its attributes assume the *default values* as declared in a so-called *user profile.* Otherwise, the caret adopts the attribute values of the character immediately preceding it. Later, we will generalize this rule.

TABLE II. Lara's Textual Commands and Operands

Command	Selection	Caret
Copy	Text to be copied	Destination
Move	Text to be moved	Destination
Delete	Text to be deleted	—
Find	Text to be found	Starting position

Lara also recognizes a few control characters. The DEL code deletes a single character (preceding the caret) within a paragraph. Two codes, EOP and EOC, indicate the end of a paragraph and the end of a chapter, respectively. If one of these codes is typed, a new structural unit of the respective type is created. A chapter end is displayed by a dotted line. Finally, a TAB indicates a jump to the next tabulator position. Note that the set of tabulator positions is an attribute of paragraph.

Formatting

The full power of the Lara command menu is still not exhausted. In fact, we have discussed so far only the first of several partially overlapping *menu cards.* A new menu card appears when the cursor is moved beyond the right edge of the shaded region of the current card. Each card contains a logically related set of entries. The first card we have seen contains the textual commands (Figure 5). The remaining menu cards are connected with *formatting actions*, or, more precisely, the assignment of attribute values to selected text. The menu entries correspond to the attributes of the current selection level (i.e., the set of attributes, and hence the menu entries, is determined by the *level* of the selection). Figure 8 shows the available set of attributes for each respective structural unit. As suggested in Figure 8 (p. 953), a separate card, called a *submenu*, is associated with each symbol. The entries in the diverse submenus refer to individual attributes.

Having selected an attribute via the menu, a *parameter* value must be specified; one way of doing this is by typing a number or a font name. If just the RETURN key is hit, a corresponding default value is assigned. Here, however, Lara introduces a second, particularly

attractive and powerful method of specifying parameter values: That is, if, instead of hitting a character key, the P-key is clicked at a text location within any document window on the display, the attribute value is taken from that text (Figures 9a and b, p. 954).

This method immediately generalizes to *sets* of attributes. Suppose, for example, that a nonformatted paragraph and the Paragraph symbol in the menu are selected. Then, if the P-key is clicked within an appropriately formatted paragraph, all that paragraph's looks, including adjusting mode, left and right margins, and leading space, are immediately copied to the selected paragraph as shown in Figures 9c and d (p. 955).

Up to now we have discussed only *re*formatting, that is, assigning new attribute values to existing text units. Although this is the most frequently used formatting method, it is nonetheless desirable to be able to specify attribute values *in advance*. To do this, we define the menu as being related to the caret, if no other alternate selection is made. That is, the valid attributes of the caret depend on its location, as shown in Table III (p. 956). If no value is explicitly assigned, caret attribute values are defined as follows: If the respective structural unit is the first within the next higher level, default values are taken; otherwise, the caret inherits the values of the preceding unit. For example, to create a new title at the current end of a text with default looks (where each title is a new paragraph), the user would

1. enter an EOP character to terminate the previous paragraph;
2. call the formatting menu (ensuring that no selection is active), select the Character symbol, and point to a previous analogous title to get Character looks (e.g., large font and underlined);
3. select the Paragraph symbol and point to a previous title a second time to also get Paragraph looks (e.g., centered);
4. write the title and enter another EOP to terminate the title (paragraph);
5. reset Character and Paragraph looks to default by selecting the respective symbols again and just hitting the RETURN key.

Returning now to the more global level, it is sometimes desirable to set a common *style* for similar documents. The goal of unified style is partially met by means of the *user profile*, which permits the specification of a default value for each attribute. A more flexible method of ensuring a uniform style is using the convention of *sample documents*. Sample documents are normal Lara documents that typically contain a collection of sample characters in different styles, and sample titles and carefully formatted sample paragraphs with different margins, adjusting modes, and leading spaces (see Figure 10, p. 957). Previously edited documents, or even the current document, may also serve as samples.

Combined with the facility for copying attribute values, the sample-documents method has turned out to be enormously powerful. Text as well as formatting attributes may be taken from the sample document; this is particularly useful if special characters are required (e.g., mathematical or Greek symbols). If the sample contains a collection of these symbols, they can be copied directly into the current text without involving the keyboard. (Remember that Character looks are always transferred).

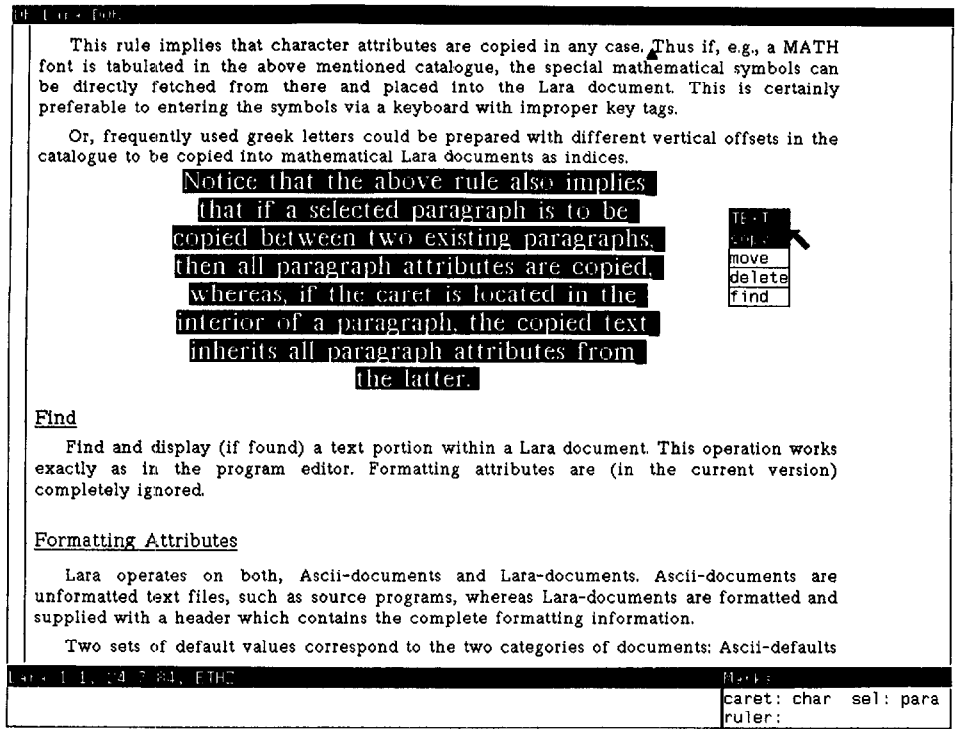
Sometimes, it is imperative to change the global style of a document after its initial creation: that is, by replacing one or more attribute values—for example, font size or line spacing—by others throughout the document. A natural way to introduce such changes is to extend the find/replace command from a purely contents-oriented to a more general context that includes the formatting considerations. Although we have not implemented this extension in the current version of Lara, we plan to do so.

So far in our discussion of formatting, we have focused on the *text region* of document windows. However, each document window contains two additional regions: the *title bar* and the positioning zone or *scroll bar*. The scroll bar is a vertical bar along the left side of the window. In addition, there are two universal windows (see Figure 4): The first contains the *dialogue* with the operator, while the second displays the current state of the selection and the caret and indicates the position of the so-called *ruler*, a thin vertical line used for measuring out documents. Each window and each region have their own area of responsibility. Since the menu entries can and do depend on the current state of the editing process, it is not a great leap to make the menu *itself* depend on the current location of the cursor.

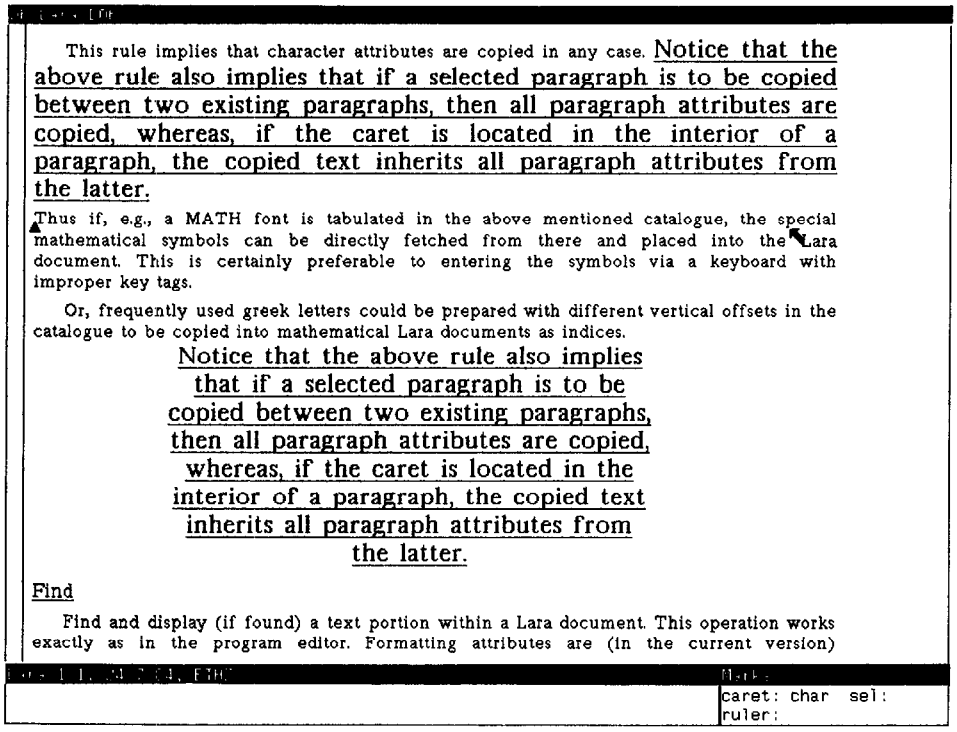
When the M-key is depressed in the title bar, the *title-bar menu* appears. It contains commands that affect either the window or the document as a whole—for example, opening and closing subwindows, redefining the extent of the window, and closing the document. The P- and S-keys also have distinct meanings within the title bar: Clicking the P-key places the window on top of a hypothetical pile of documents, whereas clicking the S-key places it on the bottom.

The scroll bar functions in much the same way. Clicking the P-key makes the text scroll up, whereas the S-key scrolls the text down. A click of the M-button positions the document absolutely. The upper end of the scroll bar corresponds to the beginning of the document, the lower end to the end. Notice that, having a display of the whole-page format at one's disposal, the use of the scrolling technique is somehow inconsequential and ill-compatible with the adage "what you see is what you get." Since Chapter attributes, for example, are not directly visible, it might be a better idea to replace scrolling by *leafing* commands particularly if a display is able to show two pages.

Universal commands are achieved via a third menu that appears when the M-key is depressed within the *dialogue window* (see Figure 4). The universal com-



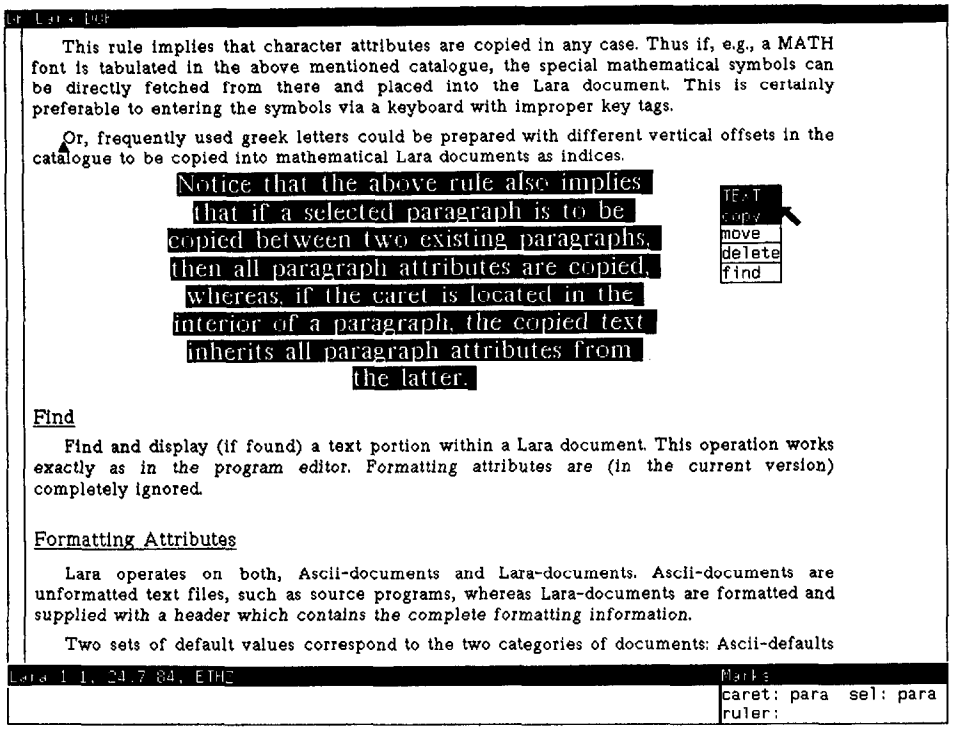
(a) Copying a Paragraph to the Interior of an Existing Paragraph



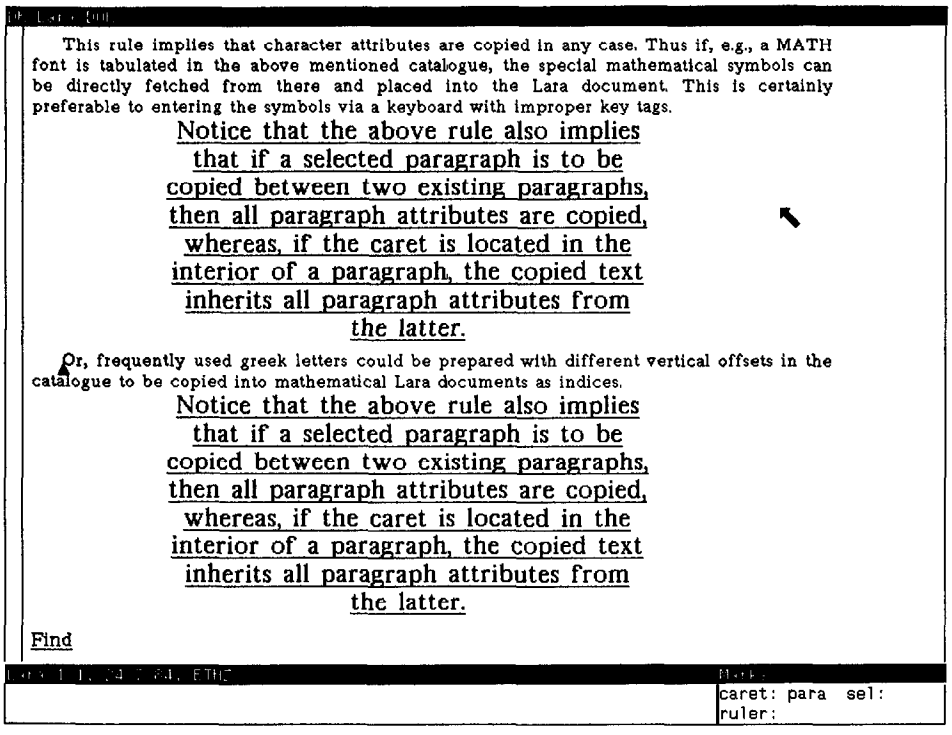
The remaining attributes are taken from the paragraph in which it is embedded.

(b) The Paragraph Is Copied Along with Its Font, Underlining, and Vertical Offset Attributes

FIGURE 6. Effects of the Copy Command

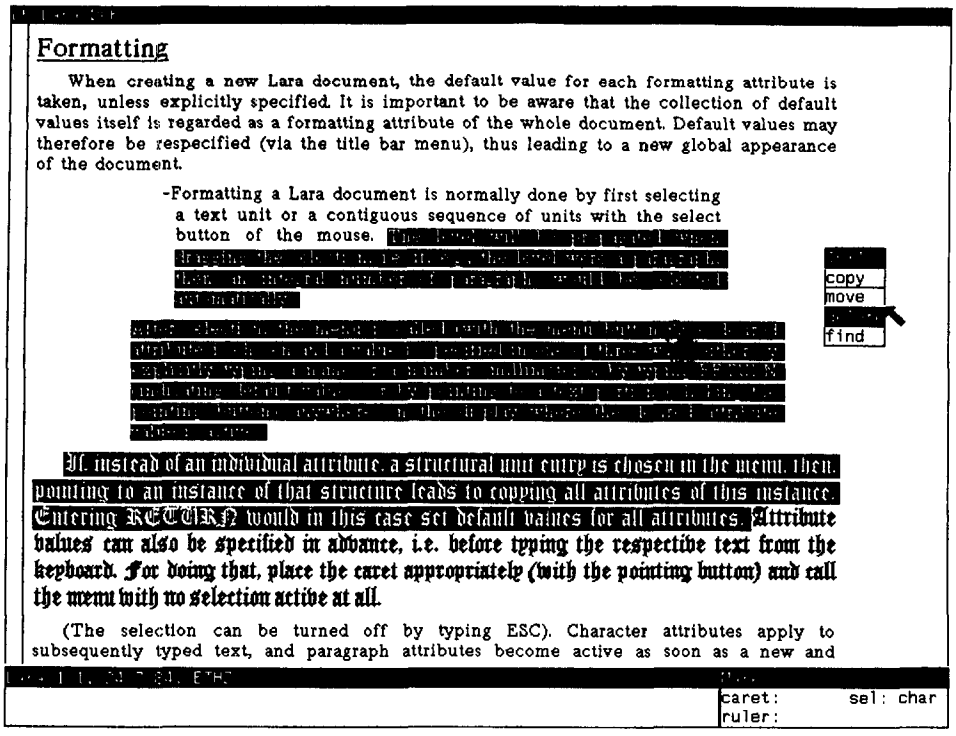


(c) Copying a Paragraph between Two Existing Paragraphs

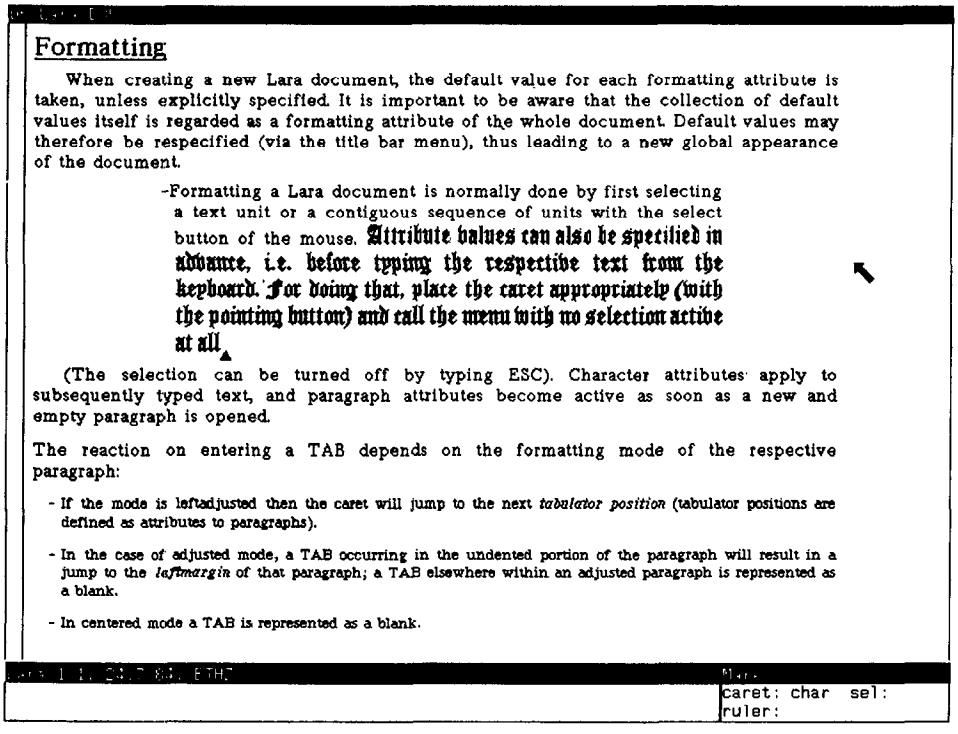


(d) The Paragraph (6c) Is Copied Along with All Its Attributes

FIGURE 6. Effects of the Copy Command



(a) The Delete Command When the Selection Overlaps with Structural Units of a Higher Level



(b) The Remaining Structural Unit Is Subsumed by the Higher Structural Unit

FIGURE 7. Global Side Effects of the Delete Command

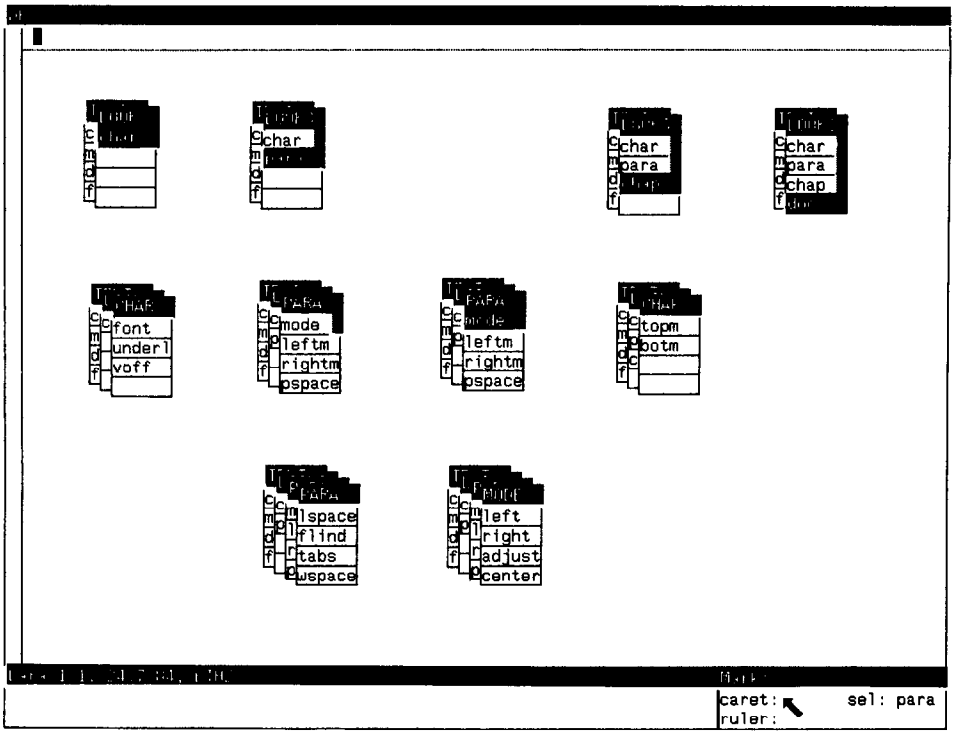


FIGURE 8. The Hierarchy of the Command Menu

mands consist of either opening a new document or leaving the editor. In addition to leaving the editor, the exit command closes the so-called *replay file*—a safety precaution against power, system, or program failures. An entry is made into the replay file whenever a new state of the mouse or a character from the keyboard has been read in the course of the editing process. If Lara is left without using the exit command, the replay file remains open. When Lara is later invoked, it will first process the previous replay file and thus restore the latest state of the document.

Through the use of these techniques, several documents can be edited together, each being displayed within its own window. The mouse and the menu technique are used to specify commands. The menu reflects the current location of the cursor and the state of the editing process. Most commands are generic: They use the selection and the caret as implicit operands. Looks can be copied from elsewhere on the display, for example, from a sample document. As a perfect symmetry exists between ordinary and sample documents—even the current document may serve as a sample.

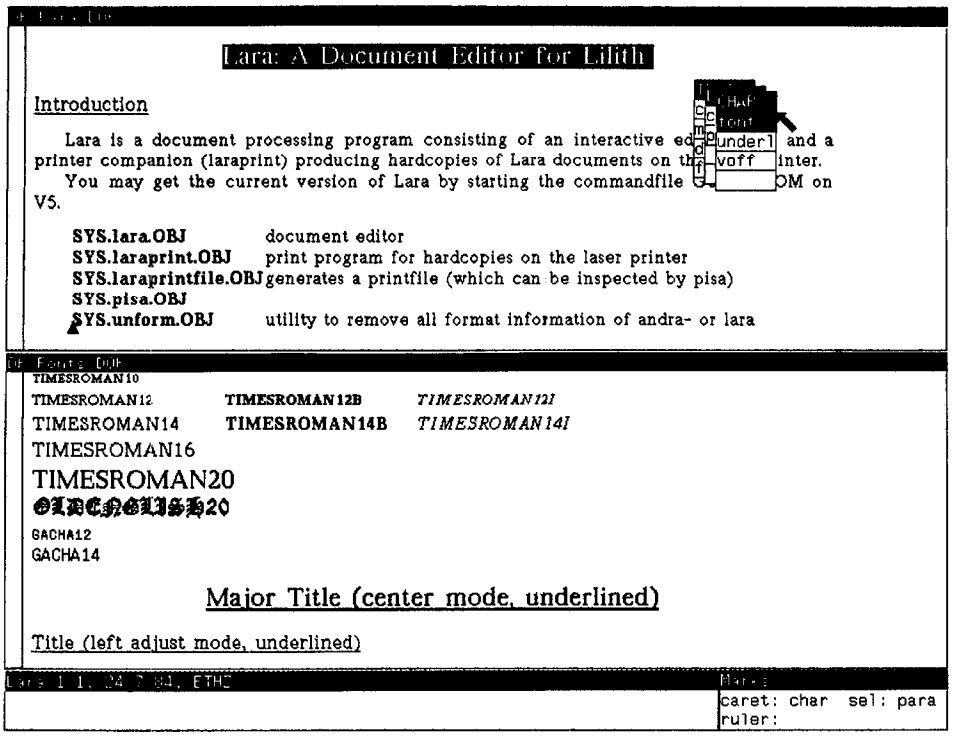
Perhaps the most important characteristic of Lara discussed so far is the close connection between internal data structure, displayed text, and the objects of the editing process. These objects correspond exactly to their counterparts in the data structure. The data structure itself is completely determined by the current *static* representation of the text. Since the data structure can be seen as a true mirror image of the latter, it is thus not explicitly dependent on the *history* of the editing process.

PROGRAM STRUCTURE

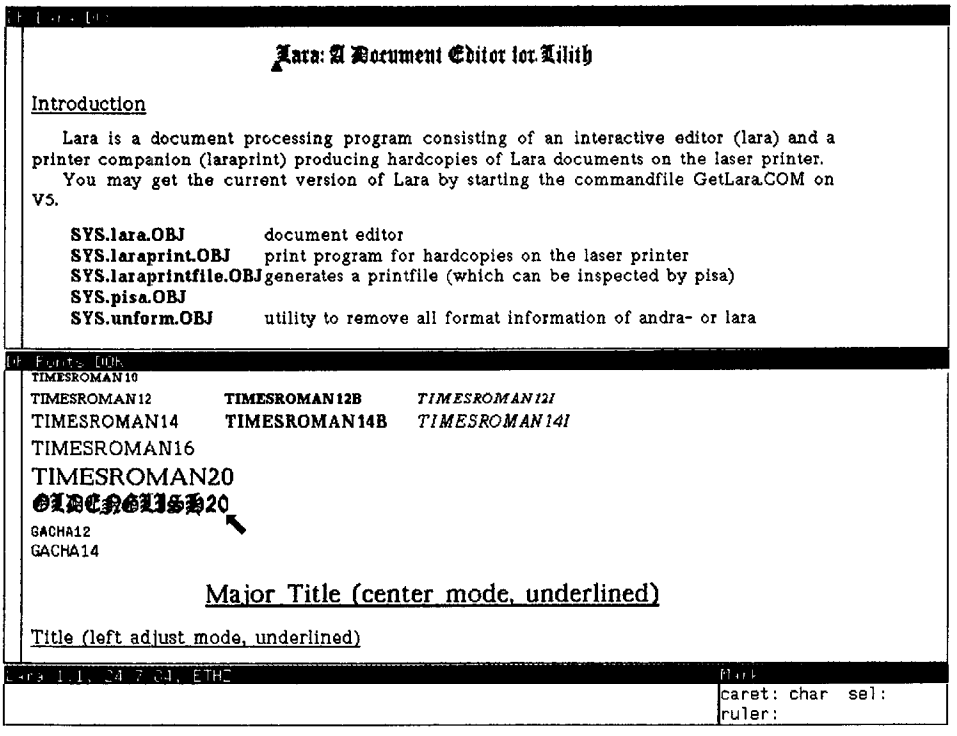
The fact that modern personal workstations are particularly well suited as editing devices also makes them attractive as *program-development machines.* In fact, the workstation Lilith was constructed as a program-development machine from the very beginning, and its development is intimately connected with the development of the programming language *Modula-2* [7]. (Lilith is programmed exclusively in Modula-2.)

Modula-2 is a high-level language designed for the development of *program systems*, including operating systems. The principal characteristic of a Modula-2 program or system is its modularity. Modules constitute the static elements of programs. They normally *import* data and procedures from other modules. Each module is split into two parts: a *definition* and an *implementation.* Exported objects are declared in the definition part and realized in the implementation part, while the definition part serves as an *interface* to the module's clients. Since the client modules depend solely on the interfaces, implementation parts may be redesigned without the need to adjust client modules. The operating system presents itself to the Modula-2 programmer as a collection of library modules; programming in Modula-2 therefore amounts to suitably extending the library.

Lara is programmed in Modula-2. The Lara program consists of the following modules: *command interpreter,*

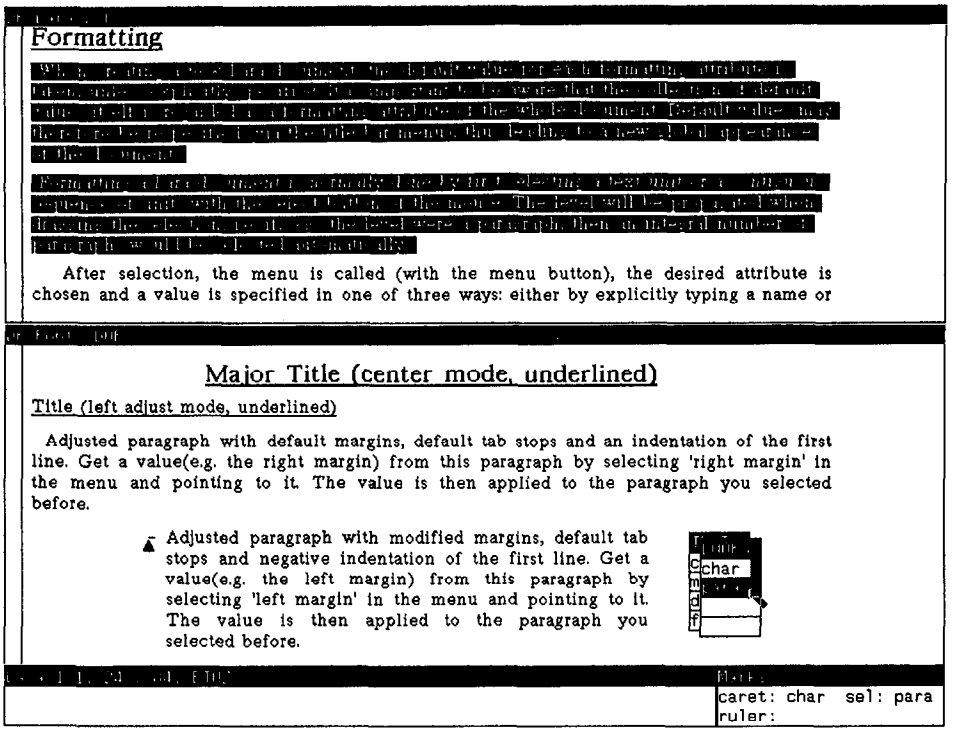


(a) Copying a (Character) Font Attribute Value

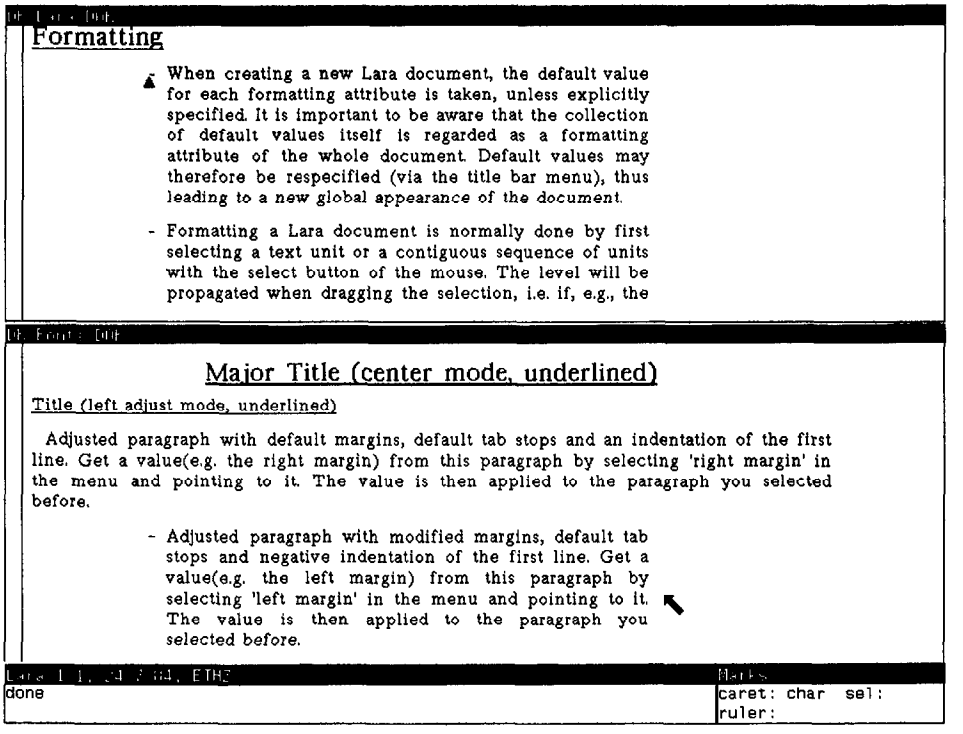


(b) The Character Parameter Value Is Transferred from the Sample Document Window

FIGURE 9. Copying Character and Paragraph Looks from a Sample Document



(c) Copying a Set of (Paragraph) Attributes from the Sample Document Window



(d) The Paragraph's Attributes—Adjusting Mode, Left and Right Margins, and Leading Space—Are Transferred

FIGURE 9. Copying Character and Paragraph Looks from a Sample Document

TABLE III. Caret Attributes (Formatting)

Caret position	Character	Paragraph	Chapter	Document
Beginning of empty document	*	*	*	*
Beginning of empty chapter	*	*	*	
Beginning of empty paragraph	*	*		
Arbitrary	*			

document handler, and *display handler*. The command interpreter, the main program, tracks the mouse and reads the keyboard for input. It recognizes commands and controls their execution by invoking procedures from the document and the display handlers. The core of the display handler is the *formatter*. It receives its information from the document handler, which administrates the main data structure and therefore constitutes the actual database. The dependency relationship among the three Lara modules is shown in Figure 11 (p. 958). To preclude any misunderstanding, it is important to emphasize that this diagram illustrates a static rather than a dynamic view of the interplay between modules. By this we mean that module boundaries do not surround temporal but rather logical entities. An arrow from a certain module box *N* to another module box *M* would signal that *M* imports from *N*. A temporally elementary action, such as selecting a text portion, might very well lead to a frequent switching of control between the different modules.

The Document Handler

The document handler provides two *channels* connecting it with its environment: one for *input* and one for *output*. It accepts editing commands (influencing the data structure) on the input channel and directs a sequential stream of characters and formatting signals to the output channel.

This means that the input channel is controlled by the command interpreter, while the data flow on the output channel is driven by the display handler. To accurately implement this model, we divided the document handler into three modules: *LaraDocument*, *LaraEdit*, and *LaraStream*. LaraDocument serves as the common base, LaraEdit corresponds to the input channel, and LaraStream corresponds to the output channel.

The Display Handler

To subdivide the display handler into functional units, we examine the formatter (or, more precisely, the *line formatter*) that is responsible for correctly breaking and adjusting individual lines. The Lara formatter treats words as elementary units that must not be broken. If a line must be adjusted, the formatter calculates the remaining space widths and distributes them among the word gaps. Obviously, the formatter must have access to the characteristic data (although not to actual raster data) of the involved fonts, such as font height, base height, and character widths. (Notice that *proportional fonts* and hence variable character widths are the normal case. It is this facility that greatly increases the complexity of the display handler and also makes a high-powered processor mandatory.)

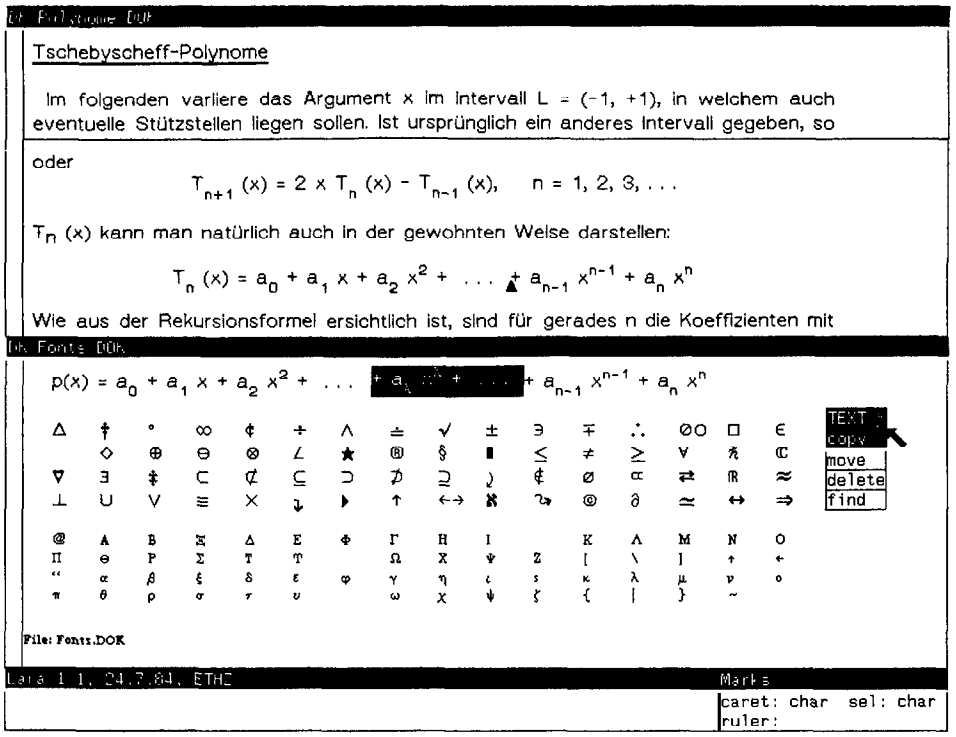
The actual formatting process is controlled by the previously mentioned stream from LaraStream. Given any starting position, LaraStream converts the internal data structure into a *linear* stream of characters and formatting information. A public variable indicates the current position within the format tree and thus signals the presence of new formatting information. The formatter accesses this information via a record that is declared in the interface of LaraStream—a technique that can be regarded as a local version of its counterpart in ordinary off-line formatters.

The display part now presents itself as two separate modules: the line formatter *LaraLine* and the actual display handler *LaraDisplay*. The three principal responsibilities of LaraDisplay are displaying lines of characters, converting screen positions into document positions, and displaying marks (selection and caret). In displaying lines of characters, LaraDisplay is assisted by LaraLine, which hands over to LaraDisplay a completely formatted line. The second display activity—converting screen positions into document positions—is called into effect when the caret or selection is defined and a location specified from which looks are to be taken. To maintain the *characteristic data* of the line being transferred—its height, number of characters, and word gap widths—LaraDisplay maintains chains of so-called *line descriptors*.

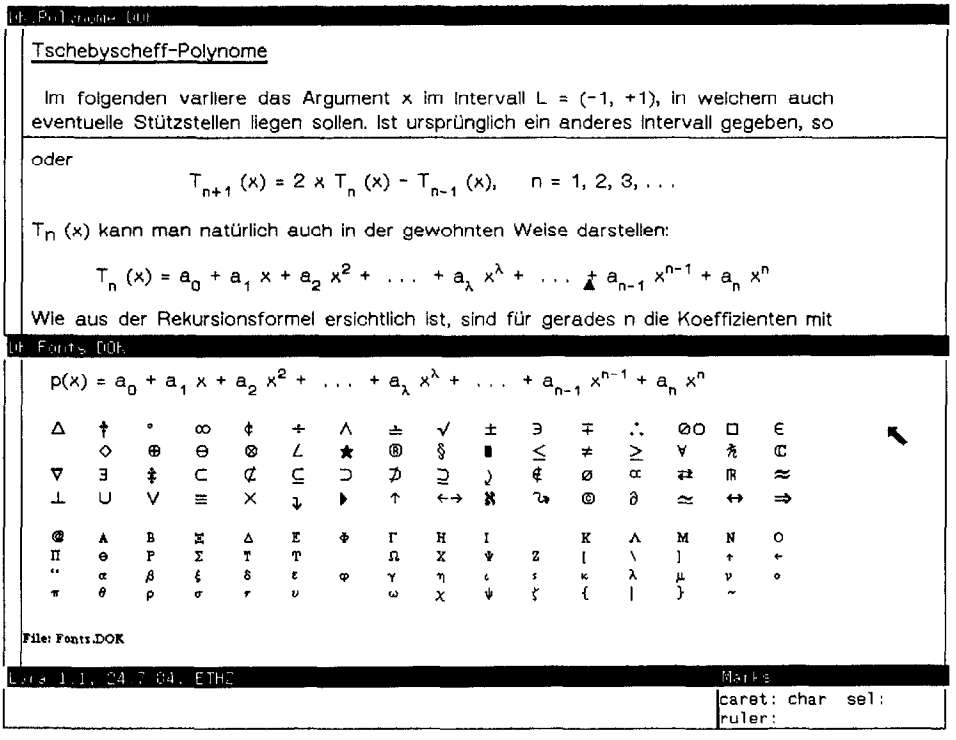
However, the exact data structure of LaraDisplay is even more complicated than has been presented here since the possibility of subwindows means the existence of subwindow descriptors as a logical extension. The merit of the subwindow and line descriptor concepts is that reformatting the entire window can be avoided whenever the caret or selection is set. The final data structure of LaraDisplay is given in Figure 12 (p. 958).

Printing

Lara documents are printed by means of a companion to Lara called *LaraPrint*. Instead of displaying a Lara document on the screen, LaraPrint creates a so-called *print file* that contains information for a laser printer: It specifies the fonts used and the blocks of characters to be printed, together with their exact positions on the page. Figure 13 (p. 959) shows an excerpt of the print



(a) Copying a Mathematical Expression ($+ a_\lambda x^\lambda + \ldots$)



(b) The Expression Is Now Inserted

FIGURE 10. Copying Special Symbols from a Sample Document

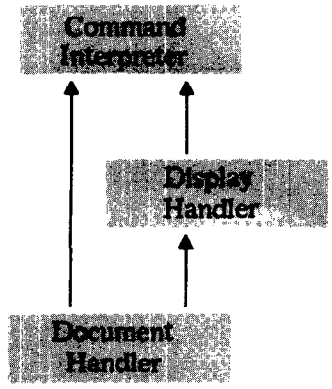


FIGURE 11. Overall Module Dependency

file that corresponds to the document described in Figure 3.

Although LaraPrint does not require the whole infrastructure of the interactive part of the Lara system, it is based on exactly the same document format, and therefore it is natural that it should use the same document handler. Moreover, in view of the conceptual coincidence between the bitmap display and the laser printer, even the formatter can be adopted. The possibility of using crucial constituents simultaneously for both Lara and LaraPrint is proof of both the power of Modula-2 in developing program systems and the adequacy of our modularization.

LaraPage is a new module designed for LaraPrint that is without counterpart in Lara. It is the *page formatter* exactly as LaraLine is the line formatter. The overall module structures of the interactive part (Lara) and its printer companion LaraPrint, as explained in this section, are summarized in Figure 14 (p. 960).

CONCLUSION

Lara is a modern text editor developed for the personal workstation Lilith. Written in the high-level language Modula-2, it is in principle transportable to similar computers, provided their operating system offers appropriate interfaces. The critical (i.e., machine-

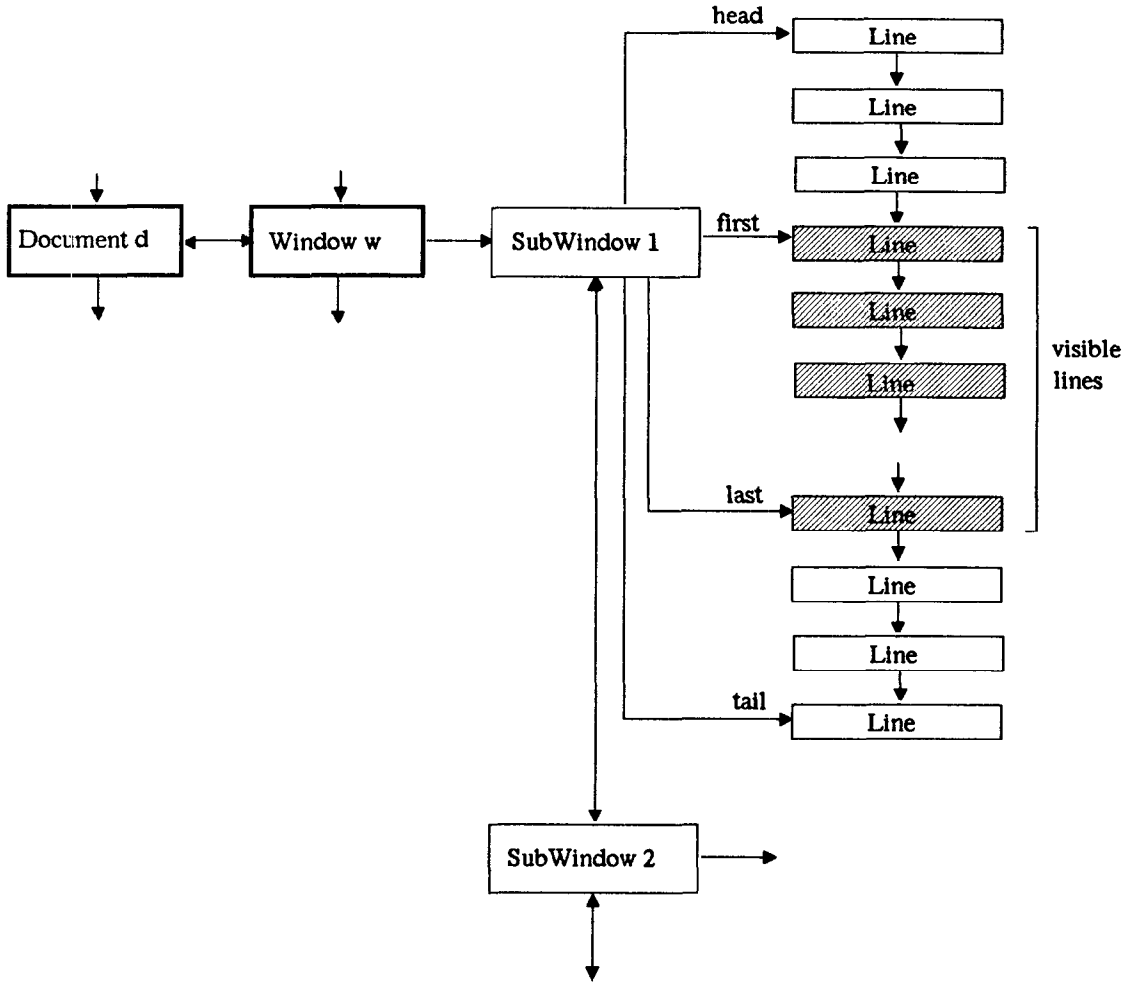


FIGURE 12. The Data Structure of the Lara Display

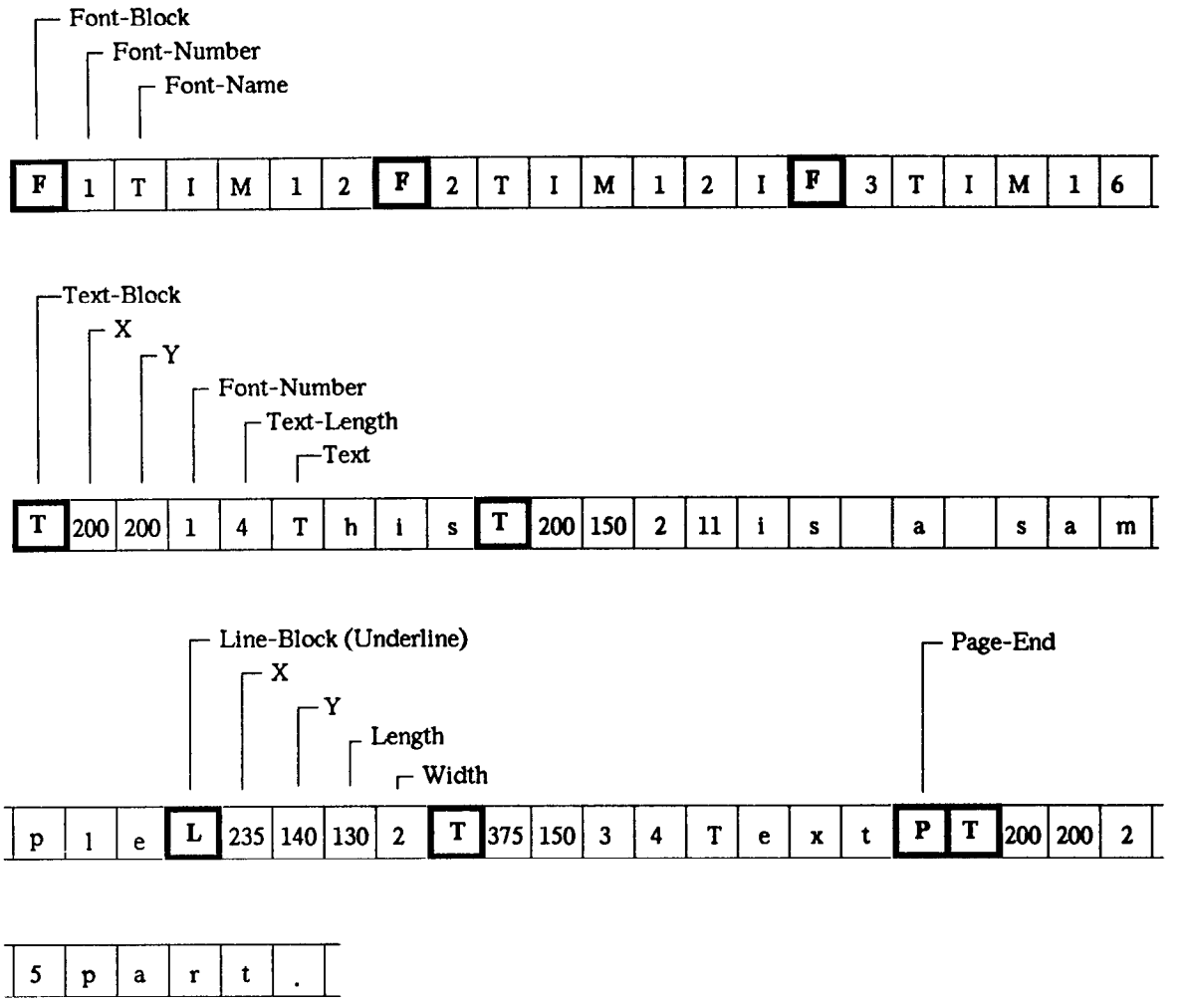


FIGURE 13. The Print-File Excerpt for the Document Referenced in Figure 3

dependent) interfaces relate to the handling of the mouse and the bitmap display; their specification and implementation on Lilith are described in detail in [1].

In developing Lara, emphasis was placed on building a sound and flexible base and exploiting the characteristics of personal workstations rather than implementing all possible features. For example, no provision is currently made for spelling-checking, hyphenation-on-the-fly, or automatic indexing.

Both the data elements handled by Lara and the commands that manipulate them are directly derived from the structure of a text as displayed on the screen and on paper. This encourages users to use these elements and commands extensively and thus to exploit Lara's possibilities whenever appropriate—in contrast to editors (and typewriters) where the character is the only element normally dealt with.

Our future plans for Lara concern extensions in two directions. First, we intend to introduce the notion of document group. Experience using Lara for writing books (e.g., [7]) has shown that it is often desirable to be able to view a group of logically connected documents as a unit. In particular, looks (such as style of titles) should be adjustable for the whole document group at a central place. This extension fits perfectly our concept of document trees of a fixed depth; we simply expand the tree at its root by one level.

The second extension we have in mind is even more important. So far, we have restricted Lara to the simplest imaginable page layout—one column form—which hardly exploits the full range of the laser printer's capabilities. By overlaying Lara with a global document composer, we plan to combine a more general allocation of text on a page with the integration of figures. The composer will manage a data structure indicating how the pages of a document are partitioned into

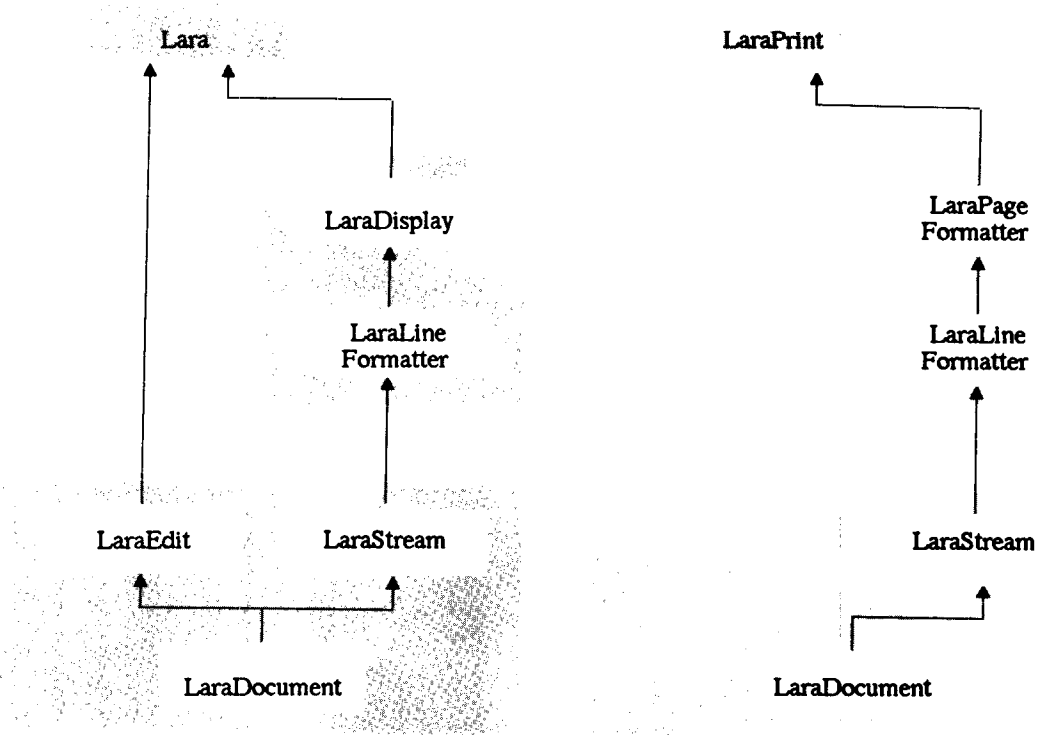


FIGURE 14. Lara and LaraPrint Module Structures

rectangles and what kind of objects is represented in each individual rectangle. A rectangle can either contain a figure or a section of a text stream.

Acknowledgments. I would like to express my gratitude to N. Wirth for the stimulus to this project, for numerous profitable discussions, and, above all, for making available such a fascinating machine. In fact, the Lilith turned out to be not only a perfect processor for the *product* Lara, but also an invaluable tool for its *construction.* My thanks go also to H. R. Schär for his substantive involvement in the Lara project, particularly his enthusiasm for the project and his competent implementation of a major part of Lara.

REFERENCES

1. Gutknecht, J. System programming in Modula-2: Mouse and bitmap display. Rep. 56, Institut für Informatik ETHZ, Sept. 1983. A comprehensive treatise on the handling of the mouse, menus, and windows on the Lilith computer. Presentation of a consistent set of base modules for interactive programs. In particular, these modules underlie Andra and Lara.
2. Gutknecht, J., and Winiger, W. Andra: The document preparation system of the personal workstation Lilith. *Softw. Pract. Exper. 14*, 1 (1984), 73–100. A detailed description of the Andra text editor and its implementation.
3. Johnson, L. Apple MacIntosh user education. MacIntosh MacWrite Manual, Apple Corp., Cupertino, Calif., 1983. User manual of the document editor MacWrite for Apple's MacIntosh computer.
4. Lampson, B.W. Bravo manual. Alto User's Handbook, Xerox Corporation, Palo Alto, Calif., 1978. A user-oriented description of Xerox PARC's Bravo editor and its capabilities.
5. Teitelman, W. A tour through Cedar. *IEEE Softw. 1*, 2 (Apr. 1984), 44–73. A written version of a video-tape tour through the Cedar programming environment. Cedar was developed at Xerox PARC. Among other things, the tour brings the reader into contact with the Tioga editor, an integral part of Cedar.
6. Wirth, N. Lilith: A personal computer for the software engineer. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, Calif., Mar. 9–12). ACM, New York, 1981, pp. 2–15. A technically oriented presentation of hardware and software concepts underlying the Lilith workstation.
7. Wirth, N. *Programming in Modula-2.* Springer-Verlag, New York, 1985. An introduction into programming in general and into programming in Modula-2 in particular. May serve as a concentrated course introducing the world of Modula-2, as well as a reference manual for the language Modula-2.

CR Categories and Subject Descriptors: D.2.2 **[Software Engineering]**: Tools and Techniques; I.7.1 **[Text Processing]**: Text Editing; I.7.2 **[Text Processing]**: Document Preparation; K.8 **[Personal Computing]**
General Terms: Design, Human Factors
Additional Key Words and Phrases: Modula 2, personal workstations, text formatting, user interfaces

Author's Present Address: J. Gutknecht, Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304 (through 10/15); thereafter: J. Gutknecht, Institut für Informatik, ETH-Zentrum, CH-8092 Zurich, Switzerland.