LEXX— A programmable structured editor

by M. F. Cowlishaw

Many sophisticated and specialized editing programs have been developed in recent years. These editors help people manipulate data, but the diversity complicates rather than simplifies computer use. LEXX is an editing program that can work with the syntax and structure of the data it is presenting, yet is not restricted to just one kind of data. It is used for editing programs, documents, and other material and hence provides a consistent environment for the user regardless of the editing task. The new *live parsing* technique used by LEXX means that it can be programmed to handle a very wide variety of structured data. The structure information is, in turn, used to improve the presentation of data (through color, fonts, and formatting), which makes it easier for people to deal with the text being edited.

Introduction

Broadly speaking, there are two types of editing programs (*editors*) in wide use today. There are text editors that treat data files as simply streams or lines of characters, and there are editors which are aware of the syntax or semantics of the data that are being edited. This latter type, which I shall call *structured editors*, has attracted many designers over the last twenty years [1, 2]. Structured editing is seen as one of the best ways of improving the quality of the tools available to computer users.

Most structured editors are specialized to a particular application. Some of these have selected a particular programming language (such as LISP [3]) or a text-markup language (such as GML, used by JANUS [4, 5]), and have built an editor specifically optimized for that language. Others (for example QUIDS [6] and Grif [7]) deal with more *ad hoc* representations of documents.

The greatest disadvantage of these approaches, though, is the natural resistance felt by computer users towards learning a new editor for each kind of data that is to be edited. General text editors such as EMACS [8] or XEDIT [9] are popular because they can be used quite successfully for most tasks, can offer a consistent environment from session to session, and can be customized to improve usability for specific kinds of data.

Even so, there are many advantages in using an editor that has specialized knowledge of the data being edited. The editor can improve the presentation of the data in a variety of ways, using appropriate formatting, color cuing, and fonts; it can provide checking of the syntax of the data, or even of the semantics; and it can provide specialized mechanisms to simplify or speed up the entry of data or of modifications. All of these improve the usability of the tool. At present, however, even quite minor enhancements to presentation often require changes to the editor itself [10].

The conflict, then, is between the goal of providing an editor which is general enough to handle most of a user's editing tasks, and that of providing an editor which is specialized for each task (or, at least, the major tasks). A separate editor for each major task does not seem to be a good long-term solution; even if all the editors were designed to a consistent specification, the duplication of effort and likelihood of confusing differences would be considerable.

Instead, a new kind of general-purpose editor is required, one that supports the concept of structured data in its primitives and yet is not specialized to a particular kind of data. The primitives and presentation capabilities need to be powerful enough to make it relatively easy to provide the editor with the appropriate specialized knowledge for a wide variety of types of information.

LEXX is a new editor that has been designed to this specification. (The name LEXX was chosen because the editor was originally developed as a tool for lexicographers during the author's secondment to the New Oxford English Dictionary project at Oxford University Press.) It achieves its goals by amalgamating a number of concepts and philosophies, of which the most novel is the *live parsing* mechanism. This technique has proved to be crucial in allowing the design of a general-purpose editor that can edit both documentation and programs while making evident the structure of the data being edited.

This paper discusses the more important design considerations, with emphasis on the aspects that most affect the user of the editor. The examples, reproduced with the kind permission of Oxford University Press, are taken from one of the most interesting projects for which LEXX is used, the New Oxford English Dictionary. It should be remembered, however, that the editor is not specialized towards handling text; it can just as easily be, and has been, customized for editing programs—LEXX is used for editing its own source, macros, and on-line documentation.

Structured editor design

The design of an editor is a complicated task, with many requirements and objectives that sometimes conflict. The goal (for any editor that is not just experimental) is that it should be a usable and effective tool for interacting with data. If any interactive program does not meet the requirements of the people that are to use it, it has failed to achieve this goal.

For a general-purpose editor such as LEXX, many of the common requirements (such as the ability to move through the data, the ability to search for strings, reliability, and so on) are well known and will not be discussed here. The more complicated requirements, including those that apply specifically to structured editing, are listed below.

Data classification A file that is to be edited usually contains several different types, or *classes*, of data. In a document there may be text and markup; in a program there will be a different set of classes, such as variables, keywords, and commentary. The class of a piece of data must be maintained by the editor since different treatment and presentation are often required for different classes.

Data structure The editor needs to be able to formalize the relationships among objects in the data. Documents and programs have structure, often expressed in terms of nesting or depth, and this structure must be understood by the editor so that it can be properly presented to the user. Classification and structure analysis of data correspond to syntactic and semantic analysis of language.

Data presentation The data being managed by the editor have to be displayed to the person using the tool. Since structured data have attributes other than just character shapes, it must be possible to display these attributes. Formatting, coloring, the use of different fonts, and highlighting may be appropriate, according to the capabilities of the display terminal being used.

Data modification Although the program may be used simply to view data, for it to be an editor its design must also provide for the modification of the data. In the case of a structured editor, any modifications must properly update not only the raw characters of the data but also the underlying classification and structure.

Programmability The editor must be flexible, easily programmable, and easily customizable. Human preferences and expectations vary so much that it is extremely unlikely that any single configuration will satisfy all users.

Performance To be usable, a tool must respond quickly to users' demands and requests. The mechanisms provided by the editor should not be inherently inefficient.

The remainder of this section discusses (and illustrates, where appropriate) each of these requirements in more detail, and describes the approaches taken in the LEXX design to meet these objectives.

Data classification

When LEXX is first requested to edit a file, it loads the file from a filing system or database in the usual way. A typical file, presented without any further processing, is shown in **Figure 1.** In this figure (an example of an entry in the Oxford English Dictionary, using an early form of markup), the entry has been *marked up* with tags. These tags are delimited with the characters "⟨" and "⟩", in the SGML [11] notation, and describe the structure of the entry; this particular entry is composed of a headword section (⟨hwsec⟩) and a single sense (⟨sen⟩). Each of these parts is in turn composed of other parts, which, in this example, form a rather simple hierarchy. However, when first loaded, the tags are not very easy to read and the structure is not apparent. It is not easy to edit a file in this format; try, for example, to read the quotation dated 1642, or to count the number of quotations.

Once the file has been loaded, LEXX determines the type of the file (usually from part of the name, called the *filetype* or *extension*), and then calls the appropriate command for processing this type of file. The command used is called a *parser* and is specific to a particular variety of data. One purpose of the parser is to analyze the data which were loaded and then identify and classify related pieces of those data.

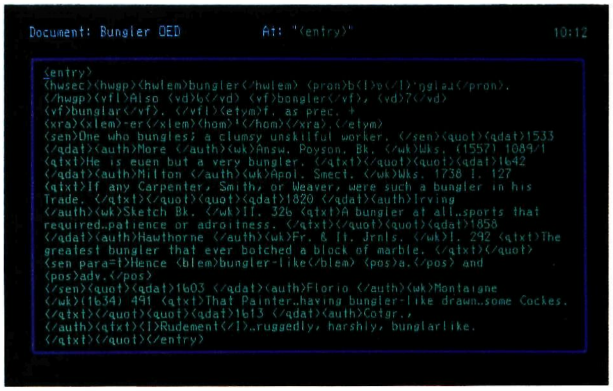
For example, when the marked-up text document shown in Figure 1 is first loaded, it is simply a series of lines of characters. These lines include both the text content of the document and the tags that describe its structure and formatting. The parser will analyze the document and determine which characters are part of markup and which are part of the text. Each sequence of characters that belong to a particular class is assigned to an *element*, the atomic unit which LEXX uses to hold data. (Characters are collected together into elements mainly for reasons of efficiency, but the concept of elements also simplifies many operations on the data.)

Each element, therefore, holds characters that all belong to the same class (or classes, since sequences of characters may belong to more than one class). This classification can then be used by the editor to help control which data are to be displayed, which data may be modified, and so on. The parser defines and names the classes used; the editor simply provides general primitives for naming and manipulating the classes.

The ultimate purpose of classification is to help the user read and edit the data; if the different classes can be distinguished visually when the data are displayed, a document can be much easier to read and to edit. The parser can achieve this by associating with each element a *symbolic font*. These fonts are then represented by different typefaces and colors on the display. An initial mapping of each symbolic font to a standard typeface and color is usually set by the parser, but this may be changed at any time by the user or a program according to preference and the typefaces available.

In **Figure 2,** markup is being displayed in a default typeface, colored turquoise; text is shown in a variety of green typefaces which indicate the font style that would be used for each segment of text if it were to be formatted for printing. The bold font is also emphasized in yellow to further enhance visibility; it is now very much easier to find the quotation dated 1642. The parser has also associated some simple *formatting* information with each element (for example, to indicate whether it should start a new line or be preceded by a blank line). This formatting also helps make the data more readable and easy to comprehend.

The parser is dynamically loaded when first required (as all LEXX commands may be). Since it is not a fixed part of the editor, skilled users and application developers may develop their own parsers (or modify existing ones). Parsers are usually written in a compilable language, though it is possible to use an interpreted language; the latter is especially appropriate for trying out new parsing ideas.

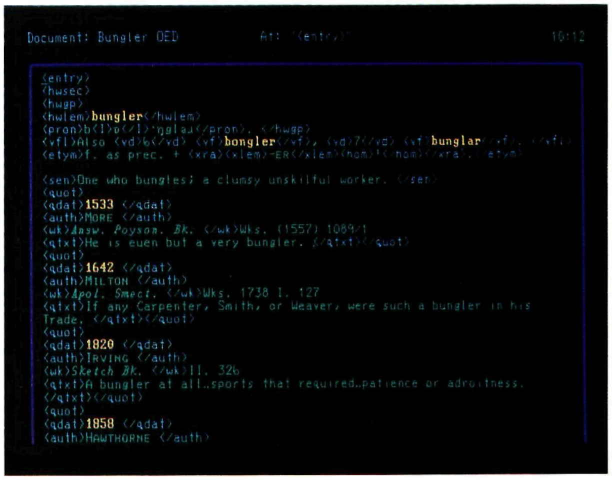


```
Document: Bungler OED          At: "<entry>"                    10:12

<entry>
<hwsec><hwgp><hwlem>bungler</hwlem> <pron>b<I>ɒ</I>'ŋglaɹ</pron>.
</hwgp><vfl>Also <vd>6</vd> <vf>bongler</vf>, <vd>7</vd>
<vf>bunglar</vf>. </vfl><etym>f. as prec. +
<xra><xlem>-er</xlem><hom>¹</hom></xra>.</etym>
<sen>One who bungles; a clumsy unskilful worker. </sen><quot><qdat>1533
</qdat><auth>More </auth><wk>Answ. Poyson. Bk. </wk>Wks. (1557) 1089/1
<qtxt>He is euen but a very bungler. </qtxt></quot><quot><qdat>1642
</qdat><auth>Milton </auth><wk>Apol. Smect. </wk>Wks. 1738 I. 127
<qtxt>If any Carpenter, Smith, or Weaver, were such a bungler in his
Trade. </qtxt></quot><quot><qdat>1820 </qdat><auth>Irving
</auth><wk>Sketch Bk. </wk>II. 326 <qtxt>A bungler at all..sports that
required..patience or adroitness. </qtxt></quot><quot><qdat>1858
</qdat><auth>Hawthorne </auth><wk>Fr. & It. Jrnls. </wk>I. 292 <qtxt>The
greatest bungler that ever botched a block of marble. </qtxt></quot>
<sen para=t>Hence <blem>bungler-like</blem> <pos>a.</pos> and
<pos>adv.</pos>
</sen><quot><qdat>1603 </qdat><auth>Florio </auth><wk>Montaigne
</wk>(1634) 491 <qtxt>That Painter..having bungler-like drawn..some Cockes.
</qtxt></quot><quot><qdat>1613 </qdat><auth>Cotgr.,
</auth><qtxt><I>Rudement</I>..ruggedly, harshly, bunglarlike.
</qtxt></quot></entry>
```

Figure 1

A marked-up document loaded as a plain file.



```
Document: Bungler OED          At: "<entry>"                    10:12

<entry>
<hwsec>
<hwgp>
<hwlem>bungler</hwlem>
<pron>b<I>ɒ</I>'ŋglaɹ</pron>. </hwgp>
<vfl>Also <vd>6</vd> <vf>bongler</vf>, <vd>7</vd> <vf>bunglar</vf>. </vfl>
<etym>f. as prec. + <xra><xlem>-ER</xlem><hom>¹</hom></xra>.</etym>

<sen>One who bungles; a clumsy unskilful worker. </sen>
<quot>
<qdat>1533 </qdat>
<auth>MORE </auth>
<wk>Answ. Poyson. Bk. </wk>Wks. (1557) 1089/1
<qtxt>He is euen but a very bungler. </qtxt></quot>
<quot>
<qdat>1642 </qdat>
<auth>MILTON </auth>
<wk>Apol. Smect. </wk>Wks. 1738 I. 127
<qtxt>If any Carpenter, Smith, or Weaver, were such a bungler in his
Trade. </qtxt></quot>
<quot>
<qdat>1820 </qdat>
<auth>IRVING </auth>
<wk>Sketch Bk. </wk>II. 326
<qtxt>A bungler at all..sports that required..patience or adroitness.
</qtxt></quot>
<quot>
<qdat>1858 </qdat>
<auth>HAWTHORNE </auth>
```

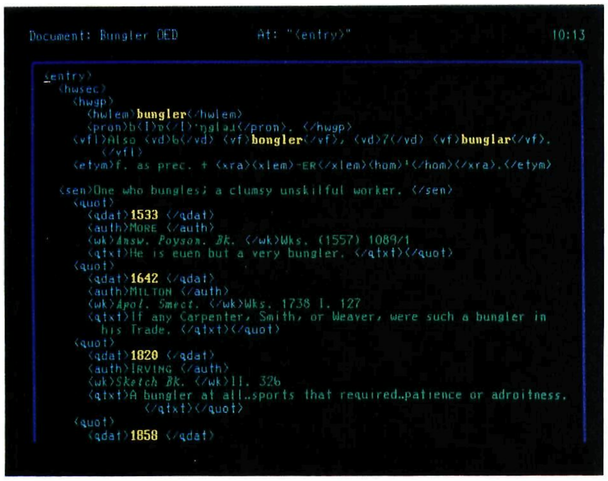
Figure 2

The document after classification and font assignment.

If a file is loaded but no appropriate parser is available, the default settings (as seen in Figure 1) permit the file to be edited simply as a series of lines of data. Therefore, even if no parser exists for a particular type of data, LEXX is still a useful and effective text editor.

Data structure

As well as classifying the data in a file and adding information to improve readability, the parser may determine the structure of the data. Each element can have a depth indication associated with it, showing its depth within the structure of the file. This depth can be an absolute value (for example, in a book, the book as a whole might be at



```
Document: Bungler OED              At: "<entry>"                    10:13

<entry>
  <hwsec>
    <hwgp>
      <hwlem>bungler</hwlem>
      <pron>b<I>ʌ</I>ˈŋgləɹ</pron>. </hwgp>
    <vfl>Also <vd>6</vd> <vf>bongler</vf>, <vd>7</vd> <vf>bunglar</vf>.
        </vfl>
    <etym>f. as prec. + <xra><xlem>-ER</xlem><hom>¹</hom></xra>.</etym>

  <sen>One who bungles; a clumsy unskilful worker. </sen>
    <quot>
      <qdat>1533 </qdat>
      <auth>MORE </auth>
      <wk>Answ. Poyson. Bk. </wk>Wks. (1557) 1089/1
      <qtxt>He is euen but a very bungler. </qtxt></quot>
    <quot>
      <qdat>1642 </qdat>
      <auth>MILTON </auth>
      <wk>Apol. Smect. </wk>Wks. 1738 I. 127
      <qtxt>If any Carpenter, Smith, or Weaver, were such a bungler in
        his Trade. </qtxt></quot>
    <quot>
      <qdat>1820 </qdat>
      <auth>IRVING </auth>
      <wk>Sketch Bk. </wk>II. 326
      <qtxt>A bungler at all..sports that required..patience or adroitness.
            </qtxt></quot>
    <quot>
      <qdat>1858 </qdat>
```

Figure 3

The document with structure shown by indention.

depth 1, and each chapter might be at a deeper level, perhaps 3). The depth can also be relative to the preceding element. (A list would normally be defined as a depth lower than its surroundings; nested lists then naturally show increasing depths.)

After the parsing, then, the document held by LEXX can have a structure, or hierarchy, determined from the content of the data. Like the classes and formatting information, the structure of the document can affect both the display of the data and the actions of commands that will work on the data. Use of the hierarchical structure model supported by LEXX is optional, since it is not appropriate for some kinds of data.

As with classification, there is little point in adding structural information to a document if this information cannot be conveyed to the user when required. This is achieved by automatic indention. Unless requested otherwise, LEXX will display each element with indention proportional to its depth in the structure of the document. The effect of this can be seen in **Figure 3** (which is the same as Figure 2 except for the indention). As for programming languages, sensible use of indention significantly aids comprehension of the structure of a document.

The embodiment of the structural information is worth noting. Instead of mirroring the structure in the organization of the data (by linking the elements in a conventional tree structure, as in ED3 [12] and STET [13]), the hierarchy is simply recorded as changes of depth associated with each element. The elements themselves form a simple linked list [14]. This considerably simplifies the traversal of the data during common requests such as context searches, and also makes manipulation of the hierarchy easier.

Data presentation

The usability of an editor is determined to a large extent by its presentation of the data being edited, and it is in the presentation of data that structured editors can really excel. LEXX knows the class of each data element and it knows the hierarchy of the elements; as has already been seen, this information can be applied very effectively to help the user. Beyond this initial presentation, flexibility of display styles can further enhance usability. In LEXX there are many optional variations, including the following:

- The structure of the data is normally indicated by indenting data elements according to their depth in the hierarchy. This indention can be increased or decreased to either emphasize or hide the structural information.
- The content of the display can be varied according to the structure. Inclusion of elements can be made contingent on their being higher than a certain depth, or lower than a certain depth, or within certain depth bounds, etc. This is especially useful for data and languages that have a formal hierarchical grammar.
- The content of the display can be varied according to the classification of the data. Elements of selected classes can be included in (or excluded from) the display as required. For example, if all the elements in a document that form chapter or section headings are assigned the class HEADER, then requesting a display of only HEADER class elements will show the table of contents of the document. Note that this table of contents is not extracted from the document in any sense, but is composed from the actual header elements in the document (which, therefore, can be edited directly). Similarly, depending on the classification carried out by the parser, the display can show just footnotes, or indexing entries, or whatever combination may be required.
- The effects of the formatting information associated with each element can be controlled. For example, the blank lines added for readability may be switched off to get more data on a small screen. Surprisingly, even on small (24-line) screens, people usually prefer to keep the blank lines; in this case, the improved readability seems to be more important than the quantity of data on the screen.
- Each element has associated with it an optional fonts string, which consists of a single character corresponding to each character of the content (data) in the element. The font character symbolizes a particular font, and the user may select the color and character set (typeface) used for each symbolic font, according to the capabilities of the display being used. Thus the font "I" might represent "Italics," and be displayed in an italic typeface, in red.

The fonts are initially assigned by the parser, usually on an element-by-element basis, and often correspond closely to the classes of data (though they are independent of the classes). The fonts associated with any element may be changed easily, as may the mapping of the fonts to visual cues. Individual characters (such as punctuation) within an element can therefore be assigned different fonts, for emphasis. Careful use of fonts in this manner can further improve the transfer of information from the display to the user.

This partial list gives an idea of the flexibility of the LEXX display processor. It is in fact a formatter, much like a document formatter, which has to select the elements for display according to a number of criteria, then format the data to fit the screen or display window available. Each element is formatted according to its associated rules for formatting and its content (data and font information). This is based on the analysis of the data by the parser and may therefore follow or ignore the original layout of the data, as appropriate.

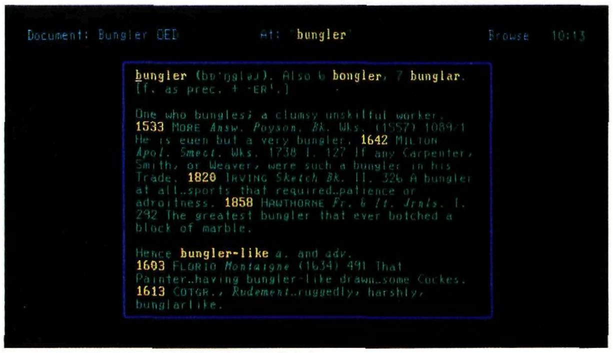
Formatting must be done every time the data are to be displayed to the user (for example, while scrolling up or down a document)—considerably more often than required by document formatters. Efficiency is therefore an important consideration, and the current formatter makes considerable use of "look-aside" information in order to achieve good performance, especially when the format of a section of a document is heavily dependent on previous elements.

It is important to appreciate that the formatting provided by the editor does not (in the case of text documents) necessarily mimic the appearance of a final or printed document. Rather, the display normally indicates the general structure and content of the data—just as markup in a document is descriptive of the document rather than of a particular final formatting.

The flexibility of the display options means that the data being displayed may be selected as appropriate for the editing task. In a marked-up document, both the markup and the data can be shown for convenience of editing. During proofreading, the markup can be excluded from the display to give an uncluttered view, and *presentation elements* (added by the parser purely to improve the appearance of the displayed file) can be included. Such a view of the dictionary entry shown in earlier figures can be seen in **Figure 4.** This view is produced by the same formatter, and at the same speed, as the earlier views; a narrower width is used to give an idea of the usual presentation style of this material.

Note that only four parameters have been changed from Figure 3. The width has been reduced, elements have been flowed together, markup has been excluded, and presentation elements (such as the parentheses around the pronunciation) have been included.

It can even be useful to display only the markup, without the intervening data. Styles may be evolved to suit the application—the choice is made by the user, not the editor.



```
Document: Bungler OED              At: "bungler"                    Browse   10:13

              bungler (bʌ'ŋgləɹ). Also 6 bongler, 7 bunglar.
              [f. as prec. + -ER¹.]

              One who bungles; a clumsy unskilful worker.
              1533 MORE Answ. Poyson. Bk. Wks. (1557) 1089/1
              He is euen but a very bungler. 1642 MILTON
              Apol. Smect. Wks. 1738 I. 127 If any Carpenter,
              Smith, or Weaver, were such a bungler in his
              Trade. 1820 IRVING Sketch Bk. II. 326 A bungler
              at all..sports that required..patience or
              adroitness. 1858 HAWTHORNE Fr. & It. Jrnls. I.
              292 The greatest bungler that ever botched a
              block of marble.

              Hence bungler-like a. and adv.
              1603 FLORIO Montaigne (1634) 491 That
              Painter..having bungler-like drawn..some Cockes.
              1613 COTGR., Rudement..ruggedly, harshly,
              bunglarlike.
```

Figure 4

The document shown without markup, in galley style.

Data modification

As shown in the previous section, the data presented to the user for modification can be abstracted from the original document and arranged in quite sophisticated ways. Within the data area of the display, the characters visible are selected from some subset of the elements of the document. Each line of the screen may contain several elements, or an element can be formatted so that it is spread over more than one line.

Once the data have been displayed, the user can (in normal operation) over-type, insert, or delete characters anywhere in the data. Elements of different classes may be altered, and the characters corresponding to whole elements of data may be deleted with a single keystroke. Every change made to the display not only must update the corresponding characters in the data structure, but also must cause appropriate changes to the other attributes of the data: the classes, structure, formatting information, and fonts.

At first sight, it seems that it is necessary to keep track of every character displayed in order to interpret its correlation to the elements in the data structure. Furthermore, the rules for updating the other attributes have to be formalized and communicated to the editor.

The first of these two tasks is expensive but possible. The second, however, is not possible, since the editor designer cannot be aware of all possible rules that might apply to data that will be edited. Restricting the editor to data that can be described by a simple set of rules or tables can be quite successful (as in Wood's Z [15] and Hall's ZEDSE*), but cannot provide for the general case and hence fails to produce a general-purpose editor.

Fortunately, there is a solution. As already described, the editor enlists the aid of a specialized parser when the file is first loaded. Since the file was provided by some external agency (it may have been edited by another editor, for

* S. T. Hall, IBM UK Laboratories Ltd., Hursley Park, Winchester, England; personal communication, 1986.

example), the parser must be clever enough to classify and deduce the structure of any raw sequence of characters. All that is needed, then, is for the editor to provide a mechanism to invoke the very same parser to reparse any data that are changed by the user. This technique makes a true general-purpose structured editor a possibility, and is called *live parsing*. It also has the advantage that the editor does not have to "see" every keystroke made by the user, and hence makes practical the use of displays that provide hardware editing features.

Live parsing is used for any change made to the data, not just those made by typing on the screen. As the editor processes each change, the element affected is added to a list of changed elements. At appropriate intervals (not after every change, since some changes may be interrelated, but before the screen is displayed), live parsing is triggered, and the parser is called once for every element on the list. The sole difference from the initial call of the parser is that only the local context (often just the element itself) around the changed element needs to be processed, not the whole file.

For example, if the name of the author of the 1642 quotation (Milton) were changed on the display shown in Figure 3, the parser would be called with the changed element identified by the "current position" information. It would then determine the local context that might be affected by the change, which in this case is the complete quotation (bounded by the ⟨quot⟩ and ⟨/quot⟩ tags). [Here the parser can determine the context simply from the structural (depth) information, but parsers for other kinds of data might need to inspect the elements more closely or make use of static information.]

When the local context has been determined, all the elements in that local context are parsed. This takes place as though they had been concatenated to form raw text as originally loaded from a file system. As each syntactic token (tag or text) is identified, it is classified and its structure is determined, and the new element, classification, and structure replace the old. In practice, of course, the parser will find that most of the elements in the local context are already correct, and so few real changes or data movements are required.

The parser may make any changes to the document and associated state information, without restrictions. It may alter, insert, or delete elements, and may change the fonts and attributes of elements. Changes made by the parser are not noted and reparsed, but they are recorded and hence can be "undone."

The size of a parser is very much dependent on the complexity and regularity of the data to be parsed. As an indication, the parser for dictionary entries has about 700 lines of PL/I [16] that are directly involved in parsing, together with a further 300 lines of initialization. (Initialization consists mainly in processing tables of special symbols and tags needed for the New Oxford English Dictionary—new tags are introduced by modifying a tag table rather than the parser itself.) Writing a parser is a significant piece of work but is a task very much smaller than writing an entire editor.

Programmability

All editors designed for serious use are programmable and customizable to some extent, and LEXX is not an exception to this rule. Programmability in LEXX is based on its command language: Every primitive operation is defined as a command, and every command can be issued either manually or from a program. All state information that can be altered can be queried on command and passed to a program, and much other information (such as the characteristics of the current display device) is also available through the command interface.

In the current implementation, commands are generally written in PL/I or REXX [17]. The first of these languages (PL/I) is compiled, and the object code is dynamically included in the editor when the command is first invoked. The second is interpreted; commands written in this language are by convention called *editor macros*, though conceptually there is no difference between macros and PL/I commands.

The kernel of the editor is a very small program that initializes certain permanent data areas mainly concerned with the operating environment. It then calls the first command, named "SHELL," which is the command dispatcher. (The default SHELL command is written in PL/I, but users can write their own in either of the languages mentioned.) From this point, execution of the editor is entirely command-driven and programmable. Commands are available to build (format) data for display, display them, and wait for user input; these primitives allow a certain amount of asynchronous operation—a command may continue to carry out background processing while awaiting user input. Since the formatter is itself a subroutine of a command, different or more sophisticated formatting than is provided in the current formatter (such as formatting that displays the data as they would appear if printed or otherwise processed) could be provided by replacing this command. The formatter is complex, so this would be a harder task than writing a typical parser.

Some of the functions of the parser commands calling for live parsing have already been described, but since parsers are themselves programmable commands, they make possible completely new kinds of interactive data-dependent processing. (The practical limit is that amount of processing which would start to increase editor response time significantly.) A parser for text documentation can carry out spelling or grammar checking, and suggest indexing entries; a parser for a programming language might warn of dangerous constructs or of poor coding and commentary standards, or expand macros as they are entered; or the data could be in the form of a spreadsheet, with the parser carrying out the calculations and updating the display as changes are made. In the last case, one would have a spreadsheet that had the full power of the REXX language and the editor directly available.

Performance

A crucial requirement for any interactive software tool is good performance. Since this depends to a great extent on the hardware being used, absolute figures will not be meaningful to every reader. As a guide, however, a response measurement made as I typed this paragraph showed an elapsed time of just under a quarter of a second. That is, it took less than a quarter of a second from the time a change was made to the time the data had been parsed and updated, a new screen formatted, and the new display fully written on the screen. (This was on a busy, shared IBM 4381 processor running the VM/CMS operating system, and using a 3279 display with a variety of fonts and colors.)

More interesting than the total time (which is quite adequate, though not as fast as might be wished) is an analysis of the processing and time delays involved. Exactly half of the elapsed time went in system overheads (almost wholly delay in the display controller). Of the remainder, which is central processor (CPU) time, the percentages spent in each part of the operation were

35%	in formatting the text for display and building the rest of the screen (status information, etc.).
29%	in sending the formatted screen to the display device.
18%	in reading changes from the display and correlating them with the data structure.
19%	in live parsing.

Of course, the actual sizes of these four tasks will vary considerably according to the hardware and software environment, the data being edited, and the complexity of the changes made. Even so, it is notable that both the formatting and the live parsing require resources that are comparable to those needed simply to drive the display terminal. It was anticipated that the interactive formatting would be very expensive, but in practice careful implementation has been able to keep this within acceptable limits. The live parsing, too, is less of a load on the processor than was expected, even though no special care has been taken over implementation in this case.

The performance of the editor does not deteriorate with large documents, except that, of course, the time needed to load the document from the file system and to parse it initially is proportional to its length.

Conclusions and further directions

LEXX is currently implemented as a "mainframe" editor, but its concepts, command language, and design are intended to be general. It would be especially interesting to implement LEXX for more personal workstations. The live parsing technique has proved to be sufficiently general that parsers for a number of programming languages have been written, together with parsers for several kinds of documentation markup. Future work will probably investigate the use of the editor for other kinds of data, while at the same time enhancing its capabilities when it is used without a specific parser.

Using LEXX as an everyday editor has confirmed that editing can be made much easier for users if full advantage is taken of the implied attributes of the data being edited. For many years it has been commonplace to show the structure of a program using indention, but it is rare for this technique to be used for documentation. Similarly, the use of color and fonts can greatly enhance the readability of data presented for editing, but few editing programs provide support for both these means of emphasis.

LEXX is an editor that can be programmed not only to understand and present the structure of data, but also to display those data in a variety of styles and colors in order to best match the data to the user and to the task being performed. The live parsing mechanism means that the changes made by the user can trigger other changes to the data or attributes of the data in a powerful and general way. The result is a programmable editor that can be customized to suit the data, the task, and—most important of all—the user.

References and note

1. A. van Dam and D. E. Rice, "On-Line Text Editing: A Survey," *ACM Comput. Surv.* **3,** No. 3, 93–114 (1971).
2. D. C. Englebart and W. K. English, "A Research Center for Augmenting Human Intellect," *Proceedings of the AFIPS 1968 Fall Joint Computer Conference,* 1968, pp. 395–410.
3. C. N. Alberga, A. L. Brown, G. B. Leeman, Jr., M. Mikelsons, and M. N. Wegman, "A Program Development Tool," *IBM J. Res. Develop.* **28,** No. 1, 60–73 (January 1984).
4. C. F. Goldfarb, "A Generalized Approach to Document Markup," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* **16,** No. 6, 68–73 (June 1981).
5. D. D. Chamberlin, O. P. Bertrand, M. J. Goodfellow, J. C. King, D. R. Slutz, S. J. P. Todd, and B. W. Wade, "JANUS: An Interactive Document Formatter Based on Declarative Tags," *IBM Syst. J.* **21,** No. 3, 250–271 (1982).
6. G. F. Coulouris, I. Durham, J. R. Hutchinson, M. H. Patel, T. Reeves, and D. G. Winderbank, "The Design and Implementation of an Interactive Document Editor," *Software—Pract. & Exper.* **6,** 271–279 (1976).
7. V. Quint and I. Vatton, "Grif: An Interactive System for Structured Document Manipulation," *Proceedings of the International Conference on Text Processing and Document Manipulation,* Cambridge University Press, UK, 1986, pp. 200–213.
8. R. M. Stallman, "EMACS: The Extensible, Customizable Self-Documenting Display Editor," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* **16,** No. 6, 147–156 (June 1981).
9. *Virtual Machine/System Product: System Product Editor Command and Macro Reference,* Order No. SC24-5221-2, 1983; available through IBM branch offices.

10. A. Lippman, W. Bender, G. Solomon, and M. Saito, "Color Word Processing," *IEEE Comput. Graph. & Appl.* **5,** No. 6, 41–46 (1985).
11. ISO—Information Processing—Text and Office Systems "Standard Generalized Markup Language (SGML)," *Draft International Standard,* ISO/DIS 8879 (1985); available from the American National Standards Institute, 1430 Broadway, New York, NY 10018.
12. O. Strömfors and L. Jonesjö, "The Implementation and Experiences of a Structure-Oriented Text Editor," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* **16,** No. 6, 22–27 (June 1981).
13. M. F. Cowlishaw, "Improvements in Text Processing Systems," *UK Patent Application GB-2-043-311-A,* Patent Office, London, 1980.
14. The current implementation holds documents in storage for editing and is therefore limited by the amount of virtual memory available.
15. S. R. Wood, "Z—The 95% Program Editor," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* **16,** No. 6, 1–7 (June 1981).
16. *OS and DOS PL/I Language Reference Manual,* Order No. GC26-3977, 1984; available through IBM branch offices.
17. M. F. Cowlishaw, *The REXX Language,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

Received June 18, 1986; accepted for publication July 13, 1986

Mike Cowlishaw *IBM UK Ltd., Sheridan House, 41–43 Jewry Street, Winchester, Hants, SO23 8RY, England.* Mr. Cowlishaw joined IBM's UK Laboratory at Hursley in 1974, having received a B.Sc. in electronic engineering from the University of Birmingham, England, in 1974. Until 1980 he worked on the design of the hardware and software of display test equipment; simultaneously, he pursued various aspects of the human–machine interface, including implementation of the STET Structured Editing Tool (an editor which gives a treelike structure to programs or documentation), several compilers and assemblers, and the REXX programming language. In 1980 Mr. Cowlishaw took an assignment at the IBM T. J. Watson Research Center, Yorktown Heights, New York, to work on a text display with real-time formatting and also on the specification of new facilities for IBM's interactive operating systems. In 1982 he moved to the IBM UK Scientific Centre in Winchester, England, to work on color perception and the modeling of brain mechanisms. He joined the IBM UK Technical Strategy Unit in the Research Projects Department in 1986. He has recently been on secondment to the Oxford University Press, working on the New Oxford English Dictionary project. Mr. Cowlishaw has received two IBM Outstanding Technical Achievement Awards [one for the conception, design, development, support, and marketing of the REX (former name of REXX) language and the REX Exec Processor, and the other for the development of LEXX], as well as an IBM Invention Achievement Award.