Flexible Text Display with *lector*

Darrell R. Raymond

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

ABSTRACT

lector is a module that provides flexible text interaction for X.11 applications. *lector* handles descriptively marked-up text, supports multiple styles, and interacts well with other programs, including other invocations of *lector*. Appropriate selection of styles enables *lector* to act as text previewer, database browser, code prettyprinter, or menu utility. The result demonstrates the utility of generalized text interaction software.

Keywords: text formatting descriptive markup modular software menu layout user interface

1. Introduction.

Text is ubiquitous in the computer environment. Despite strong interest in iconic interfaces and visual programming methods, text's importance is undiminished. Source code, documentation, electronic mail, electronic billboards and online manuals are clear cases of text-rich applications, but text is also important in its small and frequent instances such as online help, menus, diagnostic messages, and the output of operating system utilities. The pervasiveness of text argues for powerful, general methods for text display and interaction.

Text-rich systems have tended to cling to long-established applications constraints, rather than evolving towards more general methods. Source code, for instance, is still generally handled with character-based text editors, despite extensive research into structure-based editors and code prettyprinters. Online documentation tends to be supported by batch-oriented typesetting programs and associated previewers. Program diagnostics and usage messages are hardwired in code and displayed by dumping to terminal programs or files.

There are often good reasons for different approaches to text, but the result has been an overall lack of consistency in output quality and interaction. Standard text editors, for example, permit interactive editing, whereas compiled-in code and the output of operating systems commands do not.[1] Typesetting programs and previewers provide infinite variety in style, while editors are usually restricted to a single font and style of display. Both editors and typesetters are too slow or bulky for use in small or dynamic applications, such as error messages and menus. Systems such as electronic mail and bulletin boards tend to converge on the lowest level of device capability, so they can be used over a wide range of output devices. Hardwired text is very efficient but typically provides only a bare minimum of formatting capability.

Inconsistency makes it difficult to raise the overall quality of text in the computer environment. Since text display is usually not the primary activity of applications, and since general solutions are often not available, programmers quickly settle for familiar solutions. Thus in order to significantly affect overall text quality, we must provide both software and philosophy that encourages better handling of text. Programmers should have simple and effective means for high-quality text interaction, and should be persuaded that the advantages of improved display quality are worth the effort.

We have been exploring the unification of a range of text management situations with *lector*, an X.11 application for flexible text interaction. *lector* was originally devised as a simple, rapid formatter for the display of online text databases. As our experience with *lector* developed, we discovered that it exhibited useful capabilities for a wide range of text applications. While *lector* itself is probably not an optimal solution, it has interesting implications for the possibilities for generalizing text interaction. These possibilities are described in the remainder of this paper.

2. A tour of *lector*.

lector takes as input a text and a style specification; as output it displays the formatted text in a single X.11 window. Figure 1 shows the entry for *comma* from the *Oxford English Dictionary*, reproduced both from the printed page and as seen in a *lector* window. The text contains bold, italic, and roman typefaces, as well as characters from the Greek and International Phonetic Alphabet, presented in two point sizes. *lector* reproduces most of these elements quite legibly, but does not attempt to provide an exact simulation of the printed page. Some notable differences are the use of a sans serif font, the lack of right justification, and lack of accents on Greek characters.

In addition to simulating the printed text, *lector* supports multiple alternative styles, four of which are shown in Figure 2. These are a skeleton style, a quotations style, a style suppressing quotations, and a view of all text and tags, showing the typesetting actions applied to the tags that caused the actions. This latter style can be used to debug the style specification itself. Figure 2 also shows *lector*'s menus, with the menu pane containing the list of styles on top.

Figure 2. Styles for viewing *comma*.

Note that the windows in Figure 2 are of different sizes and employ multiple columns. *lector* formats text to fill the user's chosen size of window, responding dynamically to window resizing rather than filling a predetermined page. Page breaks, line breaks, and column size are determined interactively, unlike typesetting previewers that operate with predetermined page and line breaks.

The input text is marked with structural tags that are bound to typesetting actions through a style specification file. The tags are compatible with Standard Generalized Markup Language or SGML.[2] Figure 3 shows the text of the entry *comma* as it has been tagged at the University of Waterloo. The structural tags may be purely logical (so-called "descriptive" markup; e.g., the <Q> tag for "quotes" in Figure 3), or they may be tightly bound to typesetting semantics (so-called "procedural" markup; e.g., the <i> tag for "italic" in Figure 3). All that is required is that the tags be uniquely distinguished from the remainder of the text.

The style specification file contains definitions of fonts and styles. A small part of the style specification file for the Oxford English Dictionary is shown in Figure 4. Each font is given several symbolic labels to be used in the remainder of the specification. Each style consists of a starting format, a set of tags, and a set of typesetting actions to be taken upon encountering the tags. A relatively small set of typesetting actions is supported, including the ability to change font, to display a fixed string of text, to vary indentation levels, and to change vertical or horizontal spacing.

```
<E><HG><HL><LF>comma</LF><SF>comma</SF><MF>comma</MF></HL><MPR>
k<i>o&hook.</i>&sd.ma&breve.</MPR><IPR><IPH>&sm.k&rfa.m&schwa.<
/IPH></IPR>. </HG><VL>Pl. <VF>commas</VF>(formerly <VF>-aes</VF
>); as L. or Gr., <VF>commata</VF><MPR>k<i>o&hook.</i>&sd.ma&br
eve.ta&breve.</MPR><IPR><IPH>&sm.k&rfa.m&schwa.t&schwa.</IPH></
IPR>. </VL><ET>a. <L>L.</L><CF>comma</CF>,<L>Gr.</L><gk>ko&acu.
mma</gk>stamp, piece cut off, short clause, etc.:-* <gk>ko&acu.
p-ma</gk>, f. <gk>kop-</gk>root of <gk>ko&acu.ptein</gk>to stri
ke, cut.</ET><p><S4><#>1</#><S6><DEF>In <CF>Greek</CF><LB>Rhet.
</LB>and <LB>Prosody</LB>: A phrase or group of words less than
a colon (q.v.).  Hence, &dag.A short member of a sentence or pe
riod.</p></DEF><QP><EQ><Q><D>1586</D><A>A. Day</A><W>Eng. Secre
tary</W> <sc>ii. </sc>(1625) 85 <T>The last word of a comma, or
member of a sentence.</T></Q></EQ><Q><D>1607</D><A>Shaks.</A><W
>Timon</W><sc>i. </sc>i. 48 <W>Poet.</W><T>No leuell’d malice I
          nfects one comma in the course I hold. </T>
```

Figure 3. Tagged text for *comma*.

The display can be manipulated in several ways. The window itself can be arbitrarily resized or repositioned, causing the text to be redisplayed using the newly defined width and height. Resizing is also a convenient way to set the column width; the user sizes the window and then presses a key to adjust the column size to the current window width. Multiple columns in the new width can be obtained by resizing for a wider window. The user can page forward and backward, or jump to the start or end of the text, by using the appropriate function keys, mouse buttons, or menu selections.

lector has one other special navigational feature of note: the user can point to a line of the displayed text and cause the display to update so that the indicated line is at the top of the window. When combined with a style such as the skeleton style of *comma*, this feature permits rapid navigation. The user switches to a skeleton style, points to the desired subsection, and then returns to a full content style for detailed perusal.

```
# Font definitions

<Font>
      <Name>-oed-helvetica-medium-r-scaps--12-*</Name>
      <Family>Helvetica</Family><Type>Small Caps</Type><Size>12</Size>
</Font>
<Font>
      <Name>-oed-helvetica-medium-r-normal--12-*</Name>
      <Family>Helvetica</Family><Type>Roman</Type><Size>12</Size>
</Font>

# Standard style

<Spec>
#     Startup values

      <Name>Standard</Name>
      <Family>Helvetica</Family><Type>Roman</Type><Size>14</Size>
      <PrText>on</PrText><Tabs>3 6 9 12 15 18 21 24 27 30</Tabs>

#     Tag definitions

      <Tag><Name><E></Name><String>   </String></Tag>
      <Tag><Name></E></Name><LineBreak>on</LineBreak></Tag>
      <Tag><Name><HL></Name><Type>Bold</Type><Size>14</Size></Tag>
      <Tag><Name><IPR></Name><String>(</String></Tag>
      <Tag><Name></IPR></Name><String>)</String></Tag>
      <Tag><Name><Q></Name><Size>12</Size></Tag>
      <Tag><Name><D></Name><Type>Bold</Type></Tag>
      <Tag><Name></D></Name><Type>Roman</Type></Tag>
      <Tag><Name>&Ae.</Name><String>&198.</String></Tag>
      <Tag><Name>&Beta.</Name><String>&66.</String></Tag>
      .
      .
</Spec>

# No Quotes style

<Spec>
#     Startup values

      <Name>No Quotes</Name>
      <Family>Helvetica</Family><Type>Roman</Type><Size>14</Size>
      <Tabs>3 6 9 12 15 18 21 24 27 30</Tabs>
      .
      .
</Spec>
```

Figure 4. Style specification file for Oxford English Dictionary.

Software for text display must handle three basic structures. The first is the content itself, including its subdivisions and ordering, as well as the rules that distinguish content from embeded layout instructions or other codes. Second is the layout structure, which describes the basic display features of the software and the ways that these features can be combined (i.e., the programs that can be written). Third is the structure of potential interaction with the text, the role that the text software plays for the user and for other software packages.

Text systems differ in their handling of the three basic structures. Hardwired text, for example, treats strings and characters as the basic subcomponents, supports character-based layout codes, and has a single interaction possibility: the text can be read. Typesetting systems, on the other hand, allow the user to define many types of subcomponents, such as keeps, tables, and figures, and they have powerful layout functionality. Their interaction structure, however, is virtually the same as for hardwired text. Online text databases support a few types of subcomponents (documents, paragraphs, sentences, words) and query interaction, but typically have limited layout capabilities. *lector* is a unique combination of features for the three structures.

3. Content.

lector distinguishes between content and tags. Tags are any strings that conform to a given syntax. Tags that are explicitly declared in the style specification are *defined* tags; strings that do not appear in the style specification, but which conform to the syntax, are *undefined* tags. All tags, defined or undefined, are context-free; nesting and tag-pair relationships are not maintained. All strings that are not tags are content. Content fragments are displayed in the order that they appear in the input, and cannot be reordered.

Where most systems have a fixed syntax for defining markup, *lector* permits the user to define (within limits) the syntax of the tags. In other systems, undefined markup is illegal; in *lector*, undefined tags can cause some typesetting actions to take place. Since it is not possible to distinguish between instances of undefined tags, any instance causes the same action. Most systems supply a limited amount of state maintenance, such as the ability to save and return to a previous font. *lector* does not preserve state. This strategy was chosen partly to ensure that *lector* would robustly handle erroneous texts, and partly to simplify formatting under arbitrary navigation.

In the present implementation, a tag syntax is a unique pair of start and end characters. SGML-like angle brackets are commonly used. The present implementation also permits multiple syntaxes to be used in each style, and styles can use different syntaxes. By varying tag syntaxes it is possible to treat markup as a kind of content.

4. Layout.

Layout rules are specified in the style specification file. This is a user-defined mapping between markup syntax and semantics. The style specification describes the tag syntaxes, fonts, and styles that are employed in a given invocation. The styles are intended to be simple to write; they are declarative in nature, with no control flow statements, variables, procedures, or types. Each tag, font, and style specification is independent of all the others. A style specification is essentially a symbol table containing a mapping of tags to typesetting operations.

An example of a style specification file is shown in Figure 4. Style specifications can be quite lengthy if many tags and styles are defined. One such file for the Oxford English Dictionary consists of nine styles, each defining nearly 200 tags.

The bulk of the style specification file is a set of operations to be performed when specific tags are encountered. A small set of operations is supported, including change of font (absolute or relative to the current font), highlighting (currently implemented as reverse video), change of indenting level, "page break" (i.e., stop filling the window), printing of an arbitrary fixed string, and vertical or horizontal motion. These few features are sufficient to display many types of running text. A very important additional feature is suppression, the ability to discontinue or restart the display of text at a given tag.

Many other text systems rely on so-called "procedural" markup, in which the layout actions are bound to system-defined tags. *lector* allows the user to define the mapping between layout actions and markup. Another difference is the small number of typesetting capabilities provided by *lector* — the quality of its output is mainly due to the use of multiple proportional fonts. Full-fledged typesetting systems usually provide many more capabilities, such as keeps, page furniture, right justification, kerning, automatic hyphenation and so on. *lector* lacks these, but provides user-controllable suppression of text, an essential capability in interactive use and in the definition of multiple styles. Finally, in other systems markup is usually suppressed by default. *lector* facilitates both the display of markup and the definition of a variety of display styles for that markup.

5. Interaction.

Flexible interaction is *lector*'s main theme, and is achieved in several ways. First, its easy readjustment to the current window size encourages the user to choose a window size that suits the purpose. Second, rapid access to multiple styles and the use of repositioning makes it possible to navigate the text quickly, without excessive paging or the use of a scroll bar. Third, in addition to displaying text, *lector* supports the use of the text as an input medium. In menu mode, pointing to a line in the text causes the output of the file offset of that line. This information can be directed to other programs to perform some related task. In effect, the text has become a menu. Finally, *lector* can accept either new texts or new positions on its input stream; thus, an existing invocation can be reused many times, and an existing text can be redisplayed under program control.

Most previewers require the whole text to be preprocessed into displayable format before they can begin display. *lector* can begin formatting at arbitrary points in an unprocessed text, and reads only as much of the text as is necessary to fill the current window. Most previewers simply display the text, limiting interaction to paging. *lector* treats text as an interactive display, and uses styles to support more powerful navigation. Finally, where most previewers are restarted for each text instance, *lector* is most often used as a persistent utility for many texts. This cuts down on process invocation costs and relieves the user of the need to manage new windows.

6. Applications.

6.1. Online documentation.

lector is useful for the display of online documentation, since its multiple formats and index mode permit flexible navigation. Most existing online documentation is not tagged with descriptive markup, or uses tags whose syntax is more complicated than the present implementation can recognize. One such collection of text is the Unix man pages, which are tagged with the *-man* macros. Such texts can often be preprocessed with shell scripts to install more tractable tags.

Figure 5 shows the *csh* man page as seen via a combination of *awk* preprocessor and *lector*. Three styles are shown: a skeleton style, for use as an index; a standard view, for reading; and a style showing the tags installed by the shell scripts. These invocations are pipelined so that pointing to the skeleton style results in the two other styles being updated to the indicated position. This interconnection assists in searching long man pages, which are usually consulted simply to check a point of syntax.

Converting the *-man* macros via a preprocessor may appear to be an unnecessary, complicating step. One benefit of this approach is that it separates the definition of the styles from the *-man* macro syntax, and so the styles can be resued for other documents. TEX documents, for example, could be displayed using the same styles, if a preprocessor was written to convert TEX macros to the tagging used by the style specification. Separating markup syntax from markup semantics enables style definition to be treated as an independent activity.

Figure 5. *lector* styles of *csh* man page.

Figure 6. *lector* styles of C code.

6.2. Source Code.

On many computer systems the largest body of text is source code. Comprehension of this text is thought to benefit from improved presentation.[3] Code prettyprinters typically transduce the syntax of a specific programming language into a specific typesetting language. The output is also usually confined to a single style, is not sensitive to window geometry, cannot flexibly suppress code fragments, and does not employ the display as an input medium. The examples shown in Figure 6 uses *lector* to support all these capabilities for previewing C. As in the case of the online documentation, the source code is preprocessed before being sent to *lector*. Since C code is syntactically more complex than typesetting macros, a *lex* preprocessor is employed to mark up the program. The preprocessor tags line breaks, reserved words, braces, comments, strings, and other simple syntactic features.

The code displayed in Figure 6 is the main routine of the X.11 server, a reasonably non-trivial nested program. The *C Grind* style presents a traditional layout of the code. The *Nesting* style is a compressed view, showing only the major control flow statements and their nesting. Multiple fonts and point sizes are used in the *Nesting (Reduced)* style, which shows the main control flow statements in a large proportional font, with all other statements in the tiny *nil* font. This style gives the programmer a feel for both the nesting and the relative sizes of the code blocks. Finally, there are styles displaying the comments and the string constants in the code.

lector supports both user-adaptable and language-independent prettyprinting of source code. [4, 5] The style specification is provided by the user, so styles are inherently user-adaptable. Language independence results from using installed markup, rather than developing styles based on a specific programming language syntax. In effect, the tags are a kind of indirection table mapping language syntax to conceptual structures. As a result, the same styles can be used for viewing many languages; new languages are accommodated simply by constructing the appropriate preprocessor.

lector's simple model of text is an unexpected advantage for dealing with source code. Prettyprinters or editors that depend on correctly nested code have trouble with incorrectly nested programs, as is often the case with programs under development. *lector*'s specifications assume no nesting, so they are more robust in the face of nesting errors. Some code viewing tasks, however, expose limitations in *lector*'s approach. Since tag definitions are static, *lector* cannot support the highlighting of individual variables or function names dynamically specified during an interactive session. [6]

6.3. Menu utility.

lector can be used as a generic menu utility. To implement such a menu, a tagged text file containing the menu options is constructed, and note is taken of the file offsets of each option. *lector* presents the menu to the user, who selects an option by pointing to a line of text. The corresponding offset is output to the application program, which then takes the desired action.

Figure 7. *lector* menus for structured editor.

Figure 7 shows *lector* used a menu utility for a structured text editor. There are three invocations of the editor in the figure. Each invocation shows the editor window and a *lector* window, the latter displaying an abbreviated set of the editor's commands. The rightmost invocation shows the menu to the left of the editor; the bottom invocation shows the menu above the editor. The only difference between these two menus is the number and size of columns displayed. The leftmost invocation of the editor uses a different style for viewing the menu text. This style displays the editor formats which can be used for viewing the text, all of the editor's commands, and the help text associated with each command. Not all of this information fits in the chosen window size, but the user can page, changes styles, or use *lector*'s other navigational features to assist in scanning the menu file.

The editor's functions were originally accessed by a pop-up, multi-pane menu interface that was excessively slow and clumsy. The *lector* menu has several advantages. First, it relieves the editor of the need to handle menu interaction, simplifying the editor's design. Second, the menu can be configured by the user, who can choose the position, size, and orientation of the menu; this is how the two rightmost invocations were arranged. Third, the content of the menu can be adjusted through style changes; suppression can be used to hide menu options, help text, or other information to simplify the menu. Moreover, all of these features can be configured without access to the editor's source code.

The main disadvantage of *lector* menus is that they require a process invocation, and hence are slower to start up and consume more resources than linked libraries. On the other hand, *lector* menus encourage modularity by isolating menu activity from the rest of the application. In addition, *lector* menus exhibit several novel and desirable properties. They support multiple styles, run-time layout arrangement, embedded information, and flexible option suppression by means of style changes. The built-in capabilities for scrolling, paging, and multiple columns enable *lector* to easily accommodate large menus.

7. Discussion.

lector embodies a four-part philosophy of text display and interaction.

Modularity. *lector* is useful for building weakly coupled text applications constructed of reusable modules. Such systems can be quickly prototyped and are easily modified. Weak coupling tends to concentrate display-specific activity in *lector* and semantic operations in the application itself. *lector* attempts to do only text display and interaction well, relying on other modules to handle other activities. An additional level of modularity is the explicit separation of style from content.

Efficiency. Fast, efficient text display was a basic design goal. To this end, a minimal set of typesetting features was supported. Furthermore, conservative memory policies were employed, leading to processing documents on the fly rather than preprocessing them. Additional efficiency is gained by calling X.11 directly through *Xlib*, bypassing the use of a toolkit. As a result, the executable is very small (a current SPARC version is 68 kbytes) and fast, especially useful properties in environments without shared libraries or where hardware is at a minimum. *lector* is fully capable of being used on low-end X terminals with minimal memory. The ability to re-use an invocation for a new document adds extra savings on process startup time, memory management, and most importantly, the user's effort in window management.

Interactivity. *lector* treats text as a pliable medium that is useful both for input and output. This necessitated layout that is responsive to window geometry, rather than to a predetermined page size. It also encouraged the development of suppression and other style features that expand or contract the view of the text. In all cases, pointing at a text has a defined result, which can be employed to make the text an index to itself or to an application's operations. The ability to work well with other modules is another important form of interactivity.

Flexibility. *lector* tries to address as many text display problems as possible, including online documentation, scholarly texts, system utilities, and code mangement. A wide range of applications enforces simple but broadly applicable approaches to specifying, displaying, and interacting with text. Flexibility is greatly enhanced by modularity, efficiency, and interactivity, but is also a design constraint on its own. Its main effect is to temper the tendency to add application-specific features to *lector*.

Many current text management systems attempt to incorporate modularity, efficiency, interactivity, and flexibility. Object-oriented systems encourage reusable, modular text processing facilities. [7] Structured editors provide text manipulation based on abstract levels of structure. [8] Advanced typesetting systems increase the quality and variety of text layout. Text database software promotes the notion of the text itself as a modular entity, capable of being retrieved and displayed in a variety of ways.[9] *lector*'s approach differs from these other systems in two major ways.

The first difference between *lector* and other methods for handling text is its weak coupling to applications and its use of process decomposition for rapid development. The primary advantage of closely coupled applications is efficiency. The primary disadvantage of such applications is their monolithic construction and lack of independence from the underlying environment. Smalltalk and InterLisp programmers sometimes complain that applications developed in integrated environments are not easily portable outside the environment.[10] Weakly coupled *lector*-based applications, on the other hand, are highly portable. Since process communication is employed as the coupling interface, these applications are also easily distributed across workstations and file servers.

The second difference is that *lector* treats text and menus as views of a single entity. Especially in object-oriented hierarchies, graphical and textual display are distinctly different operations or objects. Menus are traditionally considered graphical control or system objects, while texts are data or user objects. *lector*'s textlike menus and menulike texts show that this decomposition is not necessarily appropriate. By treating menus and texts as interchangeable objects, we gain the advantages of interactivity for texts and the advantages of flexible layout for menus.

Both *lector*'s philosophy and implementation could benefit from further work. The style specification language is most in need of redesign, having evolved in an ad hoc fashion. The existing language capabilities needs to be enhanced to handle more types of text, and to handle more types of formatting. Some thought should also be given to reducing the difficulty of style creation. This has often turned out to be a non-trivial task, especially when many styles are defined for a complex document. If style creation becomes an important form of erogramming, then it is advisable to support it with a more carefully constructed language.

lector's interaction capabilities are still too limited, particularly in providing feedback about the user's pointing activity. Many text-rich applications such as mail and electronic billboards require better input than *lector* can at present provide. For such applications, weak coupling can lead to a clumsier design. Yet it seems worthwhile to provide multiple styles of news and mail, or to use these styles as indexes into a database. Similarly, we saw that the separation of style from markup was a stumbling block in display of specific identifiers in code previewers, though it is otherwise a natural method for defining multiple styles.

Much of *lector's* design emerged through juggling tradeoffs between performance and generality. It is important to keep this tradeoff in mind as new improvements are considered. Adding right justification of text, for example, would provide a neater appearance to some output. Its contribution to readability is questionable, however, and its execution cost is relatively high, especially if one attempts to avoid rivers of white space or to provide good hyphenation. Another example of a problematic enhancement is display of tables. Tables can be quite informative, but are not robust to all style changes. [11] The effect of table layout must also be considered in the light of *lector*'s ability to extend into display tasks not typically encountered by text previewers. How should table layout be handled in the face of frequently changing window geometry, for instance? What sorts of table facilities would prove useful in the layout of iconic displays? Identifying the most flexible solution is a difficult task, but is ultimately worthwhile if the goal is the unification of a wide range of text-based applications.

8. Conclusions.

lector was created to support a specific interactive text application. As we developed an appreciation of the general need for powerful text display and interaction, *lector* matured into a module useful in a variety of applications. *lector* exploits the use of descriptive markup for structuring text, and shows the potential advantages of this technique for menus, interactive displays, database browsers, and code prettyprinters. Focusing our attention on general text display and interaction problems has provided us with unexpected insights into a wide variety of application problems.

9. Acknowledgements.

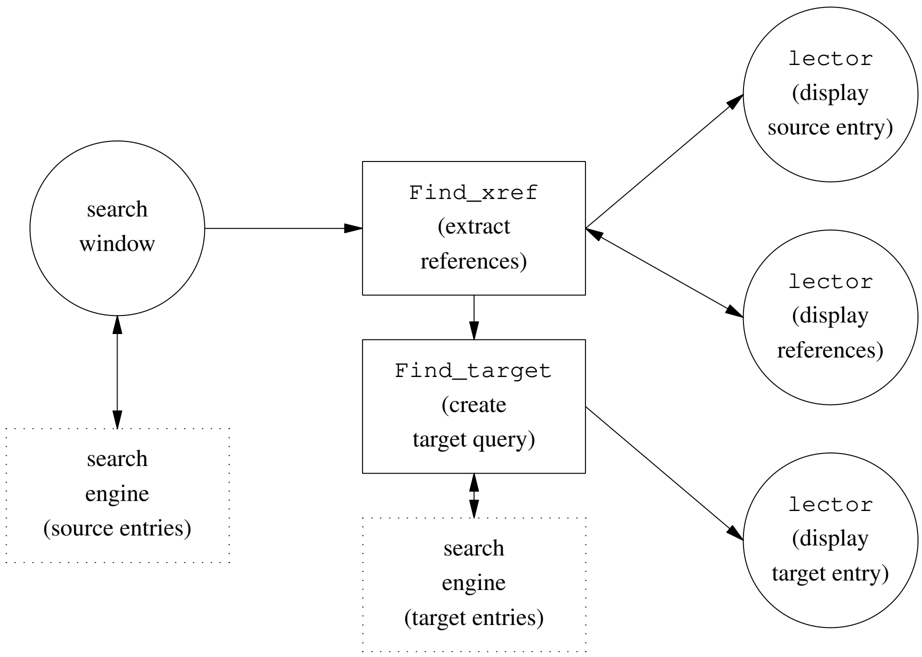
lector's design and implementation were influenced by the comments and suggestions of all the members of the Centre for the New Oxford English Dictionary. My particular thanks go to Heather Fawcett, who suggested interconnecting multiple *lector*s, to Gord Broom, who built several of the applications, and to Frank Tompa, for his continual stream of ideas and comments on many early drafts. Extensive criticism by the anonymous referees also significantly improved this paper. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada through the University-Industry Program under grant 0039063.

Multiple process structuring

Multiple process structuring is a long-established technique that supports rapid application development and software reuse. In a multiple process environment, component programs are designed to be general utilities of broad applicability, they share conventions about input and output, and they are supported by a simple mechanism for process interconnection.[12] *lector* was designed to participate fully in such an environment.

The power of multiple process structuring techniques is evident in the design of the cross-reference finder for the *Oxford English Dictionary*. In the figure, the top-left window (the only one that is not an invocation of *lector*) is used for entering search commands. Entries containing matches to string searches are displayed in their entirety in the bottom lefthand window. As these entries are displayed, they are filtered by a small program that extracts the cross-references from the entry. The extracted information is then tagged and sent to the middle window, where it appears as a menu of possible targets. Pointing at one of these cross-references results in a search for the target entry, which is then displayed in the rightmost window. In the figure, the entry for *lecturer* contains four cross-references, whose labels are displayed in the middle window. The user has just selected the cross-reference to *lector* and the target entry has been displayed in the target window.

The process structure for the cross-reference application is shown in the accompanying diagram. The search window is used to locate a source entry of interest. The program `Find_xref` extracts the cross-references from the source entry and sends them to the menu; it also forwards the entry to the source entry display window. When a reference is selected from the menu, `Find_xref` is informed and it calls `Find_target` to locates the target entry. This is done using a second search engine. The result is then shown in the target display.



Developing the cross-reference application was greatly simplified by using interconnected multiple processes. The two filtering programs are no more than a few hundred lines of code, written in less than a week. Another week was spent developing a simple shell to set up the process structure (standard Unix shells are not very useful for constructing non-linear process structures). The speed of implementation is only one benefit, however. Another is that the system is easily distributable. The search engines, filters, and display processes can reside on separate machines — even the *lector* processes need not run on the workstation that is used for display, since their connection to the X server is also based on multiple process communication. The application exhibits a high degree of modularity; display logic is isolated in *lector*, search logic in the search engines, and cross-reference resolution logic in the two filter programs. As a result, the application can be easily modified and extended, even by programmers who do not have access to the source code of the search engine or *lector*.

Flexible display of graphics

The advantages of multiple styles, multiple windows, and the use of the display as a menu need not be restricted to text. Given a font with appropriate iconic elements, *lector* can support some interesting graphical interfaces as well.

The illustration shows several views of the chess game played between the programs Phoenix and Deep Thought at the 20th annual ACM North American Computer Chess Championship.[13] Each view in the figure is an invocation of *lector*, including the displays of the game board. A simple program was written to convert standard chess notation to a tagged text description of the board at each state of the game, and that file is the source of each of the views shown in the figure. The text style shows the full content of the tagged text file. The notation style uses suppression to show only the individual moves, while the annotation style suppresses all but the annotations. The board styles result from mapping the board descriptions to appropriate glyphs in a chess font; shown are a complete board view and a view of only the knights and bishops. Multiple styles give the chess player many ways to study the game.

Navigation through the game is assisted by the interconnection of the windows. The player can page backward or forward in a board view, which shows the next or previous move. Paging in the notation or annotation view permits jumping several moves at a time. Pointing to a specific move or annotation results in positioning the board views at that point in the game.

The chess application is designed for use with a game log, but could easily mediate a game in progress. Simple filters could be written to convert move notation to board descriptions, and to convert from the user's input to move notation. Flagging of invalid moves and records of captured pieces can be provided by extending the filters to embed appropriately marked-up information in the text, and then devising a style to display the feedback. Captured pieces, for example, could be presented by showing their icons or by displaying a bar chart showing the total value captured.

Advanced text display systems provide interesting insights into new types of flexibility for graphical user interfaces. In most such systems, graphical objects such as buttons, sliders, check boxes, and scroll bars are hardwired into a program. User configurability, when it is provided, is often tediously specified by means of a default file. An alternative approach, based on flexible text display, is to specify a graphical interface or control panel with a text file that is mapped to appropriate iconic fonts, as in the chess example. By treating graphical interfaces as a kind of text, we gain a number of features not found in current direct manipulation interfaces. Such a system supports the display of very large control panels (accessed by means of paging or window resizing), the selective display of icons and controls (by means of suppression), a choice of panel layouts (by means of multiple styles), and user-selectable icons (by means of font changes). Moreover, each of these capabilities is supported in exactly the same way as it is in the case of text display, making it easier for users to transfer their abilities from one task to another.

The management of graphical interfaces as a kind of text has important implications for system decomposition. Instead of constructing a program to manage a panel of buttons and sliders, one constructs a program to manage a ‘‘text’’ describing the logical structure of that panel. This text is essentially a display-independent specification of an iconic interface. Instead of compiling fixed rules about panel layout into a program, one constructs a style specification which describes a number of possible layouts, subject to display limitations (i.e., window geometry). This style specification is essentially an application-independent specification of layouts for a class of related interfaces. By focusing attention on these two interfaces, we take another step towards the interchangeability of text and interface objects.[14]

References

1. Christopher W. Fraser and Balachander Krishnamurthy, “Live Text,” *Software — Practice and Experience,* 20, 8, pp. 851-858 (August 1990).
2. ISO, Richard Rubinstein, “Information processing — text and office systems — Standard Generalized Markup Language (SGML),” ISO 8879-1986, International Organization for Standardization, Reading, Massachusetts (1986).
3. Ronald Baecker and Aaron Marcus, “Design Principles for the Enhanced Presentation of Computer Program Source Text,” *Proceedings of the CHI ’86 Conference on Human Factors in Computing Systems,* pp. 51-58, Boston, Massachusetts (April 13-17, 1986).
4. G. Blaschek and J. Sametinger, “User-Adaptable Prettyprinting,” *Software — Practice and Experience,* 19, 7, pp. 687-702 (July 1989).
5. Matti O. Jokinen, “A Language-Independent Prettyprinter,” *Software — Practice and Experience,* 19, 9, pp. 839-856 (September 1989).
6. Darrell R. Raymond, “Reading Source Code,” IBM Canada Laboratory Technical Report TR 74.070, IBM Canada, Toronto, Ontario (October 1991).
7. Mark A. Linton, John M. Vlissides, and Paul R. Calder, “Composing User Interfaces with InterViews,” *IEEE Computer,* 22, 2, pp. 8-22 (February 1989).
8. Vincent Quint and Irène Vatton, “Grif: An Interactive System for Structured Document Manipulation,” *Proceedings of the International Conference on Text Processing and Document Manipulation,* pp. 200-213, University of Nottingham (April 14-16, 1986).
9. Dennis E. Egan, Joel R. Remde, Louis M. Gomez, Thomas K. Landauer, Jennifer Eberhardt, and Carol C. Lochbaum, “Formative Design Evaluation of SuperBook,” *ACM Transactions on Information Systems,* 7, 1, pp. 30-57 (January 1989).
10. “Panel: Confessions — What’s Wrong With Our Systems,” *Hypertext ’89,* Pittsburgh, Pennsylvania (November 5-8, 1989).
11. Richard J. Beach, “Tabular Typography,” *Proceedings of the International Conference on Text Processing and Document Manipulation,* pp. 18-33, University of Nottingham (April 14-16, 1986).
12. Brian W. Kernighan, “The Unix System and Software Reusability,” *Proceedings of the Workshop on Reusability in Programming,* pp. 235-239, Newport, Rhode Island (September 7-9, 1983).
13. Monty Newborn and Danny Kopec, “The Twentieth Annual ACM North American Computer Chess Championship,” *Communications of the ACM,* 33, 7, pp. 92-104 (July 1990).
14. Eric A. Bier and Aaron Goodisman, “Documents as User Interfaces,” *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography,* pp. 249-262, Gaithersburg, Maryland (September 1990).