`lector` — An Interactive Formatter for Tagged Text

Darrell R. Raymond

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

ABSTRACT

`lector` is an X.11 application that provides highly interactive text formatting. Unlike text previewers, `lector` handles descriptively marked-up text, supports multiple styles, and interacts well with other programs, including other invocations of `lector`. Appropriate selection of texts and styles enables `lector` to act as text previewer, database browser, code prettyprinter, menu utility, and iconic interface. `lector`'s implementation revolves around a set of tradeoffs involving efficiency, simplicity, and generality. The result demonstrates the utility of generalized text display tools.

1. Introduction.

Displaying text is one of the most troublesome, yet indispensable, parts of programming. Virtually every program generates some text, if only diagnostic or error messages. As a result, virtually every programmer must make some choice of output device and layout. These choices are becoming increasingly complicated, thanks to the staggering variety of terminals, printers, plotters, and bitmapped workstations available in modern networked environments. Typically physical devices are not dealt with directly, but are accessed instead through one or more layers of abstract device, including character-based codes such as ANSI, custom libraries such as *curses*,[3] page description languages such as PostScript,[2] text formatters such as TEX,[14] *troff*,[18] or Rich Text Format,[16] and window systems, such as X.11[24] and NeWS.[26] The variety in abstract devices compromises their effectiveness at reducing the output problem. The situation is further complicated by the idiosyncratic collection of programs intended to transform data between abstract devices.

Choosing a layout is also difficult, in part because layout depends on device capabilities. Even the ubiquitous 24x80 terminal presents numerous layout possibilities, since it supports columnation, justification, hyphenation, tables, and highlighting. With workstations or other advanced devices one may also have to deal with proportional fonts, kerning, colour, graphics, and even animation. Whatever the device's capabilities, a key layout issue is user configurability. Configurability is typically ignored by programmers, who often provide only a single layout, or perhaps a few pre-canned layouts selected via command-line options. Users who want to configure their output then typically resort to "programming" the device by embedding device-specific commands in the text.

Though the text display problem is shared by most programs in an environment, it is not usually solved in a shared fashion; programmers readdress the problem with each new application. This results in inconsistent display quality, device-specificity, and erratic levels of configurability. Since text display is not always the primary purpose of an application, programmers tend to choose the simplest acceptable device and layout, clearly a shortsighted approach.

This report describes `lector`, a text display utility designed for use in a multitasked workstation environment. `lector` is similar to document previewers such as *psview*[1] and *xdvi*,[8] providing a reasonable facsimile of typeset text in a bitmapped window. `lector` differs from existing previewers, however, in emphasizing interactivity rather than painstaking simulation of a printed page. One aspect of interactivity is speed; `lector` processes text more quickly than most other previewers, so it can be used in places where other previewers would be too slow. Another aspect is size; `lector` contains a minimal number of typesetting capabilities and makes other concessions for the sake of compactness. As a result, it is reasonable to develop applications consisting of many invocations of `lector`. Speed and size are less important than the flexibility conferred by `lector`'s use of a style sheet to define multiple views of a text. These styles are useful not only for viewing, but can be employed as interactive indexes to the text.

`lector`'s interactivity enables it to fill roles not normally accommodated by previewers. `lector` finds extensive use as an output engine for full text searching programs, for example.[20] With the appropriate preprocessing filters, `lector` can become an integral part of a hypertext system; multiple invocations of `lector` have been used to support view-defined navigation in texts.[21] `lector` can also be used as a general menu utility or as a graphical display tool for applications such as chess games. `lector` can be delegated much of the text display activity normally handled by an application, relieving programmers of the need to support output devices and user-definable layouts. It is probably unwise to expect to handle every text display problem with a single tool, but `lector` supports a broad class of common activities. In summary, `lector` embodies the following philosophy of text display:

> *separation of display from application* — display and layout intelligence is concentrated in `lector`, rather than in each application.
>
> *multiple views* — text is viewed in one of several user-defined text-dependent styles.
>
> *balance of typesetting ability and interactivity* — `lector` sacrifices exact mimesis of the printed page for interactivity.
>
> *process level decomposition* — `lector` is accessed by other applications through interprocess communication.

2. A tour of `lector`.

`lector` takes as input a file of 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*,[25] 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 in quite readable fashion, 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 accents missing from the Greek characters.

In addition to simulating the printed text, `lector` supports multiple alternative views of the text. Four invocations of `lector` displaying some of these views are shown in Figure 2. The views include a skeleton view of *comma*'s structure (including various combinational forms); a view of the quotations; a view suppressing the quotations; and a view of all text and SGML-like tags, showing the typesetting actions active on the tags that caused the actions. This latter view is sometimes employed for debugging the style specification. Figure 2 also shows `lector`'s menus, with the menu pane containing the list of formats or styles on the top.

Note that the windows in Figure 2 are of different sizes and employ multiple columns. Part of `lector`'s flexibility is that it formats text to fill the user's chosen size of window, rather than filling a predetermined page. Page breaks, line breaks, and column size are determined interactively, unlike previewers that operate with predetermined page and line breaks. This feature permits `lector` to respond dynamically to window resizing.

The input text to `lector` is marked up with structural tags that are bound to typesetting actions through a style specification file. 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

Original text   `lector` view

Figure 1. *comma* from the OED.

"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 *OED* 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 list of tags, and associated typesetting actions to be taken upon encountering the tags. The typesetting actions include the ability to change

Figure 2. Views of *comma*.

the font, to display additional text, and to vary indentation or add 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 `lector` display can be manipulated in several ways. The window itself can be arbitrarily resized or repositioned using mouse buttons or function keys. The specific keys or buttons used for resizing and moving the window are determined by the the user's choice of window manager, rather than by `lector` itself. Each window resizing causes the text to be redisplayed using the newly defined width and height. Resizing also serves as a convenient way to request a new column size; 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.

Figure 5 shows `lector`'s cursors. Normally one of the cursors depicting a mouse is displayed. These cursors show which mouse buttons are active (filled buttons are active), and the symbols under the buttons convey the button's action — paging up, paging down, or accessing the menu. The mouse cursors also give rough information about `lector`'s current position in the text; for example, if the paging-up button is not filled, then the current display position must be the beginning of the text.

Figure 5. Cursors.

The crosshair cursor indicates that `lector` is in *index* mode. In this mode, when the user points to locations in the window, the corresponding file offset of the selected string is echoed to standard output. This mode is often useful when `lector` is employed in conjunction with other

```
# 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 OED.

processes. The stopwatch cursor is displayed whenever `lector` is recomputing the display. Since display computation is fast, this cursor is usually not noticed by the user. However, if the system load is high or if a large volume of text must be scanned to fill the window, the stopwatch cursor is a useful way of reassuring the user that `lector` is still operating.

3. Model of text.

All text formatters embody rules that distinguish between content and markup, and that specify the relationship between elements of the markup with the content and with each other. These rules constitute the formatter's model of text.

`lector` obeys a simple model of text. Tags are any strings that conform to a given tag syntax. If a string conforming to the tag syntax is defined in the style specification file, it is a defined tag; otherwise it is an undefined tag. Tags specify the formatting that `lector` applies to the content. All strings that are not tags are content. All tags, defined or undefined, are independent entities; no nestings or tag-pair relationships are maintained. Content fragments are displayed in the order that they appear in the input; content cannot be reordered. The input text need not contain line breaks.

`lector`'s model of text can be contrasted with that of other systems. Where most typesetting systems have a fixed syntax for defining markup, `lector` permits the user to choose (within limits) the syntax of the tags. The ability to define syntax introduces the notion of an undefined tag. In other systems, undefined markup is illegal; in `lector`, undefined tags can cause some typesetting actions to take place. The only difference is that it is not possible to distinguish between instances of undefined tags, and so any instance causes the same action. Most typesetting systems permit a limited amount of context maintenance; for example, the ability to return to a previous font. `lector` does not support any context maintenance. This strategy was chosen partly to ensure that `lector` would robustly handle erroneous texts, and partly because context-sensitive formatting is more complicated in an interactive environment.

In the present implementation, a tag syntax is specified by a unique pair of start and end characters for the tag. The default SGML angle-bracket tags (e.g., `<chap>`) and entity reference characters (e.g., `&umlaut.`) satisfy this definition. The set of tags need not be the same in every style, a style may contain multiple syntaxes, and the syntaxes need not be constant over styles. In one style, for example, only strings demarcated by "`&`" and "`.`" might be considered tags, while in another style only strings demarcated by angle brackets would be considered tags. By varying tag syntaxes it is possible to treat markup and content interchangeably.

4. Model of specification.

All text formatters embody rules that describe how the user specifies the layout of the text. These rules constitute the model of specification.

`lector`'s model of specification is based on the separation of the semantics of the markup from its syntax. A *style specification file* is a user-defined mapping between markup syntax and semantics. Each style specification file lists the tag syntaxes, fonts, and styles that are to be employed in a given invocation. The files were designed to be simple to construct; 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 large symbol table mapping tags to typesetting operations.

An example of a style specification file is shown in Figure 4; the BNF for the style specification is given in Figure 6 (terminal strings are shown in bold font). Style specifications can be quite lengthy if many tags and styles are defined. One such file for the *OED* consists of nine styles and is 83 kbytes in size. Each of the styles defines nearly 200 tags.

Most style specification files begin with a set of font specifications. A font specification consists of the system identifier (the X.11 logical font description), an absolute symbolic label, and a three-part relative symbolic label consisting of values for `Type`, `Family`, and `Size`. Each component of this latter identifier defines an equivalence relation on the total set of fonts, which provides a way to specify relative font changes. Most typesetting programs provide

```
BlankLines   :-  <BlankLines> integer </BlankLines>
Family       :-  <Family> string </Family>
Highlight    :-  <Highlight> boolean </Highlight>
Indent       :-  <Indent> integer </Indent>
Name         :-  <Name> string </Name>
PageBreak    :-  <PageBreak> boolean </PageBreak>
PrText       :-  <PrText> boolean </PrText>
PrTag        :-  <PrTag> boolean </PrTag>
PrUndef      :-  <PrUndef> boolean </PrUndef>
Size         :-  <Size> integer </Size>
String       :-  <String> string </String>
TabList      :-  <Tabs> {integer}* </Tabs>
Type         :-  <Type> string </Type>

Font         :-  <Font> Name   Family  Type   Size </Font>

Tag          :-  <Tag>
                   Name
                      { Family } { Type } { Size } { PrTag }
                      { PrText } { PrUndef } { Indent } { String }
                      { Highlight } { BlankLines } { PageBreak }
                 </Tag>

Style        :-  <Spec>
                   Name
                      { Family } { Type } { Size } { PrTag } { PrText }
                      { PrUndef } { Highlight } { TabList }
                      { Tag }*
                 </Spec>
```

Figure 6. Specification BNF.

something similar, but in `lector` there is the added flexibility that the values assigned to a font's `Type`, `Family`, and `Size` need not have any relationship to the actual type, family, and size of the font. Thus the partition induced by the equivalence class can be maintained while the actual fonts that are displayed are altered as desirable.

The remainder of the style specification file is a list of styles. Each style consists of a set of default typesetting values and a list of tag specifications. Tag specifications are mappings of tags to a small set of useful typesetting operations. These include the ability to change font (absolute or relative to the current font), to insert blank lines, to highlight (currently implemented as reverse video), to indent, to "page break" (i.e., stop filling the window), and to print an arbitrary (but fixed) string. These few features suffice for a large class of running texts. A very useful additional feature is text suppression, the ability to discontinue the display of text. The fields `PrTag`, `PrText`, and `PrUndef` control the state of suppression for defined tags, text, and undefined tags, respectively.

`lector`'s model of specification can be compared with that of other systems. First, most typesetting systems rely on so-called "procedural" markup, in which the semantics of the markup is bound by the system. `lector` binds the markup to semantics under user control. Second, `lector` provides a very small number of typesetting capabilities — the quality of its output is mainly due to the use of multiple proportional fonts. Other typesetting systems usually provide many more capabilities, such as keeps, headers and footers, automatic page numbering, right justification, kerning, automatic hyphenation and so on. On the other hand, `lector` provides an essential capability for interactive use, namely user-controllable suppression of text. Appropriate use of suppression is the key element in the definition of the interactive styles shown in Figure 4. Finally, unlike most other systems, where markup is suppressed by default (and cannot be displayed without escape sequences or other special measures), `lector` facilitates both the display of markup and the definition of a variety of display styles for that markup.

5. Model of interaction.

All text formatters embody rules that control input, output, and the interactive behaviour of the formatter. These rules constitute the formatter's model of interaction.

`lector`'s basic interaction is to accept multiple texts on its input stream, display them one at a time in the current window, and redraw the display whenever the user requests a change in window size, output style, or position in the text. This is the *preview* mode. `lector` also supports *index* mode; in this mode `lector` displays as in preview mode, but the user can also employ the text display as a menu. By pointing at positions in the text, the user causes `lector` to output the file offset of the indicated position. Applications that can access the file are often able to act on this offset.

Text for display is obtained in one of two ways: either from the standard input stream or from a disk file. When text is received on the input stream, `lector` allocates sufficient virtual memory to hold the entire text. Multiple texts can be sent on the stream, thus enabling re-use of the virtual memory and process invocation. In such situations, it is common for the sending program to preprocess the text before sending it to `lector`, usually inserting tags of one type or another to flag important sections of the text.

When text is obtained from a disk file, the whole text can be placed in memory, or it can be handled in 8 kbyte pages. This latter capability is a necessary tactic when very large files are to be displayed. Empirical observation shows that paged access is not noticeably slower than full allocation, presumably because full allocation itself often results in paging by the operating systems and therefore similar disk activity. Since the standard input stream is not otherwise being used when text is obtained from disk, `lector` will accept reposition requests on this stream. These requests, which consist of individual file offsets, cause the display of text to begin at the corresponding position in the text file. A common use of this capability is the construction of a `lector` pipeline; several invocations of `lector`, each in index mode and accessing the same disk file, are connected through standard input and output. Any upstream invocation can then be used as an index to the downstream invocations. Several examples of this technique are provided in the section on applications.

`lector`'s model of interaction can be compared with that of other systems. First, most previewers require the whole text to be present, or at least preprocessed into displayable format; `lector` permits formatting from arbitrary points in the text. Most previewers have an extremely limited interaction model, consistent with a narrow function of previewing text; input is generally limited to user commands to page forward or backward, and the only output is the display of the text. `lector` is a first step towards more general interaction methods, accepting input not only from the user, but also from other applications. Moreover, while other previewers treat the text as a static output object, `lector` treats text as an interactive display. Finally, where most instances of previewers are bound to a single text file, `lector` can be treated as a persistent utility that is reused for many texts. This cuts down on process invocation costs and relieves the user of the need to manage new windows.

6. Implementation.

The current implementation of `lector` consists of two main parts, a window controller and a scanner. The window controller creates and manages a single window, including menus, cursors, and other window-specific features. The scanner manages the style specification file and the text. It parses the style specification, creates a symbol table mapping tags to typesetting functions, and uses this when scanning the text. This separation of tasks simplifies the porting of `lector` to other window systems, as well as enabling the style-specific functionality to be adopted by other software packages such as editors or paper-based typesetters.

The window controller component is built around *Xlib*,[11] the C interface to the X.11 window system protocol. *Xlib* is a large library consisting of more than two hundred routines and several different data types. Since even the simplest *Xlib*-based program requires some 25 different *Xlib* calls,[22] applications programmers are normally expected to employ a toolkit built on top of *Xlib*, rather than employing *Xlib* directly. A number of factors dissuaded us from using toolkits such as the X Toolkit.[15] First, many *Xlib* calls are fairly straightforward and are executed only once; thus, many calls is not necessarily overly complicated. Second, we wanted `lector` to be as small as possible; linking a toolkit library seemed to fatten up the executable by a factor of two. Third, the limitations of existing toolkits were too severe for our intended applications, for example, not permitting enough fonts. Finally, at the time work was begun, the robustness and stability of the toolkits was more questionable than that of *Xlib* itself. Thus it seemed better to design `lector` with the interface least likely to change.

The window controller manages two basic data types: a window type, which contains information about the display, such as its width, height, foreground and background colours, title, and pointers to various *Xlib* data structures; and a text type, which contains the text, its size, the extent of text currently being displayed, the current number of columns, the margin size, the symbol table of tag-to-typesetting mappings, and so on. In addition to these base types, the window controller also manages cursors, menus, events, graphics contexts, labels, colours, and all other necessary *Xlib* data types. Each of these types requires a significant amount of initialization and error handling, most of which is relatively straightforward.†

The window controller's main task is box filling, which requires simultaneous control of both *Xlib* and the scanner. Each line, paragraph, and column in the display is treated as a box. Box filling is the slowest and most costly activity, as well as the most visible one, so it is important to avoid inefficiencies. In particular, string copying is avoided wherever possible, and the box filling routines operate in a very dirty fashion, using global variables and representation-dependent tricks.

The text stream is treated as a crude pipeline; tag-delimited fragments of this stream are marked for display, while the tags are used to set typesetting variables that control the state of the formatting. The main loop of the filling routine calls the scanner on the text buffer, looking for the next tag. If a tag is found, the scanner returns the set of typesetting operations defined for that tag, which are then used to reset the typesetting variables. If text is found, the scanner returns the text to be used in filling the current box. Line breaks are identified using the simplest possible algorithm: take the longest prefix that fits on the current line. It is not sufficient to simply count the characters, as the text is usually presented in a proportional font.

The window controller does not maintain a screen map of the displayed text, other than a list of offsets that are necessary for output in index mode. While a screen map might speed up window repair (i.e., redisplay of newly exposed regions that were occluded by other windows), two factors complicate its management: the dynamics of the window, and the use of multiple styles. Since the user can change the window size and aspect ratio, any internal representation of the pages or line breaks is liable to be invalidated at any time, and recomputation may be an expensive proposition. Secondly, when multiple styles are present, there are multiple line and

† Although chock full of complicated interactions and boundary conditions. The handling of these types constitutes the most bug-prone and least satisfying part of `lector`. One of the advantages of toolkits is that they are (presumably) more robust in this area.

page breaks for the same portion of text; the tagging itself may be defined differently in the different styles, so that not even a map of tags is necessarily fixed. Empirical evidence of patterns of interaction is needed before we consider more complex internal representations.

The scanning component of `lector` is independent of X.11. During initialization, the scanner is called to read the style specification file and produce a mapping between tags and typesetting activities. Since each tag is independent of any other, the mapping is not difficult to produce; a record is constructed for each tag, containing the values defined for that tag. The scanner constructs a list of characters that start and end tags. These lists are used when scanning the text for potential tags. Finally, the tags themselves are sorted into a list for binary searching.

The actual scanning of text is relatively simple. The scanner searches the text for characters that begin tags; usually only one or two such characters need to be checked against the input. When a potential tag start is found, the end of the tag is identified by finding the corresponding end character (if no such end character is found within a reasonable distance then the potential match is ignored). The potential tag is compared with the list of tags. If it is defined, the corresponding record of typesetting actions is returned; if not, then the typesetting values corresponding to the undefined tag are returned. The window controller uses this record to adjust the current state of the formatting variables, and displays the current text fragment.

`lector` is approximately 3500 lines of C code. The first prototype was built in about two weeks; it has been refined over a period of two years.

7. Applications.

An obvious application for `lector` is display of online documentation, since its multiple formats and index mode are quite useful in viewing descriptively marked up texts. Figure 7 shows an example of the views defined on an online user guide for a full text searching program.[9] Figure 7 shows six different views, including a ‘‘table of contents’’ view, a view showing only the glossary definitions, a view showing the computer commands embedded in the table of contents, a view showing the chapter summaries, a view of the full text as it appears on the printed page, and a view of the tagged text. Six different invocations of `lector` are used in this example, but the number of invocations, their default styles, the geometries of their windows, and their relative placement on the screen are all customizable by the user. Thus the user can choose how much screen real estate to devote to the display of the manual, according to his or her present needs. The invocations are in index mode and are pipelined together; thus, pointing to a position in one view causes the others to be updated to that position. In Figure 7, each of the views shows the text starting at the section on ‘‘Your First Search’’. Note that not all of the text is accessible in each of the views; summary and glossary information, for example, does not normally appear in any other view. Note also that each of these views is created on the fly. In particular, the document does not contain a table of contents — it is produced by rapidly scanning the whole text.

The views shown in Figure 7 have taught us interesting lessons about the virtues of text suppression. The different contexts induced by suppressing various portions of the text improve both the user’s comprehension and navigation. Suppression also serves the author, who can evaluate the content of a fragment in a variety of contexts. In the case of the documentation shown in Figure 7, the *Summary* view of an early draft of the guide exposed the variable length and erratic style of the summaries. These observations led to a rewriting of the summaries for more consistency.[10]

Most existing online documentation is not tagged with descriptive markup, or uses tags whose syntax is more complicated than the present implementation of `lector` can recognize. One such collection of text is the Unix man pages, which are tagged with *troff* macros. Such texts can often be preprocessed with *awk* or *sed* scripts to install more tractable tags. Figure 8 shows the *csh* man page as seen via `Xman`, a combination of awk preprocessor and `lector`. Three

Figure 7. `lector` views of computer documentation.

views are shown: a skeleton view, the man page formatted for reading, and a view showing the tags installed by the shell scripts. As with the user guide shown in Figure 7, these invocations are pipelined in index mode; thus, pointing to the skeleteon view results in the two other views being updated to the indicated position. This interconnection is particularly useful for long man pages, because users often access them only to check a single point of syntax (in the case of the *csh* man page, the necessary content is usually buried in the middle of the page) or to remind themselves of the location of related files (found near the end of the page).

The main difficulty in producing tagged man pages suitable for `lector`'s use is that the *man* macros, like many other typesetting macros, do not employ closing tags; thus the shell scripts used for tagging must recreate the same contextual rules obeyed by the *man* macros in order to generate closing tags. On the other hand, marking up the man page permits the reuse of styles developed 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. The separation of markup syntax from markup semantics enables style definition to be treated as an independent activity.

The largest body of text found on most computer systems is source code. Comprehension of this text is said to benefit from improved presentation.[4, 5] *vgrind*[19] and *tgrind*[12] can produce improved presentation of C using *troff* and TEX, respectively, but these previewers suffer from several limitations, including restriction to a single view, formatting that is insensitive to window geometry, and the inability to suppress text or to employ the view as a query tool. `Xgrind`, seen in Figure 9, is an application that uses `lector` to display source code. As in the case of `Xman`, the source code must be preprocessed before being sent to `lector`. Since C code is more syntactically complex than *troff* macros, a *lex* preprocessor is employed to mark up the program. In the current implementation this preprocessor marks line breaks, reserved words, braces, comments, strings, and other syntactically simple features. A more complex preprocessor could recognize and mark up more subtle semantic components.

The code displayed in Figure 9 is the main routine of the X.11 server, a routine which has a reasonably non-trivial nested structure. A traditional view of this code is presented in the *C Grind* format; this view is similar to what a programmer normally sees, except that the use of multiple fonts helps distinguish the components of the code. In addition, proportional fonts generally provide more text per unit space than constant width fonts, and hence reduce the need to break lines at inconvenient places. A compressed view of the code is seen in the *Nesting* view, which shows only the major control flow statements and their nesting (for variety, this view uses Courier font). The use of multiple fonts and point sizes is exhibited in the *Nesting (Reduced)* view, which shows the main control flow statements in a large proportional font, with all other statements in the tiny *nil* font, thus giving the programmer an idea of both the nesting and the relative sizes of the blocks. Finally, there are views displaying the comments and the string constants in the code. As with any other application of `lector`, all of these views can be used as an index to the code; the user merely needs to point to the appropriate statement to jump to that position, and can then switch to the *C Grind* format to read the code.

`lector`'s basic capabilities naturally support both user adaptible[7] and language independent[13] prettyprinting of source code. The style specification is provided by the user and so the format of the prettyprinting is easily adapted by the user. Since the style specification is based on installed markup rather than a specific programming language syntax, the same styles can be used for viewing many languages; the new language is accommodated simply by constructing the appropriate preprocessor. Unlike most other prettyprinters, `lector` has the additional ability to employ the text of the program as an index to itself, and to interact with other applications such as debuggers and compilers.

Presentation of source code highlights a benefit of `lector`'s simple model of text; the inability to deal with tag pairs or tag nesting means the resulting style specifications are actually more robust for programs that have nesting errors. Thus `lector` can handle structurally invalid or incomplete programs that cannot be handled by many prettyprinters [13] and structure-based editors.[27] Prettyprinting also exposes some of the limitations of `lector`'s model of specification. Though `lector` can easily show all instances of a given class of text fragments, there is no way

Figure 8. `lector` views of *csh* man page.

to show specific instances determined during the session, because the style specification is bound before display begins. While we could construct a style that would highlight all instances of identifiers, for example, we could not construct one that would highlight only a single identifier determined by the user during the session. More powerful models of specification and of text are needed to support this type of interaction.

Figure 9. `lector` views of C code.

`lector`'s index mode eventually led us to consider the use of `lector` as a generic menu utility. In these applications a tagged text file describing the content of the menu is constructed and sent to an invocation of `lector`. The user's interaction with the menu is then handled by `lector`; the menu is manipulated like any other text, using multiple formats, paging, or resizing. When an option is selected from the menu, `lector` outputs the corresponding offset to the applications program, which performs the associated action.

Figure 10. Cross-reference tool.

Figure 10 shows the cross-reference tool for searching the *Oxford English Dictionary*. The top left window (the only window in the figure that is not an instance of `lector`) is used for entering string searching commands. Matches that are found by the searching engine are displayed in their entirety in the bottom lefthand invocation of `lector`. As these entries are displayed, they are also filtered by a small program that extracts the cross references from the entry, constructing a text that is displayed as a menu by the middle invocation of `lector`. Selection of a cross reference from this latter window results in a new search for the target, whose entry is displayed in the final invocation of `lector` in the rightmost window. In Figure 10, the entry for *lecturer* contains four cross-references, whose labels are displayed in the cross-reference window. The user has selected the cross-reference to *lector* (note the position of the cross-hair cursor) and the target entry is displayed in the target window.

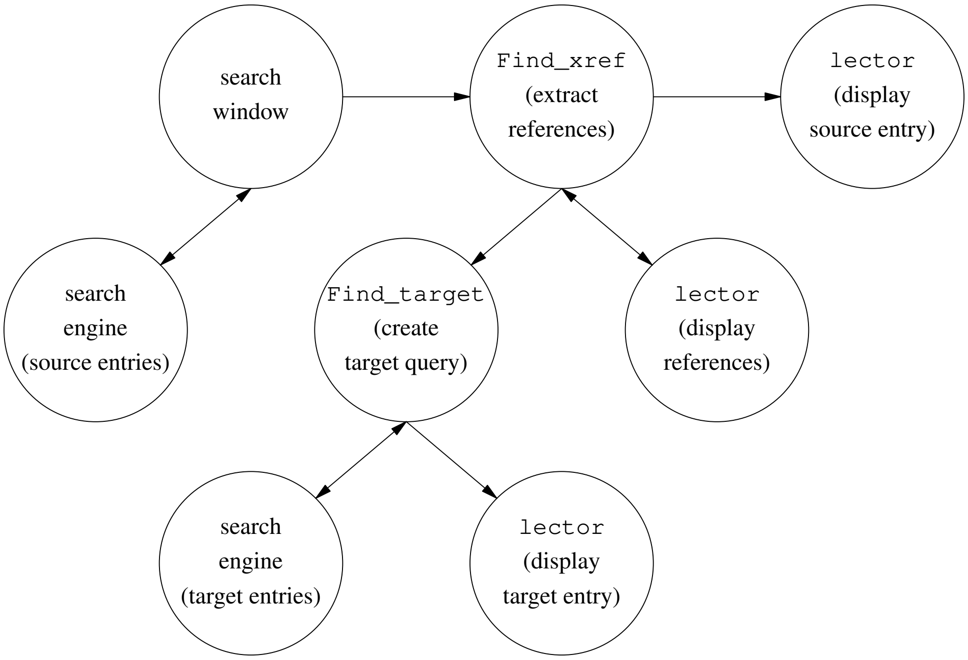

Figure 11. Process structure diagram for cross-reference tool.

Figure 11 shows the process structure diagram for the cross-reference tool. There are two invocations of the search program; one for searching cross-reference sources, and the other for searching cross-reference targets. The user interacts only with the source search engine, through the search window; the other search engine is controlled entirely by the user's selection from the cross-reference menu. Two small programs were written to provide the interaction between the search engines and the `lector` displays. `Find_xref` accepts source entries from the user's search, sends the entry to the first invocation of `lector`, extracts the cross-references, and sends them to the menu invocation of `lector`. User selections for this menu are returned to `Find_xref`, which forwards the text of the cross-reference to `Find_target`. `Find_target` uses the text of the cross-reference to develop a query for the target entry, which is searched using the second invocation of the search engine. The target entry is then sent to the third invocation of `lector`.

Process decomposition enabled rapid prototyping of this application. `Find_xref` and `Find_target` are the only parts of the application that were custom-built for the task, and can be altered without modifying either the display or search components of the tool. Another advantage of process decomposition of the application is that its various components can be easily distributed about a network in order to balance the computational load; for example, the search engines can be located on a file server, with the window processes located on users' workstations. The only complicated part of the cross-reference tool is the setup of the process structure, something not easily done with current Unix shells, which are designed for pipelines. Process setup is currently accomplished with a simple custom-built shell. The cross-reference tool is being evaluated by the Oxford University Press for use in checking the consistency of cross-references in the *Shorter Oxford English Dictionary*.

Figure 12. `lector` menus for structured editor.

Figure 12 shows `lector` used both as a menu utility and a help utility for a text editor. There are three invocations of the editor in the figure. The topmost invocation shows two windows: the editor window itself is on the bottom, and an invocation of `lector` is on the top, displaying an abbreviated set of the editor's commands. The leftmost invocation of the editor shows a similar setup, except that here the user has chosen to place the editor commands on the left side of the editor window. Finally, the three-window invocation shows a view of the menu that includes both the help text for the operations and the defined styles that can be used for viewing the text in the editor window. In addition, this invocation also shows another instance of `lector` used to display tutorial information for the command ‘‘complete tag’’.

The editor shown in Figure 12 implements many complicated features, including multiple windows, multiple formats, the ability to edit structured text, and packages of user-defined functions, in addition to the basic text editing features common to all editors. The large number of operations was originally supplied via a traditional pop-up menu interface, but this proved far too slow and clumsy. The use of `lector` to present the menu has several advantages. First, it relieves the editor application itself of the need to worry about menu interaction, thus simplifying the editor’s design. Secondly, it provides a significant amount of user configurability to the design — the user can choose the position, size, and orientation of the menu, the layout of the text, the options that are present, and even the descriptive labels assigned to the operations. Moreover, the menu can contain other embedded information; in this example, the help text associated with each menu item is included in the menu file. Simply by switching views of the menu, the user can read the explanatory text.

The main disadvantage of `lector` menus is that they require a process invocation. As a result, they are probably best as static rather than pop-up menus. On the other hand, `lector` menus exhibit several novel properties. The style of the menu can be changed without recompilation, and it can be customized by the user. The menu’s layout can be re-arranged at run time without changing the input or the style specification file, simply by changing the size and aspect ratio of the window, or by changing the number of columns. `lector`’s built-in capabilities for scrolling, paging, and multiple columns enable it to easily accommodate very large menus. Finally, `lector`’s ability to suppress text means that the choices presented can be altered at run time by changing styles.

As a final example, Figure 13 shows the chess game played between the programs Phoenix and Deep Thought at the twentieth annual ACM North American Computer Chess Championship. Each view in the figure is an invocation of `lector`; the views include the board, the notation, the source text, the annotations, and a view showing only the knights and bishops. A simple program was written to convert standard chess notation to a tagged text description of the board at each state of the game; the result can be seen in the text view in Figure 13. The board views result from mapping the tags describing the board to the appropriate characters of a chess font. The file contains the log of the entire game; paging forward in a board view takes one to the next state of the board, and paging forward in the notation view takes one to the next set of moves. The text views and the board views are interconnected so that one can point to a move or annotation and update the board views to the appropriate point in the game.

The chess application is designed for use with a game log but could easily mediate a game in progress with the addition of simple filters to convert from move notation to board description and from the user’s pointing action to move notation. Thus `lector` can easily assume the bulk of the work in providing a direct manipulation interface for chess or similar games. It is interesting to note that the text describing the board state is reminiscent of a structured statement in a communications protocol; here `lector` fulfills the role of statement interpreter, rather than document previewer.

Chess applications usually supply additional feedback about the game state, such as flagging invalid moves or presenting a list of the captured pieces. This feedback can be provided in the `lector` application simply by extending the filters to embed appropriately marked-up information in the text, and then devising the appropriate style to display the feedback either textually or graphically; for example, one could present the icons representing the captured pieces, or

Figure 13. `lector` chess application.

present a bar chart showing the values of the pieces captured.

8. Discussion.

`lector` raises two classes of questions: those about its philosophy of text display, and those about its status as a particular implementation of that philosophy.

As a philosophy, `lector` is characterized by the use of lexically unique tags, the indirect mapping of tags to semantics, the provision of multiple views, and the process-level independence of text display from text manipulation. Independence implies both the independence of the application from a specific output device, and the independence of the text from a specific layout.

Despite its increased emphasis on interaction with text, a basic lack in the current philosophy is the limited model of input and output. The display of man pages and help text is not highly interactive, but other text-rich applications such as mail and electronic billboards require significant amounts of input. In such situations the separation between application and text display becomes more clumsy, especially when the separation is achieved via process decomposition. Yet the ability to provide multiple views of news and mail, and to use these views as indexes into a database, seems worthwhile. 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.

`lector`'s philosophy can be compared with the current object-oriented approach to interface design, as exemplified by NeXTStep. In these systems most instances of text are rendered by a text object that provides consistent formatting and input/output functions. The object-oriented approach thus addresses the problem of defining a useful and efficient separation between application and text display. However, current object-oriented systems tend to rely on fixed markup schemes; NeXTStep text, for example, is coded using Microsoft's Rich Text Format. Reliance on a particular markup scheme hinders the extensibility of the approach to other texts. More importantly, the object hierarchy in such systems erects an artificial separation between text display and other kinds of display; texts are not interchangeable with menus, control panels, and other interface objects. `lector`, on the other hand, encourages the treatment of menus and iconic interfaces as a limited form of text display, and texts as menus that may or may not have semantics associated with selection of individual fragments.

It is interesting to consider extending the generalization that the `lector` philosophy encourages. The chess application, for example, raises the possibility that `lector` could be useful for displaying a wide range of "texts" whose output is largely non-textual. Icon-based interfaces might be usefully generalized if the buttons, sliders, boxes, and other interface gadgets were specified in a marked-up text which is then mapped to appropriate fonts. `lector`'s interaction and specification models are currently too primitive to support such iconic interfaces; in particular, the user cannot easily specify how the controls should react (i.e., change their display) when selected. Nevertheless, it is interesting to note that the "text" file for such an application is actually a display-independent specification of an iconic interface, and the corresponding style specification is an application-independent specification of a class of similar interfaces layouts. `lector` also suggests a number of features not present in current direct manipulation interfaces, including display of very large control panels, suppression of classes of controls, user-specified layouts, and (by means of font substitution) user-specified icons. These possibilities are straightforward extensions of `lector`'s basic display capabilities, and follow naturally from the `lector` philosophy. Similar types of abstractions have only recently appeared in traditional generators for icon-based systems.[28, 29]

As an implementation, `lector` tries to occupy the centre of a set of constantly fluctuating tradeoffs involving the relative efficiency of window and operating systems, the perceived needs of applications, the direction for future enhancements, and some debatable decisions about software reuse. Much of `lector`'s design emerged through juggling these tradeoffs.

A good example of this balancing act is the issue of level of decomposition. We had the choice of providing `lector`'s capabilities either as a set of library calls, or via interprocess communication. Compiled-in capabilities would have improved efficiency by reducing the need to transmit text between processes, and by eliminating process startup cost. On the other hand, multiple processes enhance the distributability of the system, and are easier to test and replace with other components, since the process environment (i.e., `sh`, `csh`, and their ilk) is more interactive than the library environment. Multiple processes also become cheaper and more suitable with larger memory and multiple processors, and simplify the handling of multiple invocations. Process level communication is portable across window and system boundaries and tends to encourage programmers to stick to defined interfaces. Finally, casual observation has shown that text transmission is not a bottleneck for most applications, though process creation remains a problem.

The process of enhancing and extending `lector` continues to revolve around the balancing of various tradeoffs. Adding the capability for right justification of text, for example, seems a reasonable extension that would provide a neater appearance to some output. Its contribution to readability is questionable, however,[23] 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 would be the ability to display tables. This involves an increased user cost in specification as well as the additional problem that tables are not robust to all style changes. [6] Each such enhancement must also be designed while keeping in mind `lector`'s ability to extend into display tasks not normally encountered by text formatters. If tables are permitted, for example, how should the style specification take into account dynamically resized windows? On the other hand, tables might be an ideal way to constrain the layout of iconic displays.

The part of `lector` most needing redesign is the style specification language. This language has evolved in a largely ad hoc fashion, and is best suited to handling simple running text. Despite our attempts to keep the language simple, the construction of styles typically involves a non-trivial development effort. Users should not have to expend this effort without some reassurance that the style specification is, in some formal sense, complete and robust. Better style specification does does not simply mean the adoption of a standard display machine,[17] or the provision of a full programming language. Rather, it is time for a re-evaluation of the capabilities of formatters in general, in the light of the extended applicability that `lector` suggests. A new type of abstract display machine, supplemented with novel ideas about specification of output styles, would be an important step in the redefinition of the style specification language.

9. Conclusions.

`lector` was created to fulfill a specific need for interactive display of descriptively marked-up texts. As we developed an appreciation of the generality of text display and text interaction, `lector` matured into a powerful text tool useful in a variety of applications. `lector` helps vindicate the claim that descriptive markup is a reasonable method for structuring text, and extends this claim into the realm of menus, interactive displays, database browsers, flexible iconic interfaces, and code prettyprinters, demonstrating that fluid text structuring tools improve many types of application. Finally, `lector` shows that solutions to the general problem of text output require not only that applications be device-independent, but that they be independent of style and markup as well.

10. Acknowledgements.

`lector`'s design and implementation were greatly influenced by the comments and suggestions of all the members of the Centre for the New Oxford English Dictionary. My particular thanks to Heather Fawcett, who suggested the interconnection of multiple `lector`s, and to Gord Broom, whose built `Xman`, `Xgrind`, and several other derivatives of `lector`. The specification file component of `lector` was designed jointly with Tim Bray and Gord Broom. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada through the University-Industry Program under grant 0039063. My thanks also to the Oxford University

Press for the opportunity to develop the cross-reference tool during a visit to Oxford, and for their kind permission to reprint a portion of the *Oxford English Dictionary*.

References

1. "psview(1)," *Unix User's Manual* (December 2, 1987).
2. Adobe Systems Inc., *PostScript Language Reference Manual,* Addison-Wesley, Reading, Massachusetts (1985).
3. Kenneth C.R.C. Arnold, *Screen Updating and Cursor Optimization: A Library Package,* University of California, Berkeley, California (April 23, 1986).
4. 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).
5. Ronald M. Baecker and Aaron Marcus, *Human Factors and Typography for More Readable Programs,* Addison-Wesley, Reading, Massachusetts (1990).
6. Richard J. Beach, "Setting Tables and Illustrations with Style," CS-85-45, Department of Computer Science, University of Waterloo, Waterloo, Ontario (May 1985).
7. G. Blaschek and J. Sametinger, "User-Adaptable Prettyprinting," *Software — Practice and Experience,* 19, 7, pp. 687-702 (July 1989).
8. Eric Cooper, Robert W. Scheifler, and Mark Eichin, "xdvi(1)," *Unix User's Manual* (May 8, 1989).
9. Heather J. Fawcett, *PAT: The User's Guide,* Centre for the New Oxford English Dictionary, University of Waterloo, Waterloo, Ontario (1989).
10. Heather J. Fawcett, "Adopting SGML: The Implications for Writers," OED-89-03, Centre for the New Oxford English Dictionary, University of Waterloo, Waterloo, Ontario (July, 1989).
11. Jim Gettys, Ron Newman, and Robert W. Scheifler, *Xlib — C Language X Interface, X Version 11 Release 2,* Massachusetts Institute of Technology, Cambridge, Massachusetts (1988).
12. Van Jacobson, "tgrind(1)," *Unix User's Manual* (June 1989).
13. Matti O. Jokinen, "A Language-Independent Prettyprinter," *Software — Practice and Experience,* 19, 9, pp. 839-856 (September 1989).
14. Donald E. Knuth, *The TeXbook,* Addison-Wesley, Reading, Massachusetts (1984).
15. Joel McCormack, Paul Asente, and Ralph R. Swick, *X Toolkit Intrinsics — C Language Interface X Version 11, Release 4,* Massachusetts Institute of Technology, Cambridge, Massachusetts (December, 1989).
16. Microsoft Corporation, *Microsoft Rich Text Format Specification,* Redmond, Washington (1989).
17. International Standards Organization, "Information Processing — Text and Office Systems — Document Style Semantics and Specification Language (DSSSL)," JTC1.18.15.06.01 (July 1988).
18. Joseph F. Ossanna, *Nroff/Troff User's Manual,* Bell Laboratories, Murray Hill, New Jersey (October 11, 1976).
19. Dave Presotto and William Joy, "vgrind(1)," *Unix User's Manual* (August 3, 1983).
20. Darrell R. Raymond and Frank Wm. Tompa, "Hypertext and the Oxford English Dictionary," *Communications of the ACM,* 31, 7, pp. 871-879 (July 1988).
21. Darrell R. Raymond and Heather J. Fawcett, "Playing Detective with Full Text Searching Software," *SIGDOC '89,* Pittsburgh, Pennsylvania (November 8-10, 1989).

22. David S. H. Rosenthal, *A Simple X.11 Client Program or How Hard Can it Really Be to Write "Hello, World"?,* Sun Microsystems, Inc. (1987).

23. ISO, Richard Rubinstein, "Information processing — text and office systems — Standard Generalized Markup Language (SGML)," ISO 8879-1986, International Organization for Standardization, Reading, Massachusetts (1986).

24. Robert W. Scheifler and James Gettys, "The X Window System," *ACM Transactions on Graphics,* 5, pp. 79-109 (April, 1986).

25. J.A. Simpson and E.S.C. Weiner, *The Oxford English Dictionary, Second Edition,* Oxford University Press, Oxford, United Kingdom (March 1989).

26. Sun Microsystems Inc., *NeWS 1.1 Manual* (January 15, 1988).

27. Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM,* 24, 9, pp. 563-573 (September 1981).

28. Roger Took, "Surface Interaction: A Paradigm and Model for Separating Application and Interface," *Proceedings of the CHI '90 Conference on Human Factors in Computing Systems,* pp. 35-42, Seattle, Washington (April 1-5, 1990).

29. Brad Vander Zanden and Brad A. Myers, "Automatic Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," *Proceedings of the 1990 SIGCHI Conference on Human Factors in Computing Systems,* pp. 27-34, Seattle, Washington (April 1-5, 1990).