33

A Two-view Document Editor with User-definable Document Structure

by Kenneth P. Brooks

November 1, 1988

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

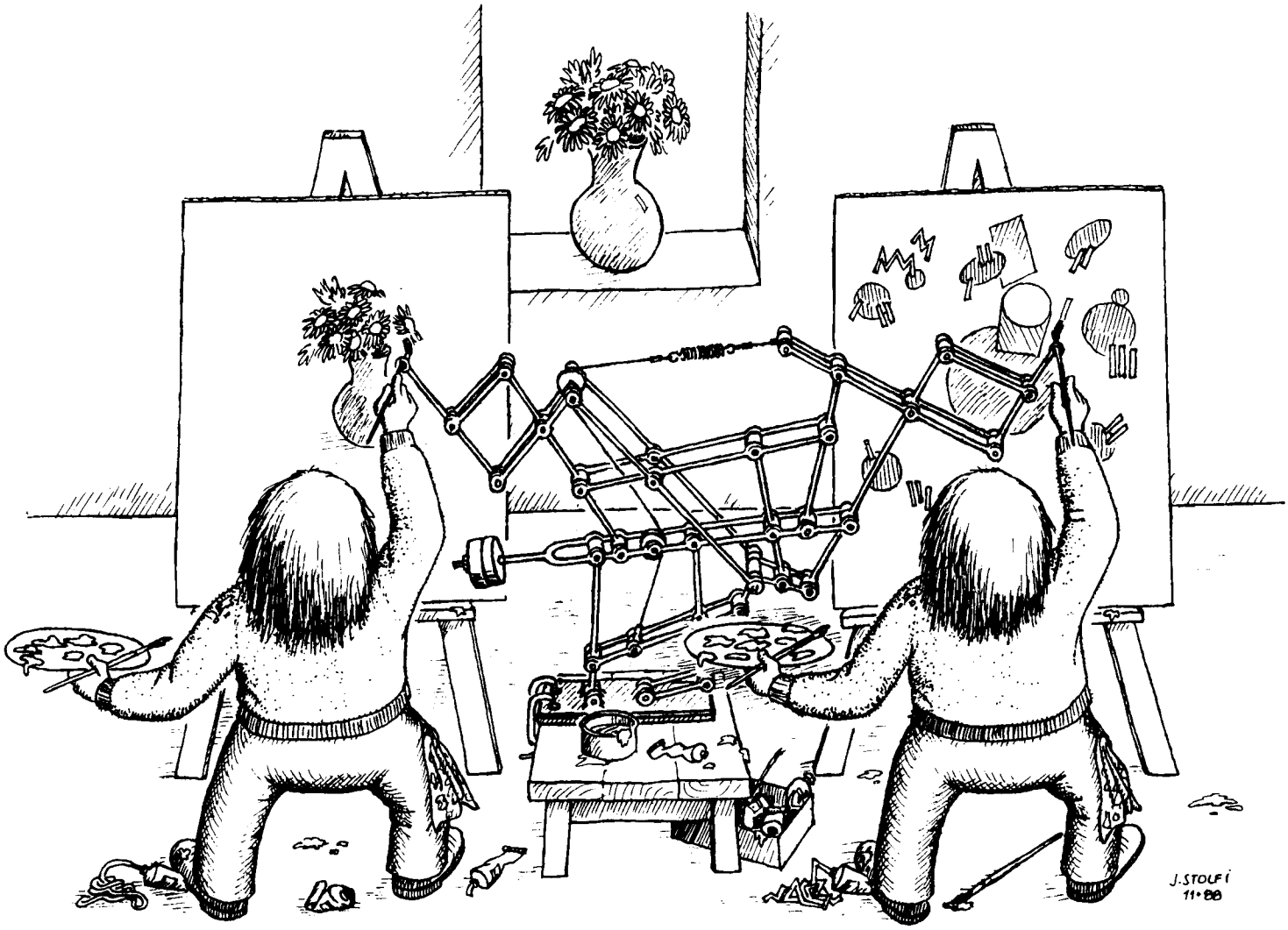
DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

A Two-view Document Editor with User-definable Document Structure

Kenneth P. Brooks

November 1, 1988



Publication History

This report reproduces, with minor changes, a dissertation submitted in May 1988 to the Department of Computer Science of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Author's abstract

Lilac is an experimental document preparation system which combines the best features of batch-style document formatters and WYSIWYG editors. To do this, it offers the user two views of the document: a WYSIWYG view and a formatter-like source view. Changes in either view are rapidly propagated to the other. This report describes both the user interface design and the implementation mechanisms used to build Lilac.

Kenneth P. Brooks

Capsule review

In this report the author describes Lilac, a working interactive system for typesetting complex documents. The novelty of the system lies in that it allows both "changes in the small" and "changes in the large" to be performed efficiently. More specifically, the user works with two views of the document. One view is a WYSIWYG view, emulating what will appear on the printed page. The other view displays the structure of the document as a tree of nested procedure calls—with the actual text appearing as arguments to those procedures. An "interpretation" of that tree yields the WYSIWYG view. The user is allowed to modify either the structure view or the WYSIWYG view. The former is preferable for large structure changes to the document; the latter for ordinary small edits.

It is the job of Lilac to update one view when the other changes, and to do so quickly. Accomplishing this has required careful design and analysis on many fronts: the language in which the document is specified, the data-structures for efficient incremental reevaluation of the document tree, the algorithms for performing selection in the document hierarchy, and finally the caching scheme used to save still-useful parts of the WYSIWYG view. All these pieces come together in Lilac, in a nicely integrated design.

This report is the author's Ph.D. dissertation from Stanford University. It is a tribute to the degree of completion of the Lilac system that this dissertation was itself typeset using Lilac.

Leonidas Guibas

Acknowledgements

I would like to thank my advisor, Leo Guibas, for suggesting the problem and for continually calling my attention to new ways of looking at it. Thanks are also due to my other committe members, Greg Nelson and Mark Linton. Greg, in particular, gave me much help in clarifying my thoughts and my algorithms, and gradually pounded into my head the concept of mathematical rigor.

I would also like to thank my father, Fred Brooks, for many hours spent proof-reading this thesis and many insights given along the way. He is particularly to be thanked for finding Chapter 3 scattered among the rest of the thesis and pointing out its existence.

I also want to thank Digital Equipment Corporation, and in particular my boss, Bob Taylor, for their generous support of my studies and provision of space and machinery for this research. Several colleagues at the DEC Systems Research Center have given helpful suggestions and encouragement along the way. Thanks to John DeTreville for developing the text display software that underlies my source view editor, and for modifying it to fit my need. And special thanks to Marc Brown for much big-brotherly advice and concern during the thesis-writing process.

Finally, thanks are definitely due to God, without whose particular providential encouragement I would surely have tossed in the towel at least once along the way.

Table of Contents

List of Illustrations

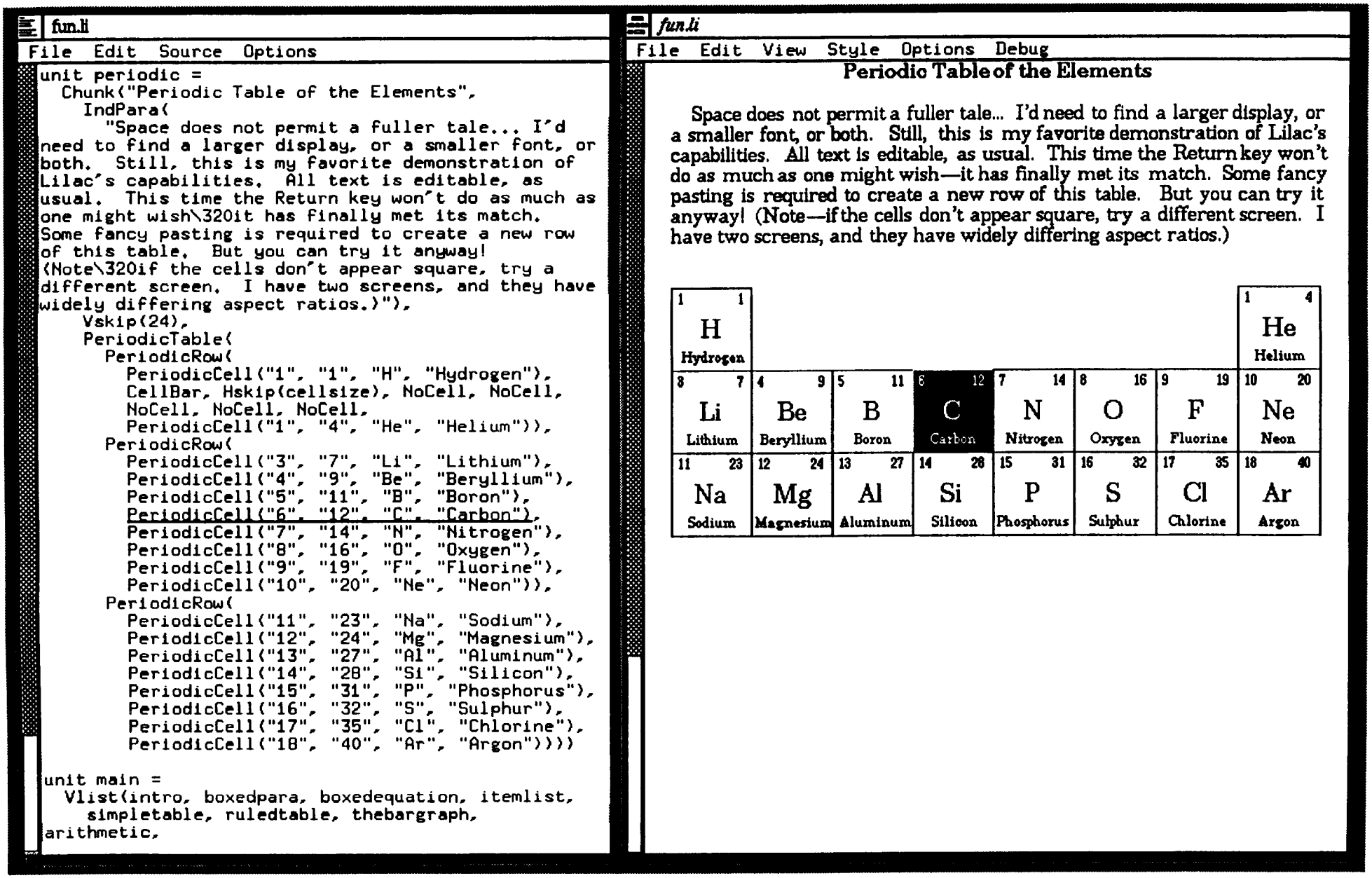


Fig. 1-1 Lilac in operation

1. Introduction

1.1. A painful choice

Electronic document preparation systems have been around for a long time. Their goal is to use the power of the computer to help authors produce high-quality documents, including such features as justification, multiple fonts, footnotes, mathematical equations, and tables. The earlier systems have a batch-processing flavor: the user prepares a manuscript in a formatter source language, compiles it into a printable result using the formatter, and then sends the result to the printer. Most recent systems use a more interactive style: the user edits, directly on the screen, a view of the document that looks approximately like the printed output. This is often referred to as WYSIWYG editing—"What you see is what you get."

A user of a batch style system has to suffer with the edit-compile-print cycle. That is, it is easy for a user to make mistakes when specifying a complicated document. He will not discover his mistakes until he goes through the process of compiling and printing the document. In practice, this cycle is often repeated many times in the course of producing one document, and it adds a significant burden to the author's job.

A user of an interactive system has to live with a restricted set of facilities. Where the best batch-style formatters embody special-purpose programming languages that allow the construction of new styles from low-level formatting primitives, today's interactive systems have far less expressive power; they offer only a fixed set of parameters with which to express style. Nothing akin to programmability is available. As soon as the user wants to format something in a way that cannot be described by the parameters provided, he is out of luck.

Thus an author who wants to produce a typeset document with today's systems faces an unfortunate choice: enjoy great expressive power but suffer with slow feedback, or enjoy instant feedback but suffer with limited expressiveness. This thesis describes a system that combines the virtues of both approaches.

1.2. The importance of programmability

Everyone who has ever grown frustrated with a text formatter knows what he really wants: an interactive document editor that is as powerful as a formatter. But to build such an editor we must deal with the issue of programmability. The most powerful formatters, such as TEX [Knuth 84] and Troff [Ossanna 76], gain much of their power from the fact that the document is in some sense expressed as a program. They have

elaborate macro systems that include aspects of programming such as conditional expressions, counting, and repetition. These facilities allow the user to write procedures to perform various common or job-specific formatting tasks, and they provide a very fine degree of control over the appearance of the text. In a sense, they allow the user to attach some intelligence to the text: not only does it have a desirable appearance (which can always be achieved by trial and error), but it will continue to have a desirable appearance even when changed.

There is another aspect to programmability: it can enable the user to express his document in terms of its *logical structure*, rather than simply as text with embedded commands that directly control the appearance. The document is described in terms of chapters, sections, etc.; in the definitions of those macros, the styling associated with them is specified. For example, the fact that section titles are set in a special font and surrounded by a certain amount of space above and below would be specified there. This ability to *bind style to structure* is an important concept. It means that a user can concentrate on the intrinsics of his document, its structure and its text, during the initial writing, and save the details of styling for later. When he is ready to work on styling, he can easily change styling decisions and have them take effect globally, just by changing the definitions of the appropriate macros.

This emphasis on logical structure occurs in varying degrees in the various systems in use. Older formatters tend to have little or none of it. Scribe [Reid 80] is entirely build around this concept, though it offers no real programmability to the user. TEX, as originally conceived, put little emphasis upon structure. But interestingly enough, Leslie Lamport found it both feasible and worthwhile to embellish TEX with LATEX [Lamport 86], a collection of macros that effectively transform TEX into a structure-oriented formatter. LATEX has gained considerable popularity, indicating a widespread desire among users to work in a structure-oriented fashion. Some recent WYSIWYG editors, such as FrameMaker [Frame 87], provide facilities to bind style to structure at the paragraph level, but there is no support for structure above the paragraph level.

I believe there is no substitute for programmability. In no other way can an author fully express his intended styling, especially in unusual cases. If the author cannot write conditional expressions, then the system designer must anticipate all the conditions for which conditional behavior will be important, and implement them. But even more important, programmability gives an author the ability to *create new structuring constructs* and to bind styling to them. If the user cannot design his own constructs, then the system designer must have the foresight to include all the constructs that

anyone will ever need. The list goes on and on: chapter, section, itemized list, definition, equation, tables of various sorts, etc. I believe that it is truly infinite. No system designer can ever design enough of them to satisfy every user's need.

This, then, is my thesis:

1. A truly satisfactory document preparation system must be interactive, and it must be based on a programmable formatting language.
2. Such a system, in order to work most effectively, requires a new document description language and a new editor design.
3. Such a system can be implemented with satisfactory performance with today's workstation technology.

1.3. The right kind of editor

What properties should such an editor have? Certainly it should support WYSIWYG editing, to avoid the tedious edit-compile-print cycle. It must also provide access to the underlying programmatic representation. Programmatic representations contain invisible information as well as visible: structure, constraints, conditional behavior. Such information cannot be seen on the printed page, nor can it easily be grabbed with a mouse and dragged into a desired state. A textual representation of the program is needed, that is to say, a document programming language.

Thus we have two views of a document: the view of the document as a program (henceforth called the *source view*), and the view of the document as it will be printed (the *page view*). To allow the user to work with both views conveniently and to see the relationship between them easily, they should be available at the same time, side by side on the screen. Stephen Trimberger, who built a two-view editor for VLSI layouts, observed that if selections made in one view are simultaneously highlighted in the other view, it dramatically aids the ability of users to see the relationship between the two [Trimberger 81]. So a good document editor should not only provide both views side by side, but should keep them instantaneously consistent and echo actions upon one view in the other view, as far as possible.

The WYSIWYG editor needs to make sense in relation to the syntax of the source view. If the user makes a selection from the middle of a section title to the middle of the first paragraph in the section and then says "delete it," what would this mean? It can hardly mean to delete all the source characters in that range—that would wreak havoc upon the structure of the document, and quite possibly would produce a

syntactically invalid program. This example shows the issue: the commands of the editor need to operate in a way that respects the syntax of the source. The source language, in turn, must be organized in such a way that a reasonable and user-friendly editor can be built within this constraint. I conclude that the best way to achieve this is to make the source syntax closely follow the logical structure of the document, and to build a structure-oriented WYSIWYG editor. Thus we have a three-way relationship: the *semantic structure* underlying the editor is specified by the *syntactic structure* of the source program, which in turn expresses the *logical structure* of the document as the author perceives it.

In addition, I believe, the editor itself should be programmable. As the author adds new constructs to his language by defining macros or procedures, so he should be able to add new commands to his user interface for creating and manipulating those constructs. This way, having gone to the source view to create a new facility, he need not go there again in order to use it. When such an editor is available to a large user community, a very convenient and powerful user interface will develop as a result of communal programming. The experience of many user communities with Emacs [Stallman 81, Gosling 82] has shown that when a programmable editor is widely available, many powerful facilities will be built upon it, beyond the wildest dreams of its original implementors. I believe that the combination of document programmability and editor programmability, together with a large and creative user community, is a recipe for producing the best WYSIWYG document editor the world has ever seen.

1.4. The research issues

The first point of my thesis could only be proved or disproved by years of experience by many people. This dissertation reports on a small (and very positive) body of experience, but its central contribution is to address the second and third points: to present an architecture and an implementation for a viable two-view document editor. In the remainder of this dissertation I discuss the technical problems that arise in building this sort of editor and my solutions to them.

Why should it be a nontrivial problem to build a two-view editor? The world knows how to build fancy document editors. It certainly knows how to build good formatters. So what's new?

What's new is that we are building an editor in a much more complex space than most document editors work in. We have given away a lot of the ability to choose a neat, optimized internal representation of the document. This editor deals with

documents that are represented as programs. The user is free to define new structural elements—which then become part of the structure that the editor must deal with. Programmability gives the user great power to specify how the document's appearance will respond to change—and the editor needs to update that appearance as specified, in response to every keystroke.

Then there is the issue of the connection between the two representations. Fundamentally, what we are doing is *editing a program by editing its output.* Thus, given a mouse click in the page image, we must find a piece of the program that is responsible for that part of the output. Given a newly typed character, we must insert it at a suitable place in the program, and also update the page image in such a way as to guarantee that a fresh execution of the modified program would in fact produce the modified output. And we must do it in 1/10 of a second, to provide adequate response for fast typists.

The problem breaks down into three major tasks:

- Language design. We need a language that can serve as the semantic basis for a WYSIWYG editor. The need for good performance in updating the page image after each keystroke also adds several constraints.
- Editor design. We need a structure-based editor that can effectively handle a user-defined structure—made of an unpredictable set of elements, and nested to an unpredictable depth. It must do everything the user needs, and for everything it lets the user do, there must be a well-defined transformation on the source program that accomplishes that result.
- Implementation. There are two main challenges here. First, we must maintain a two-way mapping between the views, so that a selection can be highlighted correctly in both. Second, we must update the screen very rapidly in response to most keystrokes, despite the fact that the screen appearance is governed by a user-defined program.

Below I discuss these problems in more detail. Strategies for their solution are outlined in Chapter 2.

1. Language design

As already discussed, the operation of the page view editor is inescapably tied to the syntactic structure of the source program. I believe that the best approach here is not to fight or hide this fact, but to use it, by designing a language in which the syntactic structure closely parallels the logical structure of the document.

The language is also constrained by the need to *assign responsibility* for each piece of the output. When the user points to a character or object to select it, we need to identify some expression in the source which is considered responsible for its existence. If the user asks to delete the selection, we would delete this expression. In a general-purpose programming language, this would in some cases be impossible. The character in question might have been computed as a result of a long and complex loop. The language needs to be constrained enough to make this problem manageable, while still being powerful enough to accomplish what authors actually need.

A third problem involves incremental interpretation. If we are to produce a correct image after each user command, and if the language has any real programming power at all, we will have to reinterpret part of the program near the change in order to determine the correct appearance. And if we are to do this 10 times per second, we will need to carefully limit the part of the program being reinterpreted. This is much easier to do if the language is strictly functional, excluding side effects.

2. Editor design

A program is by nature a hierarchy: expressions containing subexpressions, function calls with other function calls as their arguments, etc. We have chosen to make this hierarchy match the natural hierarchy inherent in a document. The challenge here is that the structure of a program is defined by the programmer. He is free to choose the elements of which it is made (subroutines) and the depth of the hierarchy. This, of course, is exactly what we wanted—user-definable document structure. But now we need to design a WYSIWYG editor that can deal with it.

Most WYSIWYG editors, even those that embody some concept of structure, have a limited set of structural elements: chapter, section, paragraph, and perhaps a few more. They tend to have commands specific to each: New Chapter, Set Chapter Style, etc. Here we cannot do that; our structural elements form an unlimited set. In designing a command set we cannot predict or provide for them all. So what we need is a *generic hierarchical editor*: one that can operate on structural elements at any level, in a hierarchy of arbitrary depth, in a consistent and comprehensible fashion.

One issue here is the unpredictable depth of the hierarchy. Whether the document is a simple paper with no more structure than sections and paragraphs, or an outline with headings and subheadings five levels deep, we need to give the user access to objects at all levels. In addition there is the problem of creating new objects. When objects are user-defined, we cannot predefine a neat set of control-key bindings to produce a new object of each type. And yet, for the common objects in his document,

the user will be very dissatisfied if he continually needs to go to the source view to create a new one. We need a generic way to make this process quick and easy.

3. Implementation

We need a two-way mapping facility between the page image and the source. This is needed for the process of making and highlighting selections. Given a mouse click at any point in the page view, we need to discover the character or expression in the source that is responsible for it; this will be the selection. Then, given a selected expression, we need to determine the area in the page image that corresponds to it. This may be much larger than the object most immediately pointed to by the mouse. In addition, there is potentially a one-to-many mapping from source to page image: unless the language is carefully constrained to prevent it, a source expression may be executed more than once, producing two or more disjoint pieces of output in the page image.

The hardest problem in implementing a two-view editor is to keep the page image up to date as the user edits. Output-based editing means keeping an accurate correspondence between a changing program and a changing image. For sufficiently complicated programs, there is no way to "outguess" the program; to determine its new output we must re-execute it. But full re-execution of a large document is prohibitively slow. We need some way to re-execute only the part that changed, and to recognize safe limits for that part. In other words, we need an incremental interpreter. In practice, we need incremental operation at several levels in order to achieve acceptable performance.

Having done the partial re-execution, we need to be able to update the screen appropriately and efficiently. This is not nearly as much of a performance problem as the re-execution of the document program. But some care is required to avoid unnecessary updating of unchanged portions of the screen, because it produces a very annoying flashing effect.

1.5. Solutions demonstrated

I have demonstrated my solutions to these problems by building a two-view document editor named Lilac. (In accordance with a tradition begun at Xerox PARC, the program is named after a beautiful feature of nature. The name is not an acronym, and it bears no relationship to the program's function.)

Lilac is a working prototype; I have used it to compose and typeset this dissertation and to prepare transparencies for several talks. In this dissertation it is responsible for all the text and several of the figures.

Lilac has been implemented on an experimental system at Digital Equipment Corporation's Systems Research Center in Palo Alto. The hardware is an experimental computer known as a Firefly [Thacker 87]; it is a multiprocessor consisting of five MicroVax II processors. Each of these processors has a speed of roughly 2/3 MIPS. Lilac does not use the multiprocessing capabilities except that it runs in parallel with the window system's inner workings. The operating system is Topaz [McJones 87], an experimental system designed for efficient multi-threaded programming. The window system is Trestle, an object-oriented window system which is particularly optimized to make use of multiprocessing. Lilac is implemented in Modula-2+ [Rovner 85], a variant of Modula-2 that includes automatic garbage collection, exception handling, and a few aids to parallel programming.

The next chapter gives an introduction to Lilac and an overview of the strategies used to solve the problems discussed above. The overview includes a road map to the subsequent chapters, where the problems and solutions are presented in full detail.

2. Overview

1. The overall structure

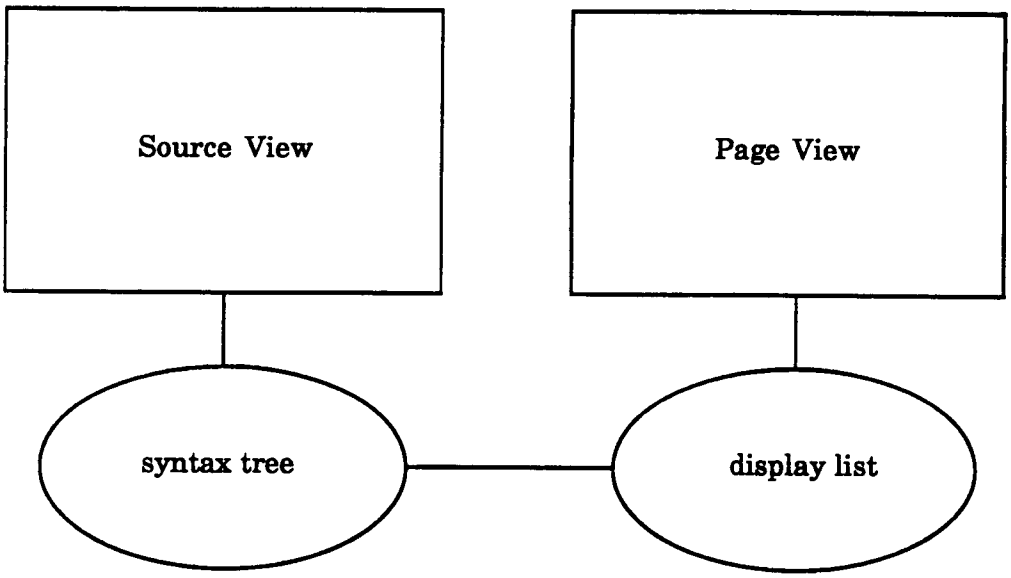


Fig. 2-1 The structure of Lilac

How does one go about building a real, usable two-view document editor? Figure 2-1 shows the overall structure of my solution. The first fundamental principle is: use a hierarchical data structure to support each view. Supporting the source view we keep a syntax tree—a parse tree of the document program. It can be derived from the text of the program by parsing, and the text can be rederived from it by unparsing. Keeping a tree rather than the text is advantageous because it embodies the structure and hierarchy of the program. It is easier to execute and easier to modify in meaningful ways. The syntax tree is the repository of ultimate truth; all other data structures can be derived from it.

Supporting the page view we keep a display list. This is also a hierarchical tree, but it is based on geometry rather than on program structure. The display list is the output of the interpretation of the syntax tree, and it serves as a map to the geometry of the page view image. It functions as an intermediary between the syntax tree and the page view, and plays a crucial role in pointing, highlighting, and updating the image. The

page view can be derived from the display list by a simple rendering process, which takes about 1/2 second for a full page.

The display list is based on "boxes and glue," a concept borrowed from TEX. A box is a rectangle which specifies some visible material ("ink") to appear within it. The page is a box; each line within the page is a box; each word within a line is a box. Boxes are usually joined to their neighbors by glue, an invisible substance that specifies spacing and the stretchability of that spacing. Boxes and glue are linked together along the horizontal or vertical axes to form horizontal lists or vertical lists, which in turn can be boxed up to produce composite boxes. The page is composite, made of lines and leading; each line is composite, made of words and spaces. Thus the display list is a hierarchy of boxes containing other boxes, a hierarchy that reflects the natural geometry of the page image.

2.2. Language design

The Lilac language is designed around three fundamental goals:

1. To provide a sound semantic basis for a good WYSIWYG document editor with a good user interface.
2. To facilitate the implementation of a two-view editor with good real-time performance.
3. To facilitate source view editing by being clean, terse, and readable.

I take it as a basic assumption that a document editor should work along structural lines. It should be easy for the user to take a paragraph, section, or chapter and operate on it as a single unit. On the other hand, the editor's commands will need to fit themselves within the structure of the programmatic representation of the document. They might hide it to some extent, but they cannot avoid it; every command must leave the document program in a syntactically valid state. So we design the language to mirror the natural structure of the document as much as possible. *Expressions are structures*: syntactic structure serves as a tool to express semantic structure. This goal can certainly be defeated by a thoughtless user or style designer, but the tools are present to make it easy to achieve.

A good user interface should be structure-based, and it should be based on the same structures that the user has in his mind. This is one of the main reasons for representing documents as programs: to allow the user to define new structural element types and then instantiate them. Thus a document description has two parts: definitions of structural element types, and the description of the actual document, using those types.

The Lilac language embodies this distinction in its syntax: *functions* define structural element types and describe their appearance; *units* contain the actual structure and content of the document, described in terms of functions.

Document structures tend to have a distinction between fixed and repeatable elements. A chapter has one title, but many sections; a section has one title, but many paragraphs. A paragraph contains any amount of material, undistinguished. A fraction contains exactly two elements, a numerator and a denominator. This distinction is reflected in the language. Most function parameters are ordinary, fixed parameters, but the last parameter of a function may be defined to be repeatable. A repeatable parameter can match any number of actual arguments in the call, including zero. The set of actual arguments matching one repeatable parameter in one function call is referred to as a *structural list*; this concept is central to the organization of the editor.

The second of our design goals is to facilitate performance—a problem that needs all the help it can get. Quickly producing the updated output from a program after a small change to the program is no easy task. But fortunately, the programs needed to express documents are not very complicated as programs go. In particular, there is a strong locality of effects: a change on page 1 is unlikely to affect anything on page 27—except the page breaks, a problem which can be isolated. Indeed, a change in paragraph 1 is quite unlikely to affect anything in paragraph 2. This locality means that it is possible to build an incremental interpreter for document programs—one which gains speed by relying on the fact that changes in paragraph 1 will not change paragraph 2. Possible, that is, with a bit of help from the language.

So we arrive at one of our vital strategies: use a purely functional language, in which expressions have no side effects. Now we have changed our observation into a *guarantee* that changes in paragraph 1 will not alter paragraph 2. On this basis we can build a very efficient incremental interpreter. We can rest assured that a change in one expression has not changed the computation of those that follow it, so we need not recompute them nor even examine them to assure ourselves that we don't need to. A functional language design also helps with the problem of localizing responsibility for a particular character in the output—it reduces the number of places in the program that need to be considered.

Another strategy for efficient implementation is to use functions rather than textual macros as basic structuring tools. A textual macro can expand into any arbitrary string of characters, including one that is not a syntactically valid unit. Allowing such macros would greatly complicate the problem of assigning responsibility for a particular piece

of output. Function calls, on the other hand, are always single syntactic units; they don't expand into anything; they execute, producing a value.

These last two issues prevented me from using the TEX language as a basis for Lilac, though it would have been very desirable to do so. Lilac simply faces too many constraints that a batch-processing text formatter doesn't have to face.

Boxes and glue, the components of the display list, are also the basic data types of the Lilac language. This means that the values returned by Lilac functions tend to describe rectangles or lists of rectangles. This helps considerably with the program-to-geometry mapping problem. It also means that the display list falls naturally out from the execution of the document program. It is simply the value computed by the top-level expression.

2.3. Editor design

If users can define structure by programming, an effective WYSIWYG editor must be prepared to smoothly handle an infinite variety of user-defined structures. Lilac's page view editor embodies several principles that help to achieve this goal.

The first guiding principle is that the editor is structure-based. The syntactic hierarchy of the document program forms the semantic hierarchy underlying all aspects of the editor's operation. Expressions are structures: each expression in the document program is, from the editor's point of view, an object which can be selected, copied, replaced, etc. Because the editor works on this basis, we can easily maintain a source view that is syntactically correct after every command.

This method of operation is reminiscent of syntax-directed editors such as Mentor [Donzeau-Gouge 84] or the Synthesizer Generator [Reps 87]. But where those systems are straightforwardly syntax-directed, Lilac is an *indirect* syntax-directed editor. The source view itself is a plain text editor, but an observer watching the source view during page view editing might well believe that he was watching a syntax-directed editor in operation. He would see all highlights and modifications (other than those in character strings) operating on whole syntactic units.

Some other basic principles of the design are:

1. Edit instances, not definitions. Editing a chapter title, for example, will always have its effects upon the contents of that particular title, never upon the formatting of chapter titles in general. The latter form of editing is not available in the page view; it is always done in the source view. In syntactic terms, page view editing always operates on units, never on function definitions.

2. Use the select-cut-paste paradigm of editing. It has been applied successfully in many contexts where a wide variety of objects are present.

3. Selections can be made at any structural level. A character can be selected; so can a word, or a paragraph, or a chapter, or indeed almost any node in the syntax tree. Pointing normally selects a character, but a readily accessed *Promote* command raises the selection to a higher structural level.

4. Insertion and deletion take place within *structural lists*. A structural list is the set of arguments that match one repeatable parameter in one function call expression. Since a structural list can contain any number of elements, deleting or inserting them will never lead to a syntactically invalid expression. As well as allowing any node to be selected, we also allow any group of consecutive members of one structural list to be selected. An insertion point is viewed as a special case of this type of selection—a selection of zero elements. Character strings are viewed as structural lists, with the characters as elements.

5. Existing structures are easily extended by *empty replication*. Empty replication makes a copy of the present structure and inserts the copy after the original—but the copy is an emptied form of the original. Wherever the original contained a character string, the copy contains an empty string; in general, the copy has the form of the original but none of the content. This makes it very convenient for the user to make more of whatever structure he is already working with, whether it is a standard one or an unusual, user-defined structure.

6. When all else fails, the user can accomplish unusual structure-editing tasks in the source view. But since that involves an unpleasant break of visual contact, we allow the user to momentarily escape to the document language using *escape commands*. If a structure called "widget" is defined, but the document doesn't contain any widgets yet, how to create the first one? Using an escape command, the user can describe a widget by typing a source-language expression in a temporary input area; the resulting widget is inserted at the insertion point in the page view.

2.4. Mapping

When the user points and clicks to make a selection, Lilac must figure out what object in the document he is pointing to. Since we are using a program as the underlying representation of the document, we need to perform a mapping from positions on the screen to nodes in the program's syntax tree. To show the user what he has selected we need to highlight an appropriate area on the screen. For this we need the inverse mapping, from nodes in the tree back to areas on the screen. Fig. 2-2

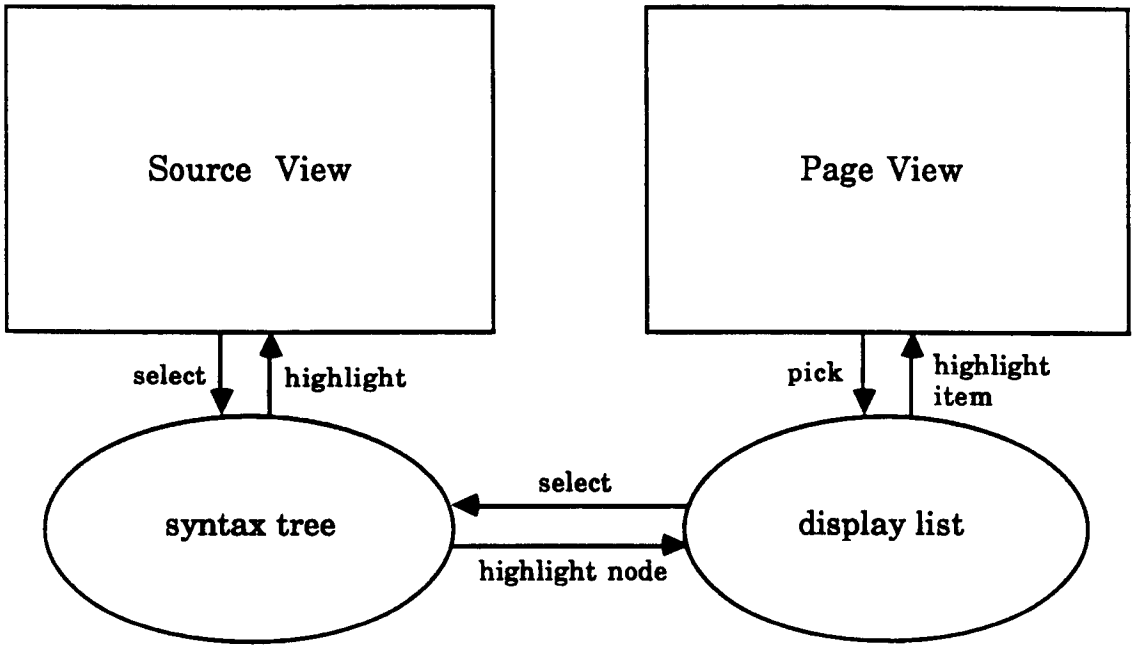


Fig. 2-2 Mapping relationships

illustrates all the mapping operations required in Lilac and how they fit into its structure.

As mentioned already, the display list is the key to solving the mapping problem. It is organized as a hierarchy of boxes containing smaller boxes, with words as the smallest unit. Each box stores a pointer to its container and coordinates giving its position relative to the position of its container. Each container stores a pointer to the list of its components. So a simple traversal up or down this tree suffices to map a point on the screen to a box in the display list, or a box in the display list to a rectangle on the screen.

Mapping to the syntax tree is somewhat harder. Before we can even begin to design an algorithm, we must settle the question of responsibility: given a box, what expression in the document program should be held responsible for its existence? Part of this question has already been settled by the decision to "edit instances, not definitions." We will always assign responsibility to something within a unit. But that doesn't finish the answer. Expressions are nested, and responsibility could be assigned at many levels in the nesting. Lilac's solution is to take the innermost: to assign responsibility to the smallest possible unit. This means that by default the user selects at the finest granularity of editable material, which is where the vast majority of editing takes place. Promotion provides access to higher levels.

In more precise terms, the algorithm for assigning responsibility is this: whenever a box is generated, there is one function currently in progress that was called from the mainline program. (All boxes are ultimately generated by primitive functions, so there is always such a call.) That function call expression is held responsible. This box may be passed as an argument to any number of higher levels of nested calls, but the responsibility remains with the call that produced it.

To map from a box to a node in the syntax tree, we store a *label* in the box—a pointer to the responsible node. This label is assigned at evaluation time when the box is generated, when the identity of the responsible node is conveniently available.

Mapping from a node to the box or boxes it produced is hardest of all. It is not always a one-to-one mapping: some expressions may be executed more than once, producing output at places widely separated on the page. This means that it will not work to store any sort of pointer in the node itself. The solution to this problem lies in a hash table called the Store. Here we store the value produced by each mainstream node, that is, each node which is part of a unit. If a node is executed more than once, it can store the several values distinctly in such a way that we can retrieve one or all of them as needed. Its principal use is in support of incremental update, but it also serves the purpose of mapping. When we want to map from node to boxes, we look up all its values in the Store. During execution, these values were (under normal circumstances) passed as arguments to other functions, which joined them into lists with other boxes and boxed them into composite boxes, and eventually they became part of the display list. So merely by finding the value(s), we have found the right place(s) in the display list, and the mapping is done.

2.5. Incremental update

After the user has given an editing command, thus changing the document program, we need to compute the new output of that program very quickly. Fig. 2-3 shows the steps required to update a Lilac document when an editing command is given. The choice of a functional language design is the key to efficiency: a change in one expression never changes the values of its neighbors. The only expressions whose values may have changed are the modified expression itself, and those that directly contain it—its ancestors in the syntax tree. We call the modified node and its ancestors "dirty", and all other nodes "clean". Only the dirty nodes really need to be re-evaluated. But dirty nodes often have clean arguments. The values of those arguments need to come from somewhere for the execution to proceed.

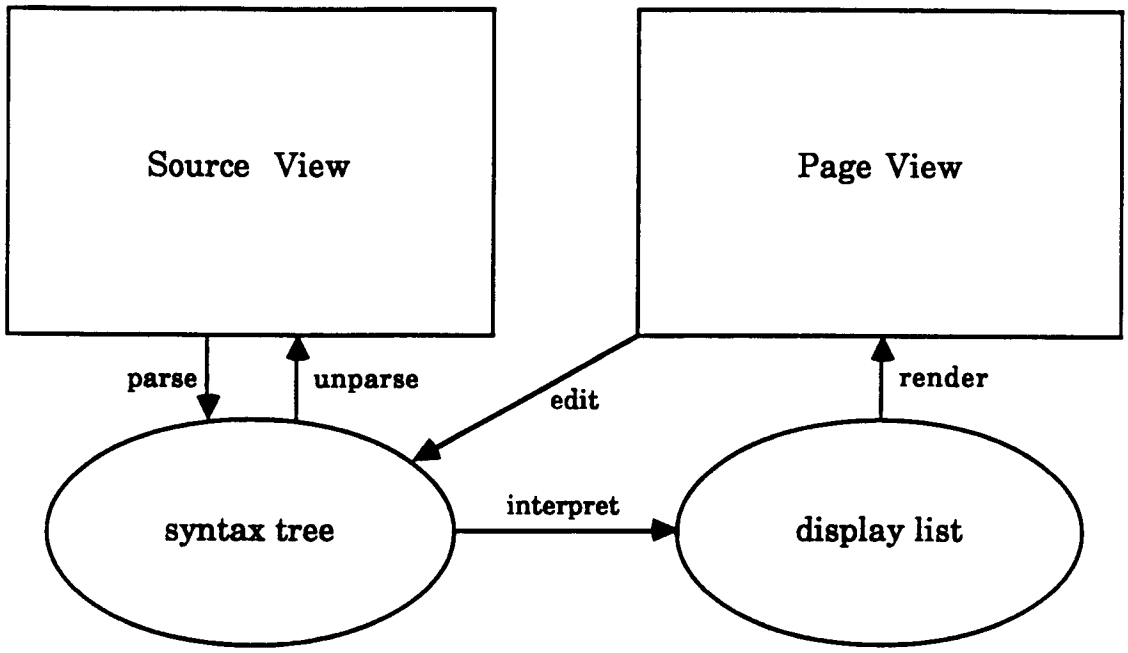


Fig. 2-3 Modification pathways

This need is met by the Store. The Store stores the value most recently produced by execution of each node in the mainstream program (and of some other nodes as well). When a dirty node has clean arguments, their values are found by retrieving them from the Store—a quick hash-table lookup. In summary: dirty nodes are re-executed; brothers of dirty nodes are looked up in the Store; all other nodes are *not even visited.* An O(n) problem has been reduced to an O(depth * b) problem, where b is the largest branching factor found in the tree.

In fact, this is not enough to bring the execution time down to 100 milliseconds. A second part of the strategy involves using incremental primitives. That is, the basic box-building functions of the language are implemented in a form that handles small changes efficiently. The following equation describes them:

old output + new input + change information -> new output, fast

The old output comes from the Store—in this case, we are looking up the value of a dirty node, but are not using it unmodified. The new input comes from the arguments to the function. The change information is attached to the new input, and tells what part of the input has been changed. This information, together with the old output, can allow the primitive function to do only a fraction of its usual work, by modifying the old output to produce the new. This is particularly important in the paragraph primitive, which is responsible for line-breaking and justification.

Part of the incremental primitive strategy is to return the same box records that were used last time wherever possible. Not only does this save allocation costs, but it helps us to gather geometrical change information. The old box records already contain coordinates giving their positions relative to their old container. As they are boxed afresh, new coordinates are computed. The difference between new and old coordinates describes how far each box has moved on the screen.

This provides the basis for the final strategy: incremental screen update. The watchword is: reuse the pixels, don't recompute them. Recomputing pixels is fast, but not fast enough to give smooth response to typing. But since most boxes contain information about both their old and new positions, we have enough information to reuse nearly all the pixels that are still valid representations of something in the text. We handle this efficiently for motions in both the horizontal and vertical dimensions, and in both directions in each dimension.

2.6. Road map to this thesis

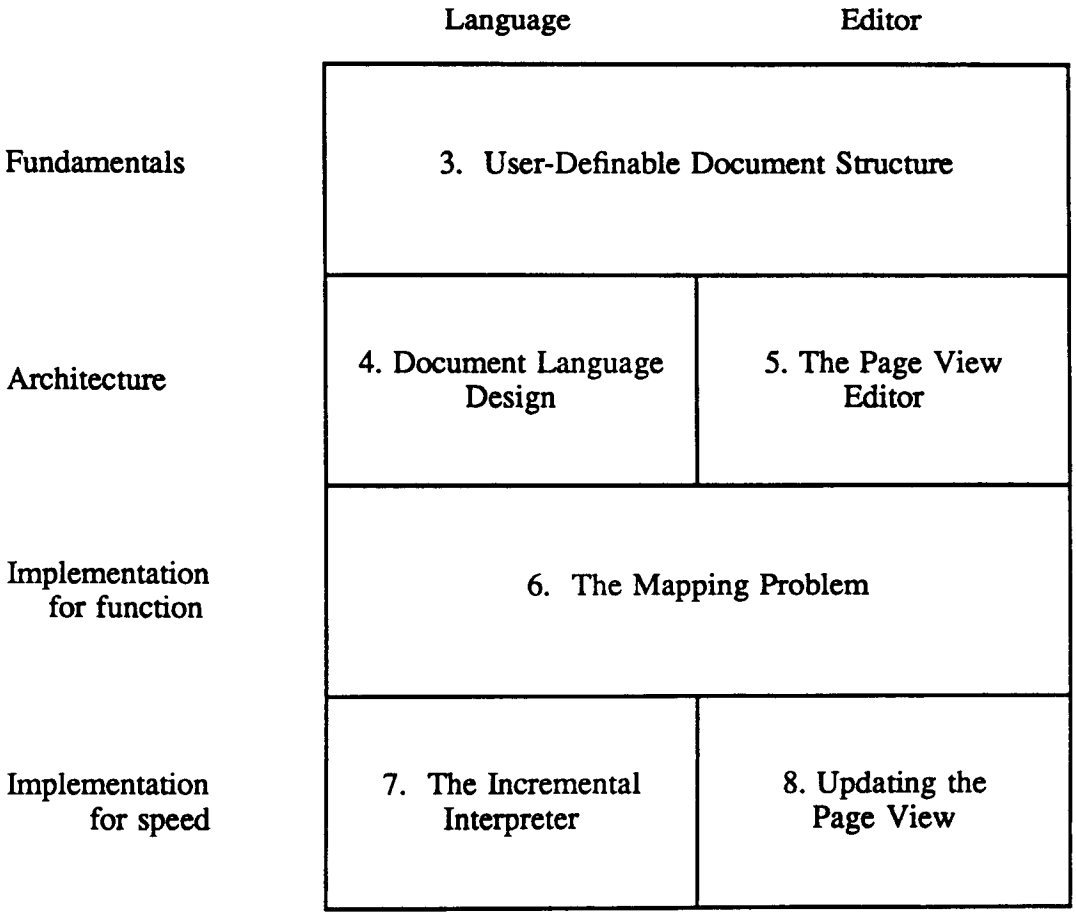


Fig. 2-4 Road map to the core chapters

Fig. 2-4 illustrates the relationships among the core chapters of this thesis. Chapter 3 discusses the structure and representation of documents in general, providing a basis for the design decisions of chapters 4 and 5. Chapters 4 and 5 discuss the design issues for the language and the editor, respectively. Chapter 6 discusses the implementation work required to make selecting and highlighting work—the mapping problem. Chapters 7 and 8 discuss implementation issues for the language and the page view display, respectively, with a special emphasis on incrementality and speed. Chapter 9 discusses the implementation of the source view, a much simpler problem but still nontrivial.

The remaining chapters discuss less well-explored issues. Chapter 10 discusses non-hierarchical aspects of document structure and how the Lilac paradigm might be expanded to deal with them. Chapter 11 discusses other future directions suggested by this project. Chapter 12 gives a survey of related work in the field. Chapter 13 reports on my experiences as a user of Lilac, and summarizes the results of this project.

2.7. A note on terminology

The units that make up the syntax tree are referred to as *nodes*. These correspond to expressions in the source language, and to selectable elements as seen from the page view editor. Text is stored in a type of node called a *character string node*, which corresponds to a quoted string literal in the source language and to a sequence of characters (each of which is a selectable element) in the editor.

The units that make up the display list, though they would be called nodes in normal data-structure terminology, will be called *items*. An item can be a box, a piece of glue, or one of a few special-purpose types. Some boxes are composite, being made up of other items. These boxes are called *containers*, and the items that make them up are called their *elements*.

3. User-Definable Document Structure

In the course of this research I have discovered that representing a document as a program really means two different things:

1. The user gets fine algorithmic control over the styling of the document.
2. He gets to define its structure.

The first was my original motivation for providing the power of programmability, and it came to me from the reports and complaints of several experienced users of document preparation systems. The second crept up on me somewhat by surprise, and has turned out to be the more significant of the two. When an author expresses his document as a program, he is free to create not only its structure but the terms in which its structure is expressed. This relates to an observation I have heard about programming: every programmer creates language, and every (nontrivial) program embodies a language of its own. The lower-level subroutines become the words of the language in which the higher-level routines are expressed. Similarly when documents are programs the macros or subroutines become the terms of the language in which the overall document structure is expressed.

3.1. The nature of document structure

Documents are by nature hierarchies of units made up of subunits, those in turn made up of smaller subunits, and so on down to the characters. One common form of this hierarchy, which we will use as a reference point, is found in many scholarly books and in this thesis:

Document
Chapter
Section
Paragraph
Sentence
Word
Character

A noteworthy feature of this hierarchy is that each unit is made up of an unspecified and flexible number of subunits. A document may consist of any number of chapters; a sentence may consist of any number of words, etc. To put it in a programmer's terms, the subunits make up *lists*. But not quite everything is a list. A chapter has exactly

one title, as does a section; the book is likely to have exactly one title page, table of contents, and index. Thus we see, amid the lists, a few *fixed elements*: elements which have a single, definite place within the containing unit.

Above we have a simple model: a hierarchy made up of seven types of elements. But in practice this neat pattern is often interrupted by less common types of elements: itemized lists, enumerations, displayed equations, tables, program examples, and of course, graphical figures. Graphical figures are outside the scope of this research, but the other elements listed are of interest: they are text, organized and typeset in special ways. They have special structures of their own, and within these structures, fixed elements are not uncommon. In a certain table, each row may have exactly four entries—no more, no less—and a particular styling for each entry. In an equation there may appear a fraction—with exactly one numerator and exactly one denominator.

In technical writing these unusual elements occur surprisingly frequently and, my experience would suggest, in virtually limitless variety. I believe they provide the strongest argument for programmability in document preparation systems.

Documents are hierarchical in their overall structure, but most of them contain a few *non-hierarchical threads of relationship*—minor aspects in which elements influence each other in ways that cut across the major hierarchy. Numberings of figures and tables often operate in this way—they are typically numbered in ascending order throughout a whole chapter, without regard for the structure of sections. The numbering and gathering of footnotes is another non-hierarchical matter, with an extra twist thrown in—it is organized by the page, which is a physical unit that bears little or no relationship to the logical hierarchical units. Cross-references are another: phrases like "see Section 4.5" or "refer to page 108" tie one part of the document to another across any number of levels of hierarchy.

A final aspect of document structure is physical structure—lines and pages. Here logical structure bows to the necessities of the printed medium. Paragraphs are broken into lines, not because of any reason within them, but because it is inconvenient to read lines that exceed a certain length. Books are divided into pages, not because the logic of the book recommends it, but because it is inconvenient to print, bind, or handle pieces of paper that exceed a certain size. So we end up with a second hierarchy, the physical, that cuts cross-grained to the logical hierarchy. A few rules relating the two are added to limit the damage: avoid widows and orphans, start each chapter on a new page, etc. And the physical structure gathers to itself a few trappings that have no place in the logical structure at all: page numbers, running headers, and the like.

3.2. Specifying a document

How real is the logical structure I have just described? It does not exist on the printed page. The page consists of black marks on a white surface, of lines of text in various typefaces at various positions. But it is real *in the mind of the author* and, if he has done his job well, *in the minds of his readers.* Now there is no law that says that a computer-aided document preparation system needs to represent any more than the appearance of the printed page. But it is a basic assumption of this work (and by no means original with me) that it is extremely useful to have the computerized system *model the document in the same terms in which the author understands it.*

Such a system models the document not as a series of lines and paragraphs in various typefaces, but as a hierarchy of meaningful units: chapters containing sections containing paragraphs, etc. Section titles are bold, not just because they happen to be bold, but because they are known to be section titles. This method of operation is advantageous because of the changing nature of documents in preparation. When styling information is bound to structure in natural ways, the computer's internal model not only generates a satisfactory appearance on the printed page, but is likely to *keep on doing so in the face of change.*

Once we have signed up for the notion that the computer should understand a document in this fashion, we need a way to communicate the necessary structural information to the computer. There are four levels of information that may be specified:

1. The text.
2. The particular structure of this document: where the chapter, section, and paragraph breaks fall.
3. The styling to be associated with each structural element type, e.g., what chapter headings look like, where they appear on the page, and how much space is left below them.
4. The structural element types themselves, and the nature of their substructures.

The first is basic, and is the material of every text editing program. The second is supported by every *hierarchical* document processing program—and even those that are not consistently hierarchical usually provide some support for the concepts of word and paragraph. The third is supported by most hierarchical document processors in some degree, though often they present only a small, limited set of styling choices. The fourth is seldom made available to users, especially in interactive editors. It is a large part of what makes Lilac an interesting challenge.

An example of the usefulness of user-defined structural element types is a "command description" structure, such as the one used in Appendix B of this document:

Cut ctrl-D

Delete the current selection and move its contents to the cut buffer. Leave an insertion point at the point of deletion.

This is a structure with three elements: the command name, the key binding, and the description. Lilac allows the user to define this structure and place each of its elements appropriately. This is where commercially available WYSIWYG editors fall short: they provide no means for the user to specify that this is one structural unit. In such systems it would appear as two paragraphs: the first with a normal left margin and a tab stop in the middle, the second with an indented left margin. A list of such descriptions would appear as a list of the two types of paragraphs in alternation. Such alternating structures are painful to produce, because paragraph properties have to be respecified in some fashion at every paragraph boundary. In Lilac, where the description appears as a single structure, it can easily be replicated to extend the list.

3.3. Documents as programs

This is the reasoning behind the original thesis—the assertion that documents should be represented as programs. The real issue is to give the user access to levels 3 and 4 in a flexible and powerful way. The experience of the TEX and Troff user communities suggests that it is not enough to provide various options and parameters for styling various constructs; the advanced user really needs *algorithmic* control over styling, so he can make styling respond to a wide range of circumstances in desirable ways. Programming provides this; it also provides a good mechanism for specifying styles that are simply unusual and unlikely to be provided for by the original system designer.

The connection between level 4 and programmability is not so obvious; indeed, it is not entirely necessary. Structural element types and their substructures could be specified by means simpler than programming, and their styling programs attached when needed. But in fact I have found that the task of specifying them fits smoothly into a programming model; they appear as declarations. In Lilac the relationship between structure and programming is as follows:

- The *header* of a function definition specifies a structural element type; the parameters and their types specify its substructure.
- The *body* of a function definition specifies the styling of that element type: what is to be done with the elements of the substructure. A function once defined

can be *overridden* with a new definition that has the same header and a different body to change the styling of that element type.

- A *call* to a function specifies a particular element of that type. The arguments to the call are the subunits below it in the hierarchy.
- The *main program* specifies the particular structure of the document. It takes the form of an expression made up of nested function calls, each specifying one structural element.
- The text of the document appears in *character string constants* within the main program. These are the most common form of argument to bottom-level function calls. Character strings usually enclose text at the paragraph level; the sentence and word levels are not explicitly represented.

We will look at a brief example. Do not expect to fully understand the syntax yet; that is explained in the next chapter. Here we are showing the general idea. We are defining a new structural element type, a Definition. It has two subordinate elements: a headword (the word being defined), and a description (the paragraph that tells what it means). The styling chosen is that the headword is set in boldface ("Bold") and outdented (negative value of "indent") and on the same line with the beginning of the description, separated from it by 12 points of space ("Hskip").

```
function Definition(headword: @Hlist, description: Hlist) =
  let indent = -16 in
    Para(
      Bold(headword),
      Hskip(12),
      description)
```

A trivial main program consisting of one definition might look like:

```
unit main =
  Definition(
    "Aardvark",
    "A burrowing mammal of southern Africa, having a stocky,
     hairy body, large ears, a long, tubular snout, and
     powerful digging claws.")
```

And would appear in print as:

Aardvark A burrowing mammal of southern Africa, having a stocky, hairy body, large ears, a long, tubular snout, and powerful digging claws.

This design has an interesting and useful feature: we use the same programming language to express both structure and styling. In the main program it specifies document structure and content; in the function bodies it specifies styling. And the

same procedures are available to serve both purposes. This is handy because it means that new object types can be defined on the basis of old ones, calling the old function as a subroutine to produce its usual styling effects. In the example we see this in action: the Para function, which might appear in a main program to specify a paragraph element, is being used to achieve a styling effect in a Definition.

The convenience of using the same language for all purposes also works the other way: sometimes, when creating a special, one-of-a-kind display element, an author may find it most convenient to skip the function definition step entirely and put styling specifications directly into the main program. This loses the some of the benefits of structure-based editing, because individual styling elements appear without any structure to show their logical grouping. Many good computer scientists would cry "Unclean!" at the very thought, but I believe there are times when it is the most efficient way to operate. I do it most often to add a bit of vertical space here and there to improve appearance.

3.4. Document types

Not all documents have the same sort of structure; it varies depending on the nature and purpose of the document. A book has one structure; a business letter has another; a cooking recipe has yet another. Each of these might be called a *document type*. In Lilac the type of a document is loosely defined as the set of structural element types defined—that is, the set of function definitions available for use within it. A document is given a particular document type by including a *style file*: an external file of ready-made function definitions. I say "loosely defined"—a document may include more than one style file, and it may add to the provided environment new definitions of its own. So we may speak of documents having "almost the same type."

Besides structural element definitions, a style file commonly includes a few other things:

- Default values of style control variables. These specify styling parameters for the whole document.
- A *page model*—a special function definition that controls page-related features such as page numbers and running headers.
- A *new document template*. This gives a skeletal structure for new documents of this type.

The new document template is particularly important because it expresses an aspect of structure not expressed in the function definitions: the proper relationship of one

structural element to another. In our standard example, the function definitions will tell us that chapters and sections and paragraphs exist, but they will not tell us that chapters should be composed of sections or that sections should be composed of paragraphs. The new document template fills in this gap by providing a minimal structure:

```
unit newdoc =
  Document(Chapter("", Section("", Para(""))))
```

This specifies an empty document containing one chapter with one section containing one paragraph—and empty character strings in all the appropriate places. The editor provides a command for creating a new document of a particular type; when it is invoked, this template is copied to provide the initial main program for the new document.

4. Designing the Document Language

Editing a program by editing its output is a difficult problem. The problem can be made much more manageable by the careful design of the programming language. As mentioned in Chapter 2, the language design has three fundamental goals:

1. to facilitate page-view editing, by having a structure that corresponds closely to the structures in the user's mind
2. to facilitate incremental updating, by making it easy for the interpreter to figure out what has changed and what hasn't
3. to facilitate structure-view editing by being clean, terse, and readable

The resulting language differs in some significant ways both from traditional document languages and from traditional programming languages. In some ways it is a midpoint between the two. Its distinctive features are:

- program-like syntax
- boxes and glue as the basis of the type system
- functions rather than macros as the basis of programmability
- declarative distinction of "mainline" from "subroutine" material
- a functional style, with side effects limited and discouraged
- dynamically scoped variables
- call-by-name arguments
- repeatable arguments as a major structuring feature
- strings as implicit lists
- limited polymorphism

This chapter discusses these features, and includes a complete grammar in Fig. 4-2.

4.1. An example document

Before we examine these features in detail, let's look at an example of a Lilac document, shown in Fig. 4-1. The source view (the program) is on the left, the page view (the output) is inset on the right. The reader is not expected to understand all of it at this point, but it should prove useful for reference from time to time.

At this stage we can point out the overall structure of a Lilac document. First comes a **use** directive, which includes a *style file*—a separate file that defines standard functions and provides standard values for style parameters: font, margins, spacing, etc. In this example the style file is responsible for the definitions of Chapter, Section, Italic,

```
use doc.li

defaults
  rmargin = 210

function Tex =
  Hbox("T", Hskip(-3),
    Raise(-3, Hbox("E")),
    Hskip(-3), "X")

unit main =
  Vlist(
    Chapter("A Chapter Title",
      Section("A Section Title",
        Para(
          "This sample document demonstrates
the structure of Lilac documents"),
        Para(
          "This paragraph shows",
          Italic("italic"), " and ",
          Bold("boldface"),
          "text and the fancy ",
          Tex, "logo."))))
```

A Chapter Title

4.A Section Title

This sample document demonstrates the structure of Lilac documents.

This paragraph shows *italic* and **boldface** text and the fancy TEX logo.

Fig. 4-1 A sample Lilac document with both views

and Bold; the other functions are primitive. Next we see the **defaults** section, with a default value declared for **rmargin**, the right margin. This overrides the default provided in the style file. 210 is an unusually small value; it allows the output to fit in the inset box we have provided for it. Next we see a user-defined function, "Tex", which formats the TEX logo. Then comes the unit "main", which contains the body of the document. Within this unit, indentation clearly shows the hierarchical structure of the text.

Now we turn to the distinctive features of the Lilac language design.

4.2. Program-like syntax

Most formatter languages have a syntax with "text on the outside, functions on the inside". Normal, running text appears in the source file simply as normal, running text. Special functions, change of font, change of formatting, etc. are expressed by a special escape notation. Scribe [Reid 80] is a good example, with such expressions as

normal text @i(this is italic) more normal text

In Lilac I have chosen a syntax that is more like traditional programming languages: "functions on the outside, text on the inside". Special functions appear as traditional

Algol-style function calls with argument list in parentheses, and normal text appears in quoted strings.

Ordinary formatters cannot easily afford this style. The source is the only mode in which the user can edit, so it must be as natural to him as possible. When he is thinking plain, running text, it needs to look like plain, running text. So they give him plain, running text, and it is the formatter wizards who suffer. Programs in a language like TEX are cluttered with escape characters (usually backslash) all over the place. In addition, there is the problem of spaces. When blank space, either vertical or horizontal, surrounds a special function, one always wonders, "is that space there just to set off the special function, or is it part of the text?" Rules can be defined to answer this question, but they are not obvious.

Lilac, on the other hand, can afford a "functions on the outside" style because of the two-view operation. If the user is editing plain, running text, he should be doing it in the page view. The source view is for programming and for establishing structure, and in that case, special functions are going to be the topic of interest. So we use a source view language that is optimized for clear seeing of structure. The page view is there for clear seeing of text.

4.3. Boxes and glue as the basis of the type system

Lilac borrows from TEX the concept of boxes and glue. Material to be typeset is organized in terms of horizontal lists of items, which can be boxed into horizontal boxes, and vertical lists, which can be boxed into vertical boxes. A list consists of subsidiary boxes, and of glue, which sits between boxes. A box can be viewed as a rigid object, having fixed dimensions. There are three of them: width, the horizontal extent; height, the vertical extent above the baseline; and depth, the vertical extent below the baseline. Glue, in contrast, is a flexible material. Glue is either horizontal glue or vertical glue, depending on the type of list it is intended to link together. It has three properties: natural size, stretchability, and shrinkability.

The fundamental operation on these objects is the building of boxes. Boxing a list transforms it from a flexible object of indeterminate dimensions into a solid object of definite, fixed dimensions. This can be done in one of two ways: to natural length (in which case all glue receives its natural size), or to specified length. In the latter case the flexibility comes into play. When the specified length is longer than the sum of natural lengths, each item of glue stretches in proportion to its stretchability, as far as is needed to fill out the specified space. When the specified length is shorter than the

Nonterminals:

Document	::=	Imports Defaults Definitions
Imports	::=	{ Import newline }
Import	::=	**use** FileId
Defaults	::=	{ Assignment newline }
Assignment	::=	Id '=' Expr
Definitions	::=	{ Definition newline }
Definition	::=	**function** Id ['(' FormalParams ')'] '=' Expr
		| **unit** Id '=' Expr
FormalParams	::=	FormalParam { ',' FormalParam }
FormalParam	::=	Id ['*'] ':' Type
Type	::=	PrimType | '@' PrimType
Expr	::=	Number | String | Variable | Call | ArithExpr |
		'(' Expr ')' | LetExpr | LocalExpr | IfExpr
Variable	::=	Id
Call	::=	Id ['(' Arguments ')']
Arguments	::=	Expr { ',' Expr }
ArithExpr	::=	Expr Binop Expr | '-' Expr | **not** Expr
LetExpr	::=	**let** Assignments **in** Expr
LocalExpr	::=	**local** Assignments **in** Expr
Assignments	::=	Assignment { ',' Assignment }
IfExpr	::=	**if** Expr **then** Expr **else** Expr

Terminals

Number	A number, optionally including a decimal point and fractional digits.
String	A string of characters surrounded by double quotes.
FileId	A filename of any syntax acceptable to the system, provided it does not start or end with blank or include newlines.
PrimType	One of the system types, listed in section 4.3.
Binop	One of: + - * / % = # > < >= <= **and** **or**

Metasyntax

[ ]	surround an optional element.
{ }	surround an element which may occur zero or more times.
|	separates alternatives.
boldface	indicates keywords which appear literally in the syntax.

Fig. 4-2 Grammar of the Lilac language

sum of natural length, each item of glue shrinks in proportion to its shrinkability. It is this flexibility that allows paragraphs to be justified.

For more information on boxes and glue see [Knuth 84], chapters 11 and 12.

These concepts, together with a few other basics, make up the primitive types of Lilac. The types above, which have to do with objects visible on the page, are called tangible types. The others are called intangible types. The types are:

Intangible types:

Num	a number (fractions are supported)
Bool	a Boolean value—either true or false
Font	a font specifier

Tangible types:

Box	a box
Hglue	an item of horizontal glue
Vglue	an item of vertical glue
Hlist	a horizontal list
Vlist	a vertical list

Some basic operations on these types are

Hbox(list: Hlist): Box

Box a horizontal list, to natural width.

HboxTo(width: Int, list: Hlist): Box

Box a horizontal list, to the specified width.

Vbox(list: Vlist): Box

Box a vertical list, to natural width.

VboxTo(size: Int, list: Vlist): Box

Box a vertical list, to the specified vertical size.

Hlist(items): Hlist

Build a horizontal list out of Box and Hglue items.

Vlist(items): Vlist

Build a vertical list out of Box and Vglue items.

Para(list: Hlist): Vlist

Break a horizontal list into lines and justify them, yielding a vertical list consisting of lines and interline glue.

4.4. Functions vs. macros

Most document formatters that aspire to be programmable at all provide macros as the means of programming. However, macros, defined in terms of textual substitution,

are slippery things. A macro, appearing as one syntactic entity in the source, can turn into several disjoint entities in the process of substitution. Or it can even turn into a partial syntactic entity, by including keywords in odd places in its body. It can cause name-conflict problems, if a name used locally inside the macro definition collides with a name used elsewhere. For these reasons, I decided to use functions rather than macros in Lilac. In fact, the decision was motivated largely by the complaints of some experienced TEX users who found macros to be a complicated mess.

What is the distinction? The semantics of the output. A macro produces a string of text at parse time; a function produces a value at runtime. There is no confusion about the number of entities in a function: it looks like one thing, and it produces one value.

Along with a function-based syntax, Lilac uses strong typing. Every parameter has a type; every actual argument must match its parameter's type. Similarly, every function returns a known type. The return type is not declared, because the parser can easily deduce it. The rigors of strong typing are eased by one implicit type conversion rule: wherever a list type is expected, any value that could be an element of that list is acceptable. It is implicitly converted into a one-element list.

4.5. Declarative distinction of mainline from subroutine material

In Chapter 3 we said that function bodies represent styling, while the main program represents the structure and content of the document. In truth, this design is a little too restrictive for comfort, so we provide means to break up that "main program" into more than one unit. But the fact remains that there is a clear distinction between this "mainline" material, which expresses the structure and content of this particular document, and "subroutine" material, which defines structural elements and describes their styling.

To keep this distinction clear, Lilac provides two structuring elements: the *function*, introduced with the keyword **function**, and the *unit*, introduced with the keyword **unit**. A function is the container for "subroutine" material— material which defines structures or patterns to be used repeatedly throughout the document. It should not contain any content material—words that contribute to the meaning of the document. Like a function in any language, it can take arguments.

A unit is the container for the content material of the document—the words and instances of structures which give it meaning. The document as a whole is described by the unit named "main", which may reference other units. A unit is normally expected to be used only once, though that is not enforced. Unlike a function, it cannot take arguments.

By providing two separate constructs, we allow the user to tell us clearly which objects are part of the content of this document, and which are being defined as structural element types to be instantiated in potentially numerous places. This gives us the knowledge needed to implement one of our principles of editing: *edit instances, not definitions*. This principle, in turn, solves a basic problem that arises in the process of editing a program by editing its output. "When I select the output of a subroutine, have I selected the *call* to the subroutine, or the *definition* of it?" Our principle gives us a simple answer: the call. We arrange matters so that regardless of where the user points, he never selects anything that is part of a function definition. Any attempt to do so will instead select the call to the containing function.

It is worth noting that the question has not entirely gone away. It still arises at the point where one unit references another: do we select the reference, or the contents of the referenced unit? However, it is not a problem in practice. Lilac's solution is, once again, to select the reference. In the rare event that the user specifically wants to replace the entire contents of the referenced unit, he can edit the source view.

4.6. A functional style

Fast incremental interpretation relies upon the fact that we need to reinterpret only a small number of nodes, and can easily tell which nodes those are. This is greatly facilitated by not having to deal with side effects. In the absence of side effects, we know that a dirtied node has dirtied its ancestors, but not its neighbors. The scope of a change is well understood. Hence Lilac uses a functional language: there is no ordinary assignment statement. Scoped assignment expressions take their place. For most of the actual needs of typesetting, this arrangement works well.

Unfortunately, it does not meet all the needs of typesetting. The automatic numbering of figures in a chapter, for example does not fit into this mold. It requires a variable, the current figure number, which is incremented each time a figure appears. The scope of these changes cuts across the boundaries of any other structures that may be present in the chapter. Lilac at present does not solve this problem. Chapter 10 presents a design for a solution within the Lilac context.

4.7. Explicit dynamic scoping

Typesetting fits reasonably well into the mold of a functional language, but it fits very badly into the scheme of passing all relevant variables as parameters. The reason is that there are a large number of formatting parameters that are needed by commonly

used primitives, but which are seldom changed. Among these are the font, the margins, the paragraph indentation, and the interline spacing. Many of these, needed in low-level operations, are unlikely to be relevant to higher-level operations. The paragrapher operates in the same fashion regardless of the font; the presentation of a chapter or a section is probably independent of the setting of the interline spacing. It would be a great syntactic burden to pass all these parameters as arguments to every function; hence the choice of dynamic scoping as an alternative.

Dynamically scoped variables, known as "environment variables", are modified and scoped explicitly, by use of the **let** construct. An example is

```
let font = TimesRoman9, lmargin = 80, rmargin = 360 in
  Para(...)
```

which typesets the paragraph with the specified, probably unusual, font and margins. The **let** construct has an assignment clause, in which one or more environment variables are assigned values, and a body, which is always a single expression, over which that assignment applies. During the evaluation of that expression, *or of any function called from it,* the new values of the variables apply. As soon as it is completed, those variables revert to their former values.

Incidentally, there are also ordinary local variables. They are assigned with explicit scope like dynamic variables, but using the keyword **local** instead. An example is

```
local a = Hglue(0, 500, 0) in
  Hlist(a, ..., a)
```

which surrounds some material with two equal and highly stretchable pieces of glue. The assignment of the local variable applies within the body of the **local** construct, but not within any functions called from it. A local variable is never seen outside of its defining function.

The set of environment variables and their values, together with the set of functions and units defined, make up the *environment.* An environment variable is normally introduced in the **defaults** section at the head of a document or a style file. Here it is given its default value, the value which it has outside the scope of any **let** expression. The default values of system-defined environment variables can also be overridden in the **defaults** section.

4.8. Call-by-name

There are certain advantages to macros in typesetting—in particular, their convenience for creating user-defined "environment functions". An "environment

function" is a construct whose sole purpose is to change the setting of one or more environment variables within a specified scope. These are constructs such as Italic(text) or Bold(text)—every formatter language has them, in one form or another. Essentially, what we want is a packaged "let" construct. If macros are the model of programming, this is easy: just write something like

```
define Italic(stuff) =
  let font = ItalicFont in stuff
```

But if functions are the model of programming, that doesn't work so well. Function arguments are evaluated before the function call! The "stuff" would be evaluated already, before the Italic function gets it, and it would be too late for the setting of "font" to make any difference. Well, that is not entirely true. If the "stuff" argument is of type String, and if the body of Italic forces the conversion of that string to Hlist, then the font will get bound at the right time. But that considerably limits the circumstances in which Italic can be applied. Its argument must be a plain character string—no superscripts, no subscripts, no special symbols. Those things can only be expressed in terms of list material (Hlist or Vlist)—and list material has the font already bound.

The solution is to allow call-by-name arguments. Not for all arguments, but as an option, the argument is not evaluated at the time of call, but at the time of use. This is specified as part of the definition of formal parameters; such a parameter is called an *unevaluated argument.* As an example, the (somewhat simplified) definition of Italic:

```
function Italic(stuff: @Hlist) =
  let font = ItalicFont in stuff
```

The argument "stuff" is defined to be an argument of type Hlist, and the '@' sign indicates that it is an unevaluated argument. At the time of call, what is passed is not, in fact, an Hlist, but a handle to an expression whose evaluation will produce an Hlist. Execution of Italic proceeds as follows. We begin work on the **let** expression, and set the "font" variable to ItalicFont. Having done that, we encounter the expression "stuff". Its type shows that it is unevaluated. So now we follow that handle, and encounter the expression which is the argument to Italic, and we evaluate it. As we do so, "font" has the value ItalicFont. That evaluation ends, and we now have in hand a real Hlist, built using ItalicFont. Now the **let** expression finishes, and "font" is restored to its former value. Italic returns, yielding the Hlist as its result.

"Environment functions" could be supported without such unusual semantics. They could be provided as a separate construct, distinct from other functions, whose definition is simply a setting of environment variables. But there is much to be gained by merging the two facilities into one. Functions which express document structure and

also manipulate the typesetting environment of their arguments allow us to bind style to structure: to make certain elements of the document take on certain styling aspects just because of their place in the structure. This is one of the fundamental benefits of programmability in a typesetting system. An example of its usefulness is the Chapter function:

```
function Chapter(title: @Hlist, body: Vlist) =
  Vlist(
    Center(let font = ChapterTitleFont in title),
    Vglue(ChapterHeadingGap),
    body)
```

In this example a chapter title is centered and set in a special font. It is set off from the body of the chapter by a specified vertical spacing. The body is set with the default font, line width, spacing, etc. for this document.

Here we can see the advantage of mixing "environment-like" and "function-like" features. In order to set the title in a special font (and still allow it to contain such complexities as superscripts and subscripts) we must change the environment before evaluating it. But in order to concatenate it into a vertical list (which is the output of the Chapter function) we must evaluate it and box it. It must be boxed in a box, with a definite height and width, before vertical list-building can proceed. And thus the unevaluated argument facility proves its usefulness: it has allowed the Chapter function to evaluate its title argument at a time of its own choosing, in an environment of its own choosing, and to do further computations upon the result of the evaluation.

4.9. Repeatable parameters as a basic structuring feature

Documents are, by their nature, composed of lists. A book is a list of chapters; a chapter is (if complicated enough) a list of sections; a section is a list of paragraphs; a paragraph is a list of words; a word is a list of characters. The essential feature of a list is that it has no prespecified length: its length depends upon its content, and is likely to change from time to time. A traditional function, taking a fixed number of arguments, is a very inconvenient construct for modeling a list. Rather than add to the language a special construct for specifying lists, I decided to make functions fit for that task by allowing variadic functions.

A Lilac function may, in addition to any ordinary parameters, take one repeatable parameter. It must come last in the parameter list. In a call to this function, any number of arguments (including zero) may be provided to match this parameter. They come after any arguments that match ordinary, fixed parameters; these must always be provided. Taken together, they form a *structural list*. As we will see in the next chapter, structural lists are important to the process of page view editing.

Any language that supports variadic functions faces the question: how is the list of arguments to be handled inside the function? How does one variable represent a whole list of values? The answer in Lilac is an ad hoc one, well suited to the needs of typesetting and probably abominably suited to most other purposes: implicit list-building. In Lilac, a repeatable parameter must be of a list type. Hlist (horizontal list) and Vlist (vertical list) are the basic list types. When the arguments corresponding to a repeatable parameter have been evaluated, they are *automatically concatenated into a list* of the appropriate type. This list becomes the value of the repeatable parameter, as seen internally. These arguments may be of the same type as the repeatable parameter, or they may be of a type compatible with elements of that list type. The former are concatenated into the list as sublists; the latter are added in as single elements.

This scheme has the disconcerting feature that the function cannot enumerate its arguments. It cannot see them as separate entities at all; they have all been rolled into one continuous list. But it has a countervailing advantage: a function that has received a list as a repeatable parameter can easily concatenate other material onto that list and pass it along to another function that takes a repeatable parameter. There is no problem of the second function seeing the list as a single argument when it should see it as several; it simply sees one list. Variadic functions are very conveniently composable.

4.10. Strings as implicit lists

The basic geometric units of Lilac data are boxes and glue, but the most common basic syntactic units are character strings. What does a character string really mean, and how does it get turned into typeset characters with a definite font and size? In every text formatter a lot of implicit work is done to every character to perform this conversion. Lilac is no exception. In Lilac, though, the language has a strong type system, and each character string appears as an argument to some function, not just a part of the general stream. So character strings need some type.

The answer is to view each character string as a horizontal list. The reader sees a character string; the function taking the string as an argument sees an Hlist, with a box for each word and a piece of glue for each space. A string looks like a constant of type Hlist—but not quite. It does not contain all the information needed to specify that list. That also requires the font, a dynamically scoped variable whose value is only available at runtime. So really, every string is an implicit function which makes use of the font variable and returns an Hlist. But few users will ever bother to think of it that way—to the common user, a string is just an Hlist.

Underneath the implicit conversion of strings into Hlists lies one of the central functions of Lilac—the ConvertString function. Purists might argue that it should be given an explicit place in the language. But its invisibility, besides keeping document programs uncluttered, serves a very useful purpose: it helps to present the desired model of a horizontal structural list. A structural list, as the reader may recall, is a list of arguments matching one repeatable parameter. A horizontal structural list is a list of arguments matching one repeatable parameter of type Hlist. The vast majority of all editing consists of inserting and deleting things in horizontal structural lists.

If ConvertString were explicit, each character string would be an entity unto itself, a separate branch of the syntax tree. If we had a line of text with one special symbol in the middle (which would require a non-string node to express it), then we would have a structural list with three elements: a string of characters up to the symbol, the symbol, and a string of characters after the symbol. But that is not the model I want to present. Rather, I want each character in those strings to appear as a first-class member of the structural list. The structural list is viewed as a list of many members, with the special symbol being one among them. The user could select the symbol together with a few characters before and after it, and copy or delete them. The list is what it appears to be: no invisible substructure clothes the characters.

A source language which corresponds to this model should have the characters as lightly clad as possible. I have not done a perfect job: character strings still have quote marks around them. This is a compromise that helps to make a parseable and comfortable source programming language. But at least they have no more than that, because their conversion to Hlist is implicit. And as any user of Lilac soon finds out, those quote marks are very evanescent things. Strings are split by the insertion of non-string material, and merged when adjacent, with no fanfare at all. The quote marks are a syntactic artifact; it is the characters that count.

It is not entirely clear that a "text on the inside" syntax is the right choice. A "text on the outside" syntax would better serve the editing model, in which every character is a first-class member of the horizontal structural list that contains it. The deletion of an embedded special function would simply delete the special expression that specifies it. There would be no mysterious disappearance of quote marks as two quoted strings fused after the deletion. Only time will tell whether this consideration is important enough to govern the choice of syntax.

4.11. Generic arguments

Consider again the style-modifier functions, such as Italic:

```
function Italic(stuff: @Hlist) =
  let font = ItalicFont in stuff
```

This function will italicize its argument, whatever it may be, with one limitation: the argument must take the form of a horizontal list. Supposing one wanted to italicize a whole paragraph? He would have to write a new function, because the Para primitive returns a Vlist. This seems a waste of effort, because the two are conceptually the same—they italicize something. To get around this problem, we define the special type Any. A function is allowed to have one unevaluated parameter of type Any. If it does, it may also return a value of type Any (and indeed, is almost certain to do so). When such a function is called, the type Any may match any actual type—and if the return value is of type Any, the function call is presumed to return the same type as the type of the Any argument. Such a function is, in every respect, simply a packaged let expression.

The most important function of this class is a primitive, UseFont:

```
function UseFont(family: FontFamily, face: FontFace,
  size: Num, body: @Any): Any
```

UseFont computes a value for the "font" variable, based upon the (family, face, size) triplet. It also sets the global variables "fontfamily", "fontface", and "fontsize" directly from the corresponding arguments, and establishes values for a few more font-related global variables. It then executes its "body" argument in that context, and returns the resulting value.

UseFont is a primitive because it is very common and needs to be fast. Because of the convenience of establishing all the font-related variables at once, it is almost always used in preference to a direct "let font = ..." expression. With UseFont explained, we can now see the true definition of Italic:

```
function Italic(stuff: @Any) =
  UseFont(fontfamily, italicface, fontsize, stuff)
```

That is, it preserves whatever font family and size were already in effect, specifies the italic face, looks up the font that corresponds to these attributes, and evaluates "stuff" with that font in effect.

Italic illustrates the nestable nature of Lilac's polymorphism. Italic takes a polymorphic argument, and it passes it on as UseFont's polymorphic argument. Whenever Italic is called, the actual type becomes bound at all levels.

This is accomplished without any overhead at runtime, because the polymorphism is quite limited. The only valid operations on a polymorphic argument are to use it as the body of a **let** expression or to pass it to another procedure that also takes a polymorphic argument. Polymorphic functions are just packaged **let** expressions.

This raises the question: would it have been better to base the language on macros? The decision to base the language on functions and strong typing has led to quite a few unusual design decisions:

- dynamic binding
- call-by-name arguments
- polymorphism

The main countervailing advantage of procedures is that they lend themselves to the use of an explicit type system, which is helpful as part of the basis for a WYSIWYG editor (see Chapter 5). More experience is needed before this question can be fully decided.

4.12. Page formatting

Not fundamental to the language, but worth mentioning, is the mechanism for describing page-level formatting, that is, such features as running heads and feet, top and bottom margins, where the footnotes go, etc. These are handled by a special Lilac function called PageModel. Its signature looks like

```
function PageModel(content: Vlist, pagenumber: Num): Box
```

The system defines it, but the user is invited to override that definition. The special feature of this function is that it is called after page-breaking has been done. In the process of formatting the document, Lilac evaluates the top-level expression, producing a vast Vlist that describes the whole document. It then breaks this Vlist into sublists that describe a page each, using the value of the "pageheight" variable to determine the length of each sublist. Then Lilac calls the PageModel procedure, once for each page, passing the sublist as the value of the "content" argument. PageModel returns a box describing the finished page.

4.13. Side effects

An interesting problem in this language involves the use of call-by-name arguments: it opens a door for side effects. One argument can be used to set the value of an environment variable which is used in the evaluation of another argument of the same function. Consider the example below. The first argument, "victim", is used to establish the value of the environment variable "name". Because the second argument,

"text", is evaluated at the time of use, it is evaluated with this environment in effect. A change to the first argument of FormLetter must now cause some re-evaluation of the second, if consistency is to be maintained.

```
function FormLetter(victim: String, text: @Vlist) =
  let name = victim in text

unit main =
  FormLetter(
    "Jones",
    Vlist(
      Greeting("Dear Mr. ", name),
      Para("We would like to offer you and Mrs. ",
        name,
        " a unique opportunity...)
      )
   )
```

We could solve this problem by prohibition: prohibit a function argument from being used, directly or indirectly, as the source of an environment variable's value. But this would outlaw some extremely useful functions, including UseFont!

At this writing, Lilac solves this problem by a published warning to the user: do not allow a function argument to establish the value of a global variable *of tangible type*. Page view editing can never touch an argument of type Num, or Bool, or Font—no means are provided for it. There is no visual way to select such a thing, because it does not directly generate any image on the screen. Page view editing applies to boxes, glue, and lists. So we warn the user not to create side effects involving variables of those types. And in fact, this leaves all the most desirable cases untouched. As one can see above, it took quite a contrived example to make this problem actually show up.

A more rigorous solution would involve storing extra information. A check at parse time can determine whether an argument is used to establish the value of an environment variable, and whether any other argument is evaluated within that environment. If so, we would attach a *function property* to the function in question. This particular function property would be of the "cross-influence" type. In the FormLetter example above, it would state, in effect, "argument 1 influences argument 2". At editing time, if we mark the "victim" argument dirty because it has been changed, the cross-influence property would cause us to mark the "text" argument node *and all its subtree* dirty as well. This would make the editing of the "victim" argument a slow process. Fortunately, in real situations, such editing is not likely to happen very often. Not very many document constructs really need to be structured that way.

Functions can, of course, pass their arguments to other functions. If this happens, while checking a function for cross-influence we would look at the property lists of the functions it calls. If we found a cross-influence property there, and if the arguments in question were derived from arguments to the main function, then the main function would receive a cross-influence property as well.

5. The Page View Editor

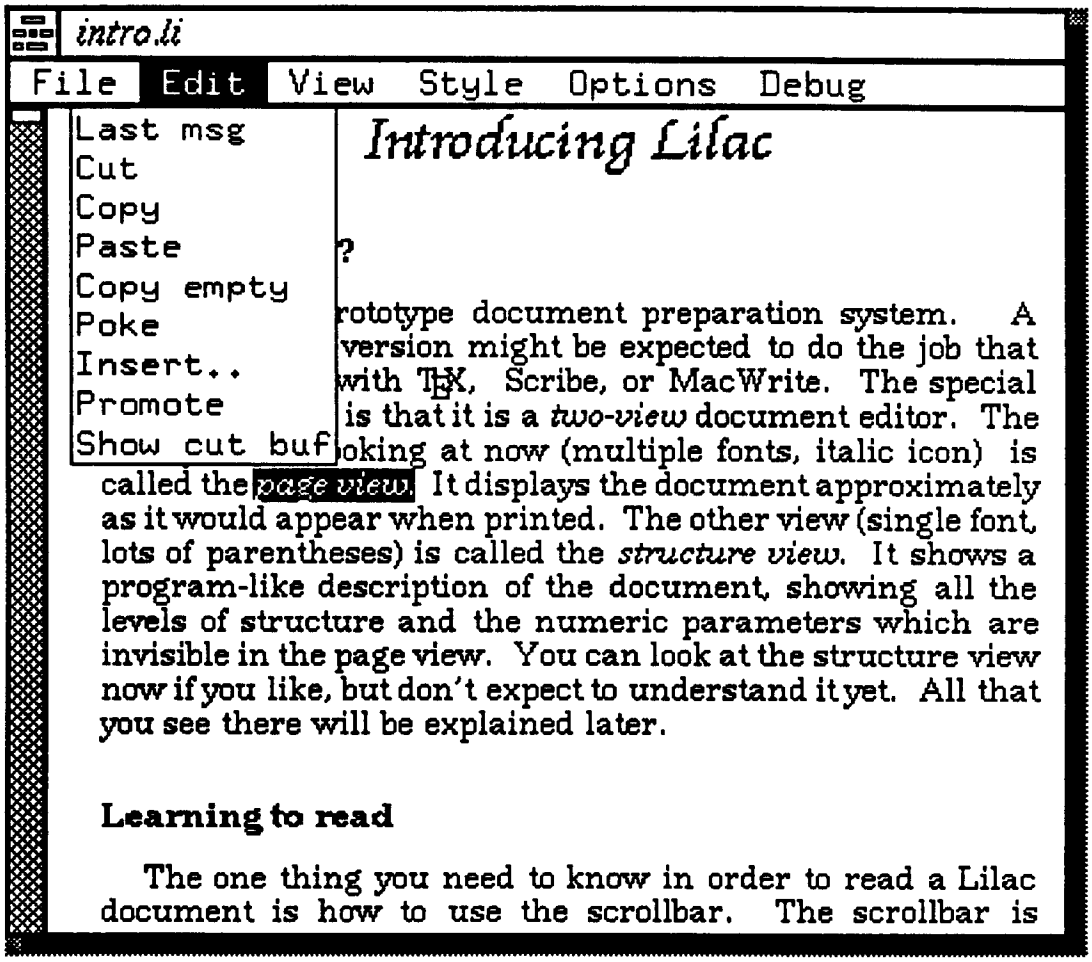


Fig. 5-1 The page view editor in action.

Building the WYSIWYG side of a two-view editor is an interesting challenge because of what it has to edit—not just text, nor a simple hierarchy of chapter, section, paragraph, word, and character, but a program, a hierarchy of arbitrary depth, built of user-defined units. So, as system designers, we cannot design commands appropriate to each type of unit, or to each level in the hierarchy. We must instead design a generic hierarchical editor, with commands that can apply usefully and sensibly to any type of unit at any level in the hierarchy.

What sort of commands are appropriate for editing a generic hierarchy? How do we give the user access to all levels, regardless of how many there are? How do we enable him to build and extend complicated structures conveniently? This chapter presents some design principles for an editor that solves these problems.

5.1. Expressions are structures

The most basic question in designing this editor is, what are the elements it talks about? "What can I select? What can I delete?" One major part of this problem has already been addressed: where, in a sea of subroutine calls many levels deep, do we place the responsibility for a particular piece of output? Answer: Edit instances, not definitions. Distinguish mainstream material from subroutines, and always assign responsibility to some part of the mainstream. This principle is embedded in the language design. Now we are left with a somewhat smaller space to work in: a mainstream expression with nested subexpressions.

To work in this space, we adopt a basic principle: *expressions are structures*. That is, the syntactic hierarchy presented by the document language will be used as the logical hierarchy upon which the editor is based. Every selection the user can make corresponds to some well-formed expression, or list of well-formed expressions, in the source program. Actually, this is not quite the whole truth, but close. Exceptions will be explained in due course.

This principle helps considerably in keeping connection between the two views. It means that for every selection made in the page view, we can highlight some syntactically reasonable collection of characters in the source view to reflect that selection.

5.2. Structural lists

The second principle is also tied into the language design. Question: how do we support insertion and deletion (common and indispensable editing actions) in the context of a program, while keeping the program syntactically valid? The answer lies in the concept of a structural list. The Lilac language is full of lists: horizontal lists, vertical lists, lists of boxes being built into other boxes being joined into other lists of boxes, etc. Lists are made accessible in the syntax through the use of repeatable parameters. A function parameter of a list type (Hlist or Vlist) can be declared repeatable, allowing it to match any number of actual arguments. In each function call these arguments form a syntactic list, a *structural list*, within which insertions and deletions are always syntactically correct.

A structural list has a conceptual, though not exact, correspondence to the geometric list which it represents. Not exact, because one argument in the structural list may yield several geometric elements. But close enough to be a significant part of the look and feel of the editor. Based upon the type of the repeatable parameter, each structural list is either a *vertical structural list* (type Vlist) or a *horizontal structural list* (type Hlist).

Structural lists form the basis for much of the editing in Lilac. We begin with the Rule of Selection: a selection may be any single expression, or any group of consecutive members of one structural list. An insertion point, viewed as a selection of zero length, may sit between any two elements of a structural list (or at the beginning, or at the end).

Any expression that is part of a structural list is called a *list element.* An expression which is not part of a structural list, but matches a non-repeatable parameter, is called a *fixed element.*

5.3. Strong typing

The Lilac editor has a type system underneath it. Almost every object in the document sits in some "slot," in that it is an argument which matches some formal parameter. Each slot has a type, the type of that formal parameter. Any object that sits in a slot must be compatible with its type. There are five tangible types (types of things that can be seen and edited), three scalar types and two list types:

Scalar types:	
Box	a box
Hglue	an item of horizontal glue
Vglue	an item of vertical glue
List types:	
Hlist	a horizontal list
Vlist	a vertical list

Scalar type slots require an object of exactly matching type. List type slots are more lenient: an Hlist slot can take Hlist, Hglue, or Box, and a Vlist slot can take Vlist, Vglue, or Box.

The consequence of all this is that Paste can fail due to type incompatibility. For example, a character string is an Hlist. A paragraph is a Vlist (and must sit in a Vlist slot). So an attempt to replace a paragraph with a character string will fail. This actually makes good sense, because a character string doesn't contain the information to tell how to relate it to its surroundings. It might be broken into lines to make a paragraph; it might be centered; it might be left- or right-justified. But some such operator must be interposed, to fit a character string into a Vlist.

The impact of strong typing is softened by the fact that the type Box is compatible with every slot. The primitive operators Hbox and Vbox can be used to convert any horizontal or vertical list material into a box in the simplest possible fashion, with no stretching or shrinking of glue. In using one of these the user recognizes that he is

doing a conversion by the simplest of many possible means. Perhaps the editor should interpose these automatically when asked to do an incompatible insertion. But I don't think so. Normally one wants to use a user-defined conversion function, designed to fit into the spacing scheme of the document, rather than using a bare, unpadded Hbox or Vbox.

5.4. Character strings

In our tree of unknown depth, made up of units of unknown type, one thing is certain: most of the leaves will be character strings. This is the one known type of unit, and since it is overwhelmingly common, we depart from generality to give it some special treatment. First of all, every character string is viewed as a structural list, with the characters as elements. But beyond that, wherever a character string is itself a member of a horizontal structural list, it is not viewed as a distinct component of that list, but rather each character in it is viewed as a component of that list. In other words, the string as a level in the hierarchy evaporates, leaving only characters. The consequence of this is that non-character elements, such as a logo like "TEX", can be inserted in the middle of a string without breaking up the unity of that string. One can select the special element together with some characters before and after it, just as if it were a character itself.

From a structure-view point of view, a consequence of this policy is that quote marks are very ephemeral things, easily created and destroyed. A non-character element is inserted into a string—snap! The string breaks into two strings, and the new element lands between them, suitably set off with quote marks and commas. Deleting the special element will reverse the process.

Before:	`"the special | logo"`
After:	`"the special ", tex, " logo"`

This convenience creates one bit of confusion. Not all character strings are members of horizontal structural lists. When the user tries to insert a special element into a string that is not, the insertion will fail, because the string cannot be split. This leaves the user probably confused and certainly frustrated. A solution under consideration is, in such cases, to surround the string with the no-op operator Hlist, which takes a repeatable parameter. Now it is a member of a structural list, and can be split. The only unfortunate thing is that we have added a level of hierarchy behind the user's back—possibly causing minor confusion later on.

5.5. Selecting in a hierarchy

Lilac uses the select-cut-paste model of editing, because it has served well in many applications where objects of several different types are being manipulated. Not so well explored, however, is the issue of selecting in a many-leveled hierarchy. The basic problem is, "How does it know what I'm pointing at?" The user points at a character; he is also pointing at many other things. A word. An entry in a table, perhaps. A row of the table. The whole table. A section. A chapter. This is the problem of *scope*, which arises in many forms of interactive editing: What is the scope of the indicated selection? How big a thing does the user mean to be pointing at?

I have approached this problem with the following design principles:

- Selection begins at the lowest level of hierarchy
- Higher levels are reached by *promoting* an existing selection
- At any given level, more material is included by *extending* an existing selection
- All these operations must be very quickly and conveniently available to the user to make a smoothly functioning editor

In close correspondence to these are the four basic selection operations of Lilac: Point, Select, Extend, and Promote. Select and Point are the most basic; they operate at the lowest level in the hierarchy. Select selects the smallest element under the mouse pointer—usually a single character. Point is similar, but it places an insertion point on one side of that element, whichever is closer. Extend is for selecting multiple elements at the same level: given an existing selection, and a new mouse position which points to an element within the same structural list, it will yield a new selection that includes the original selection, the new element, and any that lie in between. Promote is for reaching higher levels of the hierarchy: it selects the structural parent of the current selection. In the interest of convenient access, these are all bound to mouse buttons on a three-button mouse. Promote is invoked by multiple rapid clicks on the Select button.

Select

The Select operation embodies the first principle: select the smallest element that makes sense. To be more precise, it selects the smallest element that can be traced to mainline material in the document program. In most cases, this will be a single character. For an unusual case, take our friend, the TEX logo. The "T", the "E", and the "X" are specified in the "tex" function that generates this logo; they do not appear in the mainline program. So we do not allow them to be selected. Responsibility is passed on to the call to the function that generated them. So a Select on any of these characters results in the selection of the call to "tex"; the whole logo is selected.

```
use doc.li

defaults
  rmargin = 300

function tex =
  Hbox("T", Hskip(-3),
    Raise(-3, Hbox("E")),
    Hskip(-3), "X")

unit main =
  Vlist(
    Chapter("A Chapter Title",
      Section("A Section Title",
        Para(
          "This sample document
demonstrates the structure of
Lilac documents."),
        Para(
          "This paragraph shows",
          Italic("italic"),
          " and ",
          Bold("boldface"),
          "text and the fancy ",
          tex, "logo."))))
```

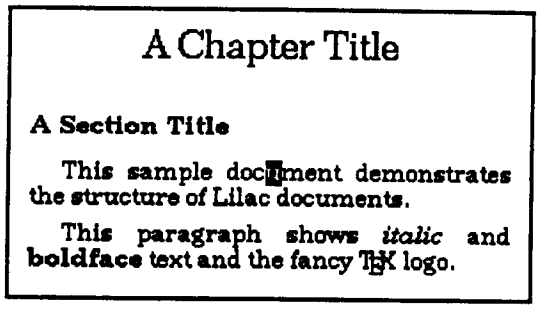


Fig. 5-2 A character selection, shown in two views

```
demonstrates the structure of
Lilac documents."),
        Para(
          "This paragraph shows",
          Italic("italic"),
          " and ",
          Bold("boldface"),
          "text and the fancy ",
          tex, "logo."))))
```

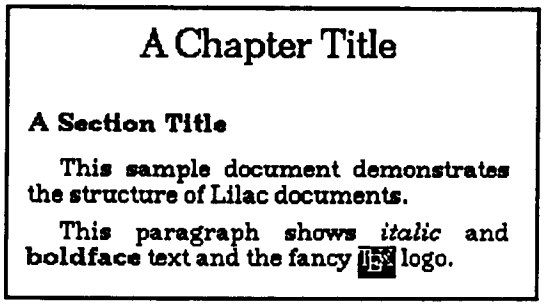


Fig. 5-3 Selecting the TEX logo

Selections, at whatever level, are shown to the user by highlighting rectangles in inverse video. Any boxes and glue items that arise from the expression selected are searched out, and each is highlighted. In case of character selections that do not begin and end on word boundaries, partial boxes are highlighted, so as to highlight only the selected characters.

Figures 5-2 and 5-3 show some simple selections, illustrated in two views. The underline in the source view corresponds to the highlight in the page view; it shows how Lilac is thinking of the selection. In 5.2 and subsequent figures, the whole source view is not shown, but only enough to give a clear context. (Page views are shown somewhat reduced, in order to fit neatly on the page.)

Note: these two-view figures are a reasonably accurate picture of what is seen in the two windows of an open document—with scrollbars, menus, and such stripped away. The underlines seen in the left-hand view (the source view) are reflected selections. These selections are shown in this fashion, both here and in the actual editor, to indicate that they are not the true focus of editing but only reflections.

Promote

The Promote command raises a selection to the next higher level in the hierarchy. The lowest three levels are not really syntactic hierarchy, but are specialized for convenient editing of character strings: character, word, whole string. Any promotion beyond that is based on the syntactic hierarchy of the source: promoting from a selected expression selects its parent expression. Any selection may be promoted repeatedly, selecting ever larger units, until the entire document is selected.

Critical to the user interface is the fact that promotion is very quickly accessible. In Lilac it is available by multi-clicking the Select button on the mouse. That is, the first click does an ordinary Select; subsequent clicks, in the same place within a sufficiently short time, invoke successive Promotes. This enables the user to select something several levels up from a character in a fraction of a second. Promote is also bound to an ordinary key, for those who might be confused by so much fast clicking.

Figure 5-4 shows five levels of a Select/Promote sequence. The entire sequence might take about one second for an experienced user. The mouse is pointing to the "u" in "document" in the page view, as the user clicks the Select button five times.

Promotion is one of those aspects in which Lilac differs most from standard WYSIWYG editors. Those systems support selection of sequences of characters and words, and the best of them offer convenient ways to select whole lines and paragraphs.

But no facility is provided to select objects at other levels, because these editors have no concept of other levels of structure.

Point

The Point operation is closely parallel to Select, but places an insertion point instead of selecting something. There is one difference: not everything that can be selected can have an insertion point next to it. Expressions which are not members of a structural list cannot. If Point is applied to such an object, it will automatically promote its level of focus until it finds an ancestor which is a member of a structural list, and put the insertion point before or after that ancestor.

Like Select, Point has an associated promotion operator, invoked by multi-clicking. It looks to the parent of the structural list containing the current insertion point, and tries to place an insertion point at that level. In case the parent is not part of a structural list, it looks on up the ancestry until it finds an ancestor that is. Unlike Promote, this operator has a choice to make: put the insertion point before the parent, or put it after? It is sometimes geometrically difficult and not always appropriate to make this decision based on the position of the mouse. So we provide two promotion operations, UpForward and UpBackward. UpForward is invoked by multiple clicks on the Point button; UpBackward is similarly invoked, but with the Shift key held down.

Higher-level insertion points raise an interesting question. When an insertion point sits between two paragraphs, how should it appear? Most document editors would show it at the end of one paragraph or the beginning of the next. But that really isn't appropriate here. This is not a character-level insertion point, and it would not accept typed-in characters. Character strings are horizontal material, but this insertion point sits in a vertical structural list!

Since it already has the distinction of horizontal and vertical structural lists, Lilac can provide a user interface to match. An ordinary insertion point sits in a horizontal structural list, and it appears as a vertical blinking bar, typically between two characters. An insertion point in a vertical structural list is called a *vertical insertion point*, and it appears as a horizontal blinking bar, between two vertical geometric elements.

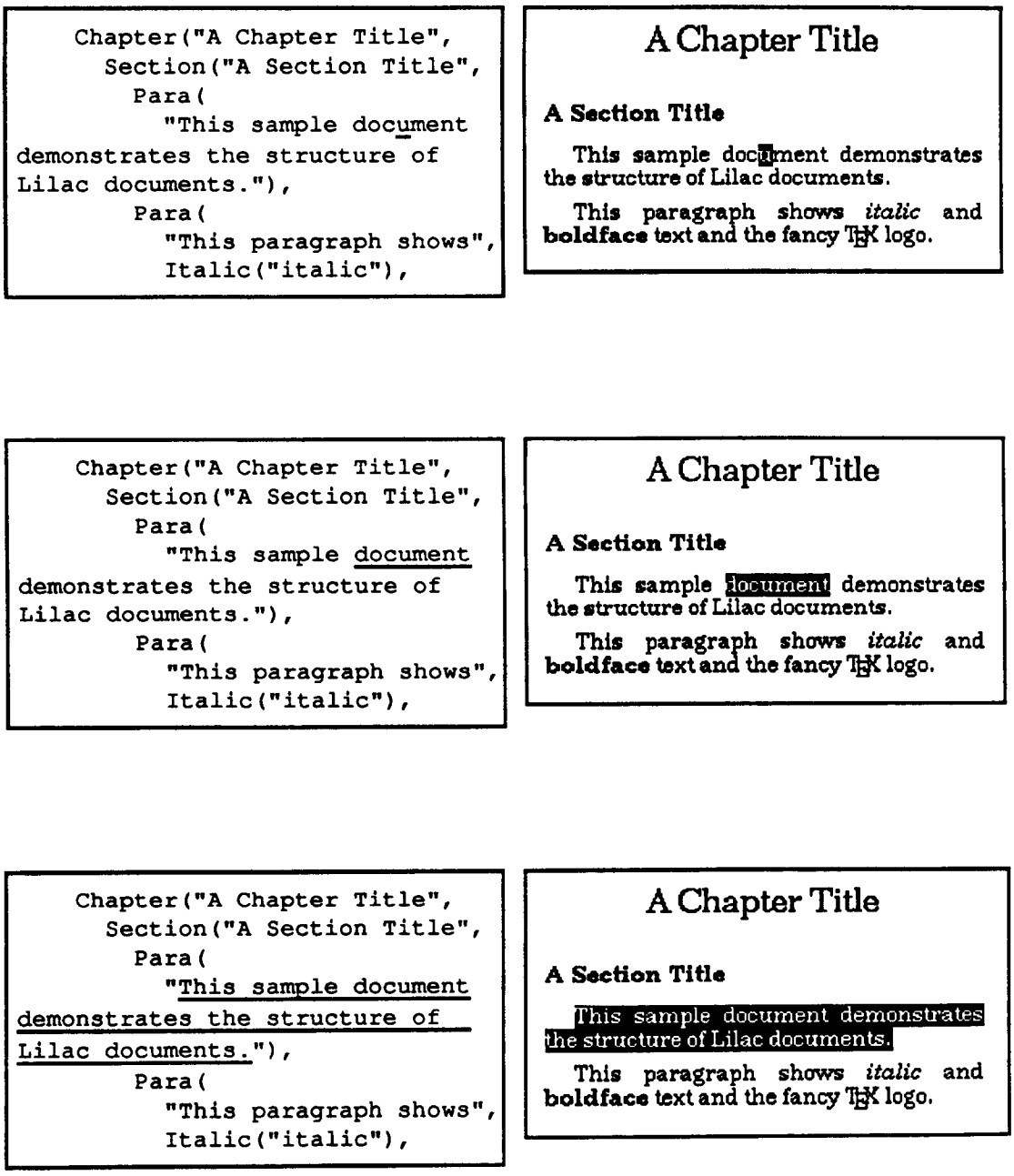
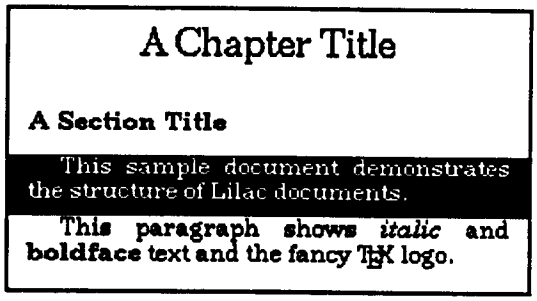


Fig. 5-4 A sequence of promotions

```
Chapter("A Chapter Title",
  Section("A Section Title",
    Para(
      "This sample document
demonstrates the structure of
Lilac documents."),
    Para(
      "This paragraph shows",
      Italic("italic"),
```




```
Chapter("A Chapter Title",
  Section("A Section Title",
    Para(
      "This sample document
demonstrates the structure of
Lilac documents."),
    Para(
      "This paragraph shows",
      Italic("italic"),
      " and ",
      Bold("boldface"),
      "text and the fancy ",
      tex, "logo.")))
```

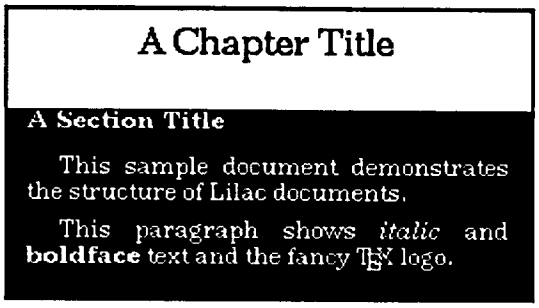


Fig. 5-4 (continued)

Figures 5-5 and 5-6 show examples of both kinds of insertion points. Note where the vertical insertion point of Fig. 5-6 is reflected in the source view—just before the second "Para" call. These two form a Point/UpForward sequence—the first click on the Point button gave Fig. 5-5, the second gave Fig. 5-6. A third would produce a Section-level insertion point, which would appear as a horizontal bar below the second paragraph.

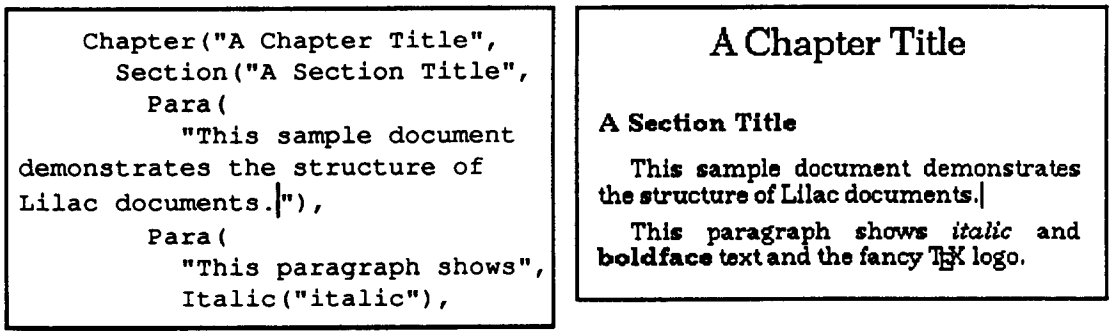


Fig. 5-5 A horizontal insertion point

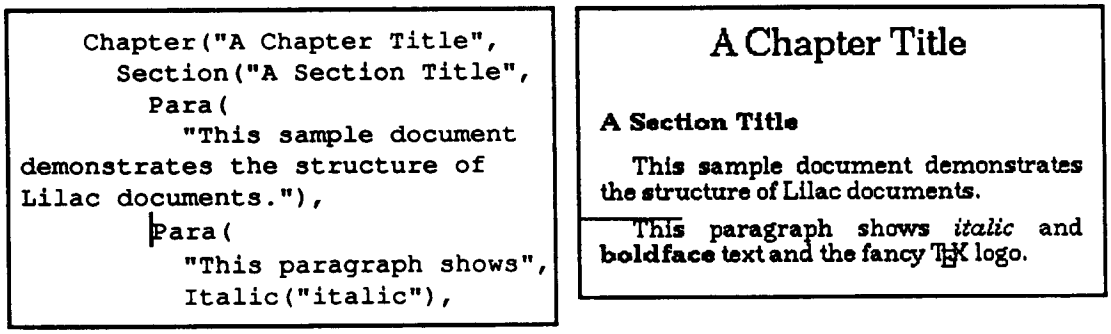


Fig. 5-6 A vertical insertion point

Extend

The Extend command extends a selection to include more elements at the same level. It is invoked with a selection or insertion point already in place. That marks one end of the extended selection; the mouse position marks the other. The new selection includes the old selection, the element pointed to, and everything in between. The interpretation of the mouse position depends on the level of the old selection. If the old selection is one or more characters, the mouse will be viewed as pointing to a character. But if it is a paragraph, the mouse will be viewed as pointing to a paragraph, even though it is also pointing to a character. A good way to look at it is that the new endpoint is implicitly promoted to the level of the old selection.

Extension fails if the mouse position is outside the structural list containing the original selection, because the rule of selection only allows a multiple-object selection within one list. Failure is obvious because the highlight does not extend, and easily correctable by moving the mouse while the Extend button is still held down.

An example: in the fourth stage of Fig. 5-4, the first paragraph is selected. At this point, a right-click anywhere in the second paragraph will cause it to be selected as well. Regardless of what exactly the mouse is pointing to, the paragraph is the ancestor at the level of the original selection, so it is selected. An example of failure: in the first, second, and third stages of Fig. 5-4, the selection is inside the character string. In this case a right-click in the second paragraph will fail, because neither the characters nor the paragraph itself are brothers of the original selection. Only other characters within the first paragraph are brothers of the original selection.

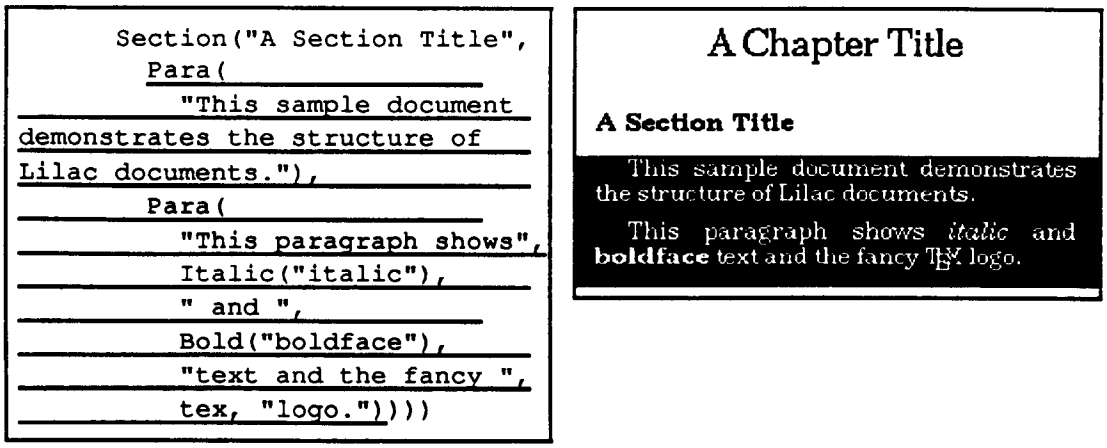


Fig. 5-7 An extended selection

Failure of extension can be frustrating. "But I really wanted to select them both!" This frustration is alleviated (quite effectively, in practice) by the PromoteExtend command, which is invoked by double-clicking the Extend button. It means, "Promote the existing selection until it can be extended to the indicated point, and then extend it." For example, if a character in the first paragraph is selected, and the mouse is pointing to a character in the second, Extend will fail. PromoteExtend will select both paragraphs, as in Figure 5-7. For another example, if the mouse is pointing into the section title, PromoteExtend will select the entire section. In this case there has been promotion but no extension, because the title is not part of any structural list. The lowest structural level that could unite it with the first paragraph sufficed to select them both, without any extension.

Summary of Selection

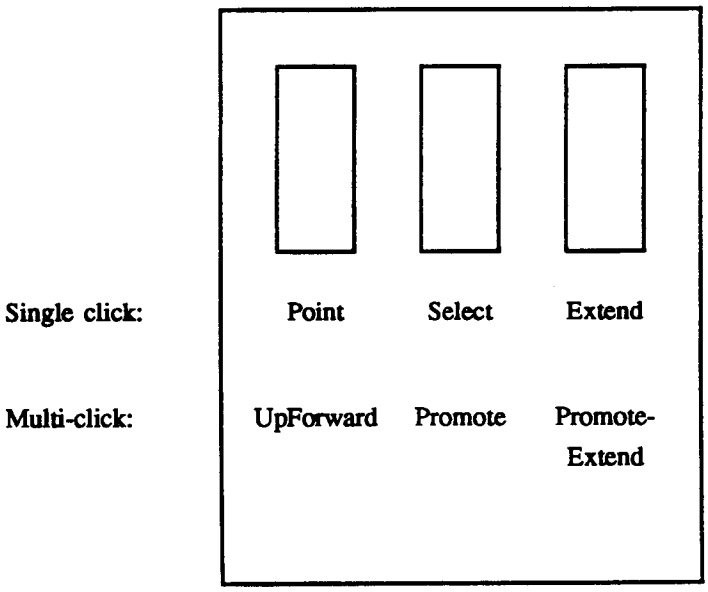


Fig. 5-8 The conceptual Lilac mouse

The user interface for selection is summarized by Fig. 5-8, a diagram of the mouse buttons. The three buttons are assigned to the Point, Select, and Extend commands. Each of these commands features continuous tracking feedback: while the button is held down, a highlight follows the mouse on the screen, showing what would be selected if it were released at that moment. Except in some large Extend cases, Lilac has no trouble doing this in real time. Each button also has a second function, invoked by subsequent clicks in a rapid sequence; all the multi-click functions are related to reaching higher levels in the hierarchy.

5.6. Creating new material

The cut-and-paste model is a good method for editing existing structures, and a very bad method for creating new ones. Consider the steps required to create a new paragraph, given that an existing one can be found:

Select the old paragraph
Copy
Put a vertical insertion point where the new paragraph should go
Paste
Select the characters in the new copy
Cut

And now, at last, a new, empty paragraph is ready to receive some new content! Faced with this task, users would soon resort to keeping empty templates around, just to get rid of the last two steps of the process. We need a better way to get new, empty structures.

Many editors solve this problem by providing a set of "New X" commands, for various kinds of X. Probably some of these are desirable in Lilac, but they will never solve the problem. The set of kinds of things is infinite, and user-extensible. An automatically generated menu of available functions could be provided, but it would usually be very long and cluttered with functions of little interest to the user. For very commonly wanted elements, I believe even a menu would be too slow for satisfaction.

Lilac's solution is to use the concept of *empty replication*. The idea is, given an existing object, to make a replica that has the same structure but none of the content. The content fields are all empty, waiting to be filled. The empty replication algorithm is recursive, as follows:

1. Copy the outermost level of the expression.
2. In place of any character string in the original, put an empty string in the copy.
3. In place of any structural list, put an empty replica of the first element of the list.
4. In place of any fixed element, put an empty replica of that element.
5. Copy numeric and other intangible arguments intact.

The one part of this that might be surprising is that we replicate the first, and only the first, element of a structural list. But we certainly need at least one element. To produce a zero-element structural list is to lose structure: replicating a section would produce a new section, but would lose the fact that a section consists of paragraphs—and quite possibly, paragraphs of some user-defined flavor. We would get a section with a title field and nothing else—most unhelpful.

Replicating only one element is an imperfect solution. It does the right thing whenever the structural list is homogeneous, which is frequently the case. If a chapter consists of sections and a section consists of paragraphs, then an empty replica of a chapter will be a chapter with one section which contains one empty paragraph, and that will be a good basis on which to build a new chapter. By contrast, a structural list that consists of two types of subunits in alternation will not be well handled. But I believe that inhomogeneous structural lists are rare enough that the copy-paste-cut-the-contents sequence is a reasonable way to deal with them.

Now we can describe the two commands that really make the machine run smoothly: Return and Tab. They are named for the keys that invoke them, because no simple name describes them. Return operates as follows:

1. Find the nearest ancestor of the current selection or insertion point that is a member of a vertical structural list.
2. Make an empty replica of it.
3. Insert that replica immediately after the original.
4. Place an insertion point in the first text field (as found by depth-first search) in the replica.

This unlikely-sounding chain of events is actually a generalization of the function that most editors attach to the Return key—it makes the Return key do what a Return key ought to do, and in a way that interacts well with an arbitrary hierarchical structure. It creates a new-vertical-whatever-you-were-working-in and puts you at the beginning of it, ready to type.

For example, if the user was typing in a paragraph, it does exactly what a Return key should do—starts a new paragraph below the first. If he was typing in a centered line, it starts a new centered line below the first. If he was typing in a specially offset paragraph with indented margins and a special font, it starts a new paragraph with all those properties.

For a more interesting example, consider a section consisting of paragraphs. Since a paragraph is a vertical element (a Vlist, to be precise), a section is a second-level vertical element. Pressing Return while typing in a paragraph will give a new paragraph, not a new section, because the paragraph is the lowest-level ancestor which is a vertical element. But a new section is easy to get. If the user first invokes UpForward (which is available by keyboard as well as by mouse) he gets a vertical insertion point following his paragraph. This insertion point sits between paragraphs, on a par with them; its immediate parent is the section. So pressing Return at this point gives a new section. New-paragraph is one keystroke, new-section is two.

Many structures, more complex than a paragraph, are based on functions taking more than one argument. In these cases, when filling in an empty replica it is handy to be able to move from argument to argument. The Tab command serves this purpose. Its function might be described as "go to the next end-of-character-string." That is, from the position of the beginning of the current selection, it searches forward for the first character string, and puts an insertion point at the end of it. Or if the selection is currently in a character string, and not already at the end of it, it puts an insertion point

at the end of the string. Repeated use of Tab will visit every string from the current selection to the end of the document. This is useful for hopping from field to field, and it is vital for reaching empty strings, which may not be reachable by mouse because they take up no space on the screen.

For an example of its usefulness, consider the table of Fig. 5-9. The Lilac code that generates this table is shown below it.

Symbol	Name	Atomic no.	Found in
H	Hydrogen	1	water
C	Carbon	6	diamonds

```
function Hbar = BlackBox(70 + 100 + 70 + 150 + 1, 1, 0)
    a horizontal bar, separating rows of the table

function Vbar = Hlist(BlackBox(1, 14, 4), Hskip(4))
    a vertical bar, separating fields within a row

function ETRow(field1: @Hlist, field2: Hlist, field3: Hlist,
    field4: Hlist) =    one row of the table
  Vlist(
    Hbox(HboxTo(70, Vbar, Bold(field1), hfil),
      HboxTo(100, Vbar, field2, hfil),
      HboxTo(70, Vbar, field3, hfil),
      HboxTo(150, Vbar, field4, hfil),
      Vbar),
    Hbar)       each row makes the horizontal bar below it

function ElementTable(rows*: Vlist) =    template for the whole table
  Center(Vbox(HBar, rows))          the table makes the first horizontal bar

unit TheTable =    the actual table, with contents
  ElementTable(
    ETRow("Symbol", "Name", "Atomic no.", "Found in"),
    ETRow("H", "Hydrogen", "1", "water"),
    ETRow("C", "Carbon", "6", "diamonds"))
```

Fig. 5-9 A ruled table in Lilac

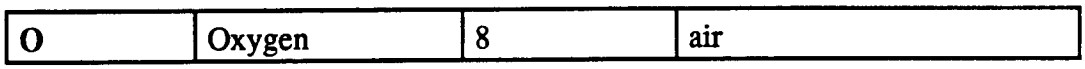
This is a good tough example, because here we are dealing with a totally user-defined structure, far removed from the structure of ordinary text. Instead of a section consisting of paragraphs each consisting of some horizontal stuff, we have an Element-Table consisting of ETRows, each consisting of *exactly four* fields.

Now let's put an insertion point at "diamonds" and press Return. The string "diamonds" is a fixed element, not a list element, so it is not eligible for replication. We look up a level. The ETRow is a vertical list element, under Vlist, so it is eligible. We replicate it, and put an empty string in place of each of the four character strings. A new empty row appears below the Carbon row, with insertion point in the first field:

With a quick sequence of keystrokes we can fill it:

O <Tab> Oxygen <Tab> 8 <Tab> air

resulting in:



And thus the table, unusual structure though it is, can be extended indefinitely by ordinary typing. And so can all structures that are composed of fixed elements and homogeneous lists.

5.7. From the whole cloth

Return and Tab form an excellent system for extending structures. But how did the first ElementTable and the first ETRow get to be there? Well, of course, they could be added by source view editing. The function definitions had to be created that way, anyway. Yes, but soon those function definitions may be buried in some style file, and the user wants to use them in a new document. Resorting to source view editing for such things soon becomes tiresome, especially if the document is large and takes a long time to reparse. The chances are good that the source view is not even open on the screen, since so much can be accomplished without it.

A smoother solution is found through the use of *escape commands*. These are commands, issued from the page view editor, that pop up a special input field and allow the user to type in something in the source view language. A good example is Insert. It invites the user to type an expression of the document language. When he is finished, the expression is parsed, and the resulting element is inserted at the current selection. It works exactly like Paste, except that the thing to be pasted does not come from a previous selection—instead, the user specifies it directly. By this means he can insert absolutely anything.

This method wins over source view editing in a great many cases because it preserves the spatial orientation of the page view. The user types a source-language

expression, but he inserts it at a page view insertion point. It is generally much easier to see where you are and be sure you are right when looking at the page view.

5.8. Paging and scrolling

Pages are an interesting issue in an interactive editor. The principle of WYSIWYG would suggest that we show exact replicas of printed pages: compute page breaks, and show the user a page on the screen. But pages are no part of the natural structure of most documents; they are an artifact of the limitations of paper. Paper (as used in modern times) has this unfortunate property: it comes in separate units, requiring a page turn and possibly a lost train of thought to get from one to the next. The computer can offer better: a seamless vertical scroll in which no artifical edges divide the text. To read beyond the visible portion, one need only slide the text up the screen, suffering no loss of visual contact. Any line may be the top of the visible region; any line may be brought to the center of the window to be seen in context.

Lilac offers the user both choices. He can see the document with its exact page breaks, page numbers, etc. (Page Mode), or he can see it as through a viewport onto the vertical scroll (Scroll Mode). Page Mode is the true "what you see is what you get" view, and it allows editing, but not with the best performance. Scroll Mode is the more common mode of operation, and it is the mode optimized for the fastest possible handling of typed input.

In either mode, vertical movement is controlled by a scroll bar at the left of the window. The scroll bar supports three operations, accessed by mouse buttons: Scroll, Page, and Thumb. Scroll slides the viewport up or down by one line—or, equivalently, moves the scroll by one line relative to the viewport. Similarly, Page moves it up or down by one screenful. Thumb takes the viewport directly to an indicated position in the document. The position chosen is proportional to the height at which the user clicked in the scroll bar: a click at the top goes to the top of the document; a click in the middle goes to the middle, etc. Scroll and Page both auto-repeat while the mouse button is held down. In Page Mode all scroll bar operations round off to page boundaries: Page and Scroll both have the effect of moving by one page, and Thumb finds the page nearest to the indicated point.

5.9. Minor topics

Style modifiers

Every document editor needs style modifiers—bold, italic, smaller font size, etc. Lilac models these as functions that take a single, generic argument. Chapter 3 showed

the Italic function as an example. Style modifying commands do their work by "clothing" the selection with a call to a style modifier function. From a parse tree point of view, they insert a node above the selected node, deepening the tree by one level. If the current selection is a partial string, the string is split to make the selection into a separate node, which is then styled as usual. If the current selection is only an insertion point, a style modifier call is inserted there, surrounding an empty string—and subsequent typing will fill in that string. When the styled text has been typed, an UpForward will pop the insertion point out of the style field, back to the level of the surrounding text.

A few common styles, like Bold and Italic, are provided as menu commands and as control-key commands. There is also a "Style" escape command. It prompts the user for a special "style template," an expression which includes one placeholder, a dollar sign. It then replaces the placeholder in the template with the current selection, and replaces the current selection in the document with the filled-in template. This is actually quite a versatile command, useful for quite a few things besides simple style changes.

Changing fixed elements

Fixed elements present a little challenge to the cut-and-paste editing model. They cannot be deleted! To do so would create a syntactically invalid document program. So we allow them to be changed with the Paste and Insert commands. Paste in general replaces an existing selection with the new material being pasted. We also allow it to replace a fixed element, provided the new material consists of exactly one element of a compatible type. Thus the traditional replacing property of Paste takes on new significance: where insertion and deletion are impossible, editing by direct replacement is still allowed.

New documents

Where do new documents come from? "Well, just give the New command, get an empty one, and start typing." But an empty what? A truly empty Lilac document would be an empty source view, which cannot even support a page view because it has no main unit. This is awfully bare rock to start from. So the most common form of the New command, always issued from an existing Lilac view, makes a sort of "empty clone" of the existing document. It copies all the use directives and all the variable-default declarations into the new document. Then it looks in the old document's

environment for a special unit named "newdoc" (usually provided by a style file). It copies this unit to the new document and renames it "main." And now we have a new document which has some context and some skeletal structure and will support a page view. We put an insertion point in the first text field, and the user is ready to start typing!

Unit "newdoc" not only makes a page view possible, but it provides a basis of structure appropriate to the document type. A very ordinary one might look like:

```
unit newdoc =
  Vlist(
    Chapter("",
      Section("",
        Para(""))))
```

This initial skeleton contains the information that that document is made of chapters, chapters are made of sections, and sections are made of paragraphs. Together with Return and Tab, it is sufficient to allow a user to type a whole document full of chapters, sections, and paragraphs without ever leaving the keyboard, much less looking at the source view. This is extremely desirable for supporting naive users with simple needs. Such users would generally be frightened of a source view, and they have no real need for one. Once a wizard has done the work to set up the structure for a document type, simple users can ride on top of that structure and scarcely even think about it.

5.10. An experiment that failed

As the design of the Lilac editor began to come together, I was concerned that the Rule of Selection (see Section 5.2) might be too restrictive. In plain text editors I am accustomed to being able to select from any character to any other character in the file, and delete the span. Occasionally I use this ability in ways that cut across the natural structure of the text—for example, deleting from the middle of one paragraph to the middle of the next. I feared that without this freedom, Lilac might prove to be a strait jacket.

So as an experiment, I provided an escape hatch to allow this freedom, to select from any character to any other character in the document. The mechanism that allows characters in a string to act like members of the surrounding horizontal structural list is actually powerful enough to support this greater freedom, so the modification was not difficult. A special form of Extend, invoked by holding down Shift while doing an

Extend, sets aside the Rule of Selection. The result is called an *incoherent selection*, because it cannot be described in terms of a single element or a single list of elements.

What can you do with a selection that cuts across structural lines? What operations apply? What do they do to the underlying structure? I came up with some experimental definitions:

Cut

Cut is defined to be a Copy followed by a Delete (an operation which is not directly available, for reasons of safety). Copy is described below. The basic principle of Delete is, delete as much as possible while allowing what remains to still make sense. The rule is: for any list element that is selected, delete it. For any fixed element that is selected, empty it. For any node that is partially selected, apply this rule recursively. "Empty" is defined similarly to the "empty replication" process, except that it leaves a structural list with zero elements, not one. An incoherent Cut does not leave an insertion point, because there is no obvious place to put it.

Copy

Copy what is selected, together with any surrounding superstructure that is required to hold it together. This is the flip side of the rule for Delete. The rule is: for any node that is selected, copy it. For any node that is partially selected, copy those parts that are selected, and those other parts that are structurally necessary because they are fixed elements. But the parts that are copied only because of structural necessity are copied "emptied".

Paste

Paste is not allowed in the presence of an incoherent selection. Note that there is no such thing as "incoherent contents of cut buffer". The Copy operation is defined in such a way that a coherent structure of nodes always results.

An interesting property of incoherent Cut is that it is not invertible. From one point of view, Delete deletes less than was selected, while Copy copies more than was selected. Delete leaves behind the structural backbone of incompletely selected nodes, and Copy copies it. A Cut followed by a Paste (after placing an insertion point to Paste at) is likely to result in two copies of the original structure, with the contents of that structure divided between them.

These facilities have been both implemented and tested. But the experimental result? In many weeks of real use of Lilac, I have never found myself wanting to use

them! The operations I really want to do almost always fit within the structural hierarchy. Of course, a definitive experiment requires a large collection of users, not just one. As Lilac becomes robust enough to attract a user community, I will be interested to see whether we still see the same result.

5.11. Experience

The editor described here works, and works nicely. I am using it, even now, to write this thesis. As of this writing, I have not yet succeeded in persuading anyone else to use it for real work, so I can only report upon a sample size of one.

I really enjoy having a user-definable structure, and I enjoy being able to edit it all with the same set of commands. Multi-clicking as a means of access to various levels in the structure works out very nicely—though occasionally I become confused as to where I am, especially in deep and unusual structures. I find the Return and Tab commands truly beautiful—in truth, they almost always do the right thing!

The major place where the user interface falls short is in inserting new types of objects. Technical documents are by no means homogeneous: besides running text they have enumerated lists, special descriptions (such as Cut, Copy, and Paste above), specially indented paragraphs, displayed equations, and many other interruptions to the normal flow. When it comes time to insert one of those, the user has only two choices: find an existing one and copy it, or use the Insert command.

These are sufficient, but they are less than smooth. A sufficiently common structure really deserves a short keyboard command to insert it. The truth of the matter is, this editor cries out for programmability. After a user has defined his structural elements, he should be able to define commands to create instances of them, and to put those commands into menus or bind them to keystrokes. This is discussed in more detail in Chapter 11.

6. The Mapping Problem

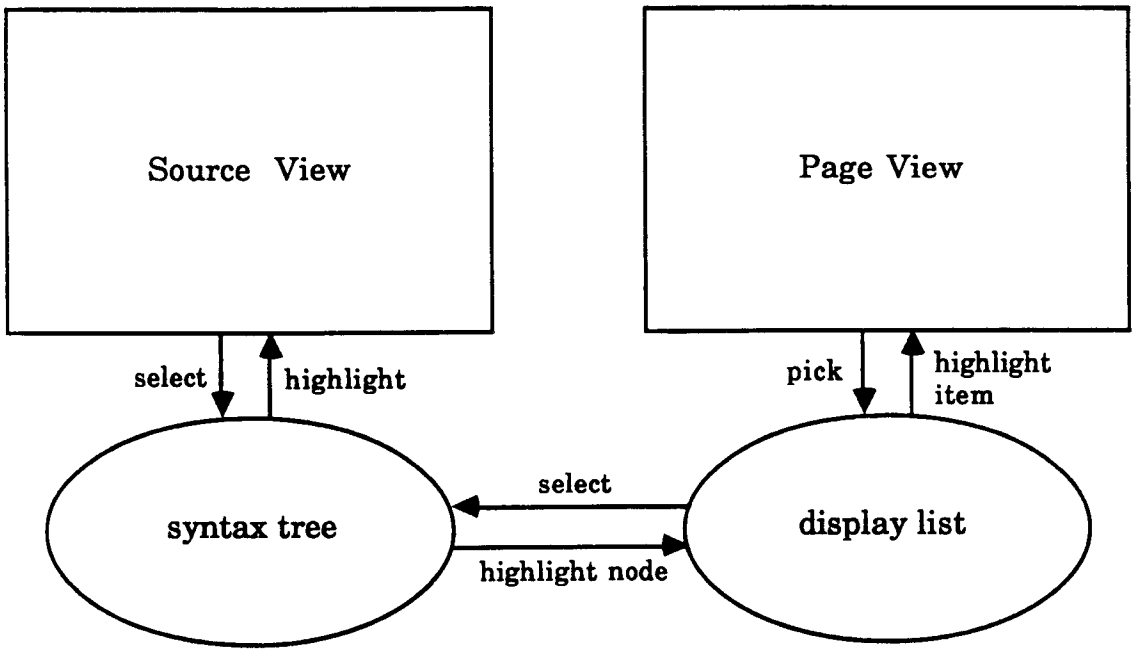


Fig. 6-1 Mapping pathways in Lilac

When a user clicks the mouse in the page view to make a selection, we have two tasks on our hands. First we need to attach a meaning to the click. Given a location on the screen, we must find some object at or near it which is eligible to be selected, and select it. Second, we must go back to the screen and find the region which belongs to that object, and highlight it.

The Rule of Selection (Section 5.2) means that selections are inherently tied to the syntax tree. To find a selectable object means to find a node or character in that tree. Because the structure of the syntax tree bears very little relation to the geometry of the image on the screen, it would be very difficult to solve either of these problems by computing directly on the syntax tree. So we use the display list as an intermediary between the two—a data structure which is geometrical enough to have a clear relation to the page view, but abstract enough to tie in conveniently to the syntax tree.

Now the selection problem is broken into four parts:

1. **Locate** an item (box or glue) at or near the mouse location
2. **Identify** the node (and perhaps the character) responsible for that box.
3. **Track down** all items that node is responsible for
4. **Highlight** the appropriate rectangles on the screen

Since we provide tracking feedback while the Select button is held down, these four steps must be fast enough to take place every time the mouse rolls.

Actually there is a fifth aspect to this task: mapping from a node in the syntax tree to characters in the source view. This is discussed in Chapter 9, which deals with all aspects of source view implementation.

6.1. The display list

The display list is based on the structure of boxes and glue, the elements used by the Lilac language to describe page geometry. In TEX, where the concept of boxes and glue originated, they can be short-lived: compute the boxes-and-glue description of a page, generate output, throw them away. In Lilac, where the output is an editable image, we keep them around permanently and attach all sorts of useful information to them to help with the mapping problem.

Boxes are joined together, usually with glue between them, and packed into larger boxes, called containers. A container is generally made just large enough to contain all the boxes in it. The page image is always represented by one big box which contains all the components of the page. To build a display list, we simply keep this structure in a tree: each container has a pointer to the list of items it contains, and each item has a pointer to the next item in its list. Each item also has a pointer to its container, for convenience in some mapping operations. Figures 6-2 and 6-3 illustrate a display list for the first verse of the poem "Jabberwocky". Fig. 6-2 shows the geometry, stretched a bit so that all box boundaries can be seen distinctly. Each word is a box; each line is a container containing its words; the page is a container containing all the lines. Spaces and leading are represented by glue, not visible here. Fig 6-3 shows the same display list as a tree structure. Here the glue items are visible, shown as circles.

One of the basic functions of the display list is to be a map that tells what is displayed in the page image and where. For this function it must represent the screen coordinates of each box displayed. But it must do this in a manner that can be quickly updated. The most obvious technique, to store the (x, y) coordinates of each box in that box's record, does not have this property. An insertion at the top of the page would change the y coordinate of every box on the screen, requiring a full traversal of the visible portion of the display list to correct them. Full traversals of the box tree are greatly to be avoided, as they must touch hundreds of items for a full page of text. A full traversal should occur only when a full screen update is required, e.g., when a new page of text is displayed.

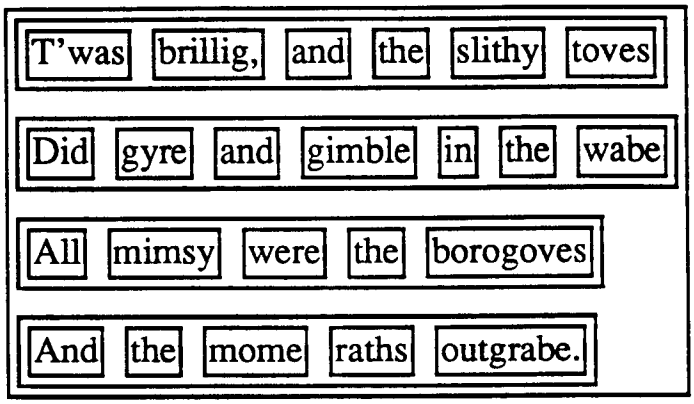


Fig 6-2 Display list geometry

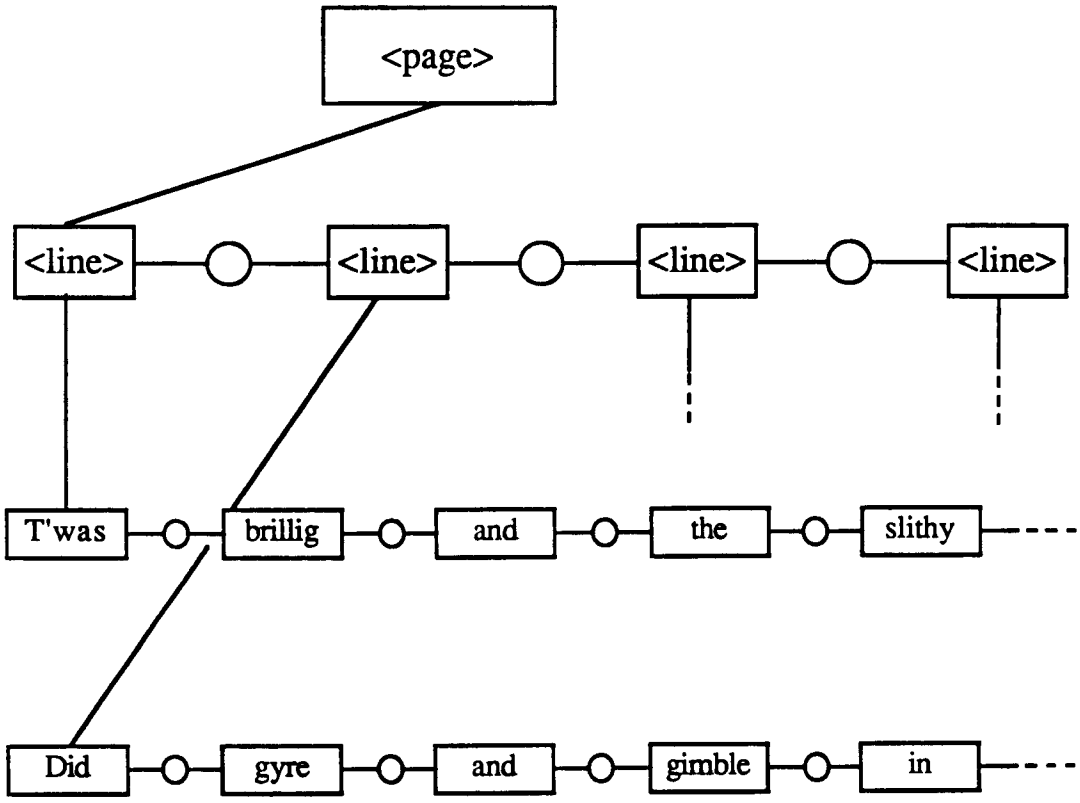


Fig 6-3 Display list data structure (circles represent glue)

My solution is to store only one coordinate with each box, an offset from the *basepoint* of the box's container to the basepoint of that box. The basepoint of a box is the point where its baseline intersects its left edge. We can get away with storing only one coordinate because of the nature of box composition: the components of an Hbox are aligned along their baselines, and the components of a Vbox are aligned along their left edges. So if we know the type of the container box, one coordinate of offset

suffices, because we know that the offset in the other coordinate is zero. The absolute coordinates of the basepoint of the top-level box are stored outside the tree.

The updating of stored coordinates is radically simplified. If an insertion occurs in a given list (or, from the tree point of view, a given set of brothers) then the offsets of all subsequent elements in that list must be updated—but not the offsets of their descendants, nor any offsets in any lists elsewhere in the tree. Instead of a full tree traversal we have a single list traversal, amounting to about 100 items in the worst case.

The pattern of working in only one coordinate at a time occurs again and again in the page view implementation. The following terms will make the discussion clearer: when a container box and one or more of its components are under discussion, the direction along which the components are laid out is called the *principal direction*—horizontal for an Hbox, vertical for a Vbox. The direction perpendicular to this direction is called the *transverse direction*. Thus we can say that each box stores its offset in the principal direction from the basepoint of its container.

For many computations it is important to know the extent of a box in its principal direction (i.e., the principal direction defined by the type of its container). This can be deduced from the dimensions of the box, which we know, and a knowledge of the principal direction. But to speed various computations we store this knowledge in a more palatable form, as two extra numbers. The **lo** field stores the offset from the basepoint of the container to the edge of the box which is nearest the top or left edge of the screen. The **hi** field stores the offset from the basepoint of the container to the other edge. These will come up again and again in the mapping and updating algorithms.

6.2. Locating an item

Given a mouse click, our first task is to discover which item the user is pointing to. This is accomplished with a recursive descent down the tree of boxes. We start out by knowing that the click location is inside the top-level box describing the page. We now want to discover which element of that box it is in or nearest to. In the common case, we are looking for a line of text. So we scan along the list of elements of the page, until we find one for which

$$lo <= click.y < hi$$

Upon finding one, we now proceed recursively, and set out to find the indicated element of that box. If this is a line of text, the test will be horizontal instead of vertical; other than that, no difference. Sooner or later we will come to a box which represents a word. A special algorithm locates the appropriate character within the word, and we are done.

The simple description above is true, provided that the mouse is inside a bottom-level item. Sometimes it is not. As Fig. 6-2 shows, some boxes, such as short lines, do not fill the full transverse extent of their container. So, by the algorithm above, a location off the right end of the third line would not hit any word, but only the line itself. But in practice, this leads to a confusing user interface. So we define the Rule of Location: *Always find a bottom-level item. Never find a container other than an empty container.* And we implement it with a proximity algorithm: If the mouse location is not in any item, find the nearest item in the list of interest and pretend that it was in that item. And then keep on descending in the display list tree. This rule means that a position to the right of the text selects a character rightmost in its line, and any position below the bottom of the text selects the character directly above it in the bottom line.

The item location algorithm is shown (somewhat simplified from reality) in Fig. 6-4. The algorithm runs in time O(b*d), where d is the depth of the display list tree and b is its branching factor. This code shows the recursive descent of the tree, the use of counted lists, and the rule of proximity—if the location is beyond the last item in the list, we work with that last item.

6.3. Identifying a node

We have seen how to map a mouse location to an item in the display list. But selections are defined in terms of nodes in the syntax tree; we need a mapping from items to nodes. This is accomplished by storing a *label* in each item: a pointer to its *node of causation.* The label is established when the item is created; it points to *the lowest level mainline node being interpreted at the time the item is created.* The interpreter is responsible for knowing what is the node of causation at any given time. So this part of the mapping problem becomes easy—find an item, you've found a node.

Now for the hard truth. The mapping algorithm described thus far works well for handling boxes, but it produces a very unsatisfactory user interface if applied indiscriminately to glue. Consider: Spaces are glue. The user must be able to select each space, because he might want to delete it. Each space has a place of its own in the structure of the document: it has an individual space character that is responsible for it. Leading (the spacing between lines) is also glue. But it does not have a place in the structure of the document. It is a *styling artifact,* thrown in by the paragraph builder as an aspect of the styling of paragraphs. By the normal rules, its node of causation would be a call to Para, or to some user-defined function that calls it. And then see the sad result: as the user moves the mouse vertically through a paragraph, he selects first

```
VAR
  topBox: Item;              the top-level container
  topX, topY: Coord;         the coordinates of its basepoint

PROCEDURE MapClick(x, y: Coord): Item
  BEGIN
    RETURN MapClick1(x, y, topBox, topX, topY);
  END MapClick;

PROCEDURE MapClick1(
    x, y: Coord;             the point of interest
    container: Item;         a box already known to contain or be "nearest to"
                             that point
    cx, cy: Coord): Item;    the absolute coordinates of its basepoint
  VAR
    i: Item;                 item currently being inspected
    ix, iy: Coord;           coordinates of its basepoint
    k: CARDINAL;             item counter—we're using counted lists
    n: CARDINAL;             number of items in this list
  BEGIN
    i := container^.first;  n := container^.count;
    k := 0;
    IF container^.type = Hbox THEN
      WHILE (k < n) DO
        ix := cx + i^.offset;
        IF ((x >= ix) AND (x < ix + i^.width)) OR (k = n-1) THEN
          IF (i^.type = Hbox) OR (i^.type = Vbox) THEN
            composite box, recurse
            RETURN MapClick1(x, y, i, ix, cy);
          ELSE RETURN i;   primitive item, we're done
          END;
        END;
        i := i^.next;  INC(k);
      END;
    ELSE    a Vbox
      WHILE (k < n) DO
        iy := cy + i^.offset;
        IF ((y >= iy-i^.height) AND (y < iy + i^.depth))
            OR (k = n-1) THEN
          IF (i^.type = Hbox) OR (i^.type = Vbox) THEN
            RETURN MapClick1(x, y, i, cx, iy);
          ELSE RETURN i;
          END;
        END;
        i := i^.next;  INC(k);
      END;
    END;
  END MapClick1;
```

Fig. 6-4 Mapping a click to an item

a character, then the whole paragraph, then a character on the next line, the the whole paragraph again, then a character, etc. This is not liveable. When the user points between lines, he almost certainly intends to point at one line or the other, not between them.

Now, how to turn this observation into a general and consistent algorithm? My solution is to distinguish two kinds of glue. A glue item that is *pickable* is individually represented by some element in the structure of the document—a character or expression in some unit. In the mapping process it acts like a solid object—it can be found just like a box, and it can be selected. Glue that arises as a styling artifact, on the other hand, is *non-pickable*. Such glue is generated within the internals of some function, primitive or user-defined. In the mapping process it acts like empty space—it isn't there, and it can't be found. A location within that space will be treated as if it were in the nearest pickable neighbor—just like a location beyond the end of the last item.

The interpreter does the work of making this distinction, and stores a boolean value, **pickable**, in each item. This is the algorithm it uses:

- every box is pickable
- every item that results from string conversion of a mainstream string is pickable
- every glue item that results *as a single item* from a mainstream function call is pickable
- all other glue items are non-pickable.

In particular, any glue items that are generated and concatenated into a list within a function are non-pickable. This covers the case of leading items generated by Paragraph, as well as any other glue that is implicitly inserted by a function. When the user points to such an item, the policy is to look for the nearest pickable item within the same list, and to act as if the mouse click had taken place within that item. If there is no such item, we turn our attention upward in the tree and use the container of the indicated item instead. Thus the problem of leading is solved: if the user points to leading, he will select a character in the line above or in the line below, whichever is closer.

6.4. Tracking down items

Selections are indicated to the user by highlighting, that is, by showing the selected items in inverse video. But that states the case too simply. *Items* are not selected; *nodes* are selected. The *items resulting from the selected node* are shown in inverse

video. To implement that we need a mapping from nodes to their resulting items. Reinterpretation, which is the technique used to display actual changes to the syntax tree, is one possibility. But in a high-quality mouse-oriented text editor, we want to provide real-time selection feedback as the user moves the mouse while making a selection. This requires an interaction loop that takes place at least 10 times per second, and preferably more—and objects much larger than a character may be highlighted or unhighlighted within that time.

Well, then, one might argue, just highlight based on the indicated item, and don't worry about nodes until the selection is complete. That would be a reasonable fallback position, but in some cases it would give inaccurate feedback to the user. This can occur for two reasons: 1. because a node may produce a list of items; 2. because a node may be evaluated more than once. In either case, the item-based highlighting strategy will highlight less than is really being selected—when the user releases the mouse button, he is in for a surprise. Figs. 6-5 and 6-6 show some examples.

```
use examplestyle.li

defaults
  contractdate = "Oct 14, 1987"

unit main =
  Vlist(
    Para(
      "According to the
agreement signed by both
parties on",
      contractdate,
      "tenant is entitled
to..."))
```

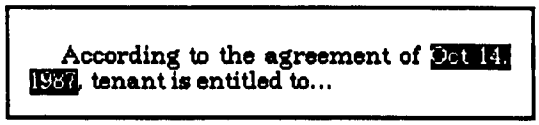


Fig. 6-5 Node produces several items in a list.

```
use examplestyle.li

function TwoPara(p: @Hlist) =
  Vlist(Size10(Para(p)),
    Vskip(8), Size14(Para(p)))

unit main =
  Vlist(
    TwoPara(
      "This is a node whose
output appears in two different
places"))
```

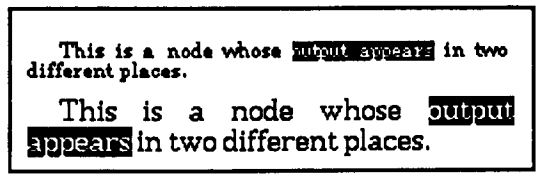


Fig. 6-6 Node produces multiple sets of output.

In fact, we can get accurate highlighting with real-time performance, by implementing a node-to-item mapping facility. For this purpose we use a hashing data structure called the Store. The Store's main purpose is actually to support incremental interpretation, so it is discussed in greater depth in Chapter 7. But it has a feature that makes it very well suited to this mapping task: it contains the output of every execution of every mainline node during the most recent reinterpretation of the syntax tree. For those nodes that were evaluated more than once, it contains multiple entries.

Unfortunately, those entries are not keyed to the node directly. They are keyed to a pair of values (node, fingerprint), where the fingerprint is a function of the list of nodes on the stack at the time this node was executed. There is no good way to reconstruct a fingerprint without reinterpreting the tree. So we add an auxiliary hash table to the store, one that is keyed on the node alone. For each stored node, this table contains an index into the main hash table, telling where to find the a result of that node. The main hash table is augmented with an extra field in each entry, the "chain" field. This field, if non-null, tells where to find another piece of output from the same node that produced this entry. Thus each auxiliary hash table entry points to the first of a chain of main table entries describing all the results of one node. By traversing this chain we can rapidly enumerate—and highlight—all the items derived from a given node. This process is shown, slightly simplified, in Fig. 6-7.

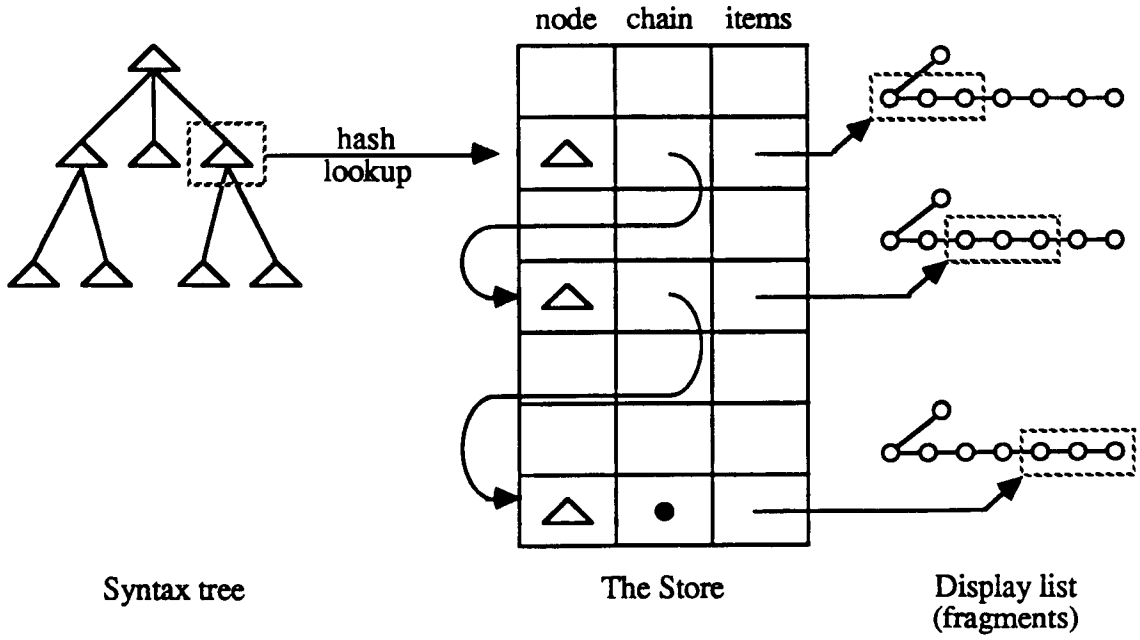


Fig. 6-7 One-to-many mapping

6.5. Highlighting on the screen

The same data structures that enabled us to locate an item from a mouse position allow us to find the screen rectangle for a given item, even faster. The one additional requirement is that there be uplinks in the display list, linking each item to its container. These can easily be provided at the time each container is built. The container's elements must be traversed anyway to determine its size, so we can add uplinks while traversing.

It now becomes a simple matter to find the absolute coordinates of an item. We go up the display list tree, keeping two coordinates to which we add the container-to-item offset at each level. Fig. 6-8 shows the algorithm. This algorithm runs in time O(d), where d is the depth of the tree.

```
PROCEDURE FindPosition(item: Item; VAR x, y: Coord);
  VAR
    tx, ty: Coord;        our current information about item's position
    container: Item;
  BEGIN
    tx := 0;  ty := 0;
    container := item^.container;
    WHILE item # topBox DO
      IF container^.type = Hbox THEN tx := tx + item^.offset;
      ELSE ty := ty + item^.offset;
      END;
      item := container;
      container := item^.container;
    END;
    x := tx + topX;
    y := ty + topY;
  END FindPosition;
```

Fig. 6-8 Locating the position of an item

We know the dimensions of each box, so having found its position, we could highlight it to those dimensions. But once again, the simplest algorithm turns out not to produce the best user interface. Using each box's own dimensions turns out to produce a very jagged appearance for highlights. In lines where some words contain tall letters and some don't, the highlight would be uneven, tall at the tall words and shorter at the shorter words. And what about spaces? Being glue, they have no height at all. How should they be highlighted?

The answer that seems to work best is to use the dimensions of the box in the principal direction, and the dimensions of its container in the transverse direction. So every word is highlighted to the width of the word, and to the height of the line. When lines are highlighted as units, each line is highlighted to the height of the line, and to

the width of the page. This provides a good answer for glue, which has no transverse dimension—just use the dimension provided by the container. And it makes extended highlights uniform and smooth.

6.6. Items outside the viewport

The highlighting algorithm is complicated by the fact that the whole document cannot (usually) be displayed at once. The problem is that items are present in the main display list that are not represented on the screen. Recall the mapping process described in Chapter 6. A node is selected; looking it up in the Store, we find the items in the display list that stem from it. Using the **offset** coordinate stored in each item and the uplinks from items to their containers, we determine each item's position on the screen and highlight its rectangle. But what if the item is not in the viewport? Then it has no rectangle on the screen. If it has ever been on the screen, it will have an uplink to the page box and an **offset** coordinate left over from its most recent screen position. Because of incremental box-building we are indeed still using the same box record to describe the page box, but this item is no longer an element of that box. Proceeding naively, we will highlight a totally irrelevant rectangle that probably cuts across lines and characters at random and certainly does not help the user's comprehension of what is going on.

This is a bit of a perplexing problem. We want editing and paging and scrolling to perform well; we don't want to have to track down items that depart from the screen to mark them "not on screen." So we need an inexpensive way of testing whether an item is currently in the viewport. The solution to this and several related problems is to use generation numbers. The *current generation* is a property of an open document as a whole; it is incremented every time a new reinterpretation begins. Every item has a field to store a generation number. The strategy is as follows: whenever we build or rebuild a container, we store the current generation number in each of its constituent items (but not in the container itself). To test that an item is indeed a member of the container it claims to be a member of, we test:

1. That the container has any elements at all.
2. That its first element has the same generation number as the item in question.

If either test fails, we conclude that the container, when most recently rebuilt, did not include our item. If an item-to-screen mapping is under way, we abort it and conclude that this item is not on the screen.

This strategy meets the need for efficient performance. Building or rebuilding a container requires a traversal of its elements anyway, so the additional cost of storing

a generation number is negligible. Verifying membership during item-to-screen mapping requires time of order O(d), which is of the same order as the cost of that mapping without it.

6.7. Complexities due to overlapping boxes

Where negative glue is used, boxes can overlap. This is generally a rare occurrence, but when it occurs it complicates the relationship of screen to display list. There never was a one-to-one mapping of boxes to screen space; containers and their elements occupy the same space. But with overlap there is not even a one-to-one relationship of bottom-level boxes to screen space. The complication strikes in two places: locating an item and highlighting rectangles on the screen.

When a mouse click falls in a region where two boxes overlap, which of them should be picked? The picking algorithm, operating straighforwardly, will pick the one that comes earlier in its container's list of elements. There will be absolutely no way to pick the other one. To alleviate this problem, we add an extra parameter to the picking algorithm: *picking depth.* With a picking depth of n the algorithm will reject n-1 successful picks, continuing on with its scan of the display list after each. It will report the nth pick or report failure if there are not that many levels of overlap at that position.

The user interface to this facility is to hold down the Option key while clicking. The first such click operates with a picking depth of 2; subsequent multi-clicks increase the picking depth by 1 each time. This replaces the usual function of multi-clicking, which is to promote the selection. In such cases the user must invoke Promote from the keyboard if he needs it.

Overlap also complicates highlighting. Highlighting is done by inverting pixels within the rectangle of interest, so if two overlapping rectangles are both highlighted, the region of overlap will not be inverted. Due to its rarety we make no attempt to solve this problem. Overlapping boxes are accompanied by strange visual effects; we view this as a feature which the user can learn to live with.

7. The Incremental Interpreter

7.1. Maintaining the result of a changing program

The appearance of a Lilac document is the result of executing the document program. This is the model we present and the invariant we must maintain. How should we maintain it in response to small changes?

Clearly, re-executing the program at every keystroke does not make sense. For one thing, it is prohibitively expensive in terms of cycles. But besides that, it takes no advantage of the structure of the situation. Though it is useful to represent a document as a program, in most cases it is not a very complicated program. In particular, there is high locality of effects. An ordinary change in ordinary text is likely to change the formatting of the paragraph containing it, but probably not the breaking of the page containing it, and almost certainly not the fonts, margins, spacing parameters, etc. of other parts of the text. So it should be possible, in the common case, to limit computation to the current paragraph.

What we need is a quick way to *test that a given editing action has had only local effects*. If so, we can handle them locally without rethinking the rest of the document. One approach, which I considered early in the design of Lilac, is to "boil down" the information in the upper levels of the document structure. At low-level nodes in the syntax tree, store the *local environment*—such information as the current font, indentation, spacing, whether we are inside a paragraph, and if so, its extent. In response to editing, reformat the text locally, using this summary of global formatting information, and ignore the upper levels of structure entirely. In the unusual case where information about the local structure is used at higher levels in nontrivial ways, this summary must contain an indicator that says, "not a local problem". And then we must give up on quick and easy reformatting for edits in that section, and must reinterpret a larger part of the document.

That is the problem with the local environments method. When it works, it works nicely; when it gives up, it gives up catastrophically. In designing Lilac for general and extensible use, I felt that it would be unwise to try to predict what information I would need in the local environment to meet almost all real needs.

An excellent example of a task that defeats this method is a boxed paragraph (Fig. 7-1). We write a function that formats a paragraph and puts a box around it. The width of paragraph and box are predetermined, but the height is determined by the amount of text in the paragraph, and the height of the box adjusts to match. Now the

user types some text into the paragraph. The local environment cannot just specify, "format this paragraph"—that could result in a gap in the box, if the paragraph grew longer. It must say, "not a local problem". And then, the whole paragraph, as well as its box, must be reformatted at every keystroke. One might propose supporting boxed paragraphs in the local environment—but then I will devise a function in which the size of the box varies as some nontrivial function of the size of the paragraph. The problem is endless.

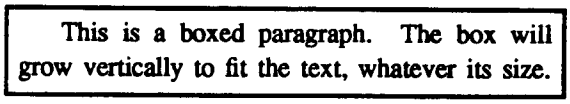


Fig. 7-1 A boxed paragraph

Lilac's approach is, instead, to revisit the syntax tree top-down, assuming locality of effects at every level unless proven otherwise. In order to focus attention only where it is needed, we invalidate those paths through the tree that have changed, or might have been affected by side effects. The local environment information, rather than being stored in many nodes, is regathered as the interpreter works its way down the tree.

The grammar is designed to facilitate this approach by encouraging a purely functional style and discouraging side effects. Changes to environment variables are associated with grammatical scopes, so that a local environment can be recomputed by top-down traversal of a single path, without reference to nodes aside of that path.

As we revisit the tree to deal with changes to the document, how do we relate them to the unchanged part of the document? This is a question that can arise at many levels. In the example above, the modified word or words must be connected to the other words in the paragraph; the modified line or lines must be related to the other lines. Then, if the height of the paragraph has changed, the box must be recomputed. And then, that box must be connected in with the rest of the document.

My view is that the most straightforward solution to this problem is to adopt the model that we are in fact reinterpreting the document. The relationships between changed and unchanged parts are established in the same way that they were established in the beginning, when we first interpreted it. A good plan—if it can be done in real time.

7.2. The Lilac interpreter

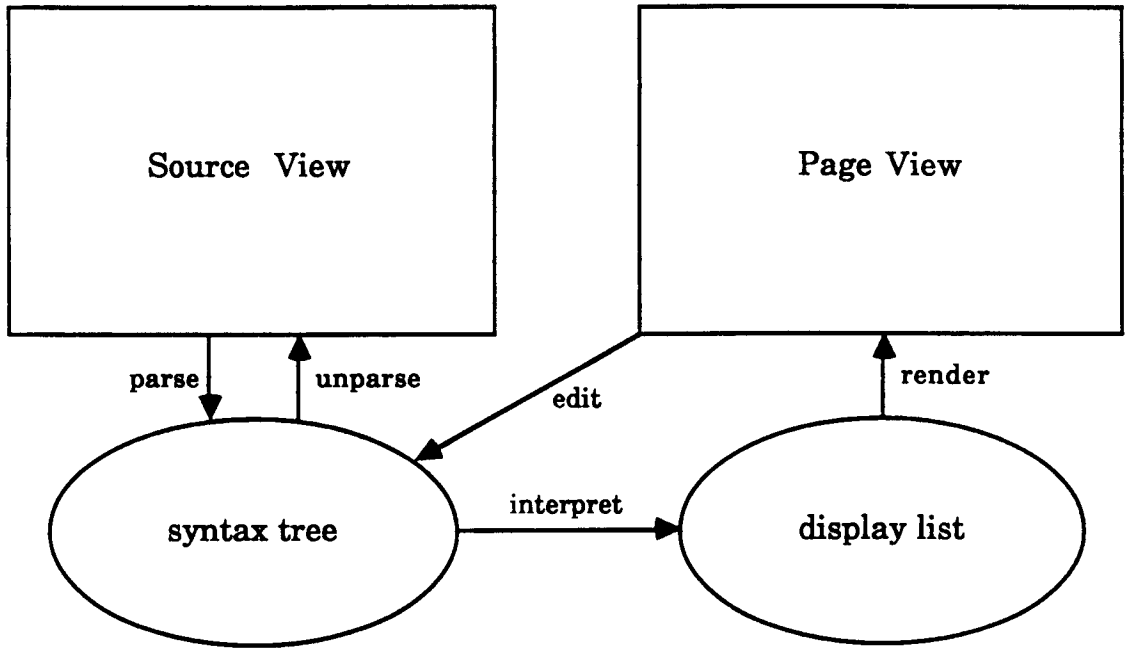


Fig. 7-2 The interpreter in context

The Lilac interpreter is designed to accomplish this task. In its overall structure, it is a reasonably ordinary interpreter. It interprets the syntax tree by recursive descent, invoking the main evaluation routine once for every node evaluated. It keeps a data stack for storage of function arguments and local variables. But in addition, it has some special features to fit it for the task of fast reinterpretation:

- It stores and retrieves intermediate results, to avoid recomputing them.
- It supports incremental versions of the critical Lilac language primitives, using retrieved previous results to save computation.
- It implicitly builds box-and-glue lists from structural lists, and uses incremental strategies to do it quickly.

These features are the focus of the rest of this chapter.

7.3. Incremental interpretation

Reinterpreting is a realistic approach because of the locality of effects of editing: most of the document has not changed. For all those parts which have not changed, we don't really need to reinterpret; we can retrieve the results of earlier interpretation.

What really needs to be reinterpreted? Any node which has directly been changed, and any procedure call whose arguments or local environment have changed. Because the Lilac language has been designed to be strictly functional, these are easy to find. They are the ancestors of a modified node, and the descendants of a modified environment-affecting node. The user interface does not support modification of environment-affecting nodes except by complete replacement. So we merely need to reinterpret ancestors.

Lilac handles this task by storing a "Dirty" bit in each node of the syntax tree, and by connecting the syntax tree with uplinks as well as downlinks. So when a node is edited, we mark it Dirty, along with any new nodes that might have been inserted beneath it. Then we trace its ancestry, all the way back to the top-level node of the main unit, marking each ancestor Dirty as we go. All other nodes in the tree are Clean.

Reinterpretation then proceeds as follows, starting with the top-level node: For any node that is dirty, we obtain the values of its children (i.e. arguments) and then execute

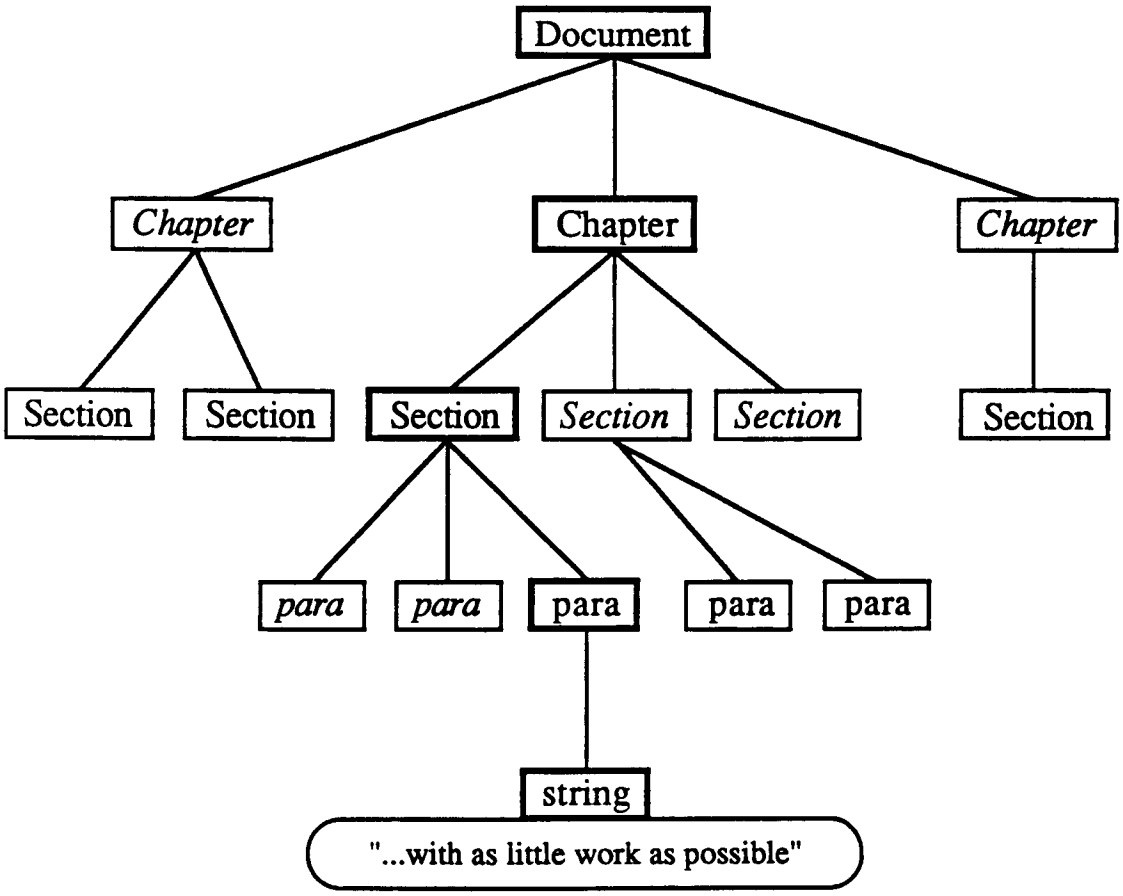


Fig 7-3 Incremental interpretation

it. For any clean node that is a child of a dirty node, we obtain its value by retrieval. All other nodes are not even visited. Whenever we execute a node, we store its value for retrieval next time, in case it may be needed. Fig 7-3 shows an example. We will suppose that the user has just typed the "e" on the end of "possible" in the string node. Dark outlines show the nodes that are marked Dirty, requiring re-evaluation. Italics indicate the nodes whose values will be retrieved. The other nodes (and their many descendants, not shown) will not be visited during this reinterpretation of the document.

7.4. Storing intermediate results

This strategy requires a way to store the intermediate results of interpretation in a quickly retrievable fashion. A couple of schemes present themselves as possibilities. One is to base the retrieval on the identity of the node that is being interpreted. Let us call this scheme *node-result storage*. The other is to base the retrieval on the identity of the operator being performed and the values of its arguments—making, in effect, a cache of operations performed. We will call this scheme *operation-result storage*.

Node-result storage faces the difficulty that many nodes are interpreted more than once, in different contexts. Most nodes of the mainline text are interpreted only once, but special cases can arise that cause such a node to be interpreted more than once. Nodes inside of functions, on the other hand, are routinely interpreted many times in one document. For example, the node that specifies the bar in a Fraction function is interpreted at every occurrence of a fraction. The node is not a unique key. One could consider a simplified scheme that stores values only for mainline nodes that are unique keys. But for reasons which will become apparent, that will not get the job done.

Operation-result storage does not have the problem of uniqueness. If the operator is the same, and the operands are the same, the result will be the same. But that scheme has an efficiency problem: an operator together with its arguments is a variable-length list, an unhandy data structure to manipulate and compare. Means could be found, perhaps, to compress that structure and speed its computation. But this method has its real downfall for another reason, which also rules out the simplified node-result storage scheme: even some of the basic system primitives, applied to arguments of a realistic size, are too expensive to repeat at every keystroke. There are two that stand out. One is the implicit ConvertString operation, which converts a plain character string into a horizontal list of boxes and glue, with sizes dependent on the current font. The other is the paragrapher, which converts a horizontal list of elements into a vertical list of lines, based upon the current line width setting. Each of them has the property that it must often deal with the entire horizontal list for a paragraph (200 words are not

uncommon). We cannot afford to rebuild this list, or recompute the linebreaks for it, at every keystroke.

Fortunately, we do not really need to. Only a small part of the list really changes at each keystroke. The common keystroke touches only one word; the rare keystroke splits one into two; no keystroke ever touches more. A similar benefit helps the paragraphing problem: the typical keystroke does not cause any change in line breaks. So we can save a great deal of time by reprocessing old results: compute the new list by patching the old; compute the new paragraph lines by rechecking and, where possible, reusing the old ones.

In order to do this, we need a scheme that will allow us to retrieve old results for use in computing new ones. Operation-result storage becomes an immensely harder problem: Given an operator and its arguments, find the previously computed result of this operator and *almost* the same arguments. I know of no efficient solution to this problem.

The simplified form of node-result storage, which only stores results for singly-used mainline nodes, also fails to serve this need. Not every call to Paragraph is a mainline node. Indeed, it is quite likely that format designers will wrap the Paragraph primitive in some other function which sets up some context, adds spacing, etc. Perhaps we can use the mainline call to that function as a storage key? Not with full generality. Consider the case of Fig. 7-4: a user-defined function, TwoPara, makes two calls on the Paragraph primitive. The mainline text makes two calls on TwoPara. Now there is no node, anywhere, that will uniquely identify one of the four resulting paragraphs.

```
function TwoPara(p: @Hlist) =
  Vlist(
    Size10(Para(p)),
    Vskip(8),
    Size14(Para(p)))

unit main =
  Vlist(
    TwoPara(
      "This is the first
paragraph.  It appears in
two different sizes."),
    Vskip(14),
    TwoPara(
      "This is the second
paragraph, also in two sizes."))
```

This is the first paragraph. It appears in two different sizes.

This is the first paragraph. It appears in two different sizes.

This is the second paragraph, also in two sizes.

This is the second paragraph, also in two sizes.

Fig. 7-4 Paragraphs not traceable to a unique node

7.5. Call paths and fingerprints

No node will serve as a unique identifier for one of these paragraphs. However, each paragraph can be uniquely identified by a *pair* of nodes: the call to TwoParax, together with the call from TwoPara to Para. This is an example of the general concept of *call path*: in the absence of loops, the sequence of function calls in progress, from the mainline down to the primitive of interest, is a unique identifier of one particular call on that primitive. It will never occur twice during one evaluation of the document. Equally important, it is also repeatable: in the absence of structural changes to the document tree, it will occur again if the document is evaluated again. Ordinary keystrokes cause no structural change to the tree; they are simply inserted into an existing leaf, a character string. So this sequence, the call path, meets our need for a key: it will uniquely and reliably retrieve the previous result of a computation whose input has been revised by editing.

It has some problems. For one thing, the above definition allows no loops. It is a rather restrictive language that has no looping constructs! However, uniqueness can be preserved if we can identify a loop counter which is unique for each iteration of a given loop. We include the loop counter's value in the call path. This may seem a strange mixing of types—we are including an integer in a sequence of nodes—but it is unique and repeatable.

A second problem is that this sequence is a variable-length object. Variable-length objects are inconvenient to store and inconvenient as hash keys. So instead of actually using the call path, we use an approximation, the *fingerprint* of a call path.

A fingerprint is a 64-bit quantity derived from any string of bytes. Chief among its useful properties is that is extremely well distributed. The probability that any two distinct strings in a collection of n strings of maximum length m will have the same fingerprint is less than approximately

$$\frac{n^2 m}{2^{58}}$$

Another useful property is that it can be extended: given two strings A and B, we can compute Fingerprint(AB), where AB is the concatenation of A and B, from B and Fingerprint(A), as cheaply as we can compute Fingerprint(B) by itself. For a string which grows periodically by small increments, this is ideal. This is the case with call paths.

The fingerprint algorithm was devised by M. O. Rabin [Rabin 81], and was implemented at SRC by Andrei Broder, Debbie Weisser, and Bill Kalsow.

Lilac uses this scheme in a hash-table data structure called the Store. A value for each instance of execution of each mainline node is stored there, and values of some non-mainline nodes are stored there as well. This hash table uses the pair (node, fingerprint) as a key, where "node" is the node of interest itself. Lilac maintains the fingerprint of the call path down to but not including the call on the node of interest itself; that final node is kept as a distinct element of the path. This saves time by reducing the number of fingerprints computed, and also helps with node-to-output mapping (discussed in Chapter 6). It also helps considerably in preventing collision: n becomes the number of call paths *ending in a particular node* rather than the number of call paths in the interpretation of the entire document. If we had a much-used function with 256 uses in one document, and a maximum call path length of 16 calls (contributing 4 bytes of fingerprint each), the probability of a collision occurring on a node in that function would still be less than one in 2^{36}.

The Store might be viewed as a cache, but it is not quite a cache. In a cache, performance benefits if the desired item is found there, but in the Store, all relevant items are required to be there. It grows as needed to handle the size and complexity of the document.

7.6. Incremental primitives

Lilac spends a large part of its time doing two list processing tasks: converting character strings into lists of words (ConvertString), and breaking lists of words into lists of lines (Para). These two tasks have a lot in common: they consist of scanning a lower-level list and producing from it a shorter, higher-level list made of larger pieces. A third task is similar—breaking lines into pages. This is not such a time-critical issue, because Lilac is organized so that it never needs to worry about more than one page at a time. But the first two are. They both frequently operate on the text of a whole paragraph at a time, and they simply cannot be done from scratch 10 times per second.

Fortunately, there is a way to reduce the amount of computation. These tasks have a strong property of locality: a small, local change in the lower-level list usually leads to a small, local change in the higher-level list, often affecting only one item. Hence we can win by doing incremental list processing. That is, during reinterpretation we obtain the higher-level list by doing minor surgery on the previous version of that list, rather than by rebuilding it from scratch. This equation summarizes the strategy:

old output + new input + change information → new output, fast

The problem then breaks into three parts: retrieving the old output, finding the change information, and doing the work. The new input consists of the arguments to the function, so it is readily available.

To retrieve the old output, we once again turn to the Store. Now the usage is a little different from the usage we saw in Section 7.5. Instead of retrieving the output of a Clean node in order to pass it along without doing any work, we are retrieving the output of a Dirty node in order to work on it a little. In fact, we are often retrieving the output of a subroutine node, outside of the mainline text and therefore outside all consideration of cleanness or dirtiness. The call to Para may be buried in a function two or three levels down from the mainstream node responsible for it. No matter. It is the previous output of Para, not of the responsible mainstream node, that we need for incremental processing. That is why the call chain/fingerprint concept is vital—it guarantees that we can retrieve the output of one particular invocation of Para (or some other primitive) no matter how deeply buried in subroutine calls.

Change information is needed to tell the incremental operator which part of the new input is really new—hence which part of the list needs work. To keep this information short and simple, we take advantage of a feature of editing: the change resulting from one command almost always occurs in one place. A group of objects are deleted, or inserted, or replaced, but the Rule of Selection guarantees that the change affected one contiguous section of one list. So we represent a change to a list with a simple triplet of integers, known as *(b, e, x)*. The value *b* (beginning) is the index of the first item that changed. The value *e* (end) is the index of the item following the last item that changed. The value *x* (expansion) is the net number of items added during the change. It can be negative, indicating a net deletion. Note that *e* is an index in the old list; the index of that item in the new list is $e+x$.

These *change records* are carried through the process of intepretation by attaching them to list values. Every list data type in the system supports this attachment. The editor, when changing a character string, attaches a change record to the relevant String node in the syntax tree. Incremental ConvertString, processing that node, attaches another change record to the resulting Hlist. And throughout the system, every incremental list processing operator, taking in a changed list with a change record attached to it, attaches a new change record to its output, describing the changes to that list. Thus the change information percolates through the process of interpretation in a cascading fashion. Incremental Para, processing an Hlist, attaches another change record to the resulting Vlist. That change record, finally, is useful to the page-breaking machinery.

7.7. Incremental list-breaking

Having gathered the needed information, we then have to do the work: update the output without computing any more than is absolutely necessary. This task varies from operator to operator, of course, but the most interesting ones have surprisingly much in common. They all involve transforming a low-level list into a higher-level list which is divided into larger meaningful units, each made up of several of the lower-level units. ConvertString transforms a list of characters into a horizontal list, whose units represent words and spaces. Para transforms a horizontal list into a vertical list, whose units represent lines of text. And the page builder, conceptually, transforms the vertical list for the whole document into a list of pages. I call these operators the list-breaking operators, because they involve finding the dividing points in the lower-level list that form the breaks between one higher-level unit and the next.

There are some differences among these operators. Para and the page builder break when a certain space is filled up; ConvertString breaks based on character type (letters vs. spaces vs. punctuation). But it turns out that the algorithms, and especially the incremental aspects of those algorithms, have much in common. Here I present a sketch of that common algorithm.

This algorithm uses the following "generic" subroutines, which would be replaced by routines specific to the type of list-breaking taking place:

MakeGroup(input: InputList; begin, end: integer): OutputItem
Makes a single item of "broken" output from a section of input list.

ItemLength(item: OutputItem): integer
Given an output item, tells how many input items it was made from.

BreakingCriterion(input: InputList; begin, end: integer): boolean
Given a section of input list, assuming that there is a break at begin, tells whether there should be a break at end.

The following universal list-managing routines are assumed:

Length(list: List): integer
Tells the number of items in a list.

Append(list: List; item: Item)
Adds an item to the end of a list.

Replace(list: List; begin, end: integer; newlist: List)
Replaces the part of list between begin and end with the contents of newlist.

To every Lilac list type, a change triplet (b, e, x) can be attached. In these examples, that attachment is accessed by

PutChangeInfo(list: List; b, e, x: integer)

Attach a change triplet to list.

GetChangeInfo(list: List; var b, e, x: integer)

Retrieve the values in the change triplet attached to list.

Fig. 7-5 shows the outline of the standard, non-incremental list-breaking algorithm. Reality, of course, is more complicated than this. There are spaces as well as words; there are interline spaces as well as lines. The WHILE loop, in practice, keeps running totals of various quantities in order to make the breaking criterion cheaper to evaluate. But this outline sets the stage for the description of the incremental list-breaking algorithm.

The incremental algorithm is shown in Fig. 7-6. Notice the four sections of processing, in place of one main loop for the non-incremental case. In the first part we find our way to the beginning of the changed section. Much time is saved here: this part of the input list gets no scrutiny, and no new items are created.

In the second part, we actually do the list-breaking work. This is essentially the same work that was done in the non-incremental case, but it is usually applied to a much smaller section of list. But note that it may continue well beyond the end of the

```
PROCEDURE BreakList(input: InputList): OutputList
VAR
  n: integer              number of items in input
  i, j: integer           counters that work their way through the input
  group: OutputItem       one item of the output list
  output: OutputList      the output list
BEGIN
  n := Length(input)
  i := 0 ; j := 0
  output := nil
  REPEAT
    WHILE (j < n) AND NOT BreakingCriterion(input, i, j) DO
      j := j + 1
    END
    group := MakeGroup(input, i, j)
    Append(output, group)
    i := j
  UNTIL i = n
  RETURN output
END
```

Fig. 7-5 The list-breaking algorithm

```
PROCEDURE IncrBreakList(input: InputList; oldout: OutputList)
    : OutputList
VAR
  n: integer              number of items in input
  i, j: integer           counters that work their way through the input
  group: OutputItem       one item of the output list
     New variables for incremental action:
  L: integer              number of items in oldout
  k: integer              a counter that works its way through oldout
  l: integer              a counter that keeps track of our position in terms of items in
                          the previous input list
  b, e, x: integer        the change triplet for input
  b2, e2, x2: integer     the change triplet for output
  newout: OutputList      the partial list formed by processing changed input
  added:                  number of items in newout
  output: OutputList      the final resulting output list
BEGIN
     1. Find the place to start processing
  n := Length(input)
  L := Length(oldout)
  GetChangeInfo(input, b, e, x)
  k := 0 ; l := 0
  REPEAT
    l := l + ItemLength(oldout[k])
    k := k + 1
  UNTIL (k = L) OR (l >= b)
  k := k-1
  l := l-ItemLength(oldout[k])

     2. Process the changed section
  b2 := k
  i := l
  j := i
  newout := nil ; added := 0
  REPEAT
       Make an item of new output
    WHILE (j < n) AND NOT BreakingCriterion(input, i, j) DO
      j := j + 1
    END
    group := MakeGroup(input, i, j)
    Append(newout, group)
    added := added + 1
    i := j
       Scan past old output, in search of a match-up
    WHILE (l < e) OR (l < i) DO
      l := l + ItemLength(oldout[k])
      k := k + 1
    END
  UNTIL (i = n) OR (i = l + x)
     We have now finished making new output items
```

Fig. 7-6 Incremental list-breaking (continues on next page)

```
    3. Now, if necessary, update indices in the remainder of the list
  WHILE k < L DO
    <update oldout[k]>
    k := k + 1
  END

    4. Patch the new material into the old list
  e2 := k
  x2 := added - (e2-b2)
  output := Replace(oldout, b2, e2, newout)
  PutChangeInfo(output, b2, e2, x2)
  RETURN output
END
```

Fig. 7-6 Incremental list-breaking (continued)

change as specified by *e*. We cannot stop computing new breaks until we find one that matches a previously computed break. In the worst case, this may not occur before the end of the list; this sometimes occurs in paragraphing, where an overflowing line may cause the whole remainder of the paragraph to be reformatted. The same is true of page breaking. For string conversion this is not a problem; one word break has no influence upon the next.

The third section is not always necessary. But in the case of string conversion it proves useful to store an index in each item, so there is no need to compute *l*. It is available at any time as oldout[k].index. That is why the third section is present: it allows us to update the index field of each item following the change. The incremental paragrapher has no such section; it computes *l* as it goes instead of storing it.

The fourth section performs the actual change to the output: a fragment of the old output list is snipped out, and the newly computed list fragment is grafted into its place. The destructive modification of the old output list is crucial—it prevents the allocation and copying that would otherwise be required.

7.8. Implicit list construction

We now come to the third special feature of the interpreter, its handling of structural lists. The language definition has set us up for an unusual piece of work here, because it allows repeatable parameters and provides that the resulting list of argument values should appear to the callee as a single value of a list type. In other words, the arguments that make up a structural list are to appear to the callee as an actual list, a list of boxes and glue. In practice this means that the interpreter has to do the work.

The work is done by one of the important "invisible operators" of the system, MakeList. Here is what happens. When interpreting a function call with a repeatable parameter, we push all the arguments onto the data stack, keeping count of the repeatable arguments (those that belong to the repeatable parameter). Then we call MakeList with a stack address and count. It takes the arguments, each guaranteed by the type system to be either a list of the appropriate type or a single element compatible with that list type, and joins them into a single list. MakeList returns this value, which we then place on the stack in place of the whole collection of repeatable arguments. We are now ready to call the function, with a stack which has as many arguments as the function has formal parameters. The repeatable arguments have all been joined into one.

MakeList, like all the other list-processing operations, must be done incrementally to avoid intolerable computation per keystroke. Indeed, it turns out to be the most frequent and time-consuming operation in the system, so incrementality is particularly vital here. Incrementality, however, poses a problem for the representation of lists. When we do a MakeList we join the sublists together into one big list. But those sublists are still stored in the Store, and at the next reinterpretation they must reappear from the Store, as sublists. But if they have already been joined together, and are ordinary linked lists, then the first sublist will appear to be as long as the whole list, causing much confusion. A standard solution might be to build the longer list by copying the items of the sublists. But here, both for reasons of performance and because we want a display list rebuilt of the same items each time, copying is out of the question.

My solution has been to avoid the ordinary linked-list data structure, and use a modified version, the *counted list*. That is, a list is represented by linking its items in a chain, but its end is never determined by the presence or absence of a null link field. Rather, each list is described by a header node which contains, among other information, a count of the items in the list. A list of this sort can be joined into a larger list without losing its own identity, provided that the larger list is never destructively modified in a way that breaks up the sublist. A list of this sort can also be made from a sublist of a larger list without copying any items. This property is vital for paragraph-building and page-building, activities which also cannot afford to do any copying.

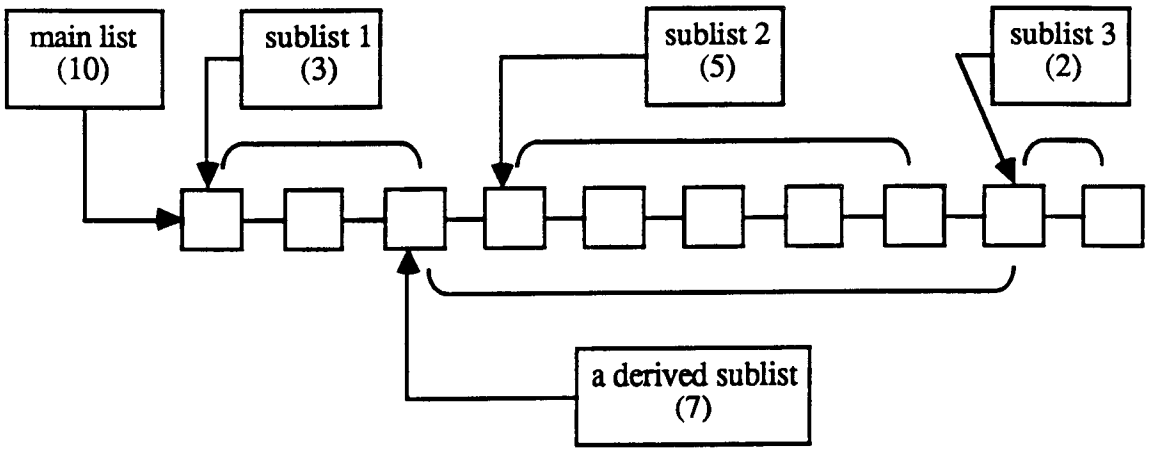


Fig 7-7 Counted lists

9. Incremental list construction

The task of MakeList is to build one counted list out of several sublists. In the ordinary course of things, this requires visiting every item in the resulting list: traverse a sublist to find its last item; make a link from that last item to the first item of the next sublist; then start traversing the next sublist. Now the problem: the full display list is, in general, a vast vertical list which contains all the lines on all the pages. The actual active display list is a sublist of this list, containing only as many items as will fit on the visible page. But the full display list is likely to contain many hundreds of items, and it is the output of the top-level node in the document, which is dirty on every editing operation. We cannot afford to traverse this list on every editing operation!

Fortunately we really don't need to. Since our display list is rebuilt of almost all the same items on each re-interpretation, the links we need to build are already in place—all except for those near the point of change. Indeed, the vast majority of changes to a list take place in only one sublist, so only two links need to be rebuilt: the one before the changed sublist, and the one after. Fig. 7-9 illustrates this situation. And if we can convince ourselves that the change did not involve the first or last elements of that sublist, we don't need to rebuild any links at all! The incremental primitive that built that sublist in the first place has taken care of the job. All we need to do is to update the count field and the change information for the main list.

This is clearly possible, but it requires a lot of extra information to be available at the right place at the right time. In particular, it requires a pointer into the middle of the list, to a place just before the beginning of the modified sublist, so that we can find that place without traversing a lot of items to get there. It would be quite unmanageable

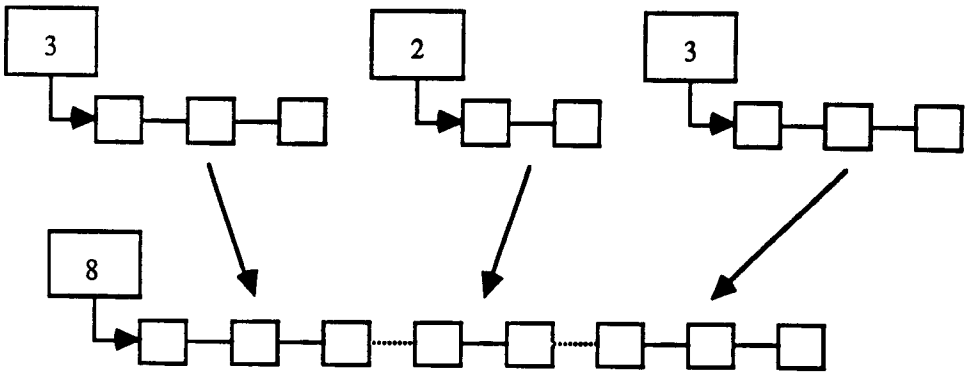


Fig 7-8 MakeList (dotted links are newly made)

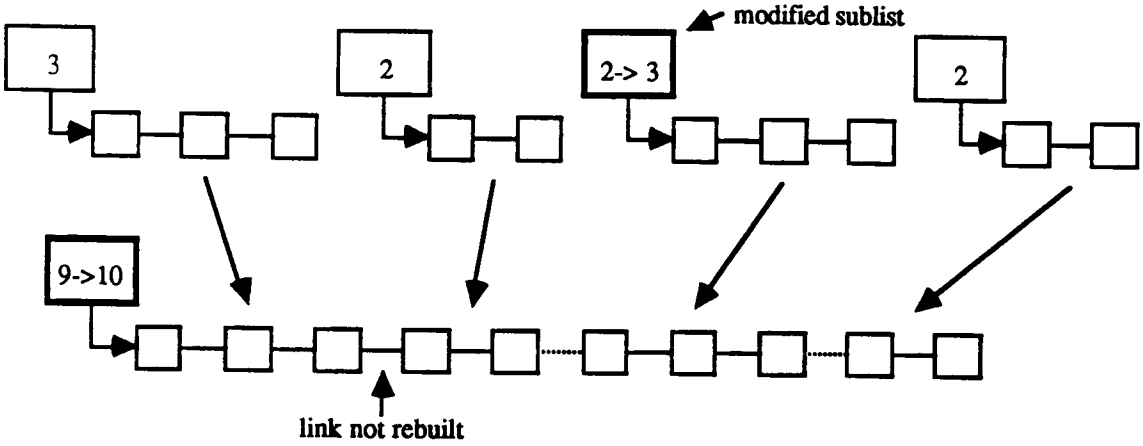


Fig 7-9 Incremental MakeList

to keep many pointers of this sort, but it is convenient to keep one per list. And so we employ the concept of a *hot spot*—the sublist of a list that is currently being worked on. And MakeList ends up with not two, but three, levels of incrementality of operation:

Level 1: Full MakeList. No incrementality.

Level 2: Slow Incremental MakeList. We discover which links need making and make them, and store hot spot information for next time.

Level 3: Fast Incremental MakeList. Using hot spot information, we avoid traversal and build at most two links, sometimes none at all.

These three levels are in chronological order: it takes at least three interpretations of a given piece of document before MakeList is operating in high gear. The first time

(level 1) it builds lists from scratch, with no prior knowledge. The second time (level 2) it rebuilds lists, noticing at each level which sublist was the focus of change, and building hot spot information accordingly. The third time, provided that the change is in or near the same place as last time, it uses the level 3 algorithm and avoids nearly all of the computation. Thenceforth, as long as the focus of editing remains in roughly the same place, the level 3 algorithm will be used.

The exact rules for hot spots are as follows. A hot spot can be built in Level 2 provided that only one sublist has been changed. A hot spot can be used in Level 3 provided that:

1. The same node has been edited that was edited last time.
2. There is a hot spot for this list.
3. There have been no deletions or insertions of whole arguments in the structural list.

If any of these conditions are violated, Level 3 will fail over to Level 2, which usually can set up a new hot spot for fast operation next time.

Thus we have avoided the need to traverse the display list on every edit. This is particularly important because MakeList is called several times during even the simplest reinterpretation—once for every structural list in the chain of dirty nodes! Once to join paragraphs into a section; once to join sections into a chapter; once to join chapters into a full display list. And perhaps one or more times, even earlier, to build the horizontal list that formed the paragraph. Fig. 7-10 illustrates the situation.

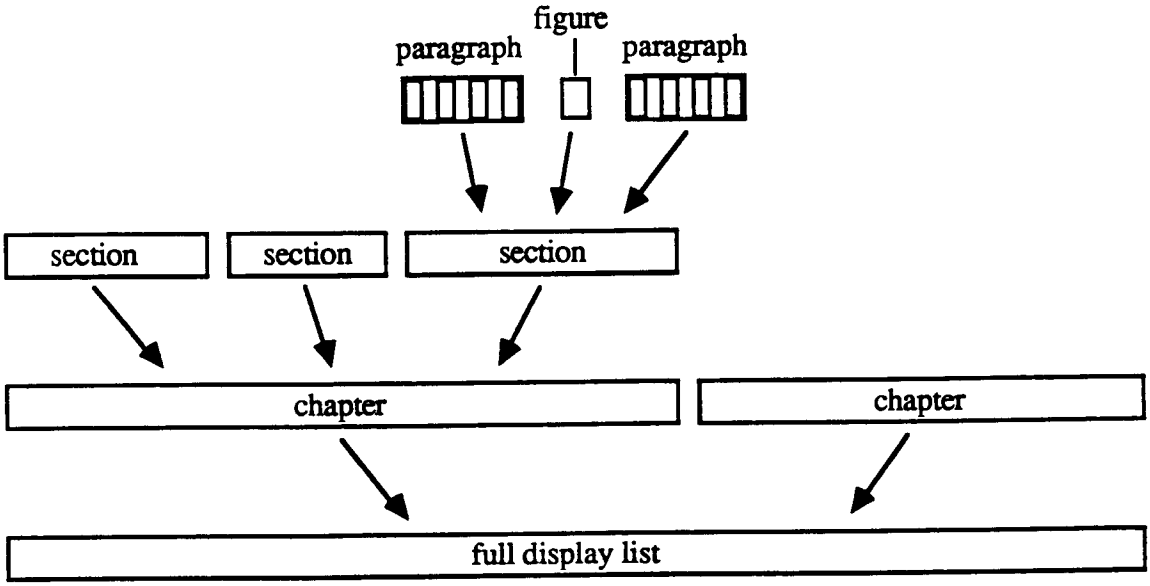


Fig 7-10 Several levels of MakeList

7.10. Incrementality within functions

At this point, there is one important place where incremental efficiency has not reached: user-defined functions. These functions, with their potential to call subroutines many levels deep, can be quite time-consuming. And they can take up more time than they really need to.

If the user edits the title of a section, should we have to rebuild the list of paragraphs in that section? Or if he edits the text of a bulleted item, should we have to regenerate the bullet? Fundamentally, no: the output of both these tasks is already stored in the Store, and their input has not changed. But how to discover this? In a function body (the table-row function, for example) we cannot mark nodes Dirty, because the function may be used any number of times, and we are focusing only on one of those uses. So, it seems, we will be forced to re-execute the whole function with all its subroutines, every time a keystroke changes any one of its arguments.

Actually, because of the way incremental list-building is done, this is not only inefficient but intolerable. Consider the bulleted-item function (which is responsible for the bulleted items that appear later in this section):

```
function BulletItem(h*: Hlist) =
  let lmargin = lmargin + 40, indent = -14 in
    Para(Bullet, Hskip(8), h)
```

This is the problem: a change record assumes that changes to any one list take place within a contiguous range. If discontiguous changes occur, they will be recorded as a contiguous span that spans them all. Now, if Bullet is re-executed to regenerate the bullet, it will appear as a new item. If the user is typing at the end of the list *h* (as it quite likely) then there will be a new item there. The Hlist returned by BulletItem will show a change span that includes the whole list. The paragraph-builder (an instance of incremental list-breaker), seeing a "completely changed" list, will do its line-breaking from scratch, which takes too long. Worse yet, it generates lines which appear as new items, so that the screen updater (described in Chapter 8) repaints them from scratch, with the result that the whole paragraph flashes at every keystroke. It is imperative that we avoid producing "new" items that are not genuinely new.

Thus, both for efficiency and for correct computing of change records, we do not want to re-execute every expression in a function. What we really want is to re-execute every expression within that function *that depends on a changed argument.* Those expressions that depend on unchanged arguments can safely be handled by looking up their value in the Store. Those subroutine calls that depend on unchanged arguments do not need to be made. Subroutine calls that depend on no arguments, but only on

constants (such as the bullet in the bulleted-item function) never need to be re-executed. All this, however, requires that we be able to detect changed values.

Items show change in a field in the item record. Items, however, are not the only values of interest. Even such simple values as integers and Booleans may change during editing. For example, an integer, derived by taking the size of one box, may be passed to a function which will generate another box sized to match. Integers have no fields in which to indicate change. The solution I adopted (quite reluctantly, at first) is to add a tag bit to every value, indicating whether it is old or new. All values, as handled by the interpreter or stored on the interpreter stack, carry this bit. It is set by the following rules:

- Every value retrieved from the Store is *old.*
- Every value stored in the default environment is *old*, except when source view editing has just changed default values.
- Every value given as a constant is *old.*
- All other values (i.e. those derived by fresh computation) are *new.*

When a function is invoked, then, it will see unchanged arguments with a tag of *old* and changed arguments with a tag of *new.* When interpreting a function, the interpreter does a sort of data flow analysis on these tags. Preparing to execute a subroutine call from this function, it gathers the values of the arguments. If any of them are tagged *new*, the subroutine call proceeds as usual; otherwise, we retrieve its value from the Store. If any argument within the repeatable part of an argument list is tagged *new*, we call IncrMakeList to redo the implicit list-building; otherwise we retrieve the implicitly built list from the Store.

This strategy bears an interesting relationship to the strategy used for mainstream nodes. There we mark nodes Dirty; the descendants of a clean node are not even visited. Here we trace tags on values; every node in a function must be *visited*, to see what it depends on, but not every node is *executed.* The big advantage comes in the fact that some subroutine calls are avoided; thus the nodes of the subroutine are not even visited.

This strategy also has a weakness: if user functions are not carefully designed, it can lead to a situation in which some things are neglected that should have been re-executed. If a value which can be changed by source view editing is passed as an argument to one function, which then stores it in a global variable with a **let** and thus passes it on to a subroutine which uses that variable, the subroutine's usage of it will not be detected. The subroutine may not be re-executed when that value changes. Thus

users need to design their subroutines in such a way that all such values are passed explicitly as arguments. This is an area that needs improvement, perhaps by improving the definition of the language.

7.11. Incremental paragraph building

Of the various incremental primitive operators in Lilac, one is interesting enough to be worthy of more discussion: the paragraph builder. Knuth and Plass, in [Knuth 81], present a detailed discussion of algorithms for breaking paragraphs into lines. The main result to be achieved by such an algorithm is to produce nice lines—lines in which the spaces do not have to stretch or shrink very much, as compared to their ideal size. This can be expressed in terms of the *adjustment ratio (r)* of a line, which measures the degree to which the spaces have stretched (positive r) or shrunk (negative r).

They discuss three algorithms:

1. First-fit, in which we choose the first legal break we find which yields an r value within an acceptable range.
2. Best-fit, in which we choose the legal break the minimizes $|r|$ out of all the possibilities for that line.
3. Optimal-fit, which minimizes $\sum|r^3|$ over all lines in the paragraph.

TEX uses the optimal-fit algorithm, and it would seem logical that Lilac should do likewise. But in fact that proves to be a bad choice, for two reasons. First, it is not a good algorithm for interactive use. Optimization over the whole paragraph means that any given line can affect line-breaking decisions made anywhere else in the paragraph. A character typed in line 5 can change the breaking of line 2. Though I have not built and tested such a design, I am convinced that it would be disconcerting and distracting to a user.

Second, the data structures for the optimal-fit algorithm are complex and do not lend themselves to incremental updating. Computations that involve the whole paragraph would have to be repeated at every keystroke, rather than only on occasional keystrokes. The result is prohibitively expensive.

Lilac, instead, uses the best-fit algorithm. It simplifies the TEX model a bit further: where TEX allows the user to influence breaking decisions by adding penalty items, Lilac provides no such option. The only criterion for choosing among line breaks is $|r|$.

This leads to a fast and straightforward method for finding the best fit. To break a line, we process its items in sequence, keeping a running total of the widths of all items

and the stretchabilities and shrinkabilities of glue items. Whenever we find a legal break, we remember it, always keeping the memory of one legal break behind the current point of processing. This proceeds until we find a break for which line width >= desired width ("the long break"). We compute r for it, and for the most recently remembered break ("the short break"). Whichever yields the smaller $|r|$ is chosen.

The advantage of leaving penalties out of the design is that it makes it easy to know when to quit. Penalties can artificially make the "long break" less desirable than one that produces an even longer line. We would have to continue checking breakpoints until we found a line of such a length that not even a large negative penalty could make it desirable. Without penalties, it is easy to find two breaks surrounding the optimal point, and to determine which of them is better.

A production version of Lilac might reasonably include a special command to set paragraphs by the optimal-fit method. (An author might want to do this just before printing.) The way the Lilac data structures work, a paragraph thus set would stay that way until it is edited; then by a process of gradual degradation it would return to best-fit quality as editing proceeds.

7.12. The incremental interpreter algorithm

We complete our discussion of the incremental interpreter with a sketch of its algorithm in pseudocode form. It is shown in Fig. 7-11. This summary leaves out much, but it shows the central incremental features: the two uses of the Store, the use of incremental primitives, and the maintenance of the fingerprint of the call path.

The global variables are:

```
fingerprint: Fingerprint        the fingerprint of the current call path
stack: array of Value           the interpreter's stack
sp: integer                     the interpreter's stack pointer
```

The usage of **stack** and **sp** is not really illustrated, except to show the means of passing arguments to primitives.

A Node is viewed (a vast simplification) as a pointer to a record with the following fields:

```
RECORD
  class: NodeClass              what kind of node this is
                                  (the value CallNode means a function call)
  mainline: Boolean             whether this node is part of a Unit
  dirty: Boolean                the "Dirty" mark for this node
END
```

A ProcRecord is a record as follows:

```
RECORD
  CASE primitive: boolean OF
    true:
      normal: PrimitiveProc     the function that implements this primitive operation
      incremental: IncrProc     the incremental version thereof
    false:
      topNode: Node             the top-level node of the defined function
  END
END
```

Fig. 7-11 shows the central core of the interpreter:

```
PROCEDURE Eval(node: Node): Value
VAR
  value: Value          the result of this evaluation
  oldValue: Value       previous result of this evaluation
  storeMe: boolean      a flag meaning that this value should be stored
  proc: ProcRecord      the function being called, for a CallNode
  savefp: Fingerprint   temporary storage for the current fingerprint
BEGIN
  storeMe := false
  IF node^.mainline AND NOT node^.dirty THEN
      we can skip evaluation of this node altogether
    Store.Get(node, fingerprint, value)
    RETURN value
  ELSE IF node^.class = CallNode THEN
    proc := <find the function called by node>
    <evaluate and push the arguments, do a MakeList if appropriate>
    IF proc.primitive THEN
      IF (proc.incremental # NIL) THEN
        Store.Get(node, fingerprint, oldValue)
        value := proc.incremental(stack, sp, oldValue)
        storeMe := true    so that the new value will be stored
      ELSE
        value := proc.normal(stack, sp)
      END
    ELSE    not a primitive
      savefp := fingerprint
      fingerprint := Fingerprint.Extend(fingerprint, node)
      value := Eval(proc.topNode)
      fingerprint := savefp
    END
  ELSE    other node classes
    value := <appropriate computation for this node class>
  END
    now, update the store
  IF (node^.mainline) THEN
    Store.Put(node, fingerprint, value, true)
  ELSE IF storeMe THEN
    Store.Put(node, fingerprint, value, false)
  END
END Eval
```

Fig. 7-11 The incremental interpreter algorithm

Several features are noteworthy. The first use of the Store is to skip evaluations entirely. If a mainline node is clean, its value is retrieved directly from the Store; it is not evaluated and its descendants are not even visited. The second use of the Store is to support incremental operations. Only primitive incremental operators are supported by this design; user-defined incremental operators are another research topic. But if a node is a call on a primitive that has an incremental version, then we maintain its value in the Store, and we use that stored value to provide the "old output" argument to incremental operators.

The treatment of non-primitive, that is, user-defined functions shows how fingerprints are maintained. A call to a user-defined function represents a lengthening of the call path; the current node is being added to it. So we compute the fingerprint of the new call path by Fingerprint.Extend, which adds a new node onto the existing fingerprint. The fingerprint of the existing call path is saved away in a local variable; when the function returns, and the call path shortens again, we restore the fingerprint to its former state.

A call on a primitive is, technically, a lengthening of the call path. But since primitives never call anything else, they have no need of the fingerprint, so we can save time by not computing it. This is one reason why it is useful to use the (node, fingerprint) pair as a storage key, rather than simply the fingerprint: fewer fingerprints need to be computed.

The two calls on Store.Put are distinguished by their use of the "store by node" feature of the Store. The fourth argument, when true, indicates that the entry for this (node, fingerprint) key should be linked into the auxiliary data structure which uses the node alone as a key. As discussed in Section 6.4, this data structure allows us to retrieve all the output of a node in order to highlight it when the node is selected. This is a second reason why (node, fingerprint) is a more useful key than a fingerprint by itself. Maintaining the auxiliary data structure has some cost, so for non-mainline nodes, which cannot be selected, we do not bother. Such nodes are stored only for the benefit of incremental primitives.

7.13. The modified display list

The net effect of incremental list-building and incremental list-breaking, cascaded over several levels of hierarchy, is that reinterpretation produces a new display list that differs very little from the old one. The new display list is a tree of items which not only resembles the old one in its structure, but consists almost entirely of *the selfsame*

item records. After a typical keystroke there are only one or two freshly created items in the entire tree: a word and/or a space adjacent to the insertion point. Apart from that local change, the display list has reassembled itself from its elementary components.

I enjoy a picturesque image of this process: A tree (the display list) has been chopped into bits and pieces and scattered on the ground (components stored separately in the Store). On command, the pieces rise and begin to arrange themselves, each springing into its appointed place. In a very short while, the tree has magically reassembled itself almost exactly as it was before—the same tree, built of all the same pieces in all the same places except for one changed twig.

For efficiency of interpretation, this is excellent. It nearly eliminates allocations, a costly operation in most systems. But even more important, it gives us an opportunity to compute the information needed to incrementally update the screen. Each box contains a coordinate describing its position, relative to the basepoint of its container. When that container is being incrementally rebuilt, we can retrieve the old coordinate at the same time we are computing the new one. By storing both the old and the new coordinate in the box, we now have information about the motion of the box on the screen. This allows the screen updater to reuse the pixels for this box, rather than regenerating them.

8. Updating the Page View

We now turn from the problems of knowing what is on the screen and highlighting it, to actually changing it. Here, as in other parts of the system, we have a complex piece of state (the image) that we cannot afford to recompute at every keystroke. We must reuse most of it, recomputing only the part that has actually changed.

8.1. Reusing the pixels

After an editing operation has taken place and the syntax tree has been reinterpreted, we have a display list that describes an image different from the one on the screen. Various items in various parts of the display list may have been added, deleted, or modified in size and in content. Modifications to any item may have changed the dimensions of its container, its container's container, etc. Our task is now to update the screen to correspond to the new box tree, and to do it without doing very much work.

For this process the guiding principle is to reuse the pixels. On most bitmapped displays, it is much faster to copy a line of text from one part of the screen to another than to regenerate it from font and character information. So we set ourselves the following goal: for every box that is on the screen and unchanged from the previous state, reuse the pixels, moving them if necessary. Regenerate only those boxes that are new or have newly moved into the visible area.

The keys to achieving this are (1) knowing where each box already is on the screen, and (2) knowing which ones have changed. This information is generated during incremental interpretation and is stored in the box records. Part (1) is the geometrical change information. Each box record contains a coordinate, called **disp** (displacement), telling the distance from its container's basepoint to its basepoint in the principal direction (as seen from the container's frame of reference). When we rebuild a container, we copy the **disp** field of each component into another field called **olddisp** just before computing the new value of **disp**. Thus we know how far this component must move, relative to its container.

Part (2) is the **mark** field of an item, and it has one of three values:

Clean	This item has not changed since the last update.
Dirty	(applies only to containers) This box has changed since the last update; look at its components for the details .
New	This item is new since the last update, or for other reasons needs to be rendered from scratch.

This value is computed as follows. Whenever a primitive function generates an item it marks it New, indicating that this item has never existed before. Whenever an incremental primitive function rebuilds a container it marks it Dirty, indicating that it already exists but may need some rearrangement. Whenever the screen updater renders an item it marks it Clean, indicating that this item is now correctly represented somewhere on the screen; unless modified, this mark will be there next time to tell us to leave this item alone.

After a reinterpretation, then, the display list is marked up in the following pattern:

If a box is marked Clean, all its components (and their components, etc.) are Clean.
If a box is marked New, all its components are New.
If a box is marked Dirty, any of its components may be Clean, New, or Dirty.

And thus we can write the rules for updating the screen:

1. For any box that is New, we render it and all its components from scratch.
2. For any box that is Clean, we copy the pixels from its old position to its new position (or if those positions are the same, we leave it alone).
3. For any box that is Dirty, we descend recursively and apply these rules to each of its components.

8.2. Safe ordering

With items shrinking and expanding and appearing and disappearing in various places, we have to be careful to copy each useful collection of pixels before we destroy them by copying something else into their place. Is this always possible? This problem was originally posed, in a somewhat simpler form, by Howard Sturgis. [Sturgis 83] The answer turns out to be yes, provided that there are no permutations among the components of any container, and that items are not moved from container to container. A permutation could occur, for example, in an editor that allowed two simultaneous selections and an Exchange command to exchange their contents. An exchange of two items in the display list would make safe ordering impossible—whichever of the items was updated first would destroy some or all of the pixels belonging to the other item.

Lilac avoids these problems simply by disallowing permutations and container-to-container moves of items. The Lilac editing model is cut-and-paste—in order to permute the text, the user must cut part of it and then paste it elsewhere. The pasted material is treated as new, resulting in brand new items. If someday the user interface supports an operation like Exchange, it will still be modelled internally as a deletion combined with an insertion. And the insertion, like any other insertion, will be rendered from scratch, rather than by copying from anywhere else on the screen.

Actually, the process of rebreaking a paragraph results in moving items from line to line. To prevent this from complicating the screen update problem, the incremental paragrapher marks such items New, causing them to be rendered from scratch.

Given these restrictions, we can define a safe order for updating the items on the screen:

> A safe order is one in which no useful source pixels are overwritten before they can be copied to their new position.

The algorithm presented here (shown in detail in section 8.3) does all screen update operations in a safe order. At its core is the Update procedure, which deals with the list of components of one container. It is first applied to the components of the top-level box; by calling itself recursively, it deals with any other dirty containers on the screen. Note that this presentation describes a safe ordering algorithm, not a complete screen-updating algorithm. The issue of whitespace, which adds a considerable amount of complexity, is omitted from this description. It will be discussed in section 8.4.

This algorithm uses special terminology. The list of interest may be a vertical list or a horizontal list, so we will make no mention of directions such as "up," "down," "right," or "left." Rather, we refer to *low*, the direction of decreasing values of the principal coordinate, and *high*, its opposite direction. For the moment, the transverse direction is ignored.

The Update procedure is called with the following arguments:

C	the container, a box whose contents make up the list of interest.
base	the basepoint of this container as it should be.
oldbase	the basepoint of this container as it currently appears on the screen.

The **base** and **oldbase** arguments allow us to use the **disp** and **olddisp** fields and size information stored in each item to compute the old an new absolute screen rectangles for that item. Thus we know, in absolute terms, the direction of motion for each item.

What will it take to achieve a safe ordering? For clean items, the requirement is rather simple: a series of items that are moving in the *low* direction must be processed in low-to-high order, and a series of items that are moving in the *high* direction must be processed in high-to-low order. Items that have not moved at all need no processing. For new items, there is also a simple answer: render them last. Since they depend on no information already on the screen, no previous processing can interfere with them.

Dirty items bring in more complexity. They not only move; they may also shrink and expand. Items that expand must be updated late, lest they clobber their neighbors; items that shrink must be updated early, lest their neighbors clobber them. In the final analysis, the criterion is this: any item whose high edge moves low-ward or does not move can be updated before its successor in the list; otherwise it must be processed after it is updated. A new item is always dealt with after its successor has been updated.

Fig. 8-1 illustrates the safe ordering algorithm. Shading shows areas that are in danger of being overwritten too soon if an unsafe ordering is used. Arrows show the direction that each box has moved; numbers show the ordering chosen by our algorithm.

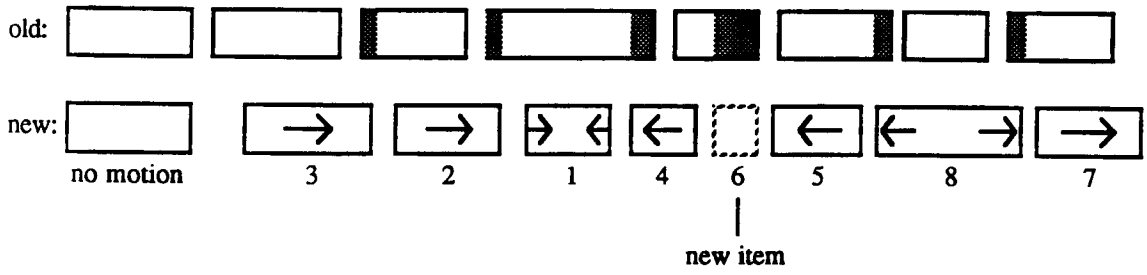


Fig 8-1 Safe ordering.

To keep the number of operations down, we group Clean items together when possible. Any run of adjacent Clean items that all move with the same offset are grouped into one *action block* and moved as a unit. Any other item is an action block unto itself.

The algorithm, then, proceeds thus: we traverse the item list in the usual (low-to-high) order, finding action blocks. As each action block is completed, we compute the fields of the action block record. These include **mark**, copied from the item; old and new screen rectangles for the block as a whole; and **himove**, the distance between the *high* edges of the new and old screen rectangles.

Then we decide how to act upon it. If **mark** = New or **himove** > 0, we push it on a stack to be processed later and keep on looking for more action blocks. Otherwise we process it immediately. After finding an item that can be processed immediately and processing it, we check the stack. If there are any action blocks on it, it is now safe to process them. One at a time, we pop them off the stack and process them, thus achieving the necessary high-to-low order. Thus the overall pattern of the algorithm is a low-to-high traversal, with embedded sections of high-to-low processing, each executed as it is reached.

In the processing of Dirty items, Update is called recursively. The outer invocation of Update guarantees that the inner invocation has been called at a safe time: all the source rectangles for the ScreenCopy calls will still contain their old contents, and all the target rectangles will be free for updating. Because Update takes **base** and **oldbase** in absolute screen coordinates and moves bits in two dimensions, the inner Update is actually doing the moving required by the outer Update as well as the moving required by its own computations. The result of all this is that during one screen update, no piece of screen is ever copied more than once.

A complication is added by the fact that the whole document is not (usually) displayed at once. Items may move into the viewport from positions partially or entirely outside it. These items are likely to be marked Clean, since they have not been created or modified during recent editing, but they have no complete representation on the screen. Some of the needed pixels do not exist. We solve this problem simply by testing for it; any such item is marked New, forcing it to be re-rendered from scratch. In Scroll Mode, where an item may dangle partially off the bottom of the viewport, this can cause some needless repainting of large items that move by a small amount. But this is not a serious problem, because the repainting only occurs when the item is drawn up farther onto the screen, not on the average keystroke.

8.3. The safe ordering algorithm

Here we present the algorithm formally, with a proof of correctness. This presentation is considerably simplified from reality, but it captures the essence of the algorithm for safe ordering in two dimensions. The most important simplification is that it considers a container to be completely filled by its components. Weakening that assumption allows undescribed white space, which is dealt with in the next section.

This algorithm resembles an algorithm for the one-dimensional case published by Knuth in [Knuth 73]. His algorithm, intended to solve a problem in memory management, does not deal with creation or destruction of elements, nor with the issue of whitespace.

Our algorithm involves two objects:

sc	The screen, a rectangular bitmap
dl	The display list, a tree of items rooted at *root*.

Its function is to modify the screen as specified by the display list. The display list is not modified.

At the outset the display list is in a freshly modified state, in which it describes two states of the screen: the old state, which is presently in effect, and the new state, which is to be achieved. Each item I stores the following information:

type(I)	Hbox, Vbox, or Leaf. Hbox and Vbox items have children, Leaf items do not.
mark(I)	Clean, Dirty, or New. Only Hbox and Vbox items can be Dirty.
rect(I)	I's *relative rectangle*, describing its shape and size. The origin of this rectangle is called the *basepoint* of I.
oldrect(I)	I's relative rectangle in the old state.
disp(I)	The *displacement vector* from the basepoint of I's container to the basepoint of I.
olddisp(I)	I's displacement vector in the old state.
count(I)	The number of children of I. Relevant only for Hbox and Vbox items.

The absolute coordinates of I's basepoint and rectangle can be computed:

$$\text{basepoint}(I) \equiv \sum_{j:\ I \text{ and its ancestors}} \text{disp}(j)$$
$$R(I) \equiv \text{rect}(I) + \text{basepoint}(I)$$
$$\text{oldbasepoint}(I) \equiv \sum_{j:\ I \text{ and its ancestors}} \text{olddisp}(j)$$
$$Ro(I) \equiv \text{oldrect}(I) + \text{oldbasepoint}(I)$$

That is, the basepoint of an item I is offset from the screen origin by the sum of the displacement vectors of I and all its ancestors.

An item of type Hbox or Vbox is called a *container*. The children of a container C are spatially disjoint and ordered in the *principal direction* defined by the type of C. We will use the shorthand:

$$\text{lo}(R) \equiv \text{if type}(C) = \text{Hbox then west}(R) \text{ else north}(R)$$
$$\text{hi}(R) \equiv \text{if type}(C) = \text{Hbox then east}(R) \text{ else south}(R)$$

and state the spatial relationship:

$$\forall_{i,\ j:\ 0 \le i < j < \text{count}(C)}$$
$$\text{hi}(Ro(\text{child}(C, i))) \le \text{lo}(Ro(\text{child}(C, j)))$$
$$\wedge\ \text{hi}(R(\text{child}(C, i))) \le \text{lo}(R(\text{child}(C, j)))$$

We also have the relationship of *spatial composition*:

$$R(C) = \bigcup_{i:\ \text{child of } C} R(i)$$

This is too strong for reality; it excludes the possibility of empty space in a container. In reality R(C) is a superset of the union of its chidren's rectangles. But this simplification does not alter the ordering relationships that are our current focus.

We will speak of a rectangle on the screen *containing* an array of pixels. For a rectangle R and an array of pixels P we define:

$$R \text{ c } P \equiv (P \text{ is congruent to } R) \wedge \forall_{p:\ \text{point in } R}\ sc[p] = P[p + nw(domain(P)) - nw(R)]$$

where nw(R) means the northwest corner of R.

Every item I describes an array of pixels P(I) to be drawn in R(I). We call P(I) the *proper pixels* of I. The screen is up to date when every item's rectangle contains its proper pixels:

$$\forall_{I \text{ in } sc}\ R(I) \text{ c } P(I)$$

The principle of spatial composition implies that the proper pixels of a container are completely described by its children. We will also speak of the *old pixels* Po(I) of an item I; this refers to the array of pixels described by I when the display list was in the state now described by the oldrect and olddisp fields.

This incremental screen updating algorithm presumes the existence of a non-incremental screen updating function, DrawItem. DrawItem updates the screen rectangle for an item I. Using Hoare's notation, its effect is:

$$\{\}\ \text{DrawItem}(I, \text{basepoint}(I))\ \{R(I) \text{ c } P(I)\}$$

(We are not interested in the results of applying DrawItem to a point other than basepoint(I).) Thus DrawItem(root, basepoint(root)) is sufficient to update the screen; our algorithm's purpose is to update it more efficiently. DrawItem is also required to have the property of *non-interference*:

$$\{R \text{ c } P \wedge R \cap R(I) = \varnothing\}\ \text{DrawItem}(I, \text{basepoint}(I))\ \{R \text{ c } P\}$$

We use another subroutine, ScreenCopy:

$$\{R_1 \text{ c } P\}\ \text{ScreenCopy}(R_1, R_2)\ \{R_2 \text{ c } P\}$$
$$\{R_1 \text{ c } P \wedge R_1 \cap R_2 = \varnothing\}\ \text{ScreenCopy}(R_3, R_2)\ \{R_1 \text{ c } P\}$$

8.3.1. The goal

We can now specify the desired behavior of our algorithm, Update:

$$\{Ro(I) \text{ c } Po(I)\}\ \text{Update}(I, \text{oldbasepoint}(I), \text{basepoint}(I))\ \{R(I) \text{ c } P(I)\}$$
$$\{R \text{ c } P \wedge R \cap R(I) = \varnothing\}\ \text{Update}(I, \text{oldbasepoint}(I), \text{basepoint}(I))\ \{R \text{ c } P\}$$

The first property is our goal of updating the screen. The second is, once again, non-interference—an Update of I must not change any pixels outside R(I).

8.3.2. The algorithm

Update operates by recursive descent through the display list. We will focus on one level, at which Update is applied to a container C:

Update(C, oldbasepoint(C), basepoint(C))

C has n = count(C) children, which make up an ordered list of items numbered from 0 to n-1. We will use the letters *i, j, k, l* somewhat ambiguously to refer both to the position of an item (*i*) and to the item itself (more formally *child(C, i)*).

This is the algorithm:

```
Update(C, oldbase, base) =
  n := count(C)
  k := 0
  for i := 0 to n-1 do
    R := rect(i) + base
    Ro := oldrect(i) + oldbase
    if (hi(R) ≤ hi(R0) and mark(i) ≠ New) or i = n-1 then
      for l := i downto k do
        DoItem(l, oldbase, base)
      end
      k := i+1
    end
  end

DoItem(i, oldbase, base) =
  if mark(i) = Clean then
    ScreenCopy(oldrect(i) + oldbase, rect(i) + base)
  elsif mark(i) = Dirty then
    Update(i, oldbase + olddisp(i), base + disp(i))
  else
    DrawItem(i, base + disp(i))
  end
```

Fig. 8-2 The safe ordering algorithm

8.3.3. Proof of correctness

Our initial condition is Ro(C) c Po(C). The progress of the main loop is described by the two variables i and k. At the head of the main loop, i marks the boundary between examined and unexamined items, and k marks the boundary between updated and non-updated items. The loop invariant is:

$0 \le k \le i$
$\forall_{j,\ 0 \le j < k}\ R(j)\ c\ P(j)$
$\forall_{j,\ k \le j < n}\ Ro(j)\ c\ Po(j)$
$\forall_{j,\ k \le j < i}\ hi(R(j)) > hi(Ro(j)) \ \vee\ mark(j) = New$

We prove this assuming the effectiveness and non-interference of DoItem:

$\{Ro(I) \subset Po(I)\}$ DoItem(I, oldbasepoint(I), basepoint(I)) $\{R(I) \subset P(I)\}$
$\{R \subset P \ \wedge \ R \cap R(I) = \varnothing\}$
DoItem(I, oldbasepoint(I), basepoint(I)) $\{R \subset P\}$

Let us examine one iteration of the loop. If the **if** condition is not satisfied, we know that $hi(R(i)) > hi(Ro(i)) \ \vee \ mark(i) = New$; we advance i but not k, and the invariant still holds. If the condition is satisfied, we will call DoItem(i, ...). Here we know (except on the last iteration) that $hi(R(i)) \leq hi(Ro(i))$. Correctness in this case depends upon the fact that the items are disjoint and spatially ordered:

$$\forall_{i,\ 0 \leq i < n}\ i < j \ \Rightarrow \ hi(Ro(i)) \leq lo(Ro(j)) \ \wedge \ hi(R(i)) \leq lo(R(j))$$

This means that $hi(R(i)) \leq lo(Ro(i+1))$, so by non-interference, DoItem(i) cannot harm the pixels of item i+1. On the other side, if $k < i$ there will be non-updated items lower than i. But if so, we already know that $hi(R(i-1)) > hi(Ro(i-1))$, hence $lo(R(i)) > hi(Ro(i-i))$, so DoItem(i) cannot harm the pixels of item i-1 either. In the special case where mark(i-1) = New, $Ro(i-1) = \varnothing$, and $Ro(i-i) \subset Po(i-1)$ is vacuously true and cannot be changed by any screen operation.

A similar argument assures the safety of each iteration of the inner loop. Its invariant is:

$0 \leq k \leq l \leq i$
$\forall_{j,\ 0 \leq j < k \ \vee \ l < j \leq i}\ R(j) \subset P(j)$
$\forall_{j,\ k \leq j \leq l \ \vee \ i < j < n}\ Ro(j) \subset Po(j)$
$\forall_{j,\ k \leq j < i}\ hi(R(j)) > hi(Ro(j)) \ \vee \ mark(j) = New$

At the end of the inner loop we reach the state

$\forall_{j,\ 0 \leq j \leq i}\ R(j) \subset P(j)$
$\forall_{j,\ i < j < n}\ Ro(j) \subset Po(j)$

Setting $k = i$+1, $i = i$+1, the outer loop invariant is restored. Since the inner loop is forced to execute on the last iteration of the outer loop, we finally end up with $\forall_{j,\ 0 \leq j < n}$ $R(j) \subset P(j)$, which implies our goal, $R(C) \subset P(C)$. Non-interference is easy to demonstrate: since DoItem is non-interfering, and is always targeted to a rectangle R(i) that is a subset of R(C), Update does not interfere with pixels outside R(C).

Now for the recursive nature of Update. DoItem(I, ...) is clearly correct in the case that I is a leaf, because then $mark(I) \neq Dirty$. It is also correct if Update is correct, and Update is correct if DoItem is correct. But since the recursion reduces the depth of the tree by one level, recursion must end at leaf items, so Update is indeed correct.

8.3.4. A note on implementation

Item lists are stored as singly-linked lists, so it is not really feasible to traverse them backward. Lilac actually uses a stack to store items that cannot be updated when first seen; the inner loop consists of popping items off the stack and updating them until the stack is empty.

The Update procedure is shown as having **oldbase** and **base** as arguments, even though they could have been computed from the **disp** fields of items. This is important for efficiency: to compute them on the spot would add a factor of the depth of the tree to the cost of the overall algorithm. Other efficiencies, not shown in this presentation, save space in item records. Since the items in a container are arranged along one dimension, we store only one coordinate each of **disp** and **olddisp**. This coordinate, already mentioned in Chapter 6, is called **offset**. The other is known to be zero, except in the case of **root**, which is specially handled. Also, we store **oldrect** only for Dirty items, which are relatively few since only containers can be Dirty. To save space in other item records, we store only an index in the item record; **oldrect** itself is found in a separate array of rectangles.

8.4. Whitespace

The boxes-and-glue model makes for a few complexities in the treatment of whitespace. Whitespace is not explicitly described by boxes and glue; it is just wherever ink isn't. There is glue—but glue describes the distance between boxes, not the absence of anything within that span. With the use of negative glue, it is possible to have boxes that exceed their "formal" boundaries, as described by their width, height, and depth. Where such boxes exist, they "slop over" into the glue that abuts them, making that glue rather useless as a descriptor of whitespace on the screen.

Besides that, containers often are not completely filled by their components. The rule of boxmaking says that the transverse size of a composite box is the maximum of the sizes of its components in that direction. Next to a component that is smaller in the transverse dimension, there is empty space. We refer to such spaces as *transverse margins*. Also, some boxes are made to a specified size in the principal direction. If the component boxes are too small to fill out that size and the component glue cannot be stretched to fill it, there is empty space left at the end. We refer to such spaces as *end margins*.

The algorithm of section 8.3 ignored the issues involving whitespace—each list was treated as a series of tightly packed components, each extending out to the full transverse extent of the container. Now we discuss how whitespace is actually handled.

Lilac treats whitespace as the boxes-and-glue model describes it: whitespace is wherever there is nothing else. For the white spaces that occur between items in a list, we determine where there is nothing else by examining the extents of the adjacent items. Actually, of the nearest adjacent *substantial* items—that is, those which might actually describe something to be drawn on the screen. Glue items are *insubstantial* items, and are ignored in screen update calculations.

The rule for whitespace is, if while doing the update computations we determine that a whitespace should exist, and we cannot verify that it is already white on the screen, we must clear it. Since there may be hundreds or even thousands of whitespaces on a screen, we need to be careful not to let that happen too often. For whitespace inside of new containers, this is simply handled: we clear the bounding box of the container before painting the container and its components.

For whitespace inside of existing structures, more care is needed. The first technique for avoiding excessive clearing of whitespace is to combine items into action blocks. As already mentioned, if several adjacent items are clean, and are moving with the same offset, we combine them into a single action block. Insubstantial items are skipped over and ignored in this process. Thus the action block describes a rectangle that includes all the items *as well as the whitespaces between them.* When this action block is moved, those whitespaces are moved with it, and thus are adequately taken care of.

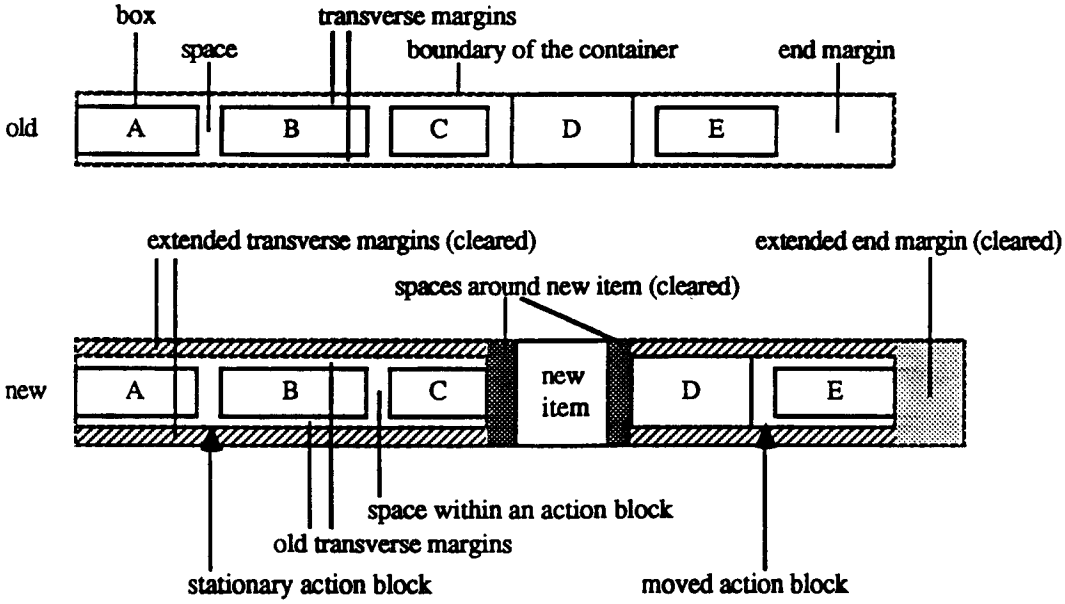


Fig. 8-3 Clearing white spaces

Transverse margins of clean items are similarly handled. The rule is, when copying, go to full transverse extent. Even if the item is smaller than its container in the transverse dimension, treat it as if it were not: copy the transverse margin with it. The only exception to this rule comes when the transverse size of the container itself is changing. Then we have to be careful: if the size is decreasing, copy only as much as will fit in the new size; if increasing, copy as usual and then clear the new portion of the transverse margin.

White spaces between items, when they cannot be scooped into an action block, are handled by the following rule: if any edge of the space has retreated, then clear it, else leave it alone. That is, the space is viewed as a rectangle, filling the gap between two items and extending out to the transverse extent of the container. If, during the update, any edge of this space has moved outward, then we clear the space. A simple movement without change of size still invokes this rule; it is cheaper to clear bits on the screen than to copy them. End margins are handled in the same fashion.

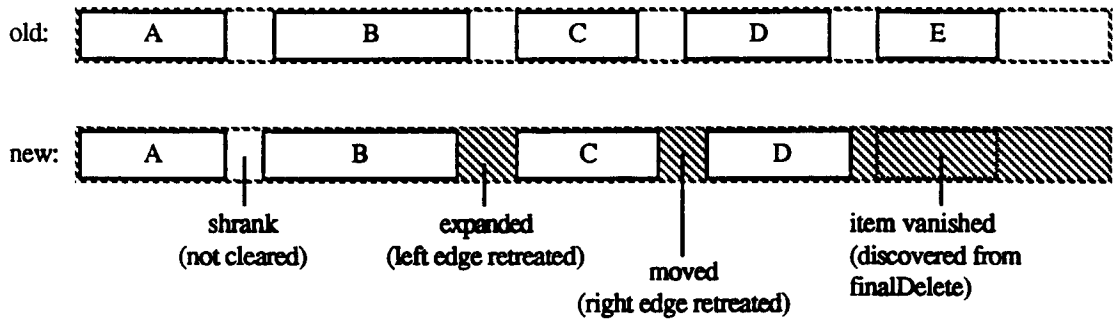


Fig. 8-4 Clearing spaces within a line

This well-optimized handling of whitespace leaves one serious problem: how do we detect deletions? Boxes and glue, suitably manipulated, can create a situation in which an item disappears from between two others without changing the distance between them at all. If we aren't careful, we will ignore the deletion and fail to clear the space, and the screen will be left incorrect. The deleted item is gone from the list, so it cannot remind us that it has disappeared. I found it necessary to add a couple of Boolean fields:

delBefore a field of each item, is true if one or more items were deleted immediately before it

finalDelete a field of each container, is true if one or more items were deleted at the end of its list of components

These fields are maintained during the rebuilding of the box tree as the syntax tree is reinterpreted. An item with **delBefore** = True forces a break between action blocks, and forces the space before that item to be cleared. A container with **finalDelete** = True always gets its end margin cleared.

8.5. Overlapping boxes

The possibility of negative glue means that items in a list may overlap on the screen. Though this is rare in practice, the possibility considerably complicates the task of updating the screen. Until now we have assumed that a box A, with a bounding rectangle R(A), completely describes the bits within that rectangle. When boxes overlap, that assumption fails. All the bits described by A lie within R(A), but not all the bits in R(A) are described by A. Some of them may be derived from other boxes whose bounding rectangles overlap R(A). The rule in such cases is that black ink dominates: if a given position is described as being black by any box, then that position will be black on the printed page.

This makes it much harder to keep a correct screen. If A is overlapped by another box B, and we clear or rearrange the pixels in R(A) in the process of updating A, then the pixels contributed by B are wiped out or, worse yet, displaced and scrambled. The result is an incorrect screen.

At one time I designed a solution to this problem, replete with auxiliary data structures and incremental algorithms to keep them correct. I have never implemented it, and by now I see little reason to do so. Actual overlap of ink (as opposed to overlap of white parts of boxes) is quite rare in practice, and is usually a mistake when it occurs. But the real deciding factor is that a full-screen refresh turns out to be so fast. Not fast enough to do on every keystroke, but usually faster than 1/2 second. So it works quite well to let the screen become incorrect when overlapping portions are edited. When the user needs to see the truth, a quick Refresh command, issued from the keyboard, does the job quite nicely.

8.6. Pagination

The display list, as described thus far, represents a long, unbroken vertical scroll of text. It naturally corresponds to the Scroll Mode style of display. But real documents come in pages, which are often decorated with marginal features such as running headers and page numbers. To get from the scroll to the pages requires a list-breaking process similar to the process of paragraph-building, plus a bit of work to put the marginal features in place.

The challenge of pagination is to maintain an accurate and stable view of the page as the display list changes during editing. A similar challenge applies in Scroll Mode—to maintain a stable view of some limited portion of the display list as the whole display list changes. In either mode, we will refer to the visible portion of the document as the *viewport.* The two cases are similarly handled; the basic strategy is to use an entity known as a vertical position pointer (Vpoint for short). This is a pointer that indicates the top of a page in relation to the full display list, and it stores the information in three forms:

- A pointer to the first item (top line of the page)
- The index of that item in the display list
- The vertical coordinate of that item in the display list, i.e., if there were really a long scroll, how far down it this item would be.

In Scroll Mode, we maintain knowledge of only one "page", namely the part of the document that is currently visible in the window. After an edit, we begin work at the Vpoint to the top of that page. We invoke a special form of the vertical box-building routine, one which includes as many items as will fit, until a certain height is met or exceeded. That height is determined by the height of the window. Having found as many items as it needs, it makes a box of them and returns it. This is the new top-level box for the page view.

Once in a while things don't work out that simply. An edit causes the deletion or replacement of the top item on the page, rendering our Vpoint's pointer invalid. It is critical that we detect this, or we will build a page box containing items that are no longer part of the display list. Here is where the index in the Vpoint proves useful. The main display list, like all other lists that get edited, has a change record telling which items were modified during the most recent edit. This information is expressed in terms of indices, so a quick comparison will tell us if our Vpoint is in the modified range, hence invalid.

In that case we have to re-establish a top-of-page point. At many times it proves convenient to keep a back pointer, a Vpoint roughly one page behind the top of the current page. If there is such a pointer, and if it is valid, we work forward from it; otherwise we work forward from the beginning of the display list. In either case, we now use the vertical coordinate in the Vpoint as our guide. The goal is to establish a new top-of-page at roughly the same position in the document scroll as the one that was invalidated. So we scan along the items of the display list, adding up heights, until we find a box at or near the desired position. There we set our new top-of-page Vpoint,

and page-building proceeds as usual. In practice this computation is fast enough that it never causes any disconcerting delays. It is seldom triggered except by large operations, which are expected to take a perceptible amount of time anyway.

Page Mode operation is a bit different. Here, instead of building a box to fit the window, we take a sublist according to the value of the **pageheight** global variable. The algorithm is just like the algorithm for line-breaking in the paragraph-building process—we may break just short of that height, or just long, whichever has fewer demerits. But here, in contrast to the line-breaking case, the job is not over yet. The Lilac function "PageModel," which the user may redefine in his document, is invoked to build the top-level box for the page. PageModel takes two arguments: the page number, and the *page content*—the sublist which we just computed. It can compute upon these in any fashion the Lilac language allows, provided that the final result is a box. That becomes the new top-level box for the page.

This is obviously more processing than Scroll Mode requires, and indeed it takes longer—not devastatingly longer, because PageModel is evaluated with the usual benefit of incremental primitives, but long enough that we offer the user no guarantee of speed in Page Mode. Page Mode is for previewing and for final adjustment of page breaks and appearance.

In Page Mode, as in Scroll Mode, we represent the top-of-page with a Vpoint, but here we maintain a list of them, one for each page. This makes it a very fast operation to flip through the pages of a document—it typically goes at a rate of 2 to 3 pages per second. During each edit, the change record of the main display list is used to build up a *dirty range*—a range of item indices within which Vpoints cannot be trusted. If the top of the current page ever falls within that range, we repaginate, starting from the nearest page that still can be trusted.

8.7. Repaginating at high speed

The above works reasonably and makes good sense, but when push came to shove, it left performance about 20% short of the goal. Surprisingly enough, that really makes a difference! When one is typing at full speed, a little slowness adds up fast, until soon the editor is many keystrokes behind, and the feeling of responsive editing is lost. Another special performance device was needed to make up the difference.

The crux of the matter is that the page is an unusually long box. It may well contain 100 items—50 lines and the glue between them—where few other boxes ever contain more than 20 to 30 items. The traversal required to rebuild this box is time-consuming.

Actually, it can often be avoided. The common keystroke adds a character to a line, but it does not change the height of that line or affect any other line. So the image geometry has changed only at the line level. And because of incremental boxing, that line is still represented by the same box record that represented it before. So in terms of information computed by boxing lines into a page, *nothing has changed at all!* If only we can detect this situation reliably, we can skip page-boxing entirely.

This is the algorithm:

If we are remaking the page, not making it afresh,
and there were no insertions or deletions in the top level of the display list
and only one box in the display list was modified
and it did not grow or shrink in the vertical dimension
Then we can skip page-boxing.

The first three conditions are easily tested: the first by the existence of a page box, the others by the change record on the main display list. The last condition, vertical growth or shrinkage, is harder to test because we do not normally store the former size of every box. But we can use locality: the line modified by one command is the line most likely to be modified next. So we can use a three-stage strategy similar to that used in list-building:

Level 1: Build the page from scratch.
Level 2: Ordinary incremental box-building. Use the same records, but recompute the geometry. Note which line was modified this time.
Level 3: Skip the job entirely, knowing that nothing of significance changed.

This works, but not quite as simply as it sounds: the screen update algorithm relies upon the geometrical change information stored in boxes. Consider what could happen after a change that made line 17 taller: Height has changed, so we can't use Level 3. We use Level 2, and note that line 17 was the one changed, and rebuild the page. All lines below line 17 move down a little bit, and all is well. Now the next keystroke. It changes line 17 but doesn't change its height, so Level 3 will work. We skip boxing. But alas! The lines below line 17, not having been re-boxed, still contain that geometrical change information that says, "move me down a little bit." And the unsuspecting screen updater moves them, and and keeps on moving them with each additional keystroke until they fall off the screen.

So we make the algorithm a little tighter. We will skip page-boxing only if there was no vertical motion this time *or last time either*. So the conditions of the first

algorithm become the conditions for recording an active line, which is to say, "Level 3 may be possible next time." When next time comes, if those conditions are still met and the modified line is the same as the recorded active line, then we skip page-boxing. Thus, when typing causes a new line to be added to a paragraph, the first two keystrokes on the new line (the one that created it, and the one after) lead to Level 2 boxing. At the third keystroke, we can return to Level 3.

How does this feel in practice? Perfectly fine. This is the fact upon which all the speed devices in Lilac depend: a little occasional slowness does not matter. The user will seldom notice that one character was echoed a little late. It is *cumulative* slowness, leading to echoing that falls farther and farther behind the typed input, that is to be avoided at all costs.

9. The Source View

The source view was much easier to implement than the page view, because it deals in a very well-understood medium—plain text. However, it is not just plain text; it is text that is tied into a larger data structure and has to reflect events taking place in the page view. This has led to a few implementation challenges worthy of mention.

9.1. Displaying from a tree

The source view looks like ordinary text, but it is not ordinary text underneath; it is text whose underlying representation is a syntax tree. The two are closely related: the text is derived from the syntax tree by unparsing; the syntax tree could be rederived from the text by parsing.

In the beginning I tried to drive the textual display by unparsing whenever characters were needed to update or refresh the display. This proved to be a failure. I could not make an unparser fast enough to meet the needs of a text editor, and the attempt produced a painfully complicated unparser. So I turned to a strategy of storing auxiliary text alongside of the syntax tree. This is done at the level of major syntactic units. The document header is one such syntactic unit; each unit or function definition is another. Each such syntactic unit may optionally have its unparsed text attached. The strategy is still based on unparsing: whenever the display code needs characters, it turns to the unparser to get them. But whenever such a request refers to a major syntactic unit which has no text attached, we unparse it and then save that text for faster access in the future.

A note is in order here: I did not write all the code that manages the source view. Low-level management of the text display is handled by the VText module, which I borrowed from an ordinary text editor written here at the Systems Research Center. Thanks are due to John DeTreville, who is chiefly responsible for this module and has made changes to help my work on a couple of occasions.

9.2. Reflected selections

The source view shows reflected selections: whenever a selection is highlighted in the page view, the corresponding part of the document program is highlighted in the source view. This is different from an ordinary selection, and it is shown to the user in a different way. Ordinary selections in the source view, made by pointing and

clicking with the mouse, look just like page-view selections: they are highlighted in inverse video. Reflected selections, in contrast, are highlighted by underlining. This is a less obtrusive highlight; it is less distracting in the user's peripheral vision, which from experience seems to be an important consideration. The contrast also has a useful meaning: it makes it clear that the source view is not the true focus of editing, although it is showing a reflected selection and will reflect changes. The distinction is important because editing does not have exactly the same meaning in the two views.

```
    Chapter("A Chapter Title",
      Section("A Section Title",
        Para(
    "This sample document
demonstrates the structure of
Lilac documents."),
        Para("This paragraph
shows ",
    Italic("italic"), " and ",
```

A native source view selection

```
    Chapter("A Chapter Title",
      Section("A Section Title",
        Para(
    "This sample document
demonstrates the structure of
Lilac documents."),
        Para("This paragraph
shows ",
    Italic("italic"), " and ",
```

A reflected selection in the source view

Fig. 9-1 Native and reflected selections

Reflected selections add an aspect to the mapping problem. Not only do we have to map from a node in the syntax tree to an area in the page view, we also have to map it to a span of characters in the source view. How is that to be accomplished?

We accomplish that by turning the source-view character count into a fundamental measuring aspect of the syntax tree. Every node in the syntax tree contains a **count** field, an integer which tells the number of source-language characters that would be obtained by unparsing that node. The count of a parent node is the sum of its children's counts plus the number of characters contributed directly by the parent.

Using these **count** fields we can relate any node to its *index*—the number of characters that precede its representation in the source view. To compute a node's index or to find a node from its index takes time of order O(d * b), where d is the depth of the

tree down to our node and b is the average branching factor encountered along the way. These computations make use of an additional node field, **filler**, which tells how many parent-level characters precede the first character representing that node. An example will be instructive.

Consider the expression: foo(alpha, beta)

If for a moment we pretend that it is a whole document, its syntax tree consists of the following nodes:

foo	count = 16, representing the characters "foo(alpha, beta)"
	filler = 0
alpha	count = 5, characters "alpha"
	filler = 4, representing characters "foo("
beta	count = 4, characters "beta"
	filler = 2, characters ", " (comma, space)

The index of node **foo** is 0, because it starts at the beginning of the "document". The index of node **alpha** is 4, its filler. The index of node **beta** is 11:

alpha.filler + alpha.count + beta.filler = index
4 + 5 + 2 = 11

These indices are used to represent selections. Most selections are actually stored in multiple representations for ease of computation, but one of those representations is always an interval of indices. (Nodes are not the only things that have indices; we compute indices for characters within a string node as well.) This is exactly what is needed for reflected selections! We simply underline the source-view characters within that range, and the job is done.

9.3. Prettyprinting

Early experience showed that a source view is useless unless attention is paid to the formatting of the program displayed in it. This formatting has to be automatic, because most of the program is never directly touched by the user—it has been created by his actions in the page view. If it is not in a legible format he will never take the time to make it legible; it simply will not be useful. So I chose to use an adaptive prettyprinting scheme: if an expression will fit on one line, fit it; if it won't, treat its subexpressions as independent units and fit them on multiple lines. Wherever a subexpression begins a new line, indent it two spaces inward from the indentation of its parent. The breaking of lines depends on the current width of the source view window. (In addition to these

adaptive breaks there are also some unconditional breaks, places in the syntax that always begin a new line.)

This prettyprinting task is a bit different from usual. The underlying representation of the source view is not text—it is a syntax tree. To prettyprint the text we cannot just add linefeeds and spaces; we need to modify the tree in some fashion that will cause it to unparse with linefeeds and spaces in the desired places. So this prettyprinter uses the tree, rather than a stream of characters, as the basis of all its computations. Four fields in each node are involved in the process:

count	As described above; the total number of source-view characters representing this node and its descendants.
flat	The count as it would be if this node were unparsed all on one line, without newlines or indentation.
filler	The number of parent-level characters, including spaces and linefeeds added by prettyprinting, that directly precede this node.
breakHere	The number of space and linefeed characters just before this node, added by prettyprinting.

The **breakHere** field stores the prettyprinting information. If zero, it means that no break was inserted before the node. If its value is some n > 0, it implies a break, and represents one linefeed and n-1 spaces. By this convention a single field represents both the line break and the level of indentation. The **flat** field is computed before the prettyprint computation begins; it allows us easily to determine whether an expression will fit on one line. The purpose of the **filler** field is to assist in the computation of indices, as described in Section 1. When prettyprinting creates a new break, we add the value of **breakHere** to **filler** to keep it consistent with this purpose.

Once this prettyprinting computation has been done, the actual task of prettyprinting is a matter of unparsing—with an unparser whose behavior is modified by the values of **breakHere** fields.

9.4. Reflected changes

Whenever the document changes as a result of page view editing the source view changes to match (though this behavior may be suppressed at the user's request) . This means that structural changes in the syntax tree need to be accompanied by appropriate textual changes in the source view.

We accomplish this by getting the page view editor to pass information to the source view manager. Each editing action can be summed up in terms of one node, the common ancestor of all that was inserted or deleted. So we get the page view editor to report this node, together with the values of its **count** field before and after the change. This allows us to locate the old characters that belonged to this node and replace them with the new ones. We derive the new characters by unparsing the node. This is actually a bit of overkill; the common parent may cover a much larger span of text than the part actually edited. But it doesn't matter. Ordinary character edits are the vast majority of all editing commands; all structural edits are quite rare by comparison, and their performance is not critical.

Actually there is another step in the process: maintaining a prettyprinted view. A newly inserted node has never had a prettyprint computation done on it; it would unparse flat, as one long line. If we allowed this to go on unchecked, we would soon have an unformatted and unreadable source view. So when the page view editor reports a modified node, we run the prettyprint algorithm over it before unparsing it to get the new text.

For simple character insertions and deletions, of course, time is critical. Here, as everywhere else in the system, we store extra information to create a "fast path" for this simple common case. Character edits have the convenient property that they never change the structure of the syntax tree; they simply add characters to a string node somewhere in that tree. So when a character insertion or deletion command is given, we record the identity of the string node being modified and compute and store its index. On later modifications to the same string node, we can use that index to quickly find the point of modification in the source view. Index computation is avoided, as are unparsing and prettyprinting.

9.5. Source view editing

When the user edits in the source view itself, it is no longer possible to maintain an exact correspondence with the syntax tree. As the user types the source view passes through syntactically invalid states, which the tree could not represent. So at such times we let go of that connection and let the source view be represented by plain text.

This takes place, usually, at the granularity of an expression. When the user invokes a source view editing operation, we find a suitable expression and make this transformation. From the index of the insertion point, or the indexes of the two ends of a selected range, we determine the lowest-level expression that completely includes

the text being modified. We transform this expression's node into a TextNode, a special type of node which cannot normally occur in syntax trees. It now has the same characters, but no syntactic substructure; it has been "flattened" into an undifferentiated sequence of characters.

As the user edits at different places, the tree may acquire any number of TextNodes. Whenever a TextNode is made, we mark its ancestors Dirty, leaving a trail that can be used to easily find all TextNodes later. When TextNodes are present, the document as a whole enters the state NeedsReparsing. In this state the page view does not change, nor can it be edited.

Actually, expression-level transformation occurs only within units. With other major syntactic blocks (the header, functions, comments) we transform the whole block into a TextNode, since these tend to be small. An editing operation that spans more than one unit will result in whole spans of units being transformed into TextNodes, also.

When at last the user believes that he is finished editing and wants to see the results, he gives the **Reparse** command. This can be invoked from the source view menu; a click in the page view will also trigger it. Reparse scans the syntax tree at top level, looking for TextNodes and units marked Dirty. In dirty units it descends down the tree to find all TextNodes present. Each TextNode is parsed and replaced with the resulting parse tree. At last, if there have been no errors, the syntax tree is back in an interpretable state, and we change the document state from NeedsReparsing to Consistent. Then we can re-evaluate the main unit. Now the Dirty marks function just as they do in page view editing: they show the incremental interpreter which expressions have undergone some change. It is just as if each TextNode had been the target of a Paste command, simultaneously. Depending on the size of the TextNodes involved, a Reparse may take a fraction of a second or many seconds.

As usual, nothing is as simple as it first appears. Reparsing is complicated by the fact that source view editing may transform one expression into several, or several expressions into one. This can result in TextNodes that are syntactically invalid within themselves, although the unit as a whole is syntactically correct. So we adopt several strategies to deal with apparent syntax errors:

1. If a TextNode seems to contain more than one expression, we try reparsing its parent instead. This will deal with the case where one expression was transformed into several.
2. We keep track of the common ancestor of all TextNodes within a given unit. If parsing any TextNode leads to a syntax error, we try reparsing this ancestor.

This will deal with the case where a pair of parentheses were added, transforming several expressions into one.

3. If for any reason that fails, we try reparsing the whole unit. If that fails, we have a genuine syntax error and we complain to the user.

Once reparsing and reinterpretation are complete, the new state is reflected in the page view. This is a fundamental asymmetry in Lilac: the source view tracks the page view at every moment; the page view tracks the source view only on command, and not always quickly. This is because the page view editor forces syntactic correctness at every step, while the source view editor ignores it. Syntax-directed program editors exist, and it might be an interesting experiment to build a source view editor on such a basis. But I have heard few reports of great satisfaction from the users of such editors.

9.6. Editing definitions

The source view allows something that the page view does not: editing of definitions. Both default variable values and function definitions may be added, deleted, or modified. This raises a question: how much to reinterpret? When the user edits a function body no part of the mainline document has been modified or marked Dirty. And yet, the change may have effects in many parts of the document—not only in nodes that call the modified function, but possibly in nodes that call other functions that call it. To track down all such effects accurately is quite a difficult problem.

In search of a simpler solution we quickly come upon an obvious one: reinterpret everything. This is easy to do, but in practice it takes too long—approximately one second per page in the document, in my experience. This makes function editing a bitter pill to swallow. The system begins to take on the feel of a batch-mode document formatter; users will go out of their way to avoid it. Looking further, we find a second easy answer: reinterpret nothing. This has the advantage of completing quickly (faster than one can blink, for most small procedures) and the disadvantage that it leaves the page view incorrect. In fact the page view gradually becomes correct, in an interesting way. When the user first edits in any descendant of a node whose styling was changed by the change to the function, that causes that node to be marked Dirty, which is all that is required. The node suddenly takes on its new correct appearance, sometimes responding to a simple keystroke with a startling jolt.

The solution I chose is to let the user tell us what he wants. By default we reinterpret nothing. The user may, however, select some part of the document and declare it to be the "region of interest." On any Reparse that involves non-mainstream

(i.e., function or default variable value) material, this expression is reinterpreted, its ancestors are incrementally reinterpreted, and the rest of the document is left alone. This is a compromise. It gives the user an optimally fast edit-and-test cycle for debugging new functions. It gives him correct appearance where he needs it most. It does not give him a guarantee of full correctness. He can have that back, when he is willing to wait 20 seconds for it, by invoking the Recompute command, which forces a reinterpretation of everything. Or he can declare the whole document to be the region of interest, to get full correctness automatically on every Reparse.

10. Non-hierarchical Relationships

10.1. The problem

The most important limitation of the design described thus far is that it makes no provision for the non-hierarchical relationships that occur in a document. These are the relationships that cut across hierarchical lines: cross-references, footnotes, end-notes, floating figures, figure and table numberings, bibliographical entries, and the like. While these relationships are far less significant or common than the basic hierarchical relationships, they are still important. If they are not correctly modelled by the document preparation system, the user ends up having to simulate them with inadequate tools, and they become extremely hard to maintain.

Non-hierarchical features split into two major groups: *structure-based* features and *page-based* features. A good example of a structure-based feature is figure numberings. Figures in a book are commonly numbered sequentially within chapters: fig. 3-1, fig. 3-2, fig. 3-3, etc. This numbering is based on a high-level structure in the hierarchy: the chapter. It starts afresh at every chapter boundary. At the same time, it cuts across lower levels of the hierarchy. Whatever section boundaries may separate the figures, it doesn't matter; their numbering is sequential within the chapter.

A good example of a page-based feature is footnotes. They bear no relationship to the logical document structure whatsoever. Their organization is based on the *physical* structure of the document—its pagination. Starting afresh on each page, footnotes are gathered up as their points of origin appear on the page, and are deposited at the bottom. Their numbering (in most books) works the same way: it begins afresh on every page and proceeds sequentially within a page.

Several of these features come under the category which I will call *asides*. That is, an object of one of these types has a position in the main stream of the text, but it also sets aside some material which will appear at a later place in the document. Its position in the main stream is called the *anchor*. This may have a visible representative, such as the raised number for a footnote, or it may be invisible.

Now we can classify non-hierarchical features. Footnotes are page-based asides. So are floating figures (which "float" to the top or the bottom of the page where they are anchored) and full-page figures (which precede or follow the page where they are anchored). End-notes are structure-based asides, based on the chapter (or sometimes, on the whole document). Bibliographic entries are structure-based asides, based on the whole document. Figure and table numberings are structure-based *counters*, usually

based on the chapter. Cross-references are something a bit different, and will be discussed later.

The necessity of supporting these features in any serious word processor should be quite evident. For a particularly bad case, consider footnotes. The user can insert them by manual means—put a superscript number at the anchor point, and manually put a dividing line at the bottom of the page and the footnote below it. Of course, getting the page break just right may be a bit of a challenge—and if the page break happens to split a paragraph, he will have to manually cut it into two. But the real problem arises when he wants to change the text. The carefully devised footnote section slides across the page boundary and onto the next page, where it is totally out of place. Such contrivances are nearly impossible to maintain. The system must be set up to do the work.

10.2. A proposed solution

I have designed a solution for these problems and built an exploratory implementation for part of it; it illustrates the feasibility, though the performance is not yet satisfactory. Actually, there are two different solutions, one for page-based features and one for structure-based features, since their practical requirements are so different. For both problems, I introduce some new objects into the language: aside-classes. These resemble environment variables, but they are not exactly variables, because they are not given values. Rather, they "gather" their values. There should be one such object for each non-hierarchical relationship being dealt with.

There are three types of these objects: **AsideClass**, for dealing with structure-based asides, **Counter**, for dealing with structure-based counters, and **PageAsideClass**, for dealing with page-based asides. Sample declarations for them are:

```
declare
  endNotes:  AsideClass
  figNumber: Counter
  footNotes: PageAsideClass(10, 1)
```

Each AsideClass or PageAsideClass is a hidden receptacle in which vertical list material is gathered. Each Counter is a numeric variable which is incremented each time its value is used. The numbers associated with the PageAsideClass will be explained in section 4.

The **Aside** and **PageAside** primitive operations set material aside in the hidden receptacle for a specified class:

```
Aside(class: AsideClass, anchor: @Hlist, content: @Vlist): Hlist
```

For example:

```
Aside(endNotes, "anchor", Para("The contents of the note"))
```

This call to Aside adds its third argument, a paragraph containing the words "The contents of the note" to the endNotes class. It returns the horizontal list described by its second argument, i.e., "anchor". Both these arguments are unevaluated; within them, a special environment variable, **asidecounter**, is available. Its value is the index of this entry in its class: 0 for the first entry, 1 for the second, etc. Thus we can have numbered end-notes, as in this EndNote function:

```
function EndNote(note*: Hlist) =
  Aside(endNotes, SuperScript(ShowNum(asidecounter+1)),
    Para(SuperScript(ShowNum(asidecounter+1)), note))
```

This puts a superscript note number in the mainstream, and in the aside-receptacle a superscript note number followed by the note itself. The contents of the receptacle are finally *reaped*, or brought into the main stream, by the AsideContent or PageAside-Content primitives. End-notes can be brought in at the end of a chapter by:

```
AsideContent(endNotes)
```

The output of this function is a vertical list composed of all the material put in by various calls to Aside(endNotes, ...).

PageAside and PageAsideContent are used just like Aside and AsideContent, except that PageAsideContent should appear only in the PageModel function. Counters are like Asides except that they have no content and no content-retrieval function. The function

```
Count(counter)
```

retrieves the current value of the counter and increments it.

10.3. Implementation

The design described above is all very well for sequential processing, as in a batch-style document formatter. But how are these relationships to be maintained in the midst of editing? What will keep the numberings from becoming misnumbered as things are inserted or deleted? Some additional data structures are needed.

One major part of the problem is recognizing the *scope* to which an aside-type entry belongs. That is, what else is being gathered or counted along with it? For sequential processing, we could simply state: a scope is everything between one call to Aside-Content and the next. For incremental processing, a span of this sort would be very hard to keep track of. Nor is it necessary. As already discussed, many non-hierarchical

features still have scopes that are rooted in the hierarchy. End-notes have the chapter as their scope; bibliographical entries have the document as their scope. So we use a mechanism similar to the **let** mechanism for global variables:

```
scope endNotes in chapterContent
```

Thus scopes are compelled by the syntax to be well-nested within the hierarchy. When execution enters a **scope** expression, a new scope is opened for the aside-class in question. If it is a Counter, it is set to zero; if it is an AsideClass, it is initialized empty. A call to AsideContent should come within the scope, but after all Aside entries for that class.

Now we have something to attach a data structure to. Each aside-class is represented as a variable, with one slot of storage assigned. When incremental interpretation enters a **scope** expression, it picks up a *scope record* (which is found in the Store) and, like a **let** expression, assigns it to that variable. Arriving at any Aside or AsideContent call, we can retrieve this record from the variable and use it to trace the relationships of this call to others within its scope.

What does a scope record contain? Primarily a pointer to an ordered, linked list of *aside records*, one for each relevant call within the scope. Each of these records contains:

- The counter value at this entry, equal to the number of entries prior to it in this scope.
- The *label* at this entry—the node in some unit which is most directly responsible for this entry. (This is the same responsibility mechanism used for selections; the label is the node that would be selected if an item generated at this point were picked with the mouse.)
- For AsideClass entries, the content of this entry—the material that was set aside at this point.

A scope record for an AsideClass also contains the label of the AsideContent call for that scope.

This provides the needed information to handle incremental editing. If the contents of an Aside entry are edited, the incremental Aside function retrieves the appropriate aside record from the Store and updates the content field therein. Then, retrieving the appropriate scope record from the environment, it finds the AsideContent label and marks that node and its ancestors Dirty. Thus incremental interpretation will proceed to two places: the Aside call (where actually nothing of consequence has happened) and the AsideContent call, where the modification actually appears. The AsideContent

function, when incrementally invoked, performs an incremental MakeList with the content fields of all the aside records in the scope as its argument. The output is the combined list of all material set aside in this scope.

Counters have no content to be edited, but they share with AsideClasses the problem of new entries. A new entry, inserted in the middle of a scope, causes all the counter values of entries after it to be wrong. It also creates a new aside record which needs to be linked into the list. To find its place in the list, we use the fact that a node can be mapped to its index (the position of its first character in the source view) quite rapidly. So doing this for the label of the new entry and for the label of each entry in the existing list, we find its place by linear search and insert it. Now, for each entry after the point of insertion, we find its label node and mark it and its ancestors Dirty. Incremental interpretation will now proceed in many directions, and will be unusually time-consuming—but when it is finished, the entire scope will have a new and correct numbering.

Deletion is a somewhat harder problem. When the label node for an entry is deleted, that entry has been deleted, and it should be removed from the linked list. But how are we to know that there was a deletion? We don't want to traverse every open scope on every editing operation to check for this—that would be too costly. Perhaps this is a place where numberings should be allowed to become temporarily incorrect—with a user command available to force the necessary traversal and renumbering. An alternative would be to reserve a bit in each node to indicate whether it is the label of an aside feature. Only the deletion of one of these nodes, a relatively infrequent operation, would trigger a renumbering computation.

10.4. Page-based features

Page-based features (PageAsideClasses) require a quite different implementation, because their scopes are differently organized. The scope of a PageAsideClass is always a page—a structure which bears little or no relation to the logical structure of a document. Indeed, it is a very transient structure. While material seldom moves from one chapter to another, editing is constantly moving material from one page to another. Among the moved material may be anchors, whose movement changes the number of entries in the scope (the current page). This may in turn change the outcome of the page-breaking process, as the amount of aside-content to be displayed on the page increases or decreases. Thus we need a way to cheaply discover the set of entries in the page and to detect any changes in that set.

The only good solution I can see is to attach new information to items. Computing the set of aside-entries on the page must be an integral part of the page-breaking process itself, since aside-contents as well as mainstream text take up space on the page and must influence page-breaking decisions. Thus the solution is to attach new information to items, and to propagate it through the box-building process.

The PageAside primitive closely resembles Aside:

```
PageAside(class: PageAsideClass, anchor: @Hlist,
          content: @Vlist): Hlist
```

But the **anchor** argument takes on new significance. PageAside returns its **anchor** argument—evaluated, and with one new item added: an Anchor item, which contains a pointer to the aside record for this entry. Whenever a container is built, the aside records from any Anchor items or other containers in it are gathered into a linked list, in the order of their appearance in the item list. These lists, like ordinary item lists, are counted lists, so that they can be incrementally reconstructed without unnecessary linking.

The result of this work is that when the main Vlist has been constructed, its items contain pointers to linked lists that include all the PageAside entries in the document. As we traverse this list at page-breaking time, each line brings in with it all the PageAside entries that are anchored within it. Now we are in a position to do page-breaking correctly with page-related asides included.

The algorithm is as follows. As each item is considered, we add its height into the counter that is keeping track of the height of the growing page. If our item has any PageAside entries attached to it, we also add in the height of those entries. As before, we proceed in this fashion until we come to a legal break whose height is longer than the desired page height, then compare its badness with the badness of the previous break, and choose between the two.

This provides an answer (though a less than perfect answer) to the perplexing problem of a footnote anchored near the bottom of the page. The problem is, if we include the footnote, it takes up space and moves its anchoring line to the next page—and then it is not appropriate to include it. But if we do not include it, there is room for its anchoring line on this page—and we should include it! But by this design, the anchoring line and the footnote are considered as one unit, take it or leave it. And the result will almost always be to leave it to the next page, leaving more than the usual white space at the bottom of this page. The user unsatisfied with this result is invited to improve upon it by manual intervention. (An even more elegant solution would

involved dividing a footnote between pages, but that is a very difficult problem in this context, and I will leave it to future research to solve.)

There are a couple of complications to the process of adding aside-material into the computed height of a page. One is that PageAside classes, such as footnotes, are likely to produce different behavior depending on whether they are empty or nonempty on that page. A dividing rule, perhaps, or extra white space between the main text and the footnotes. Another complication is that footnotes may be divided in columns; thus the set-aside material contributes only a fraction of its height to the actual height of the page. These issues are handled in the PageAsideClass declaration:

```
declare
  classname: PageAsideClass(extra, factor)
```

The **extra** parameter is extra height added to the page height when the first entry in this class is encountered; the factor is multiplied by the height of an entry in this class to determine the height added to the page for that entry.

For good results, these numbers must match the typography actually done by the PageModel. If the footnotes class is declared with extra=10, PageModel should in fact insert 10 points of vertical space (or other divider) between the main text and the first footnote. If the footnotes class is declared with a height factor of 1, PageModel should call PageAsideContent in a context where its output will contribute in a one-for-one manner to the height of the page. (And if PageModel omits a given class, that class may be declared (0, 0) to exclude it from page-breaking computations.) Class declarations tell the page-breaker when to break the page; PageModel decides how to lay out the results. Lack of coordination results in an oversized page or excessive white space at the bottom.

By its very nature, the PageAsideContent function belongs only in the PageModel, not in any part of the mainstream text. This means that in Scroll Mode, when PageModel is ignored, the contents of PageAside entries are invisible and cannot be selected or edited from the page view.

10.5. Page-based features: incrementality and numbering

Once again, the issues of incrementality arise: how to detect a change in the set of PageAside entries on the page, and how to react to it appropriately. We are helped by the fact that only one page is on display at a time. We do not have to consider the existence of multiple scopes for PageAside classes. The present page is the only scope that matters at a given time.

Within that scope, we need to be able to detect changes due to insertion, deletion, or changes in page-breaking that move entries onto or off the page. To assist with this, we maintain a secondary linking of PageAside entries. Not only are they linked into one main linked list for all entries on the page, they are also linked into a list for each class. For each class there is a scope record, containing a pointer to this list and a count of entries in it. As page-breaking takes place, these lists are rebuilt and checked as each item is included into the page. Change is discovered in two ways. Insertions, replacements, and arrivals from off the page are discovered by the fact that a new entry shows up, needing to be linked into the list. Deletions and departures from the page are discovered by the fact that the count for a class is no longer what it used to be. In either case, the usual incremental MakeList performed by PageAsideContent is replaced by a full, regular MakeList to reconstruct the class contents.

Numbering of PageAsides is very tricky. The numbering of footnotes, for example, cannot be determined until page breaks are known. But page breaks cannot be determined until items have been created, since page breaking depends on the dimensions of items in the main Vlist. But once items have been created, it is too late to go back and change the number on the footnote. Not all numbers have the same physical size; the size of the number might affect a line-breaking decision and thereby the height of a paragraph.

In fact, to keep footnote numbers correct requires a two-pass algorithm. The first pass involves the normal reinterpretation that takes place after any editing action. At the end of this reinterpretation, page-breaking takes place. During page-breaking we can discover if the membership of any PageAsideClasses has changed. If so, we begin the second pass.

Each aside record for a given class stores the most recent value of the entry counter for that class. This value has been corrected during page-breaking. Now, for each aside record in a class whose membership has changed, we find the label node and mark it and its ancestors Dirty. Once this has been done for all modified classes, we invoke the incremental interpreter a second time. All the relevant PageAside calls are reinterpreted, and they draw the value of the **asidecounter** variable from the value stored in the aside record. Thus, this time, the numbers come out correct. This second pass can be a relatively large incremental interpretation job, and somewhat time-consuming. Fortunately, it does not happen very often. Footnotes are rare in most documents, and events that change the set of footnotes on a page are quite rare compared to keystrokes. However, the user should be given the option of turning off this second pass, in which

case he gains speed in exchange for tolerating incorrect numbers on his footnotes. This option is quite desirable when one is scanning rapidly through the pages of a document.

I have done a pilot implementation of the PageAside feature, and it works quite adequately. At this writing, it needs more performance tuning before it can be included in the working system. Ordinary editing works quite smoothly within footnote material, when the editor is in Page Mode where such material can be seen. The Promote command gives an interesting result: a sequence of promotions starting in a footnote proceeds as usual until the whole footnote content is selected. Then, on the next promotion, the selection vanishes from the footnote and appears on the footnote number within the main text—the visible anchor. This is the point at which the footnote construct as a whole is selected, and a Cut would cause this footnote to vanish from the list. Further promotions select higher levels of structure in the main text, in the usual fashion.

10.6. Other non-hierarchical features

For some features I do not yet have a design, though I believe that one can certainly be provided by mechanisms similar to the ones described here. One of these is cross-references. Here something (typically a chapter, section, or page number) is set aside at one point for use at another. In constrast to the Aside design, the point of retrieval may precede the point of setting-aside. Two-pass computation is required to get correct values for these entries. Indeed, in the case of page numbers there is the remote possibility of an oscillating situation: a page reference number, by its width, causes a change in page-breaking computations which changes the number of the page being referenced, which changes the width of the reference, which changes the number of the page being referenced, etc. I have yet to hear of a word processor that attempted to handle this case.

Full-page figures are another complexity. I assume that for best maintenance of a document, they should be anchored to the related text, rather than being placed at a specified, numbered page. This, then is a variation of the PageAside situation: an entry is anchored on one page, but must appear on another (ideally, the facing page, when printing on both sides of the paper). This will require some modification of the pagination data structure, which now contains only the Vpoint indicating the top of each page.

Indexes, bibliographies, and glossaries are another issue. In preparing this latest version of this report, I added an index-building facility to Lilac, but it is not an automatic, WYSIWYG process. The user adds index entries to the document, leaving

invisible anchors (visible in the source view) in the text. A special command causes Lilac to write an external data file which records the content and page of each anchor. Then a separate, batch-mode program (which typically takes a few seconds) can be run to sort and organize this data into index entries. It produces the index as a new Lilac file, which the user can then open in Lilac to see the results.

This method can easily be applied to automatically produce glossaries, bibliographies, and tables of contents and lists of figures with page numbers. The one primitive function used to implement indexing, ExtAnchor (external data anchor), will suffice to produce anchors for all these needs as well. No further modifications to Lilac are needed, though a new external processing program needs to be written for each such feature. This method lacks some of the glories of instantaneous WYSIWYG editing, but it can handle multifile documents without difficulty, and its speed seems quite adequate in relation to the frequency with which it needs to be done.

11. Future Directions

The previous chapter presented detailed designs for a major area where I believe this research should continue. Beyond this, there are many other extensions and improvements which could usefully be made to Lilac. Some of these have to do with building a fully functional typesetting program; others have to do with efficiency and ease of use; others are explorations of ways to make Lilac yet more powerful. Beyond these, there are concepts and methods explored in Lilac which could fruitfully be put to use in other problem domains. This chapter discusses all of these possibilites.

11.1. A programmable page view editor

Of all the enhancements which could be made to Lilac, the one I would most like to explore is to make the page view editor programmable. Note the distinction: the document structure is already programmable—the user can create new structural elements and describe their styling, based on primitive elements and styling features. But I would also like to make the *editor* programmable, so that the user could create new *interactive commands* based on existing commands and other appropriate primitives. This would probably mean introducing a second language into the system, entirely different from the document description language.

This would be a very natural and appropriate enhancement to the generic hierarchical editor that is currently in place. The existing editor is reasonably good at handling a wide variety of structures, but in some cases it is not great. One such case involves outlining: in an outline, it is common that some headings at a given level have subheadings and some do not. In Lilac, if the user hits Return while working in a heading that has no subheading, he will get a new heading that also has no subheading. Adding that subheading is not so easy. It involves getting an insertion point into the empty list of subheadings (a minor difficulty) and then inserting a fresh subheading. Lilac does not know what to insert there; the user must specify it, either by Copy or by an escape command. This process would be much more convenient if the user could program a command to do it automatically:

1. Put an insertion point in the empty list of subitems to the current item.
2. Insert an empty subitem (here the command program would include a document-language expression specifying the thing to be inserted).
3. Put an insertion point in the title field of that subitem, ready for the user to type the title.

Many such specialized commands could be useful, especially for dealing with mathematical typesetting. The simplest sort of user-defined command, and one with great potential to speed up the process of document creation, is simply to insert a new instance of a specific object type—an enumerated list, a fraction, an exponent, a summation sign, etc.

The concept of programming an interactive editor has already been richly explored, and its value abundantly well proven, in the Emacs [Stallman 81, Gosling 82] text editor. I have worked in several user communities that used Emacs as their standard editor, and I have seen its programmability put to extensive use, both for personal convenience and for the building of entire embedded applications. But what I am proposing here really goes farther than that. It is a synergy between two sorts of programmability. The user defines the elements of document structure by document programming; he then can define commands appropriate to that structure by user interface programming. By binding these commands to keys or adding them to menus, he can produce a customized editor that is extremely streamlined for his particular task, just as document programming allows him to create a document structure well tailored to his particular need.

Defining the primitives to support such programming will be an interesting project, because they must operate on a rather complicated domain. Where Emacs deals with a simple sequence of characters, this programmable editor will be dealing with a tree—the syntax tree. This tree is adorned with such complexities as node types and repeatable vs. non-repeatable elements. Primitives will have to be provided for detection of type and repeatability, various sorts of tree-walking, and tree manipulation.

11.2. Expressiveness

The Lilac language is sufficient to get the job done, but it could be improved. The most significant improvement would be to support repeatable formal parameters with a mechanism for enumerating the actual parameters inside the callee. This would make it possible to build tables and similar structures with less programming and less non-textual verbiage in the source view.

Some sort of array-indexing facility is needed. In some styles footnotes are not marked with numbers, but with special characters. An array would be the best way to express the fact that footnote 1 is marked with a '†' character, footnote 2 with '‡', footnote 3 with '§', etc.

The Lilac language contains no looping constructs. It gets along remarkably well without them, but they would certainly make it more powerful for some special purposes. The reason for omitting loops is that they complicate the fingerprinting strategy. Without loops, each execution of a given node is unique within a given execution of its containing function or unit. With loops, this uniqueness is lost. A unique fingerprint can be computed, for a counted loop such as a **for** loop, by computing the loop counter into the fingerprint. For an uncounted loop such as a **while** loop, this is impossible, unless the user is allowed to provide us with some "loop key" value which is guaranteed to be different for each iteration. Lilac could support **for** loops with only moderate effort; to support **while** loops without a "loop key", it would need to treat each such loop as a "hard lump" within which all incrementality is lost. If the loop is executed at all, it is executed in its entirety, and all subroutines called from it are executed in their entirety. Given the probable highly specialized nature of **while** loop applications, this may not be unreasonable.

One piece of structure that is known to the user but not to Lilac is the ordinary relationship of structures to substructures. The fact that a Chapter normally has Sections as its substructures, or that in an outline a Heading usually has Subheadings, cannot be represented in Lilac. Or it is expressed in a rudimentary fashion in the "unit newdoc" template—but for structures that are too rare to belong in the new document template, there is no such information. It could be provided with the use of "prototype" declarations, giving a new structure template for each structural element thus declared. Then a handy variety of Insert command could be provided in the page view editor: instead of typing the whole expression to be inserted, the user just types the name of the function, and its initial argument list is provided by the prototype.

Colors and textures, although not a basic attribute of text, are sometimes used to emphasize it. A primitive variable which governs the color of text and lines would be useful; so would a new box type within which the background is changed to some color, texture, or shading other than the usual plain white.

The best document processors allow the inclusion of figures, and display them on the screen as well as in print. Figures come in many formats: simple bitmapped images, scanned (perhaps colored) images, line-drawing descriptions, PostScript programs. Perhaps some day standardization will reduce the set of such formats to a manageable size; at present, it seems impossible to find a complete set of desirable formats. So it might be useful to provide an interface whereby new item types can be defined, implemented by dynamically loadable code, to represent such figures.

Ultimately we really want integrated editing of text and graphics, which requires loading or otherwise interacting with the code that implements editing within figures of various types. This is a major area of ongoing research; the Quill project [Chamberlin 88] is probably the best example.

11.3. Language and editing

Lilac functions are allowed to have the last parameter, but only the last, as a repeatable parameter. This restriction is important to prevent ambiguity: if a function has more than one repeatable parameter, how are we to tell which of a long list of arguments belong to each parameter? However, it produces some asymmetry in editing. Consider a TableRow function, which typesets one row of a two-column table:

```
function TableRow(first: Hlist, second*: Hlist) =
  hbox(hskip(100), hboxto(200, first, hfil), second)
```

Its second parameter is repeatable. This is good, because it means that the corresponding argument list can contain multiple elements: ordinary text, emphasized text, subscripts, etc. The first parameter, unfortunately, cannot be repeatable. That means that if the user wants to italicize a word in the first column, this cannot proceed in a straightforward fashion, because it would make the first argument into a several-argument list. The obvious solution is for the editor to handle this by wrapping the first argument in the null operator Hlist, whose only function is to cause an implicit MakeList of its arguments. Hlist takes a repeatable parameter, so the first argument can then be split without difficulty.

However, now we have asymmetry in a logically symmetrical tree. The arguments corresponding to *second* are direct arguments of TableRow; those logically corresponding to *first* are actually arguments to Hlist, which is an argument to TableRow. Such asymmetry will have visible consequences for Promote; these may confuse the observant user. The very fact that the editor silently added a level of depth to the tree may be similarly confusing.

Structural lists are absolutely necessary, but this problem suggests that repeatable parameters may not be the best way to describe them. A language design that supports structural lists in a more symmetric way would be valuable.

11.4. More WYSIWYG editing

Page view editing in Lilac can only affect the contents of units. No WYSIWYG editing is provided for function definitions. For ordinary editing, this rule is a good thing; it prevents much ambiguity and potential for perplexing accidents. However, it

might be useful to provide a special Edit Function mode. The user might select a function call expression and then invoke Edit Function; that function would then be opened to WYSIWYG editing with this call serving as a concrete, visible example. An inversion of editability would take place: where the function arguments were editable, they now become hard lumps; where material generated by the function was not selectable, it now becomes selectable and editable.

Lilac also restricts page view editing to expressions of *tangible* type—those that yield items or lists of items as their values. Numerical constants, in particular, are not selectable or editable. This is reasonable—since a number has no position or appearance on the screen, it would be rather difficult to select. And yet, there are times, particularly in the adjusting of spacing, when I find myself wanting hands-on control of things that are specified by numbers. As it stands, two-view editing serves this purpose moderately well. Since Lilac unparses and reparses at the expression level and uses incremental interpretation afterward, a change to a number leads to the most trivial of reparsing tasks. Such a Reparse usually takes less than a second, making a process of trial and error rather painless.

Still, the best WYSIWYG editors do much better. Common spatial properties such as paragraph margins and tab stops can be set visually by means of a ruler. To provide this facility in Lilac, however, would be a nontrivial research project, because the specifications of these properties are often buried inside a function definition. This suggests that a system like Lilac really should provide WYSIWYG access to function definitions. Other problems arise: since Lilac allows arithmetic on numbers, we may have to symbolically interpret the function to determine the relationship of numbers in the function definition to spatial properties on the screen. And even then, the assignment of responsibility is a nontrivial problem: if a margin is specified as the sum of two quantities, which one should be modified when the user drags the margin indicator?

11.5. Screen vs. print

I have discovered through use that "what you see is what you get" is not quite what I really want. Things that appear the same on the screen and on the page do not feel the same. Different resolutions make a difference. The font size that looks lovely on the screen looks grossly oversized on the printed page; the size that looks good in print causes eyestrain on the screen. A rule that looks thin and fine on the screen looks excessively thick and dark when printed with the same width. One solution to this problem might be to present the screen display scaled up by 20% or so from the true

size. My screen, a standard 17" monitor, is not tall enough to show a page at this scale. Another solution (which I have been using by makeshift means) is to allow *different style files* for displayed and printed appearance. Actually, one of the style files can be given as a set of modifications to the other; the modifications are probably few. Where the headers of Lilac documents now include style files via **use**, they might include style files with an indication "use for display" or "use for printing". The display styling would be used for normal editing; the print styling would be used for printing, and could be requested when exact page preview is wanted, for adjusting page breaks and the like. Unfortunately, I do not know how to make the transition rapidly—a change of font requires a full recomputation of the document. An elegant solution to this problem would be valuable.

11.6. Convenience of use

Several pieces of intelligence could be added to make Lilac use easier. One is to provide help in finding the definitions of things referenced. Supposing I see a call to a Fraction procedure, and would like to see its definition. It might be defined in the current document, or it might be defined in a style file. This "find the definition" should bring its definition into view, regardless of where it may be found, opening a style file if necessary.

Similar intelligence could be used to make Paste more effective when used across documents. As it stands, Paste will work only if the target document's environment includes definitions for all functions and variables referenced in the material being pasted. Lack of such definitions will lead to a parsing error and failure. But it might instead be made to lead to the copying of definitions from the source environment into the target document. This might lead to some chaos, with definitions from one document's style file ending up scattered through another document without record of where they have come from. But it might also reduce frustration considerably for those users who want to ignore the source view.

11.7. Better typesetting

Where TEX uses an optimal-fit dynamic programming algorithm for choosing line breaks in pargraphs, Lilac only uses a local decision-making algorithm. In addition, it does not support *penalties*, which allow the user to give weight to some possible line breaks over others. The reason for this is performance: Lilac needs algorithms which are quick and simple and can be handled incrementally with minimal computation. This is quite reasonable during editing, but at final printout time, the user might well be

willing to wait to have a more meticulous job done. A command could be provided to do this: for all paragraphs within the selection, perform high-quality linebreaking. Any editing of these paragraphs thereafter would lead to a gradual degradation of quality, eventually returning to the quality that Lilac ordinarily provides.

For the same reasons of speed, Lilac is not in a position to do automatic hyphenation in its ordinary operation. (The hyphens in this document were placed manually.) Once again, this would be appropriate to do when the user asks for high-quality linebreaking. Getting it spontaneously undone when the user edits the paragraph later is a problem requiring a bit of research.

For setting the spacing between lines within a paragraph, Lilac adopts the elegant solution used by TEX. The user specifies a glue, *baselineskip*, which specifies the spacing, not between boxes, but between their baselines. The paragraph builder uses this glue, together with the actual heights and depths of lines, which may vary, to compute the actual interline glue which will give the proper, even spacing. (See [Knuth 84], pp. 78-80, for more detail on this design.)

Unfortunately, outside the paragraph builder Lilac has no very good place to do this job. Where TEX adds boxes to a Vlist sequentially, Lilac tends to build up a Vlist out of sublists. The information about the heights and depths of adjacent boxes is not so easy to get at, and Lilac currently does not attempt it. This means that where lines are made explicitly, or at such places as the boundary between paragraphs, uniform vertical spacing is not easily maintained. Lilac helps by (usually) building character string boxes with the height and depth of the font, rather than of the individual characters containing them. This means that all lines built from the same font tend to have the same height and depth. But as soon as subscripts or superscripts appear, uniformity is lost—or must be manually repaired, usually by insertions of negative glue. A better solution needs to be designed.

Lilac does not yet support multi-column pages. To do so would require a new primitive of complexity similar to Para, for breaking down a vertical list and constructing a horizontal list of columns from it. I foresee no new complexities here.

Lilac does not yet support mathematical typesetting with the elegance that TEX provides. It has no "math mode" or special math-boxes. All mathematical typesetting is still possible—but it may have to be expressed in terms that are not ideally suited to the material, thus requiring more work to maintain. For example, it is not easy in Lilac to write an equation with several terms which are fractions of different heights, and make them all line up correctly along a horizontal line. It can be done, but it requires

much tweaking. It would be useful to add facilities to Lilac to enable it to handle such typesetting more naturally.

11.8. Efficiency

Lilac currently handles documents up to roughly 20 pages in size without difficulty. Beyond that, response to keystrokes begins to become sluggish. This problem could be considerably helped, of course, by waiting a year for processors twice as fast as the workstations currently available. But there is a deeper problem than that. Incremental interpretation requires time of order O(d * b) — depth of syntax tree times branching factor. Branching factor is multiplied by the cost of looking up a node in the Store—a modest but not entirely trivial operation. As documents grow large, branching factors at some level increase until at last they become a performance problem. This could be improved upon. Incremental MakeList, when operating in its fastest mode, does not even examine all its arguments, but only those adjacent to the one that actually changed. With considerable restructuring of the interpreter, we could avoid even looking up the arguments that will not be examined. Instead of doing (d * b) lookups, then, we need do only (d * 3) lookups, with a few rare exceptions. Speed of interpretation then would not depend on the size of a document at all, but only upon its complexity and the complexity of the functions used in it.

Lilac is a space hog. Item records are large and very numerous. I believe there is no way around the fact that they are large: they are storing information to support bidirectional mapping, incremental interpretation, and incremental screen-updating, as well as typesetting itself. Something can be done about the fact that they are numerous. Users when editing often focus in one one section of the document, perhaps only a few pages, for an extended period of time. Groups of logically related items that are distant from the working area could be written out to disk and replaced with a "swapped-out" marker item that tells how to get them back in and tells the total vertical space they occupy. This marker would be stored both in the main Vlist and in the Store. Any attempt to render one of these items on the screen would lead to a swap-in. At an occasional cost in time, we could save vastly on space.

Node records are far less of a space problem, because they are far less numerous. But for sufficiently large documents, they could become a problem. This problem can be solved in coordination with the swapping-out of items. For any node whose resultant items have been swapped out, that node and its descendants could also be swapped out and replaced in the tree with a marker node. Any attempt to reinterpret that node or to select something within it would result in a swap-in.

The most serious performance problem in Lilac comes at startup. Documents are stored as text files consisting of the text of the source view. Thus, to bring up a working page view requires a full parse and interpretation of the document, requiring a bit more than one second per page for ordinary text, more for complex documents. This is not a major impediment to its use as a document preparation system, but it is a considerable hindrance to its use for presentation of on-line documents. Apart from this slowness, it would be very nice for that purpose, supporting on-line documents of a much higher quality than are available on most systems. This could most easily be solved, at a high cost in disk space, by saving the internal data structures (syntax tree, display list, Store) rather than just the text. Some time would still be required to reconstruct the trees in memory from an external representation, but the major cost, interpretation, would be saved.

A far more ambitious solution would be to implement opportunistic interpretation: read in the text and parse it, but then interpret only as much as is really necessary to display the first page. While the user is looking at the first page, continue the process of interpretation until it is complete. This would solve the startup problem quite effectively, I believe; it is also a major research project.

11.9. More than two views

The appearance of a document in the page view is governed by its *environment*—the set of procedure bodies and default values of variables defined in it or in its included style files. The same document body (i.e., main and subsidiary units), in a different environment, might have a different appearance—different fonts, or margins, or spacing, etc.

This potential can be put to good use by supporting multiple page views, some or all of which may be opened by the user at various times. One page view would be the straightforward page view already described. The others would each be defined with the help of an *auxiliary style file*. This style file is interpreted after all other definitions but before the document body, to create the modified environment for its special view. Some examples of usefulness:

- Table of Contents view. If we redefine the Section function to ignore its body argument and render only the title, then we have a view which shows only chapter and section titles. Further modification of the styling of these titles can produce a nice table of contents view. Chapters and sections can be rearranged, deleted, retitled, or created without contents by working within this view,

though the bodies are not accessible for editing. This view allows the user to work at outline level, while actually manipulating the document, not a separate entity.

- Screen vs. print views. In my experience, the fonts that look good on a 75 dpi screen are too big to look good in print, and the fonts that look good in print cause eyestrain when used for very long on a 75 dpi screen. The "What you see is what you get" model is imperfect in this regard. Two page views can be useful here. The screen view, set in a large font, would be used for text creation and early editing; the print view, set in a smaller font, would be used for final adjustment of page boundaries and placement of figures.
- Typewriter view. It may sometimes be necessary to print a document on a printer that supports only one font. A screen view that sets all text in one fixed-pitch font could be useful as a preview for such printing.
- Transparency vs. print views. Like the screen, a transparency is a medium whose constraints are somewhat different from those of the printed page. Different font sizes, line thicknesses, and spacings are appropriate. Sometimes a user might want to produce the same document in both formats; multiple views would make this work more convenient.
- Annotated views. A document might be written by one person and annotated by one or several others. Sometimes users might want to see the text with its annotations; sometimes they might prefer it uncluttered. In such documents we could define an Annotation function, which produces a null value in the standard view but renders its argument in full in the annotated view.

11.10. Application to other realms

Some of the techniques developed in this project are probably useful in other realms besides text. The realm of graphics comes immediately to mind—both general illustrative graphics, and specialized uses such as VLSI design. Two-view editors for such applications have already been built; that would be no innovation as such. But some ideas from Lilac might be worth adding.

One of these is the strategy for selecting in a hierarchical structure, using multiple button clicks to promote. All hierarchical graphical editors I have seen allow access to only one level at a time, typically the outermost level by default. Substructures can be edited only by ungrouping them or by entering some special mode such as "edit cell." The promotion strategy allows much faster access to the various levels, while leaving no ambiguity as to what is being manipulated. This could also be very useful in other

textual systems that involve a lot of structure, such as outline processors and syntax-directed program editors.

Also interesting to apply would be the language design concepts of dynamically scoped variables and unevaluated arguments. Graphics, like text, involves many style variables—line thickness, color, fill pattern, and a very important case, transformation. It is quite natural to control these values in a hierarchically nested fashion; however, existing graphics languages such as PostScript tend not to do so consistently. Asente's system uses dynamic scoping for transformations, but does not make the facility generally available for other purposes.

12. Related Work

12.1. Document formatters

Batch-processing document formatters are the oldest form of computer-aided document preparation tool, and countless such programs have been built. There are two that, in my opinion, represent the state of the art. The TEX system [Knuth 84], which has inspired many parts of the Lilac language design, uses a powerful macro language to describe a document and its layout. It emphasizes fine precision in typesetting control, and is unequalled in its ability to typeset intricate mathematical formulas. It pays no particular attention to the logical structure of a document; the TEX source for a document may or may not bear any relation to that structure, depending on how the user has chosen to use macros. TEX is an intricate machine, with many controls and options; it is reputed to be difficult for novices to master.

Scribe [Reid 80] uses a more high-level language. There is no procedural control of typesetting; typesetting is controlled by the presence of various *environments*, which enclose the text they influence. They operate by setting the values of various variables and options within their enclosed scopes. In contrast to TEX source, Scribe source often clearly reflects the logical structure of the document being described. Scribe is frequently recommended as the formatter of choice for novices. Lilac's handling of global variables is inspired in part by the paradigm used in Scribe.

An interesting old experiment in programmable formatters is PUB, [Tesler 72] which was built at the Stanford Artificial Intelligence Lab. It supported user-defined variables and provided a sizeable subset of Algol as an expression language.

12.2. WYSIWYG editors

The "What you see is what you get" direct-editing method of document preparation is newer, requiring more powerful display devices. One of the first editors of this type was Bravo [Lampson 79]. Bravo, using a bitmapped screen, displays an approximation of the printed document, with fine-grained spacing and a variety of fonts. This display is maintained constantly up to date as the user types text or does other forms of editing. Editing operations act upon the selection, a piece of text which has been marked off by the user by pointing with the mouse. The selection is indicated by underlining the selected text. This method of operation was brought into widespread use essentially unchanged by MacWrite [Apple 84] and is now used in countless commercial products. The Lilac page view editor follows in this tradition.

Some examples of the current state of the art are Microsoft Word 3.0 [Microsoft 87] and FrameMaker [Frame 87]. FrameMaker is probably the best WYSIWYG system I have seen. It provides excellent page layout capabilities and well-integrated handling of graphics. Automatic numbering is provided. It also supports named, user-defined styles—at the paragraph level only. This is the extent of support for structure in FrameMaker: properties can be attached at the character level, the paragraph level, or (for some page layout properties) the major section level. Within this limitation, however, it does very well at giving convenient and intuitive access to properties. The user can adjust margins and tab stops, either for an individual paragraph or for all paragraphs of a given named style, by moving objects on the ruler.

12.3. Syntax-directed editors

Several program editors have been built that use an abstract syntax tree as the internal representation. Mentor [Donzeau-Gouge 84] and the Synthesizer Generator [Reps 87] are good examples. These systems use knowledge of the language syntax to control the editing process, with the aim of preventing the user from writing syntactically incorrect programs instead of telling him his mistakes later. Information stored in the tree is also used for such tasks as intelligent search and replace, incremental type-checking, and incremental compilation.

The Lilac page view editor follows in this tradition: editing is based on a syntax tree and works within the syntax so that the user cannot produce incorrect syntax. In contrast to these systems, however, Lilac's page view editor is an *indirect* syntax-directed editor. The user does not see the program he is editing, but instead sees its output.

12.4. Two-view editors

The concept of providing two simultaneous views, one programmatic, the other visual, is still more recent. An early instance of this mode of operation is the Sam program [Trimberger 81], a VLSI layout editor. Sam presents the user with a *program view* and a *graphic view*. The program view displays a textual description of the underlying data structures in a largely declarative language. Subroutines (cell definitions) and iteration are supported. The graphic view displays the layout in the usual fashion, with stippled rectangles. Both views are editable. The program view uses a syntax-directed editor, so the problem of a syntactically incorrect program can never arise.

Trimberger's most important contribution is the concept of reflected selections. A selection made in either view is also highlighted in the other, providing a very useful tool to show the user how the two views relate. This concept is present in Lilac, where it proves similarly powerful.

Ignacio Zabala, a Stanford student, built the GOB system [Zabala 82] exploring the possibilities of a constraint-based description language underlying an interactively editable text structure. His is a WYSIWYG editor, not an actual two-view editor, but has a program-like representation "just beneath the surface."

The first two-view text processing system was Janus [Chamberlin 82], developed at the IBM San Jose Research Center. It is not a full-fledged two-view editor: it presents two views, but only one of them is directly editable. The *markup* view shows the document description in the Standard Generalized Markup Language (SGML). This includes the raw text together with its *tags*, which comprise a declarative language describing the document. The *formatted* view shows a page of the fully formatted output. All text and markup editing is done in the markup view; a Show command updates the formatted view after editing. There is actually a small vestige of two-view editing: page layout can be directed by graphically editing a page template in "layout mode".

Greg Nelson's Juno system [Nelson 85] applies the two-view editing concept to the realm of two-dimensional graphics. Pictures are built of procedures which are represented textually in a declarative language based on points and constraints relating those points. In the graphical view, using the mouse, the user can create, select, and move points, and express various constraints among them. Unlike Lilac, Juno does not update the textual view on every operation, but only adds procedures to it as the user invokes a "make procedure" command.

12.5. Concurrent work

Paul Asente's doctoral project [Asente 87], completed about half a year before this project, also applies the two-view concept to two-dimensional graphics. His system, Tweedle, goes far beyond Juno in that it uses a powerful procedural language to describe graphics. The Tweedle language ("Dum") can easily describe repetitive or recursive figures. Objects can be built up of subobjects, with transformations applied. Objects can also be defined as *variants* of other objects, with modification operators applied to add, subtract, or alter features inherited from the original object. Constraints can be expressed by the use of *control points*, which may be defined in one object and

referenced in another. Good interactive access to these facilities is provided in the graphical view.

Asente encountered several of the same problems that I have encountered in Lilac, and it is interesting to compare the results. Both projects need fast incremental re-execution of the description language, to provide reasonable performance for WYSIWYG editing. To allow this, both of us found it necessary to design new description languages in which side effects are carefully limited and controlled. All changes to global state must take place over a clearly indicated scope, rather than persisting indefinitely as a simple assignment might do. Functions must have no side effects. To allow otherwise is to severely compound the problem of discovering the global state in effect at a local region of the program that needs to be re-executed.

Tweedle, like Lilac, makes a syntactic distinction between *objects* and *functions*. Objects are parts of the picture; their contents are accessible to WYSIWYG editing. Functions are subroutines for use by objects; their contents are not accessible to WYSIWYG editing. (The Lilac equivalent of an object is called a *unit*.) The distinction is important to both systems for the same reason: it removes some ambiguity as to what is being manipulated as the user points and acts in the WYSIWYG view.

The other part of that ambiguity relates to hierarchy: when the user points into an object made of several levels of subobjects, at what level in the hierarchy does he mean to select? Here Lilac and Tweedle take opposite approaches. In Lilac, selection starts at the lowest level of hierarchy, and is raised to higher levels by *promotion*. In Tweedle, selection by nature applies to the top level of hierarchy (components of the main program). Lower-level objects can be selected only after a higher-level object has been opened to internal modification by the Edit command.

Don Chamberlin and his team, who did the Janus system, are now working on a new system named Quill [Chamberlin 88]. Like Janus, Quill is based on SGML, but it is a full WYSIWYG system. Quill is designed to support full integration of various sorts of graphics editing together with text editing. Quill differs from Lilac primarily in the nature of the SGML language: SGML describes only structure, not styling; the details of styling must be bound elsewhere.

Pehong Chen and Michael Harrison are working on another true two-view document editor, the VORTEX system [Chen 86, Chen 87, Chen 88]. Instead of setting out to design a document description language to support the best possible two-view editor, they have set out to build the best possible two-view editor than can be built around an existing standard language, TEX. VORTEX (Visually Oriented TEX), provides a *source*

view showing the source for a document, and a *target view* showing the formatted result. I argued that the TEX language is ill-suited to building a high-performance WYSIWYG editor, and indeed, Chen and Harrison have not attempted it. They allow text entry only in the source view; the new characters are batched and propagated to the target view by background processing. The WYSIWYG editor supports such functions as cutting, pasting, and style changes, which do not demand such fast turnaround. On the other hand, VORTEX provides the full formatting powers of TEX, which at this stage consider- ably exceed those of Lilac. It also offers genuine integrated graphics editing, where Lilac only includes graphical figures as monolithic objects.

13. Experience and Conclusions

I have used Lilac to create, revise, and typeset this dissertation. All the text and many of the figures were done using the tools provided by Lilac; the rest of the figures are either bitmaps taken from screen dumps of Lilac in action, or are MacDraw output, included as PostScript. This represents about four months of heavy use, interspersed with some programming to fix problems as they arose.

Though one user's experience is hardly sufficient to make any final judgements about a program, I have found the experience very interesting and instructive. Here are some of the conclusions I have come to in the process.

13.1. Performance

The performance goal of 10 keystrokes per second (100 msec per keystroke) has been achieved for ordinary running text in documents of sizes up to 10 pages or so. For larger documents, the large branching factors in the syntax tree begin to bring performance down noticeably. Text in deeply nested contexts also takes longer per keystroke, but this is not a serious problem because such text is relatively rare.

When a speed of 10 keystrokes/second is attained, it is quite satisfactory. Faster speeds feel even better, but at this speed I never get ahead of the echoing of characters. A speed of 9 keystrokes/second is still reasonable; 8 begins to feel definitely sluggish. From there, as performance decreases, the situation degrades very rapidly to an utterly intolerable one.

The larger commands, such as Cut, Paste, and Return, are not so highly optimized for performance. In some cases they take up to 1/2 second or even occasionally 1 second (and a very large Paste can take time proportional to the length of the material being pasted). However, these slow responses do not seem troublesome at all, since they occur relatively rarely and the really slow cases occur extremely rarely. The time taken is always far less than the time I took to think about and set up the command.

13.2. Page view editing

A major conclusion which comes as no surprise is that I really like WYSIWYG editing. The feeling that "I can see what I'm doing" has a great comforting effect, far beyond those occasions where it demonstrably speeds the process of producing a satisfactory document. The feeling of spatial orientation provided by seeing the

document in typeset form, and seeing selections indicated within that context, is very satisfying.

Vertical insertion points are a successful experiment. If I think of my document in hierarchical terms, I certainly like to see the difference between an insertion point in a character string and an insertion point between paragraphs. The biggest problem is that they don't go far enough: there is no guaranteed way (except in the source view) to tell a paragraph-level insertion point at the end of a section from a section-level insertion point that follows that section. As far as I can tell, the more means that can be devised to illustrate this sort of distinction, the better.

The Promote concept, using a sequence of Promotes to reach high levels in the structure, works out quite nicely. The multi-click approach to it is often very effective, but it is not perfect. Speed is its great advantage, and also its downfall. Sequences requiring up to 3 or 4 clicks work quite well. Beyond that, my brain cannot send the instructions to my hand "all at once," and a feedback process is required: looking at the highlight on the screen, I decide whether to click again. This seems to require a double-click time limit of somewhat over 1/2 second; for some users this figure will probably be far greater. Even when the feedback process is in use, the chances of overclicking are significant; to correct for an overclick, I must pause and start the sequence over again, which is frustrating. Therefore I believe that a "slow version" key-binding for Promote must always be provided. This is a key or key combination which will invoke Promote without any time-dependency, so that it can be repeated as quickly or as slowly as desired.

Even more effective than Promote is PromoteExtend, the command that promotes the selection to a level at which it can be extended to include a newly indicated point. When using PromoteExtend I always feel that I know exactly what I am selecting, and at what level; I often find it the most convenient way to make a high-level selection. For example, putting an insertion point in a section and then PromoteExtending to its title is a way to select the section, without danger of overclicking.

One minor problem is the issue of "invisible promotions." When promoting inside a Bold field, I will at some point select all the characters within that field, and then on the next promotion I will select the field itself. In the former case a Cut would leave an empty Bold field; in the latter, it would remove the field entirely. Unfortunately, since Bold does no boxing and adds no ink or spacing, the appearance of the highlight does not change at all as this promotion occurs. If I were not alert to the problem, I could be confused as to the level of the selection. Of course, the source view is there

to answer the question—if it happens to be already open on the screen, scrolled to the right place, etc. This is sufficient, but less than lovely. I know of no solution to this problem. Fortunately, in practice, I find that a bit of alertness usually suffices.

The Return and Tab commands, in my experience, are really great! They are my favorite feature of the user interface. They allow me to extend almost any structure in a way that quickly becomes ingrained in my subconscious so that I don't have to think about it. Tab is also very useful for hopping to the end of a growing paragraph after I have stepped back to make some minor correction in the middle of it. It is equally useful for finding empty text fields after I have left too many empty structures lying around.

The Return paradigm falls shortest in the context of mixed structures: where text is thickly interspersed with equations, or specially formatted examples, or the like. Where a change between structure types occurs, I find myself using the Insert.. command with uncomfortable frequency. I believe the only good solution to this is a measure of user interface programmability. At least this must be provided: to allow the user to create menu items and key bindings to Insert specific expressions. For example, he might create a menu item to insert an empty ProgramExample structure, or a control-key combination to insert an empty paragraph. As discussed in Chapter 11, I believe that a much deeper level of programmability would be very useful.

As of this writing, Lilac supports one form of figure inclusion: bitmaps. Here again the joy of WYSIWYG editing shows up: it is really nice to see my figure as I edit the text that refers to it. It seems to bring the document to life in a way that the text alone cannot do.

13.3. Two-view editing

Given a source view and a page view to edit in, I find that I spend about 95% of the time working in the page view alone. That is not a measured figure, but an impression gathered over months. There are whole sessions in which I never open the source view window. However, on those relatively rare occasions when I really do want a source view, it is very nice to have one available. Those occasions usually involve one of:

- Orientation. Working in some complicated or unusual structure, I want confirmation that my selection is what I think it is. Here reflected selections are a great help.
- Global styling. I want to change font, margins, or some such parameter, over the whole document.

- Special constructs. I am constructing a table (such as Fig. 5-9) or special diagram (such as Fig. 2-4). Here I find myself using the programmability of Lilac in all its glory, both to define structures and to describe their styling. This is often an iterative process, as I debug and refine the styling.
- Prototyping new style sheets. I am creating or enhancing a style sheet, and I create a main unit to provide visible examples of what I am describing. Here I am really using Lilac as a fast-turnaround document formatter: page view editing is rare, and the page view exists as a confirmation of the work I am doing in the source view. And in this case, that fast turnaround is really great to have!

I find that I resent any circumstance that requires me to use the source view when I do not have real, intrinsically source-view work to do. Thus the escape commands, Insert.. and Style.., are important. In these commands I use the source view language, but I keep the spatial orientation and continuity of the page view, and that is very valuable. Switching views often involves an unpleasant loss of continuity.

A vital aid to mitigate this problem is the "Show me" command, which, given a page view selection, scrolls the source view to display the corresponding text and briefly flashes a highlight around it. When turning to the source view after long ignoring it, I almost invariably need this command. Perhaps it would be better to have the two views automatically scroll together, but I doubt it. I think it would be more distracting than helpful. Possibly a more useful command would be the "Over" command, which would transfer control over to the other view. If the keyboard focus were in the page view, it would be transferred to the source view, where the reflected selection would become a real selection ready for editing. The other direction is less easy to define.

The immediate reflection of page view edits in the source view is not as useful as I originally thought. I seldom look back and forth between views as I work, and when I do, it is not the textual characters that I am looking at. Indeed, I often find the motion in my peripheral range of vision distracting. The main reason for supporting such reflection is that it is a prerequisite for the immediate reflection of page view selections in the source view. Interestingly enough, this feature is quite useful, providing a ready reference point to show the connection between the two views. Visual distraction is avoided because the reflected highlight is (optionally) so light that it passes unnoticed in the peripheral vision.

Some portion of my resistance to source view editing came from the fact that textifying and reparsing were done at the unit level. Units are often quite large, and

this made the Reparse command very slow. Matters were still better than a batch-mode document formatter would make them, but the same frustration was present. Very late in the project, I changed the implementation so that textifying and reparsing are done at the expression level—with backup strategies applied in case the textified portions are not locally syntactically complete after editing. This means that in many cases the Reparse command now creates and interprets only a few new nodes, requiring less than a second. Page view and source view editing became much more similar in spirit, with reparsing being far less burdensome. Immediately I found myself doing real two-view editing much more readily than I had before.

Probably the case where I find this fast turnaround most valuable is in dealing with numerically controlled styling. To "debug" the styling of a function I will copy the function definition into my working document (if it is not already there) and focus on one instance of it, which I declare to be the region of interest (see section 9.6). Then I can move into a fast-turnaround debugging mode: edit a number in the function, Reparse and look at the results; edit, reparse, and look. The cycle can often take place in a few seconds. In some cases the numerical parameter is an argument to a function call directly in the mainstream text. Then the process is even easier. Without needing to declare a region of interest, I can simply edit the number and reparse, seeing results in a fraction of a second.

The Poke and Region of Interest commands are similarly important. To remind the reader, the issue is, after a function body or variable value has been modified, how much of the document should Reparse reinterpret. To reinterpret all (which was the first policy I implemented) is intolerably slow. To reinterpret only those parts of units that have actually changed is to reinterpret nothing, in this case, thus giving the user no feedback. The solution of reinterpreting whatever the user has declared as a region of interest seems to be a good compromise. Poke allows the user to point out another region at a later time and reinterpret that also; this is also valuable. It saves the necessity of a Recompute, which is so slow that it is greatly to be avoided.

13.4. User-definable structure

The most unusual feature of Lilac is that it allows the user to define the elements of which document structure is made, and then to edit a structure made from them in WYSIWYG fashion. I find this freedom to define new elements very valuable. A normal document is made of chapters and sections and paragraphs; well and good. But then I am making a set of slides—and instead of using chapters and sections and para-

graphs, I make my document of slides and items and subitems, all of which I can freely define.

Programmability is useful for special document types; it is equally useful for recurring special constructs in an otherwise normal document. The two-view figures of Chapter 5 are an example of this, as are the program examples found in Chapter 4 and elsewhere. For such a feature I define a special function for use in that document, debug it once, and then use it as often as I need it. Rather than modeling the program example as a series of specially styled paragraphs, as I might have to do in an editor like MacWrite, I can model it as a single entity, as I conceive it. This is a real advantage.

My other use for programmability is for special, one-shot structures such as the "road map", Fig. 2-4. This may not be the best way to build such figures; it is quite time-consuming, and there is no multiplier in the payoff. But it still has the advantage that such figures, once defined, tend to be robust in the face of minor changes.

13.5. Language

The technique of using **let** expressions and unevaluated arguments to handle global variables in a well-nested way proves quite successful. It is not hard to program with, once the basic concept is clear. As I look into the finer points of document processing (Asides, automatic numbering, and the like), I keep seeing more and more places where unevaluated arguments are desirable. In the course of maintaining and improving a style file, I often find myself converting evaluated arguments into unevaluated arguments. Perhaps this means that macros are really better than functions as the fundamental element of programmability in a document formatter. Certainly tradition would support this view. But if so, Lilac would still need a specially constrained form of macro, one in which each macro expands to exactly one expression, not an arbitrary string of source characters.

13.6. Implementation issues

The Store, with its use of call path fingerprints as keys, is unquestionably the most important invention in the implementation of Lilac. Again and again, as I have expanded Lilac or added new optimizations for speed, this facility has made it easy to retrieve the right information at the right place at the right time. Its ability to answer the question, "What did I do last time I was here?" is fundamental to the success of the incremental interpreter.

The concept of maintaining a current *node of causation* during interpretation and attaching it as a label to created items is similarly important. It serves well in its principal role of supporting item-to-node mapping, but it is useful beyond that. In situations involving Asides, numberings, and cross-references, the availability of this information allows us to mark the appropriate nodes dirty when a local change has had distant side effects.

A third vital concept is the use of *counted lists*. Where incrementality and list-building meet, it is crucial that we be able to concatenate lists nondestructively. That is, each sublist in a concatenated list retains its distinct existence and its length in spite of the concatenation. The same facility allows us to extract a sublist from a larger list without altering the larger list. This is important for paragraph- and page-building.

Interpretation is by far the most costly operation in the system. Parsing, unparsing, and rendering are quite cheap by comparison, and mapping operations are utterly negligible. Why should this be true? Because interpretation involves subroutines. Parsing and unparsing in a sense involve only one operation per node, because they operate at the superficial syntactic level. Rendering involves roughly one operation per item. But the interpretation of a node that invokes a user-defined function may involve any number of operations, including further function calls to an arbitrary depth.

Within the realm of interpretation, a great proportion of the time is spent in a few fundamental operations. List-building, string conversion, and paragraph-building top the list. Heavy optimization of the incremental versions of these three operations was crucial to getting acceptable performance. Other box-building is less prominent, but still significant, in the accounting of time spent on interpretation.

An incremental interpreter whose performance depends on O(depth * branching factor) will never be really satisfactory. It is doomed to flounder when dealing with long, undivided lists of similar elements (such as a glossary) or with sufficiently long documents of any structure. The dependency on branching factor must, and can, be eliminated from average-case performance. It will remain for some highly unlikely worst-case scenarios.

The nature of characters adds considerable complexity to the basic Lilac algorithms. They are by far the smallest unit of data found in any type of document, textual or graphical. Because they are so small and so numerous, we can afford neither a node nor an item for each individual character. They must be grouped, adding an extra level of structure to both the syntax tree and the display list. This creates inhomogeneous structures: the syntax tree consists of nodes, but then at the bottom level it consists of

characters within nodes; the display list is similarly complicated. A considerable fraction of the code in the Lilac system is devoted to handling this extra, inhomogeneous level of structure.

Lilac is by its very nature a space hog. In order to handle large documents (hundreds of pages) successfully, it will inevitably need some form of swapping. Perhaps a fast paging system with a fast disk will suffice; I suspect that to work well on workstations in the near future, it will need its own smart swapping algorithms.

13.7. Contributions to knowledge

The Lilac project has made several contributions to the field:

1. For the first time, a document description language designed in the fashion of a real programming language, with functions, data types, and strong typing. With this design came the discovery that dynamic binding, unevaluated parameters, and repeatable parameters are necessary to meet the particular needs of the document description task.
2. A language specifically designed to serve the needs of two-view editing, and illustrating those needs. It is a pure functional language, which is vital for fast incremental reinterpretation. It also invites the user to explicitly express the hierarchical structure of his document, in terms of whatever constructs seem best suited to the task. This allows the page view editor to exploit that structure to give a clean, logical user interface. The data types of the language contribute to the making of an intuitive user interface.
3. A design for a generic hierarchical editor which is able to handle user-defined structuring constructs without knowing about them in advance. The basic selection operators, Point, Select, Extend, and Promote, allow quick access to all levels of a hierarchy, regardless of its depth, and are suitable for application to many other realms besides text. The Return and Tab operators allow easy extension of a user-defined hierarchy, regardless of its depth or the details of its structure.
4. A strategy for implementing a two-view document editor with fast performance. The core of this strategy is the Store/fingerprint mechanism, which allows the interpreter to recall the previous result of a given computation and use it to avoid or shorten a repetition of that computation. Also important is the handling of changing lists, with change records, incremental list-building, and incremental list-breaking.

13.8. Conclusion

The Lilac project has been a success. It has met its concrete goals in terms of performance, and despite its incomplete state, it shows considerable promise of meeting its subjective goals of user satisfaction as well. The combination of user-definable structure and WYSIWYG editing is very effective and pleasant to use. A sufficient, if not superb, user interface for such editing is possible and has been built. Already, despite its familiar bugs and limitations, I would not willingly trade it for any batch-style formatter or WYSIWYG editor that has come to my attention.

Appendix A: The Lilac Language

This Appendix gives a summary of the Lilac document description language, for the reader who would like to get a more concrete feel for it. Some material is repeated from Chapter 4.

A.1. Data types

Intangible types:

Num	a number (fractions are supported)
Bool	a Boolean value—either true or false
Font	a font specifier
Family	a font family (Times, Helvetica, etc.)
Face	a font face, one of **plainface, boldface, italicface, bolditalicface.**
String	rarely used—for external names only

Tangible types:

Box	a box
Hglue	an item of horizontal glue
Vglue	an item of vertical glue
Hlist	a horizontal list
Vlist	a vertical list

Generic type:

Any	matches any type, under certain circumstances

Tangible types are those that can be directly responsible for output on the screen; only expressions of these types can be selected. The types Hlist and Vlist are also known as *list types*. Repeatable parameters must be of a list type.

A.2. Grammar

Nonterminals:

Document	::=	Imports Defaults Definitions
Imports	::=	{ Import newline }
Import	::=	**use** FileId
Defaults	::=	{ Assignment newline }
Assignment	::=	Id '=' Expr
Definitions	::=	{ Definition newline }
Definition	::=	**function** Id ['(' FormalParams ')'] '=' Expr
		| **unit** Id '=' Expr
FormalParams	::=	FormalParam { ',' FormalParam }
FormalParam	::=	Id ['*'] ':' Type
Type	::=	PrimType | '@' PrimType
Expr	::=	Number | String | Variable | Call | ArithExpr |
		'(' Expr ')' | LetExpr | LocalExpr | IfExpr
Variable	::=	Id
Call	::=	Id ['(' Arguments ')']
Arguments	::=	Expr { ',' Expr }
ArithExpr	::=	Expr Binop Expr | '-' Expr | **not** Expr
LetExpr	::=	**let** Assignments **in** Expr
LocalExpr	::=	**local** Assignments **in** Expr
Assignments	::=	Assignment { ',' Assignment }
IfExpr	::=	**if** Expr **then** Expr **else** Expr

Terminals

Number	A number, optionally including a decimal point and fractional digits.
String	A string of characters surrounded by double quotes.
FileId	A filename of any syntax acceptable to the system, provided it does not start or end with a blank or include newlines.
PrimType	One of the system types, listed in section 4.3.
Binop	One of: + - * / % = # > < >= <= **and** **or**

Metasyntax

[ ]	surround an optional element
{ }	surround an element which may occur zero or more times
|	separates alternatives
boldface	indicates keywords which appear literally in the syntax

A.3. Types and semantics

Every expression has a type. Strong typing applies—the type of an argument must match the type of the corresponding formal parameter. A few implicit type conversions are provided:

Actual type	→	**Formal type**
Box		Hlist
Hglue		Hlist
Box		Vlist
Vglue		Vlist

A repeatable formal parameter (marked with a '*'), may match zero or more actual parameters. These may be of the formal type itself (Hlist, for example), or of types convertible to that type. The callee will see the arguments to a repeatable parameter as one unified list of the specified type. The distinctness of particular arguments has vanished; there is no way to enumerate them or to discover their boundaries. Only the last formal parameter of a function may be repeatable.

Actual parameters to a function are normally evaluated before the call. A formal parameter may, however, be declared to be an *unevaluated parameter*, indicated by an '@' sign before the name of the type. Actual parameters to such a formal parameter are not evaluated before the call, but at the point at which the formal parameter is used inside the callee. If this parameter is used more than once, the actual parameter will be evaluated more than once; if it is never used, the actual will never be evaluated.

A quoted string (String, in the grammar above) is not considered to be a constant of type String unless it appears in a position requiring that type. Rather, it is normally an expression of type Hlist. Its evaluation is nontrivial, and consists of building an Hlist of those characters (boxes for words, Hglue for spaces) based upon the current value of the **font** variable.

The type Num represents both integers and fractional numbers. It is guaranteed to handle values as large as 1 million and as small as .001. The usual arithmetic and comparison operators are provided. Wherever Num values are used to represent sizes, the units are points (1/72 inch).

Penalties, as defined in TEX, are not implemented in general. The only kind currently supported are infinite penalties, used to force a page break. (This is an area for future work.)

Local variables are declared and assigned by the **local** construct (LocalExpr, above). Their scope is local to the containing procedure and to the body of the defining **local**

expression. The right hand side of the assignment to a local variable determines its type; it may be any valid runtime expression.

Global variables are declared and assigned in the **defaults** section of the document. Their scope is the entire document. The right hand side of the default assignment of a variable determines its type; it may be any executable expression that can be stated in terms of variables and functions already defined. The values of global variables may be temporarily changed by the **let** construct (LetExpr, above). The type of the right hand side of a **let** assignment must agree with the established type of the variable. The assignment is dynamically scoped; it is in effect during the entire evaluation of the body of the **let** expression, and it is seen by all units or functions called during that time. When a **let** terminates, the variable reverts to its previous value.

A function may take one argument of generic type, specified by the type Any. It may not be repeatable. In order to be useful, it must be unevaluated. Such a function may also return a value of type Any. What this does, effectively, is to let a function serve as an encapsulated **let** construct, which can modify global variables over the scope of an expression of any type. The **UseFont** primitive function, described among the primitives below, is an example.

A.4. Units and functions

Units (beginning with the keyword **unit**) contain the specific content of a document. The top-level structure of a document is given by the unit named **main**; any other units are brought in as they are called from **main**. Units take no arguments, and should be called no more than once. It is perfectly normal for a large document to consist of only one unit; multiple units are useful for grouping unusual constructions together with their defining functions, to make the source view more readable.

Functions (beginning with the keyword **function**) are intended to define structure and specify styling, not to include content. Unlike units, they make take arguments and are expected to be called many times. Units can call units and functions, and functions can call functions, but functions cannot call units.

The type of a function or unit is determined by the type of the expression that makes up its body.

A.5. Style files

The Imports section of a document is used to import *style files*. Style files are useful for defining standard document types, such as a simple memo, a business letter, a

technical manual, etc. A style file has the same syntax as any other Lilac document except that it has no **unit main**, and thus has no page view. When a document imports a style file, all the variables and functions defined in the style file become part of the document's environment. The document can override definitions provided in the style file, with two limitations. It can override a variable's value, but not its type. It can override a function definition, but only in such a way that the number, types, and repeatability of the arguments, and the type returned by the function, are preserved.

If more than one style file is imported, the later ones can override the earlier ones. Style files may also import other style files, and may override definitions provided therein.

A style file may include a **unit newdoc**. This gives an empty skeleton for new documents of that type. When the editor is asked to create a new document based upon a given style file, the **unit newdoc** will be copied from the style file to become the **unit main** of the new document.

A.6. Primitive variables

The following variables are predeclared by the Lilac system:

font Font
The current font.

fontfamily Family
The current font family.

fontface Face
The current font face.

fontsize Num
The current font size.

protospace Hglue
Specifies the glue to be used for spaces. The parameters of this piece of glue are multiplied by the size of a space as given by the current font, to determine the glue used to represent a space.

baselineskip Hglue
The **Para** primitive, which builds paragraphs, inserts glue between lines to achieve the effect of a glue of **baselineskip** between their baselines.

lineskip Hglue
When two lines of excessive height or depth lie adjacent, such that the usual spacing between baselines would make them overlap or come too close, **Para** will instead insert a glue of **lineskip** between the lines.

lineskiplimit Num

Determines the limit between the **baselineskip** and **lineskip** modes of operation. If the **baselineskip** mode would cause two lines to come nearer to each other than **lineskiplimit**, the **lineskip** mode will be used instead.

lmargin Num

Left margin of lines in paragraphs.

rmargin Num

Right margin of lines in paragraphs, measured from the left edge of the page. The available width for text is (**rmargin** - **lmargin**).

indent Num

The indentation of the first line in a paragraph. This is added to lmargin to determine where the first line starts. Negative values cause the first line to be outdented.

pageheight Num

The height of a page. (Page mode only)

pagefill Num

The height of fixed page material, such as running heads and feet. (Page mode only.) **pageheight** - **pagefill** gives the vertical space available for content material on the page.

The following global names are constants: the Bool values **true** and **false**; the FontFace values **plainface**, **boldface**, **italicface**, and **bolditalicface**; and the infinitely stretchable glue values **hfil** (Hglue) and **vfil** (Vglue).

A.7. Primitive functions

The following functions are predeclared by Lilac:

BlackBox(width: Num, height: Num, depth: Num): Box

Produces a solid black box with the specified width, height, and depth. Useful for making rules, outlines, and underlines.

WhiteBox(width: Num, height: Num, depth: Num): Box

Produces a white (i.e. invisible) box with the specified width, height, and depth.

Hglue(size: Num, stretch: Num, shrink: Num): Hglue

Produces a piece of horizontal glue.

Vglue(size: Num, stretch: Num, shrink: Num): Vglue

Produces a piece of vertical glue.

Hskip(size: Num): Hglue

Equivalent to Hglue(size, 0, 0).

Vskip(size: Num): Vglue

Equivalent to Vglue(size, 0, 0).

Hlist(h*: Hlist): Hlist

The horizontal list constructor. Takes any number of arguments and gathers them into a single horizontal list. Arguments may be of type Hlist, Box, or Hglue. Useful for creating a structural list at a place which otherwise takes only a fixed argument of type Hlist.

Vlist(v*: Vlist): Vlist

Like Hlist, but for constructing vertical lists.

PageBreak(): Vlist

Produces a one-item Vlist containing a special penalty item which forces a page break. Forced line breaks are not currently implemented.

ShowNum(n: Num): Hlist

Converts a number into its decimal representation, using the current font.

NumVal(h: Hlist): Num

Converts an Hlist (presuming that it starts with characters which represent a number) into its numeric value. When the Hlist does not represent a number it returns 0.

Width(b: Box): Num

Yields the width of a box.

Height(b: Box): Num

Yields the height of a box.

Depth(b: Box): Num

Yields the depth of a box.

FullHeight(b: Box): Num

Yields the height+depth of a box.

Hbox(h*: Hlist): Box

Builds an unconstrained horizontal box from its arguments.

HboxTo(width: Num, h*: Hlist): Box

Builds a horizontal box, constrained to have the specified width.

Hmeasure(h*: Hlist): Num

Computes the natural width of a horizontal list without actually boxing it.

Vbox(v*: Vlist): Box

Builds an unconstrained vertical box from its arguments.

VboxTo(fullheight: Num, v*: Vlist): Box

Builds a vertical box, constrained to have the specified full height. Its depth is the depth of the last item in the list, its height is the remainder of the space.

Vmeasure(v*: Vlist): Num

Computes the natural height of a vertical list without actually boxing it.

Raise(delta: Num, b: Box): Box

Builds a box from another box, with the baseline moved down by delta. This effectively causes the box to be moved up like this with respect to the containing horizontal list. Delta can be negative, causing the box to be moved down like this.

Para(h*: Hlist): Vlist

The paragraph builder. Undoubtedly the most powerful function in Lilac, this takes a horizontal list and turns it into a filled and justified paragraph. It breaks the list into lines with the width specified by the **linewidth** variable and turns each line into a box, effectively doing HboxTo(linewidth, <line>) for each. Line breaks always occur between a box and subsequent glue. For each line it chooses the break which will require the least stretching or shrinking. (This is a local decision, not a globally optimized decision as in TEX.) The lines, once made, are glued together using **baselineskip** or **lineskip** as described in the Fonts section above. The result of all this is a vertical list of boxes alternating with glue.

UseFont(family: Family, face: Face, size: Num, body: @Any): Any

Font modification. Computes the **font** variable from family, face, and size; computes other font-related variables; and evaluates body in that environment.

SetFont(family: Family, face: Face, size: Num): Font

Similar to UseFont, but intended for use in the defaults section . Sets up defaults values of all the font-related variables. Its result should be assigned to the variable **font**.

GetFont(family: Family, face: Face, size: Num): Font

Computes a font from family, face, and size, without affecting any font-related variables. This is the inner core of UseFont and SetFont, available in case you want to "do it yourself" with regard to these variables.

GetBitmap(filename: String, baseline: Num): Box

Get a bitmap from a pickled bitmap file and make it into a box. Baseline tells how far up from the bottom of that box its baseline will be. If in doubt, put 0. It should be used only in the defaults section—assign this box to a global variable, then refer to that variable at the place where the box is wanted.

A.8. Fonts

Fonts are more complicated than the other quantities described by global variables. A font is viewed as being made up of three properties: the font family (Times, Helvetica, etc.), the face (plain, bold, italic, or bold-italic) and the size (10 point, 14 point, etc.). Thus we have four font-related global variables: **fontfamily**, **fontface**, **fontsize**, and **font**. The standard font-changing procedure UseFont takes arguments for the family, face, and size, sets the corresponding global variables to those values, computes the resulting value for **font**, and sets that; within that environment it evaluates its body argument. Where one wants to modify some but not all of the font parameters, the names of the corresponding variables can be used in place of the others. This, for example, shrinks the font size by 2 points and changes nothing else:

```
UseFont(fontfamily, fontface, fontsize-2, <expression>)
```

A.9. Pagination and page layout

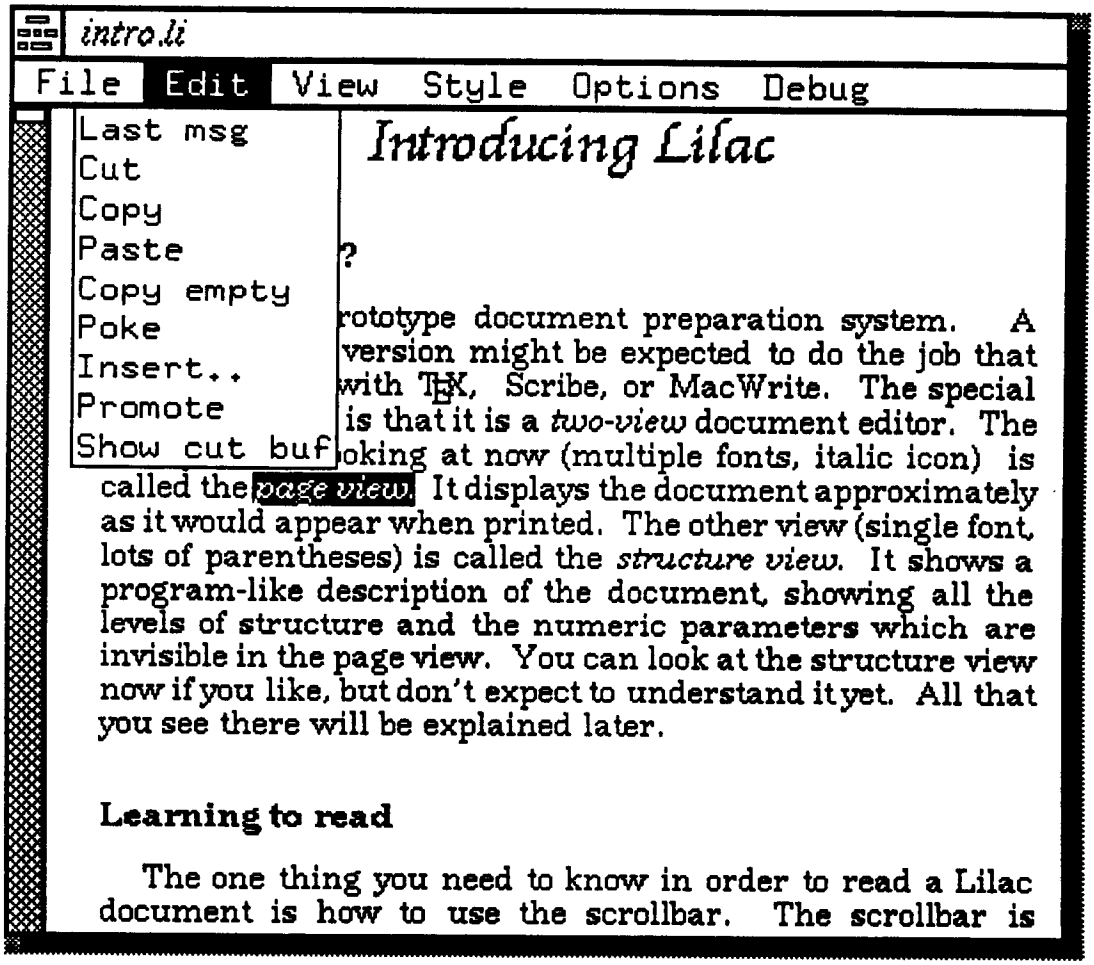
Pagination takes place only when the user requests Page Mode, or during printing. Here is how it works. The value of the top-level expression in **unit main** is expected to be a Vlist, probably much longer than a page. The page breaker scans through this list, adding up heights until a height of (**pageheight - pagefill**) is exceeded. Then it chooses the exact break, at that point or at the previous line, and thus establishes a sublist, the *page content*. This is passed to the special function PageModel:

PageModel(pagenumber: Num, content: Vlist): Box

Given the page number and page content for the current page, produce the box that represents the completed page. This may include running heads and feet, borders, and other content-independent effects. The height of any such added material should be consistent with the declared value of **pagefill**, to ensure that the page breaker makes an appropriate decision.

The system defines a trivial version of PageModel, but style files are expected to override it.

Appendix B: Editor Command Set



This Appendix gives a summary of the actual command set of the Lilac page view editor. The innovative and important commands were discussed in Chapter 5, but this appendix gives a more concise description of them all.

B.1. General Interaction

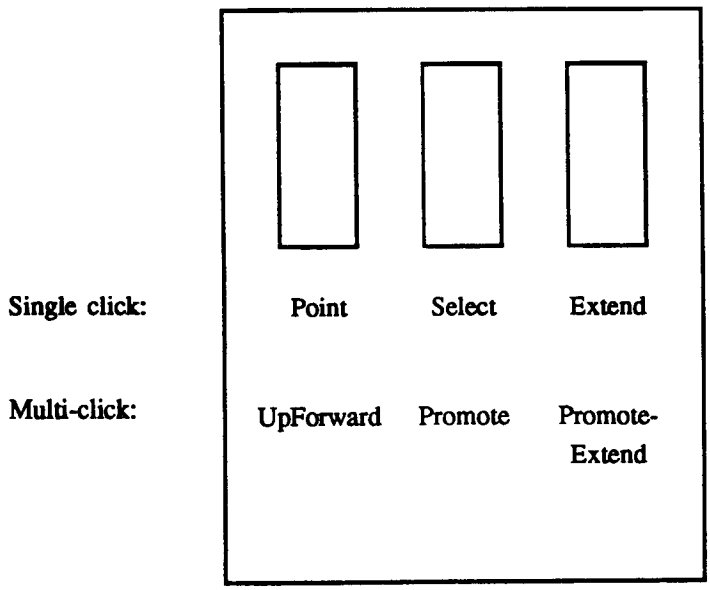
The screen dump above shows a Lilac page view window in use. The window includes a menu bar (shown with the Edit menu pulled down), a scroll bar, and a text area. Other figures in this document have shown only the text area. The scroll bar is used to adjust the portion of the document in view. The tab (small white section at the top of the scroll bar) indicates the position and size of the current viewport relative to the size of the whole document.

The menu bar area serves two other functions as well: it is a message area, and it is a special input area. When anything goes wrong, the menu bar turns into a message area displaying an error message. A click in the bar will dismiss the message, restoring the usual menu bar display. Some commands require special textual input, such as a file name. When such a command is invoked, the menu bar turns into a special input area. A prompt is displayed, followed by an insertion marker or some pre-selected text; the user can type and do simple editing in this area. To terminate the command the user presses either Return to confirm or Escape to abort, and the menu bar resumes its usual appearance.

The text area gives the WYSIWYG view of a portion of the document. All mouse actions in this area relate to making selections.

B.2. Mouse Commands

This figure gives a summary of mouse commands in the text area:



Point

Place an insertion point before or after the indicated object, whichever is nearer.

Select

Select the indicated object.

Extend

Extend the selection from the current selection to the indicated object or a suitable

ancestor thereof. If the selection cannot be extended to anything at the indicated position, nothing happens.

UpForward

Promote the insertion point. To be more precise: find the nearest ancestor of the current selection that is a member of a structural list. Put an insertion point immediately after it.

Promote

Select the parent of the object or objects currently selected. Within a character string, three levels of selection are predefined: character, word, and whole-string. Promoting from a whole-string selection, as from any other sort of selection, has the usual effect of selecting the parent.

PromoteExtend

Promote the current selection to such a level that it can legally be extended to the indicated position, and then extend it.

B.3. Keyboard Commands

The keyboard has two special modifier keys, Control and Option. Most keyboard commands are invoked by holding down one of these while pressing some other key.

Backspace Backspace key

Delete the character prior to the insertion point, if any. Or if there is a non-empty selection, do a Cut. Backspace will not delete anything but characters.

New Vertical Element Return key

As discussed in Section 5.6: find the nearest ancestor of the current selection that is an element of a vertical list, make an empty replica, and insert the replica after the original. If possible, put an insertion point in the first character string in the new element.

Next End of String Tab key

If the insertion point or selection is in the middle of a character string, place an insertion point at the end of that string. If it is not in a string, or is already at the end of one, search forward for the next character string in the document and put an insertion point at the end of it.

Forward ctrl-K

If there is a non-empty selection, put an insertion point at the end of it (if possible, that is, if the selection is part of a list). If there is an insertion point which is not

at the end of its list, advance it past one character or list element. The usual rule of horizontal lists applies: an insertion point at the end of a character string which is part of a horizontal list will be advanced out of the string and past the next list element.

Backward ctrl-J

Like Forward, but moves the insertion point to the beginning of the selection or backward past one list element.

UpForward ctrl-Space

As discussed under mouse commands. As a keyboard command, it is very useful for such things as leaving a boldface or subscript text field to return to normal-styled text.

The following commands are all duplicates of menu commands, and are described in the Menu Commands section, below.

Refresh	ctrl-L
Cut	ctrl-D
Paste	ctrl-F
Copy	ctrl-S
Copy Empty	ctrl-A
Insert	opt-I
Bold	ctrl-B
Italic	ctrl-I
Other view	ctrl-O
Save	opt-S

B.4. Menu Commands

There is a menu bar at the top of the editor window. It contains five menus: File, Edit, View, Style, and Options. Their commands are described here. Where a command has a keyboard equivalent, it is given to the right of the command name.

File

New

Create a new document of the same type as this one. In source view terms, the "use" and "defaults" parts of the existing document are copied into the new one. If any of the style files used by this document defines a "unit newdoc," this unit will be copied to become the main unit of the new document. In all ordinary circumstances, this should result in a working page view in which the user can click and start typing.

New..

Create a new document using a specified style file. Opens a special input field for the name of the style file, then proceeds like New above, but using that style file.

Open..

Open a Lilac document, whose name will be given in a special input field. If all goes well, two new window icons appear, one for the source view and one for the page view. If the document is syntactically incorrect, only a source view will appear, and it will show an error message explaining the problem. If it is correct but has no "unit main," there will be no page view and no complaint. This is the case when opening a style file.

Save opt-S

Save the document.

Save as..

Save the document under a new name. The old file is not deleted, but it becomes dissociated from the open document.

Revert

Restore the document to the state in which it was last saved. Any changes since that time are lost.

Print

Print the document.

Print page

Print the currently displayed page. (Page mode only.)

Close

Close the document. Both views will vanish. If the document has been changed

since it was last saved, the user will be asked whether to save before closing. Lilac terminates when there are no open documents left.

Edit

Last msg

Redisplay the last menu bar message. This includes error messages which the user had to click to get rid of, and informational messages which appeared for only a brief time. It also includes "flash messages" which never appeared. That is, there are some errors which are of such small consequence that Lilac only flashes the menu bar to indicate them—such as trying to Advance beyond the end of a structure. If the user wants to know why Lilac complained, Last msg will reveal an explanation.

Cut ctrl-D

Delete the current selection and move its contents to the cut buffer. Leave an insertion point at the point of deletion.

Copy ctrl-S

Copy the contents of the current selection into the cut buffer. Does not modify the text or the selection.

Paste ctrl-F

Replace the current selection with the contents of the cut buffer.

Copy empty ctrl-A

Make an empty replica of the current selection and copy it into the cut buffer.

Poke

Recompute the selected object. Useful when global defaults or function definitions have changed and a full recomputation has not yet been done.

Insert.. opt-I

Open a special input area and receive an expression in the Lilac document language. If it parses successfully, replace the contents of the current selection with the resulting object.

Show cut buffer

Shows the contents of the cut buffer in the message bar, as a Lilac Document Language expression. Effective only if the expression is short enough to fit.

View

Page mode

Switch the editor into page mode, in which exact pagination is performed and page-marginal features such as running headers are displayed.

Scroll mode

Switch the editor into scroll mode, in which pagination is ignored.

Refresh ctrl-L

Do a full refresh of the text area. Useful after editing overlapping boxes, which may cause the scrambling or disappearance of some items.

Other view ctrl-O

Transfer control to the source view. The characters which were underlined as a reflected selection now become highlighted as a real selection.

Style

Italic ctrl-I

Apply the Italic styling function to the current selection. If the current selection is a horizontal insertion point, apply the Italic function to an empty text field and insert the result at the insertion point, placing the new insertion point inside the new text field.

Bold ctrl-B

Like Italic, but apply the Bold styling function.

Bold-italic

Like Italic, but apply the BoldItalic styling function.

Style..

An escape command for styling. Opens a special input field, expecting a *template expression*. This is a normal Lilac language expression, except that it includes the special expression '$'. The substitution item '$' is replaced by the current contents of the selection, and the selection is replaced by the resulting object. Thus the current selection gets "nested" inside a level provided by the template expression. As with Italic, the template is applied to an empty text field if the current selection is an insertion point.

Options

Region of interest

Record the current selection as the region of interest. Whenever a Reparse of the source view results in changes to default values or function definitions, this region will be recomputed even though it has not been directly edited. Thus the region of interest can serve as a test bed for global styling changes, while the user avoids the cost of a full recomputation.

Other commands in this menu are too implementation-specific to be of interest here.

B.5. Source view commands

The source view editor needs little description, because it is merely a mouse-based plain text editor, more or less modelled after the page view editor. Its commands are for the most part a subset of the commands of the page view editor. However, it has one special menu of its own. These are its commands:

Reparse Escape key

Reparse whatever parts of the document may need it due to recent source view editing. Changes are reflected to the page view.

Recompute

Perform a full recomputation of the document. Useful after a change to global styling information, to cause the change to be reflected in the whole document. The user must be prepared to wait—usually just over 1 second per page.

Grand reparse

Reparse the entire document and reimport any imported style files. This is the same process that was done when the document was originally opened. Useful after a style file has been changed.

Prettyprint

Prettyprint the source view to fit the current width of the window. Useful after changing the width of the window. This is done automatically after a Reparse.

Other view ctrl-O

Transfer control to the page view. If the source view has been edited, this will involve a Reparse. The current selection is mapped over to provide the new page view selection. This cannot always be done precisely; sometimes the resulting selection is the least common ancestor of the initially selected material. If the initial selection was not within a unit, it cannot be done at all; the user must use the mouse to establish a selection.

Appendix C: Examples

This appendix provides examples of actual Lilac programming. Most of the functions here are functions that I have actually used to get useful work done; a few are included just for demonstration.

C.1. A basic style file

Here I display, with a few uninteresting omissions, the style file "basics.li". This style file defines some of the most basic functions for use in all document types; it is intended to be imported by almost all other style files.

```
defaults
  interparaglue = Vglue(6, 0, 0)
  font = SetFont(newcenturyschlbk, plainface, 12)

{ style modifiers }
```

Even though the Bold and Italic style modifiers are so basic that there are built-in keyboard commands to apply them to text, they are most conveniently defined in a style file, not as primitives.

```
function Italic(b: @Body) =
  UseFont(fontfamily, italicface, fontsize, b)

function Bold(b: @Body) =
  UseFont(fontfamily, boldface, fontsize, b)

function BoldItalic(b: @Body) =
  UseFont(fontfamily, bolditalicface, fontsize, b)

function Plain(b: @Body) =
  UseFont(fontfamily, plainface, fontsize, b)
```

The Symbol font has Greek letters for the alphabetic characters.

```
function Greek(b: @Body) =
  UseFont(symbol, fontface, fontsize, b)

function Size10(b: @Body) =
  UseFont(fontfamily, fontface, 10, b)
```

Other size modifiers, omitted here, are also defined.

```
{ basic conveniences }
```

The Para1 function is usually preferable to the primitive Para because it provides for a suitable spacing between paragraphs. The global variable **interparaglue** is declared in the defaults, above.

```
function Para1(h*: Hlist) =
  Vlist(Para(h), interparaglue)
```

Line is for text in which automatic line-turning is not wanted. It performs the services of conforming to the left margin and providing a suitable spacing between lines; these things are done automatically in the Para primitive.

```
function Line(h*: Hlist) =
  Vlist(Hbox(Hskip(lmargin), h), lineskip)
```

Center centers a line of text within the page margins in effect.

```
function Center(h*: Hlist) =
  HboxTo(rmargin, Hskip(lmargin), hfil, h, hfil)
```

Underline produces a continuous underline under a word or phrase. Unfortunately, it has to do it by building a box. Text underlined by this means cannot be broken across line boundaries in paragraphs. The negative dimension on BlackBox achieves the effect giving the Vbox the same baseline as the original text, so that it will line up correctly with non-underlined text.

```
function Underline(h*: Hlist) =
  Vbox(Hbox(h), BlackBox(Hmeasure(h), -3, 4))
```

Outline, a relatively complicated function, is extremely useful. It draws an outline of thickness *t* around a box *b*, spaced out by a margin of *space*. Here one can see local variables in use. Their use avoids some repetition of function calls, which is desirable for keeping good editing performance. (Yes, a style file designer has to worry about performance as well as about appearance and structure.)

```
function Outline(t: Num, space: Num, b: Box) =
  local
    hsp = Hskip(space),
    w = Width(b),
    r = Height(b),
    d = Depth(b)
  in
    Vbox(
      BlackBox(w + 2 * (t + space), t, 0),
      Hbox(
        BlackBox(t, r + space, d + space),
        hsp, b, hsp,
        BlackBox(t, r + space, d + space)),
      BlackBox(w + 2 * (t + space), t, 0))
```

Struts, which are invisible boxes, are useful for forcing a container to have the specified minimum transverse width.

```
function Hstrut(w: Num) =
  WhiteBox(w, 0, 0)

function Vstrut(h: Num, d: Num) =
  WhiteBox(0, h, d)

{ standard structuring tools }
```

Here we see some common structuring functions and the concept of binding style to structure. In this style, for example, a chapter title is set in 18-point type and is centered. The end of a chapter ends a page. It would be quite normal for a style file that imports this style file to override the definitions of Chapter and Section to achieve their own particular styles.

```
function Chapter(title: @Hlist, body*: Vlist) =
  Vlist(UseFont(fontfamily, fontface, 18, Center(title)),
    Vskip(24), body, PageBreak)

function Section(title: @Hlist, body*: Vlist) =
  Vlist(UseFont(fontfamily, boldface, fontsize, Hbox(title)),
    Vskip(16), body, Vskip(16))
```

Item defines an entry in an itemized list. Here each entry is marked with an outdented bullet (the special character encoded by the octal value 267).

```
function Item(h*: Hlist) =
  let lmargin = lmargin + 40, indent = -14 in
    Vlist(Para("\267", Hskip(8), h), Vskip(4))
```

EnumItem defines an entry in an enumerated list. Its formatting is similar to Item, except that an entry begins with an outdented number (or, for that matter, anything else the user may type there).

```
function EnumItem(num: @Hlist, h*: Hlist) =
  let lmargin = lmargin + 40, indent = -14 in
    Vlist(Para(HboxTo(14, num, Hskip(8)), h), Vskip(4))
```

Here are some examples of the objects defined above:

A centered line of text

A centered, outlined line of text

- An Item

1. An EnumItem

C.2. Some more complex examples

Here is a function to produce a paragraph highlighted by centering it and putting a box around it. The width parameter governs the width of the paragraph.

```
function BoxPara(width: Num, h*: Hlist) =
  Center(
    Outline(1, 4,
      let lmargin = 0, rmargin = width, indent = 0 in
        Vbox(Para(h))

unit aboxpara =
  BoxPara(200, "This is a paragraph surrounded...")
```

This is a paragraph surrounded by a box, set out from the text by a margin of 4 points.

This next example typesets a fraction. The interesting features are that numerator and denominator are centered along a common centerline and that the fraction bar is as wide as the wider of them. The CenterTo utility function centers its argument within a space of a specified width, in contrast to Center which centers its argument between the page margins.

```
function CenterTo(w: Num, h*: Hlist) =
  HboxTo(w, hfil, h, hfil)

function Fraction(num: Hlist, denom: Hlist) =
  local w = Max(Hmeasure(num), Hmeasure(denom)) in
    Vbox(
      CenterTo(w, num), Vskip(2),
      BlackBox(w, 0.5, 0), Vskip(2),
      CenterTo(w, denom))

unit somefractions =
  Vlist(
    Center(Fraction("3x - 2y", "5xy")),
    Vskip(14),
    Center(Fraction("k", "7a - 3b + c")))
```

$$\frac{3x - 2y}{5xy}$$

$$\frac{k}{7a - 3b + c}$$

Here is a simple three-column table. Like most two-dimensional layouts produced by Lilac, its definition uses two functions. The outer function, Table3, is mostly a convenience and a structuring device. The inner function, Table3Row, really does the work. Arguments to Table3 set the sizes of the left and middle columns; there is no need to specify the size of the right hand column. This table includes some styling: entries in the first column are automatically bold. Controlled column widths are achieved by building constrained horizontal boxes (HboxTo). The predefined constant **hfil** is infinitely stretchable glue, which pads out the box, making the text flush left within it. Putting **hfil** glue on both sides of each column box would have produced centered columns.

The defaults shown here are merely declarations to establish the existence of the global variables **size1** and **size2**.

```
defaults
  size1 = 100
  size2 = 100

function Table3(lsize: Num, msize: Num, v*: @Vlist) =
  let size1 = lsize, size2 = msize in
    v

function Table3Row(left: @Hlist, mid: Hlist, right*: Hlist) =
  Hbox(Hskip(lmargin),
    HboxTo(size1, Bold(left), hfil),
    HboxTo(size2, mid, hfil),
    right)

unit atable3 =
  let lmargin = lmargin + 100 in
    Table3(70, 150,
      Table3Row("H", "Hydrogen", "1"),
      Table3Row("He", "Helium", "2"),
      Table3Row("N", "Nitrogen", "7"),
      Table3Row("O", "Oxygen", "8"))))
```

H	Hydrogen	1
He	Helium	2
N	Nitrogen	7
O	Oxygen	8

Here we have a useful element found many times in Appendix B: the command description. I know of no WYSIWYG editor in which this could be defined as a single structural element, though it most certainly is one. Defining it as such makes it very painless to compose a list of such descriptions.

The Line function comes from basics.li, presented above.

```
function CmdDescription(name: @Hlist, key: Hlist, descr*: Hlist) =
  Vlist(
    Line(HboxTo(150, Bold(name), hfil), key),
    let lmargin = lmargin + 30, indent = 0 in Para(descr),
    Vskip(6))

unit adescription =
  CmdDescription("Cut", "ctrl-D",
    "Delete the current selection and move...")
```

Cut ctrl-D

Delete the current selection and move its contents to the cut buffer. Leave an insertion point at the point of deletion.

Next we have a somewhat more complicated table: a table with vertical and horizontal rules. This table was used as an example in Section 5.6; here is a better, more versatile definition, more fully described. It has a header row for column titles, set off from the rest of the table by an extra thick rule.

RT4Bar is a convenience function; it simply makes a horizontal rule of the appropriate size and position to fit this table.

```
function RT4Bar =
  Hbox(Hskip(lmargin),
    BlackBox(
      size1 + size2 + size3 + size4 + 1,
      1, 0))
```

The top-level function, RuledTable4, allows the user to specify column widths. It also takes two substantial arguments: a header, a fixed element intended to be filled by one RT4Row; and a body, a list to be filled by any number of RT4Rows. The top-level function is responsible for the top horizontal rule and for the extra thickness below the header row.

```
function RuledTable4(s1: Num, s2: Num, s3: Num, s4: Num,
    header: @Vlist, body*: @Vlist) =
  let size1 = s1, size2 = s2, size3 = s3, size4 = s4 in
    Vlist(RT4Bar, header, RT4Bar, v,
      Vskip(8))
```

The row function, RT4Row, does most of the work. It is responsible for the vertical rules within each row and for the horizontal rule below it. It is also responsible for the boldness of the element in the first column.

```
function RT4Row(field1: @Hlist, field2: Hlist, field3: Hlist,
    field4: Hlist) =
  Vlist(
    Hbox(Hskip(lmargin),
      HboxTo(size1, rtvertbar, Bold(field1), hfil),
      HboxTo(size2, rtvertbar, field2, hfil),
      HboxTo(size3, rtvertbar, field3, hfil),
      HboxTo(size4, rtvertbar, field4, hfil),
      rtvertbar),
    RT4Bar)
```

Here is an actual table of this kind. Note the use of the Plain styling function in the header row. It overrides the inherent boldness of the first column, in the interest of having all column titles in a uniform style.

```
unit aruledtable =
  UseFont(fontfamily, fontface, 12,
    RuledTable4(70, 100, 70, 185,
      RT4Row(Plain("Symbol"), "Name", "Atomic no.", "Found in"),
      RT4Row("H", "Hydrogen", "1", "water"),
      RT4Row("He", "Helium", "2", "balloons"),
      RT4Row("C", "Carbon", "6",
        "diamonds, graphite, coal"),
      RT4Row("O", "Oxygen", "8", "air, water"),
      RT4Row("Si", "Silicon", "14", "computers")))))
```

Symbol	Name	Atomic no.	Found in
H	Hydrogen	1	water
He	Helium	2	balloons
C	Carbon	6	diamonds, graphite, coal
O	Oxygen	8	air, water
Si	Silicon	14	computers

The last example is my favorite demonstration of Lilac's powers. Here we typeset an abbreviated Periodic Table of the Elements. Due to the limited size of the page and the lack of small screen fonts, I cannot do the full Periodic Table, but the first three rows fit satisfactorily.

As in the other table examples, we use a function for each structural level. PeriodicTable handles the table as a whole and establishes its cell size and row width; it is responsible for the rule along the bottom edge of the table. PeriodicRow handles a row of the table, and is responsible for the rule at the right edge of each row. PeriodicCell describes one cell; this is where most of the work gets done. Each cell is responsible for the rule at its top and the rule at its left. It is also responsible for the details of typesetting the elements within it. CellBar is a simple convenience function for building the vertical rule between cells.

```
function CellBar =
  BlackBox(1, cellsize, 1)

function PeriodicTable(size: Num, cellsinrow, v*: @Vlist) =
  let cellsize = size in
    Vlist(v,
      Hbox(Hskip(lmargin),
        BlackBox(1 + (cellsize + 1) * cellsinrow, 1, 0)))

function PeriodicRow(h*: Hlist) =
  Hbox(Hskip(lmargin), h, CellBar)
```

A cell has four elements: atomic number, atomic weight, atomic symbol, and the name of the element. Here we specify their arrangement: number at the top left, weight at the right (note the use of **hfil** to space them); symbol in the middle in 18 point type; name centered below it. Note the use of CenterTo to center things in an unusual space (not the usual page margins).

```
function PeriodicCell(atn: @Hlist, wt: @Hlist, sym: @Hlist,
    name: @Hlist) =
  Size10(
    Hbox(CellBar,
      VboxTo(cellsize,
        BlackBox(cellsize, 1, 0),
        Vskip(2),
        HboxTo(cellsize,
          Vstrut(8, 3), Hskip(4),
          atn, hfil, wt, Hskip(4)),
        Vskip(8),
        CenterTo(cellsize, Size18(sym)),
        Vskip(5), CenterTo(cellsize, name))))
```

This table has a special complication: its top row is incomplete, and the missing cells should not be outlined. The NoCell function provides a filler for these positions.

```
function NoCell =
  Hskip(cellsize + 1)
```

Here is the table itself. Everything is neatly, hierarchically structured—except the empty cell to the right of Hydrogen. Here I have resorted to ad-hoc techniques to provide a rule on the right size of the Hydrogen cell. The resulting picture appears on the next page.

```
unit periodic =
  PeriodicTable(60, 8,
    PeriodicRow(
      PeriodicCell("1", "1", "H", "Hydrogen"),
      CellBar, Hskip(cellsize),
      NoCell, NoCell, NoCell, NoCell, NoCell,
      PeriodicCell("1", "4", "He", "Helium")),
    PeriodicRow(
      PeriodicCell("3", "7", "Li", "Lithium"),
      PeriodicCell("4", "9", "Be", "Beryllium"),
      PeriodicCell("5", "11", "B", "Boron"),
      PeriodicCell("6", "12", "C", "Carbon"),
      PeriodicCell("7", "14", "N", "Nitrogen"),
      PeriodicCell("8", "16", "O", "Oxygen"),
      PeriodicCell("9", "19", "F", "Fluorine"),
      PeriodicCell("10", "20", "Ne", "Neon")),
    PeriodicRow(
      PeriodicCell("11", "23", "Na", "Sodium"),
      PeriodicCell("12", "24", "Mg", "Magnesium"),
      PeriodicCell("13", "27", "Al", "Aluminum"),
      PeriodicCell("14", "28", "Si", "Silicon"),
      PeriodicCell("15", "31", "P", "Phosphorus"),
      PeriodicCell("16", "32", "S", "Sulphur"),
      PeriodicCell("17", "35", "Cl", "Chlorine"),
      PeriodicCell("18", "40", "Ar", "Argon")))
```

1 1 H Hydrogen							1 4 He Helium
3 7 Li Lithium	4 9 Be Beryllium	5 11 B Boron	6 12 C Carbon	7 14 N Nitrogen	8 16 O Oxygen	9 19 F Fluorine	10 20 Ne Neon
11 23 Na Sodium	12 24 Mg Magnesium	13 27 Al Aluminum	14 28 Si Silicon	15 31 P Phosphorus	16 32 S Sulphur	17 35 Cl Chlorine	18 40 Ar Argon

Bibliography

[Apple 84] Apple Computer, Inc.
MacWrite
20525 Mariani Ave., Cupertino, CA 95014, 1984

[Asente 87] Paul J. Asente
Editing Graphical Objects using Procedural Representations
Doctoral dissertation, Stanford University, 1987

[Chamberlin 82] Donald D. Chamberlin *et al.*
JANUS: An Interactive Document Formatter Based on Declarative Tags
IBM Systems Journal, vol. 21 no. 3, 1982

[Chamberlin 88] Donald D. Chamberlin *et al.*
Quill: An Extensible System for Editing Documents of Mixed Type
21st Hawaii International Conference on System Sciences
Kailu-Kona, Hawaii, January 5-8, 1988

[Chen 86] Pehong Chen *et al.*
The VORTEX Document Preparation Environment
TEX for Scientific Documentation: Second European Conference
Strasbourg, France, June 1986

[Chen 87] Pehong Chen and Michael A. Harrison
Multiple Representation Document Development
Computer, vol. 21, no.1, Jan. 1988

[Chen 88] Pehong Chen
A Multiple Representation Paradigm for Document Development
Doctoral dissertation (in preparation), U.C. Berkeley, 1988

[Donzeau-Gouge 84] V. Donzeau-Gouge, B. Lang, B. Melese
Practical Applications of a Syntax Directed Program Manipulation Environment
7th International Conference on Software Engineering, Orlando, Florida, 1984

[Frame 87] *FrameMaker Reference Manual*
Frame Technology Corporation, 2911 Zanker Road, San Jose, CA 95134, 1987

[Gosling 82] James Gosling
Unix Emacs Manual
Carnegie-Mellon University, 1982.

[Knuth 73] Donald E. Knuth
The Art of Computer Programming, vol. 1, p. 246
Addison-Wesley, Reading, Massachusetts, second edition, 1973

[Knuth 81] Donald E. Knuth and Michael F. Plass
Breaking Paragraphs into Lines
Software—Practice and Experience, vol. 11, 1981

[Knuth 84] Donald E. Knuth
The TEXbook.
Addison-Wesley, Reading, Massachusetts, 1984

[Lamport 86] Leslie Lamport
LATEX: A Document Preparation System
Addison-Wesley, Reading, Massachusetts, 1986

[Lampson 79] Butler W. Lampson
Bravo Manual
Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA, 1979
Part of the *Alto User's Handbook*

[McJones 87] Paul R. McJones and Garret F. Swart.
Evolving the UNIX System Interface to Support Multithreaded Programs.
DEC/SRC Research Report 21, 1987

[Microsoft 87] *Microsoft Word reference manual, version 3.0*
Microsoft Corporation, Box 97017, Redmond, WA 98073, 1987

[Nelson 85] Greg Nelson
Juno, a constraint-based graphics system
Siggraph '85, San Francisco, July 22-26, 1985

[Ossanna 76] J. F. Ossanna, Jr.
NROFF/TROFF User's Manual
Computing Science Technical Report 54, Bell Laboratories, 1976

[Rabin 81] M. O. Rabin
Fingerprinting by random polynomials
Department of Computer Science, Harvard University, Report TR-15-81, 1981

[Reid 80] Brian K. Reid and Janet H. Walker
Scribe User's Manual, Third Edition
Unilogic, Ltd; 605 Devonshire St., Pittsburgh PA, 1980

[Reps 87] Thomas W. Reps and Tim Teitelbaum
The Synthesizer Generator: A System for Constructing Language-Based Editors
Springer-Verlag, New York, third edition, 1988

[Rovner 85] Paul Rovner, Roy Levin, and John Wick
On Extending Modula-2 for Building Large, Integrated Systems.
DEC/SRC Research Report 3, 1985.

[Stallman 81] Richard M. Stallman
EMACS: The Extensible, Customizable, Self-Documenting Display Editor.
AI Memo 519a, MIT, March, 1981

[Sturgis 83] Howard Sturgis
(Problem 83-2 in the Problems column)
Journal of Algorithms, vol. 4 no. 1, March 1983, p. 86

[Tesler 72] Larry Tesler
PUB: The Document Compiler
Stanford Artificial Intelligence Project Operating Note 70, 1972

[Thacker 87] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr.
Firefly: A Multiprocessor Workstation
DEC/SRC Research Report 23, 1987.

[Trimberger 81] Stephen Trimberger
Combining Graphics and a Layout Language in a Single Interactive System
18th Design Automation Conference
Nashville, Tennessee, June 29 - July 1, 1981

[Zabala 82] Ignacio A. Zabala-Sabeles
Interacting with Graphic Objects
Stanford University doctoral dissertation, 1982

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating go to's while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986. Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301