Abstract

Presentation by Tree Transformation

by

Vance Maverick
Doctor of Philosophy in Computer Science

University of California at Berkeley

Michael A. Harrison, Chair

Every interactive system requires a presentation mechanism, to show the user the data it handles. Often, the relationship between the data and its presentation is complex; further, it is often mediated by a style mechanism, allowing the user or a designer to describe how the data should be displayed. It is a standing engineering challenge to develop a presentation model that is *flexible*, handling many kinds of data and layout; *powerful*, giving the user extensive control over appearance; and *efficient* enough for interactive work.

In this dissertation, we propose a model of presentation by tree transformation. Because information often has a hierarchical logical structure, trees are widely used to represent documents and other data. The layout or presentation of a document is also often modeled as a computation over a tree. But these trees are not generally identical. In other words, presentation can be seen as a mapping between trees. Casting it as a formal tree transformation offers both expressive, compact style specifications and efficient implementation.

We present a general framework for presentation by tree transformation. It has been implemented as part of Ensemble, a software development environment and multimedia document system developed in our research group at Berkeley. We describe the tree-transformation mechanism, and a language for specifying presentation styles as transformations. In addition, we have developed several distinct output tree languages or "media": a pretty-printer for formatting programs, a graphing tool for presenting numerical data as x, y plots, and a general tree viewer for displaying the structure of any document.

We define four measures of efficiency that are important for interactive presenta-

tion, and show that the implementation can be tuned to provide good performance according to each measure.

Chapter 1

Introduction

Every interactive program must present information to the user. In a few cases, the logical content of this information may be identical to its appearance: for example, in a drawing program, the user edits a document whose content is the very lines, rectangles, etc. seen on the screen. But in general, the appearance of the information is distinct from the information itself. In a database, for instance, the records may contain a compact binary representation of text, numbers, etc.; to create a report, the database program generates bitmaps and character outlines, in some spatial arrangement, to represent the information on screen or on paper.

In the case of a simple database display or a basic text editor, the relationship between information and appearance may be straightforward. But in many important applications, such as a word processor, a Web browser, a report generator for a database, or a multimedia design tool, the relationship is quite complex. And increasingly, users demand sophisticated control over the appearance of the information they work with.

In this dissertation, we present a model for computing the appearance of information, based on the fundamental operation of *tree transformation*. We demonstrate that it allows the user powerful control over appearance, and that it is flexible and efficient enough for a variety of interactive applications.

1.1 An Example of the Model

For an example of the kinds of problems we address in this dissertation, consider a simple problem in the presentation of structured documents. The input document is a

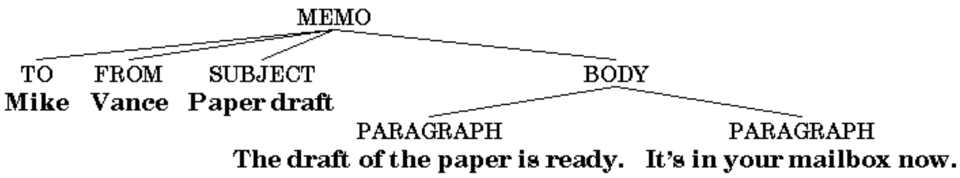


Figure 1.1: Tree structure of a sample "memo" document. The text of the memo is represented as the content of its leaves.

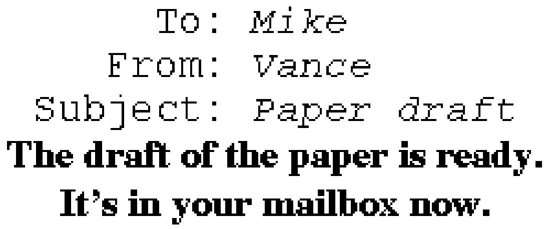


Figure 1.2: A sample layout for the memo.

memo; its logical structure is illustrated in Figure 1.1. The memo class has obligatory "from", "to", and "subject" fields, and a body with a variable number of paragraphs, two in this instance. A flexible presentation tool ought to be able to display this memo in a number of different layouts, according to style guidelines supplied by the user or a designer. For example, both Figure 1.2 and Figure 1.3 are possible realizations of this document.

There are many possible ways to compute these layouts from the given tree structure. One common and natural strategy for text presentation is to represent the layout by nested regions of vertical and horizontal formatting, like the `\vbox` and `\hbox` constructs of TeX [52]. Figure 1.4 shows how this can be done for the first layout; and Figure 1.5 shows

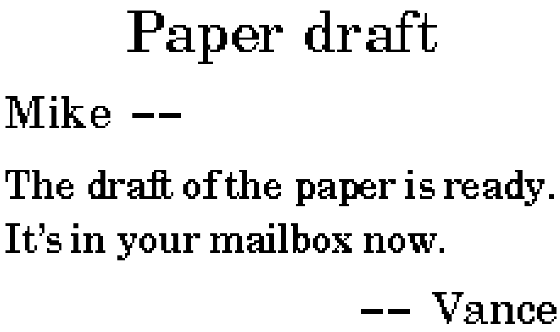


Figure 1.3: An alternate layout for the memo.

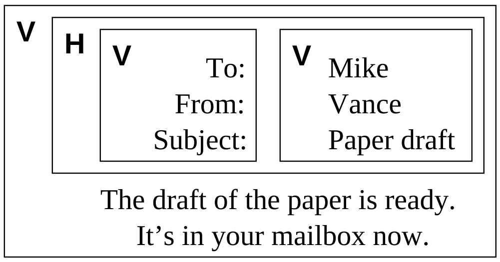


Figure 1.4: The layout of Figure 1.2, expressed as nested boxes: within each box, elements are laid out horizontally (H) or vertically (V), with justification.

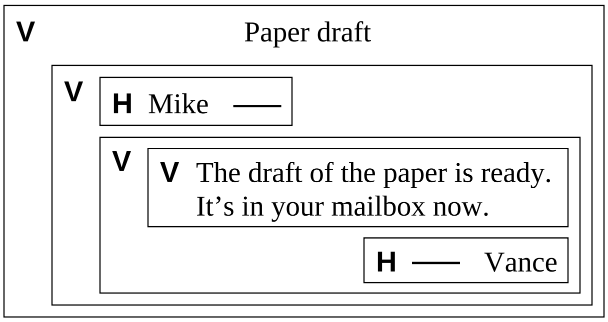


Figure 1.5: Box layout of Figure 1.3.

how to do it for the second. In both cases, the nested boxes form a hierarchy, that is, a tree.

So the input document is a tree; and using a formatting "language" of text terminals, as well as horizontal and vertical nonterminals, we can express each layout as a tree, with a different but related structure. The presentation of a document, then, can be seen as a translation from one class of trees (documents) to another (layouts). In Figure 1.6, we illustrate the two translations used for these examples.

The field of *tree transformations* is devoted to just such translations. Its history goes back at least to the 1930s, with the conversion rules of Church's lambda calculus [25]. Since that time, many variations of tree transformation have been developed, with different computational properties and specification mechanisms. A useful overview of the field is available in the dissertation of Eduardo Pelegrí-Llopart [68]. In this dissertation, we develop the idea of presentation by tree transformation, and present a particular transformation model, with several practical applications.

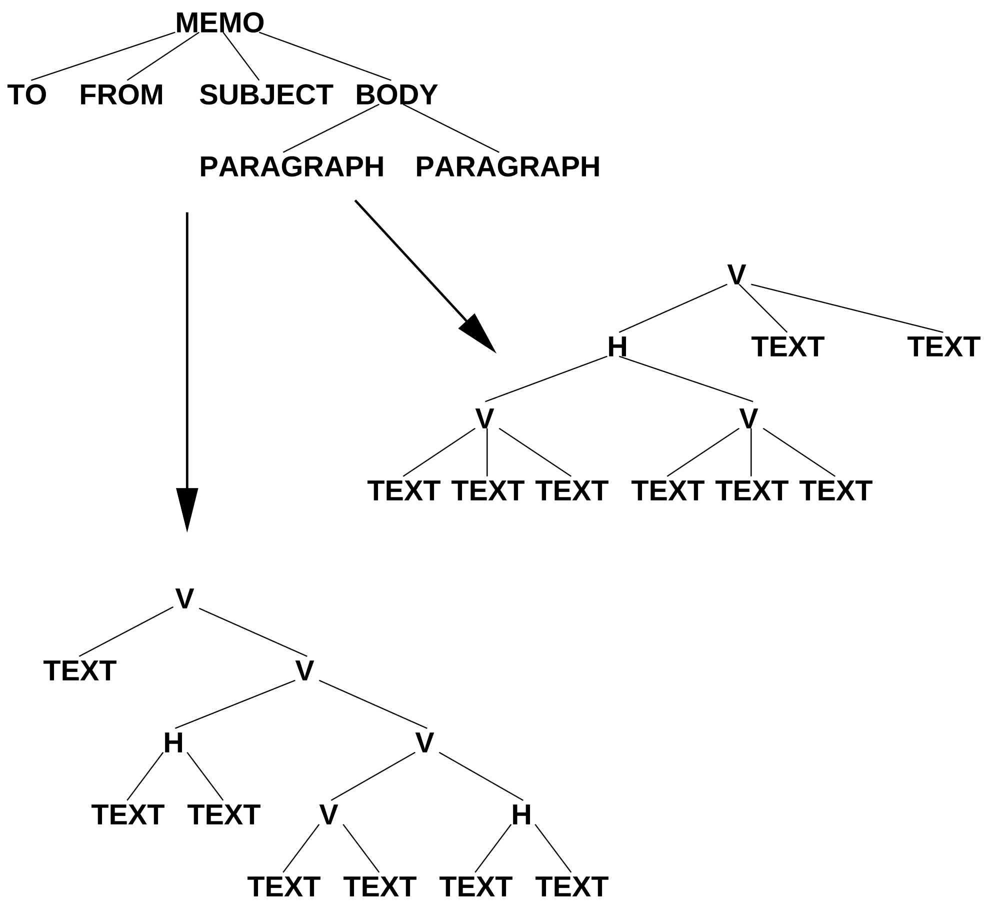


Figure 1.6: The two presentations of the memo, modeled as translations from the input tree to two distinct layout trees.

1.2 Thesis

Our thesis is that, as a presentation mechanism, tree transformation is *flexible*, *powerful*, and *efficient*. By "flexible" we mean two things. First, transformation can be used for different output media, with widely varying layout computations, not just the nested-box layout of these examples. Second, a single presentation style file, written as a transformation, can handle a broad class of documents, with a variety of structures, much richer than our simple memo example. And by "powerful" we mean that, within a single output medium, the transformation language allows the user to specify widely varying appearances for the same document.

Finally, by "efficient" we mean that it performs well according to four specific, quantitative measures. In each case, our goal is to achieve performance sublinear in the size of the document, approximately $O(\log N)$, where N is the number of nodes in the document. There are three measures which are important for any interactive system:

Startup time is the time between selecting a document and seeing it on the screen. Part of this time, of course, is due to loading the document into memory; while this is important to the user, it is outside our scope.

Refresh time is the time taken to redraw a window, when it is exposed by the window manager, or the user jumps to a new position in the document.

Size is the memory consumption of the presentation's data structures. In editing environments, it is not uncommon for users to have 100 or more documents open at once; so size must be kept low. Further, it should be possible to shrink inactive presentations, to a small fraction of normal size, or even to a fixed-size stub.

And if the tool also supports editing, then there is a fourth important measure of efficiency:

Change time is the time taken to process a single simple change. For Ensemble, the most common change is a single-character insertion; it is acceptable for less common changes to take longer.

Our goal in this dissertation will be to make change time logarithmic in document size. User-interface research suggests that, for good interactive performance, this time must be kept below a fixed interval, about 1/10 second; see, for example, Shneiderman [81, p.

284]. We believe constant change time is possible within our model, and we will discuss how it can be achieved.

1.3 Organization

In Chapter 2, we present the architecture of Ensemble, the larger system within which this work was carried out. We focus especially on the presentation aspects of the architecture, because the following chapters rely on them.

Chapter 3 is a preliminary study of presentation in Ensemble. It reports on the "line-based" presentation tool, a special-purpose tool for displaying programs. We evaluate it in light of our thesis, and show that, while useful, it does not meet all our goals.

In Chapter 4, we return to the transformation model, and show how a textual presentation schema can specify transformational rules to present documents.

In Chapter 5, we discuss in detail the three presentation media developed so far with the transformation mechanism. These are a full text medium, with many more features than the memo example illustrates; a tree-display tool for showing the structure of documents; and a graphing tool to display numerical documents as x, y plots.

Chapter 6 presents the implementation of the transformation mechanism. This is in some ways the heart of our research. We begin with a straightforward implementation, then analyze its performance according to the four efficiency measures described above. Then we show how the implementation has been improved in order to reach the goals we set.

In Chapter 7 we discuss related work, from the theoretical framework of tree transformation, to alternate models of presentation, to other applications of tree transformation to presentation problems.

Finally, we present our conclusions, and a summary of our research contribution, in Chapter 8.

Chapter 2

The Presentation Architecture of Ensemble

The Ensemble project began in the late 1980s as a fusion of two lines of software research: one in programming-language editing environments [11], and the other in document systems [23]. An important goal of the project has been to support software development and multimedia editing in one system, using a single document model. While Ensemble has not fully achieved this goal, it has reached a useful, stable state, with enough of the desired features to serve as a framework for research in programming languages [93], multimedia presentation systems [42], and user interfaces [32].

Ensemble can display and edit parsed programs and multimedia documents: Figure 2.1 shows a single session with three active documents. The chief document medium is natural-language text; but limited implementations of images, graphics and video have also been developed. Documents may be compound, that is, composed of subdocuments of various media, including programs.

Documents may be displayed in multiple different presentations at once, as in Figure 2.2. Several presentation mechanisms are available; they are largely restricted to two-dimensional layout, but apart from this are varied and flexible. The user controls presentation by editing textual style sheets, written in special-purpose languages.

Text editing is mediated by standard insertion-point cursors and contiguous selections. Using a Lisp-based extension language, the user can control the bindings of keys to edit operations, and compose new operations from the language primitives.

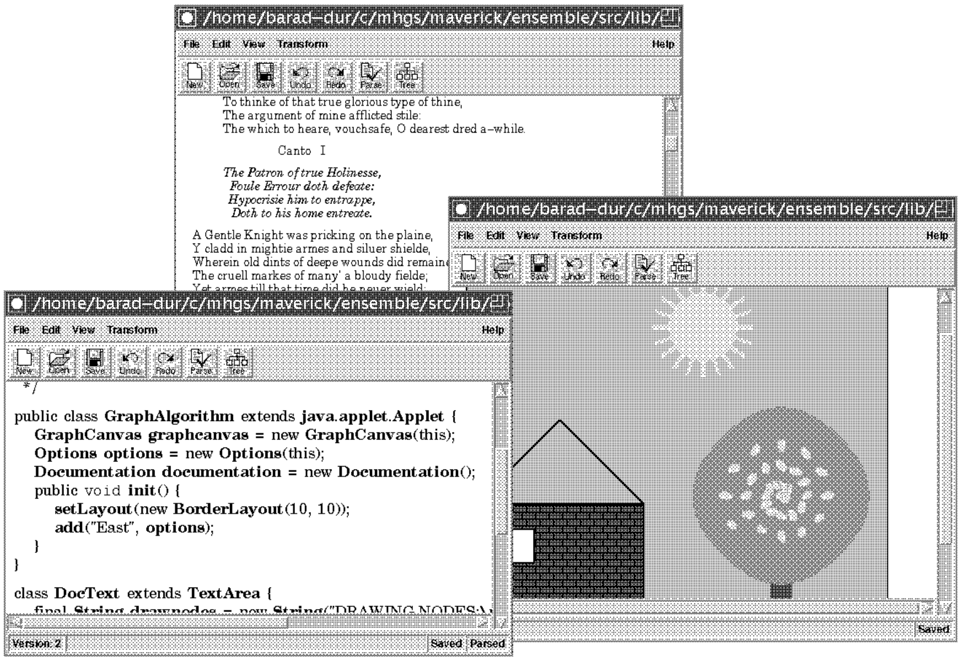


Figure 2.1: Screendump from Ensemble, showing three documents: a natural-language text, a Java program, and a graphical figure.

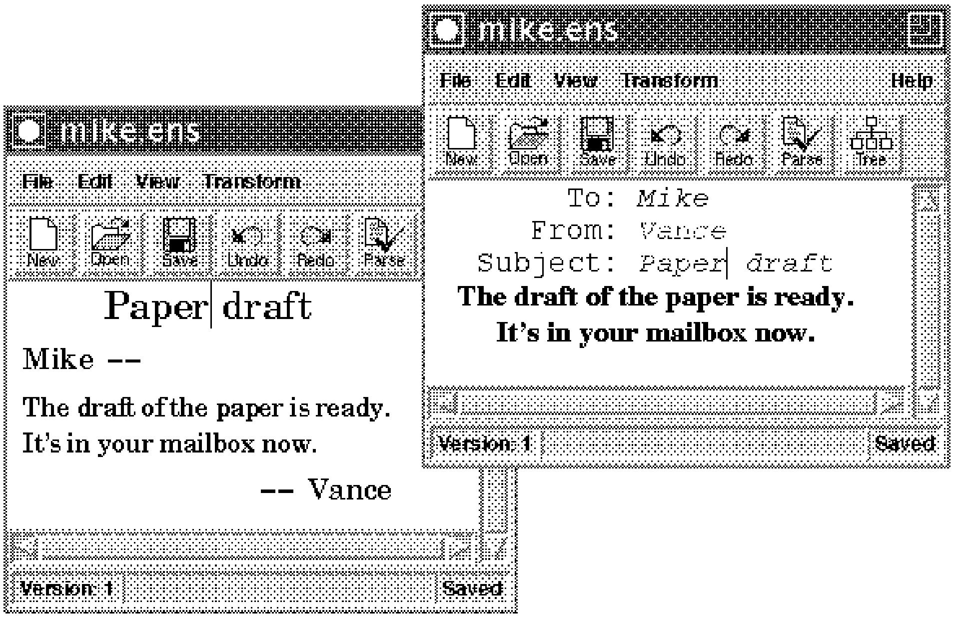


Figure 2.2: Two presentations of the same document. Both are "live" for editing.

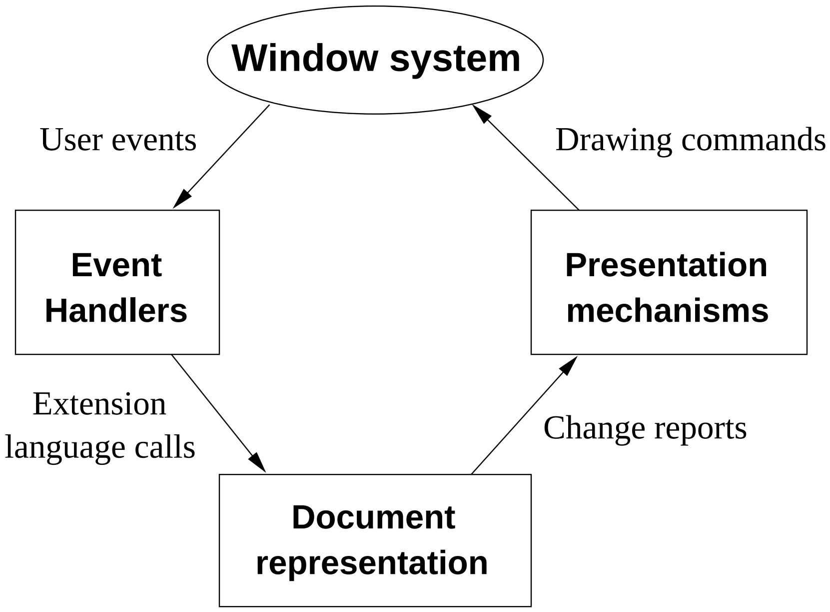


Figure 2.3: The overall architecture of Ensemble, illustrated as the flow of control between modules. Of the three main divisions of the architecture (in rectangles), our concern in this dissertation is with presentation.

Ensemble provides extensive programming-language features, including syntactic and semantic analysis. Errors discovered by analysis are reported in the presentation. All editing changes, including analyses, are recorded, and may be undone and redone.

In this chapter, we present the architecture of Ensemble. It has been described elsewhere, but this account is somewhat different. In comparison to the unpublished paper of 1995 [33], we concentrate on the presentation aspects of the architecture. And while Ethan Munson's dissertation [65] also concentrates on presentation, there has been considerable evolution in the design since 1994. We begin with the broadest view of the system, and steadily add detail.

2.1 Overall Architecture

Figure 2.3 shows how Ensemble divides the various tasks of a document system between its modules. The fundamental representation of any object is a *document*. Various presentation mechanisms convert documents to a graphical representation, and render them to physical devices, for example by issuing drawing commands to a window system. When the user presses a key, the resulting event is passed to one or more event handlers, which use extension language services to modify the document. These changes propagate through the presentation mechanisms, and are reflected in the window system.

The Ensemble data model and version control system, from which the document representation is built up, are treated in the work of Tim Wagner [93]. Event handling and extension language services are the work of Brian Dennis [32]. In this chapter, we are concerned primarily with presentation.

2.2 Multiple Presentations and Views

From the beginning, an important requisite of the Ensemble design has been to make it possible to present a single document in several ways at once [24]. For example, one might wish to view a textual document in one- and two-column formats at once; or to view a numerical document both as a graph and as an editable table. It is also useful to be able to view different parts of a document at the same time. Accordingly, the Ensemble architecture breaks down the editing and display of information into three layers. Briefly:

- The document is the fundamental representation of what the user has created, corresponding to the information stored on disk.
- The presentation is the appearance of the document, computed by one of a variety of mechanisms.
- A view is a rendering of a presentation, or part of a presentation, onto a physical device such as a monitor or printer.

There may be several presentations of a given document, and several views of a given presentation. We illustrate this division of the architecture in Figure 2.4. In the following sections, we describe each level in more detail.

2.2.1 Document Structure

At the document level, Ensemble data is structured as n-ary trees. The nodes in the trees are typed, and classified as terminals or nonterminals. The text of a document, or any other media content, is carried in the terminals. Nonterminals express the structure of the document. A class of documents, the possible configurations of terminals or nonterminals, is represented by a language, or grammar. In the case of a program, the structure

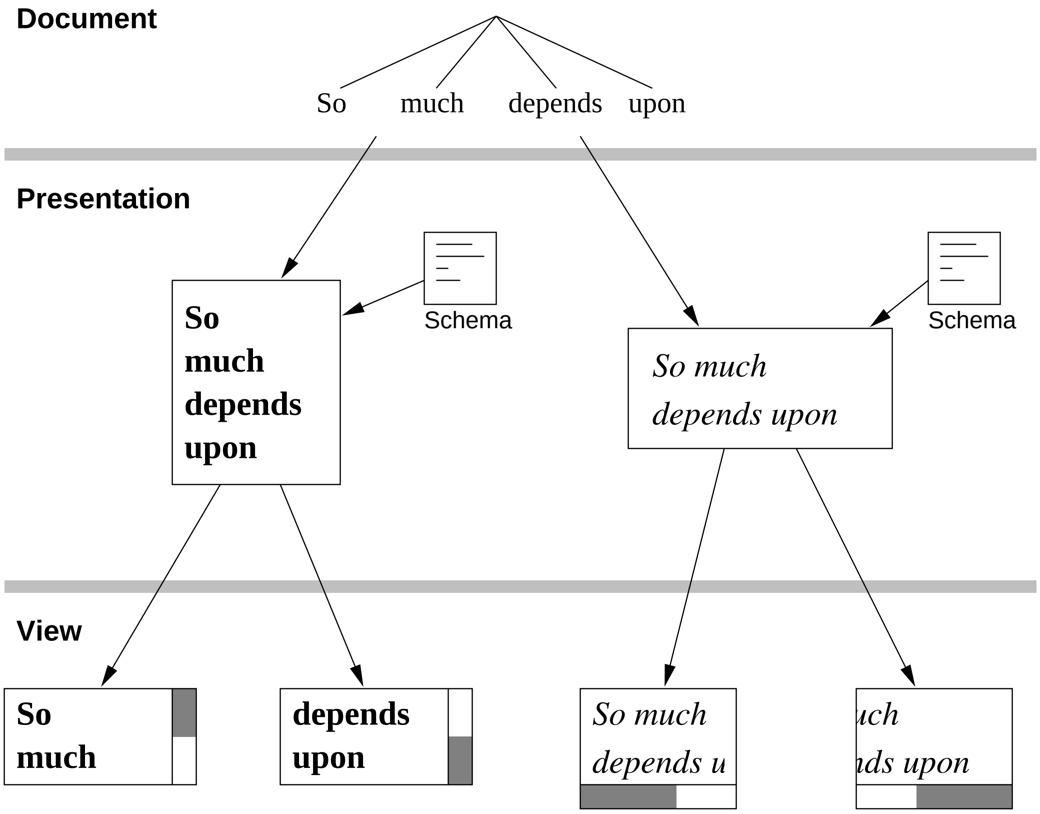


Figure 2.4: There may be any number of presentations per document, and any number of views per presentation. In addition to the document, each presentation is controlled by a *schema* or style file.

and the text are kept consistent by a parser based on the grammar. In the case of a general document, consistency is maintained by editing operations that respect the grammar.

A given tree may be viewed in different "abstractions". Essentially, these are different filters for the set of nodes. For example, a full parse tree contains nodes representing whitespace. Many presentations and other clients will wish to ignore these nodes: they do so by choosing an abstraction. (Much fuller discussions of trees at this level can be found in the work of Tim Wagner [93, 92].)

Complex documents are often broken down into multiple trees, for two distinct reasons. The most obvious reason is to share content. Consider an article containing an equation. If the equation will be reused in another context, say a letter, sharing means that the two versions will automatically be kept consistent. (There is also a space saving, which may be important with large inclusions.)

The second reason to break documents into components is to share languages. If the article with an equation is to be represented as a single tree, its language must have all the features of both articles and equations. But if the trees are divided, the article language can stay simple, and the equation language can be reused in other contexts, such as the letter.

The inclusion is represented by a special link node, which is a terminal in the outer tree. The link has a target, a node in the same or some other tree. We refer to documents with links as *compound* documents. Some clients treat the trees as separate: other clients, such as user navigation tools, treat them as fused into a single tree. Figure 2.5 shows a simple example. As we will see in the next two sections, each presentation or view contains one "general" client that views the entire document, and coordinates a group of "specialist" clients, each viewing a single document tree.

2.2.2 The Presentation Layer

Ensemble has a variety of presentation engines. In addition to the Proteus system [42, 65], there are specialized tools for program presentation (Chapter 3), and the general tree-transformation mechanism to be presented later in this dissertation (Chapters 4, 5, and 6). All these engines compute the appearance of a single document tree. In addition to the tree, each reads a *presentation schema*, a declarative specification in a special-purpose language, written by a designer to control the appearance of a class of

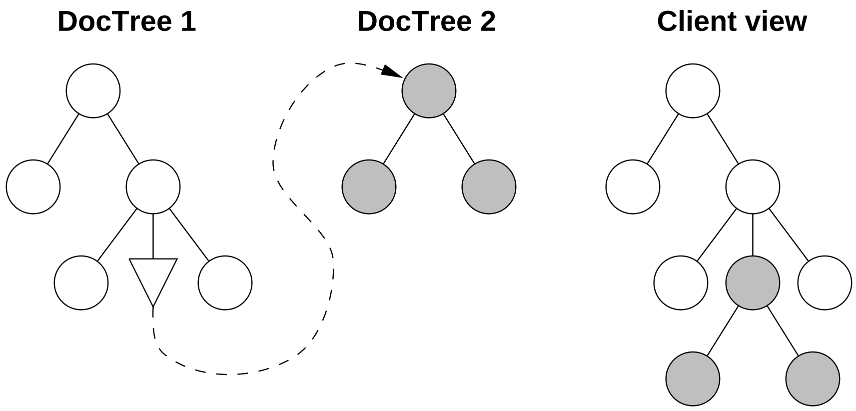


Figure 2.5: Example of a compound document. One document tree contains a link to another. To a client, they can appear as one combined tree.

documents.

Currently, most presentations lay out their contents in a single two-dimensional rectangle, but this is not a fundamental restriction of the high-level design. Paginated presentations and newspaper-style flow layout, which have more complicated layout than simple rectangles, could be handled within this framework as well. Ethan Munson implemented a video medium, which performs layout in time as well as in the two dimensions of the screen [65].

The computational model of presentation varies considerably. Proteus builds a tree closely related to the document tree, and computes its appearance by attribute computations. The tree-transforming presentations (Chapter 4) use a layout tree that is more distantly related to the document. The "line-based" program presentation tool (Chapter 3) generates a special record for each line of text, regardless of tree structure. In general, presentations may use any appropriate representation for layout; the only client is the corresponding view (see Section 2.2.3 below).

Compound documents are handled by compound presentations. Each tree of the compound document is laid out by a separate sub-presentation. These are nested in a presentation tree corresponding to the compound structure of the document. A presentation has no control over the appearance of the presentations nested within it, except that it may place them anywhere within its own rectangle.

If a subtree is linked in at more than one point in the compound document structure (for example, if an equation appears at more than one point in an article) a separate presentation is created for each occurrence. That is, even when the compound document structure is a graph, with several paths from the root to some subdocument, the compound presentation is a tree.[1]

This tree structure is called a presentation tree. (Often, speaking loosely, we will refer to the large presentation entity corresponding to a document and a view as a presentation, but the term "presentation tree" is necessary to distinguish it from individual presentations. This confusion is probably the worst historical legacy in the terminology of Ensemble.) Just as the appearance of a presentation is under the control of a presentation schema, so the construction of a presentation tree is under the control of a "style function". The job of the style function is to create and initialize the appropriate presentation for each tree in the document. It is part of the user-interface specification, written in the ExL extension language and customized by the user or a designer.

2.2.3 The View Layer

There may be any number of views of a given presentation tree, to enable the user to view different parts of the document at the same time. Each view is a tree of *renditions*: each rendition handles the display of one presentation on the physical device. Part of the task of writing a presentation tool is writing the corresponding rendition class, specialized for reading the tool's layout information. At runtime, an instance of this rendition class is generated automatically to handle each sub-presentation.

2.3 Extras: Cursors and Selections

So far, we have described the elements of a "static" document display, enough to make presentations and display them on the screen. To support user editing, Ensemble also provides cursors and selections. These are called *extras*, because they are extra information, in addition to document content, that the presentation system must display. In Figure 2.2, there is just one cursor, visible in two presentations of a document. Figure 2.6 shows the presentation abstractly: coordinate information from the cursor flows out to each of two views.

The "meaning" of a cursor is a single place in the document; the meaning of a

[1] In a production version of this architecture, it might be worth avoiding this waste of space, but in practice, few Ensemble documents have complex graph structure.

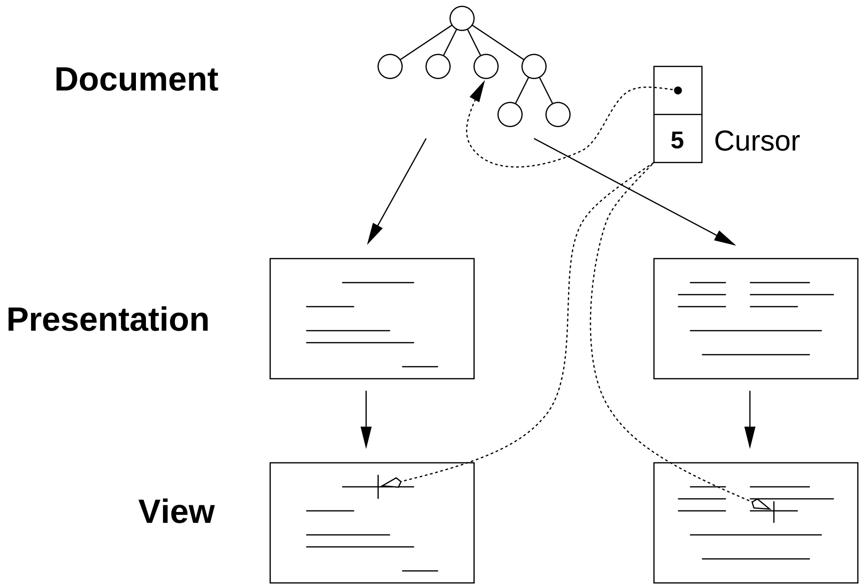


Figure 2.6: Information flow in Figure 2.2. A single cursor, defined in terms of the document structure, is represented graphically in two different layouts.

selection is a contiguous range of document content. Both of these are expressed in terms of a *location*, which is a document node plus an optional media-specific coordinate. In text, the coordinate is an integer offset into the text of the specified node. A cursor contains one location; Figure 2.6 shows a cursor defined in terms of a node plus the offset 5. A selection contains two locations, bracketing a contiguous range of document coordinates.

Each extra has a logical "home"; when the user places or moves the extra, the value at this site is updated. This home may be with a document, a presentation, or a view. It is encapsulated in an object called a *parasite*, because it is attached to the document-presentation-view architecture but is not a part of it. In Figure 2.6, the "Cursor" box is a parasite attached to the document, so that it can be drawn by any client of the document.

In order for the image of a cursor or selection to appear on the screen, its extra must be propagated from the parasite out to a presentation or view client. If the extra affects the layout, like a caret insertion cursor (`thu^s`), it goes to a presentation. In Figure 2.6, it goes to the view level; a copy of the cursor is propagated to each view, to be drawn at the point appropriate for that presentation. The presentation or view must be ready to handle any number of these extras in addition to its document or presentation input.

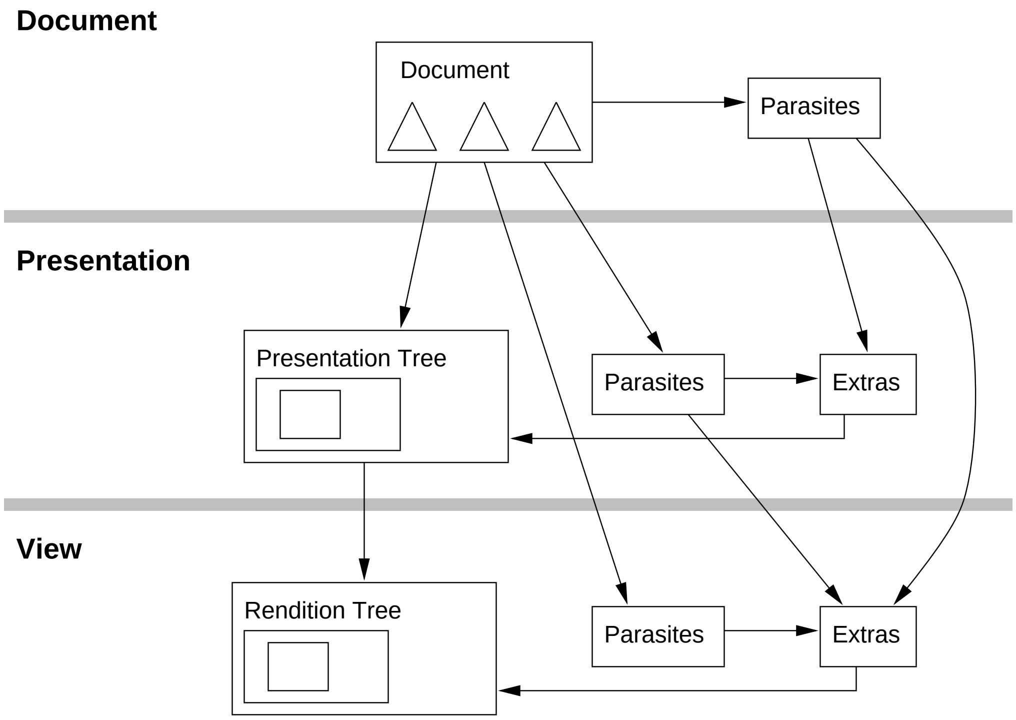


Figure 2.7: The document-presentation-view architecture, with the addition of *parasites* and *extras*.

With the addition of parasites and extras, the overall presentation architecture looks a good deal more complicated (Figure 2.7). The parasites attached at a given level are gathered in a single set. They are all dependent on the document: that is, if the document changes, they may need to reset their positions. From there, the parasite information flows to sets of extras at the appropriate drawing level. The presentation or view that draws them reads them there.

In practice, Ensemble cursors, like those illustrated in Figure 2.2, are all drawn at the view level. However, the freedom to attach them at different levels, so that they are visible just in the current view or in all views of a given presentation or document, has been valuable in building the user interface.

2.4 Change

With all these elements of the architecture in place, we are ready to examine how Ensemble actually handles a change to a document. It would be possible, of course, to recompute all the clients of the document from scratch. But to handle user edits in real time, document, presentation, view, and cursor position must all be updated within an acceptable time interval, regardless of the size of the document or the number of documents and windows currently open in Ensemble.

To achieve this, we impose a simple object model on the architecture. Each document, presentation, and view, and each cursor and selection, counts as an object. The objects are connected by an acyclic graph of dependencies: each view, for instance, depends on a presentation, which depends on a document. All these objects must provide incremental change information. The details vary from one object class to another, but the general requirement is the same: clients of an object must be able to identify rapidly which parts of it have changed.

In this section, we begin by describing how change information percolates through the network of objects. Then we turn to individual objects, and discuss in varying detail how they represent changes and handle changes in their dependencies.

2.4.1 Change Propagation

The propagation of changes through the dependency graph begins with a set of "dirty" objects. In the commonest case, this is just one document, with one changed character. Ensemble first computes the set of all objects dependent on them, directly or indirectly. Second, it sorts the objects in topological order, based on the dependencies, so that if object b depends on object a, a precedes b in the ordering. Obviously, the web of dependencies must have no cycles: if it does, this error will be detected during the topological sort.

Finally, having collected and ordered the objects to be updated, Ensemble calls a special method, `update`, on each in turn. In this method, each object examines the objects on which it depends, updates its internals according to their changes, and then returns a code indicating its own change status. This code can have the value **clean** (no changes), **dirty** (changed, but no incremental change information available), or **dirty-inc** (changed, with incremental change information). It is saved as part of the object's state, to be read further down the line by the object's own clients.

When a large document is displayed in many views, it can happen that the time to propagate a change is unacceptably long, because of the number of dependent objects and the complexity of the changes in each one. To handle this case, it is necessary to distinguish "emergency" formatting, a minimal update sufficient to reflect the change immediately to the user, and "convalescent" formatting, restoring the system to full consistency, in an interval when the user has paused and more time can be taken. This distinction is not supported in Ensemble today (see our discussion of future work in Section 8.2), but part of the solution can be provided here, using the object dependency web.

When Ensemble detects an "emergency" (perhaps keystrokes are piling up in the buffer faster than they are being processed), its first step is to identify the active document, and one active view that must be kept up to date—presumably the one in which the user is typing. Next, Ensemble identifies all the objects that lie on a dependency path from the document to the view. Then, with each new keystroke, it can update just this minimal set, in topological order. This will suffice to keep the active view up to date. Updates to objects that depend on the document, but are not in this minimal set, will be deferred until the "convalescent" pass, after the buffer has cleared.

Of course, it may still take too long to update the active view, if the user is typing fast or the reformatting is complex. If so, a further level of emergency processing will be required, in which presentations and views are allowed to become internally inconsistent. We discuss this possibility in more detail in Section 8.2.

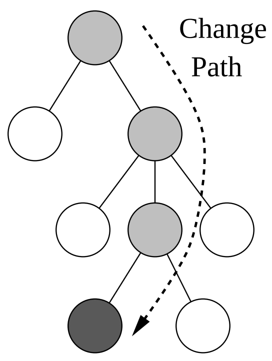


Figure 2.8: A document tree after a change. The changed node itself is marked "local changes" (dark gray), and its ancestors are marked "nested changes" (light gray). Thus, clients can find the change by following the path from the root to the change site.

2.4.2 Change Representations

In this section, we survey the representation of change information by various objects in the Ensemble architecture.

Document changes

Changes in the document tree are represented with Boolean "change bits". A node may be marked "local changes", meaning that its content has changed, or, if it is a nonterminal, that its list of children has changed. It may be marked "nested changes", meaning that at least one of its descendants has local changes. (Or it may be marked with both bits, or neither.) This scheme enables clients to find changed nodes and subtrees rapidly. Figure 2.8 shows how this works.

When there is a change in a compound document, the path of "nested changes" bits extends all the way from the root of the root document tree, through any links in the structure, to the change site, possibly in a different document tree. This is illustrated in Figure 2.9. Here there has been a change in a subdocument. A presentation of this subdocument will handle the change in the same way as in Figure 2.8. The outer presentation, though, may also have to handle a change, even though none of the structure it sees has changed. If the size of the inner presentation has changed, then the inclusion point (the triangular link node) must be treated as changed, and everything that depends on it must be updated—possibly including the relative position of the inner presentation.

In addition to these bits, individual terminals may provide more detailed change

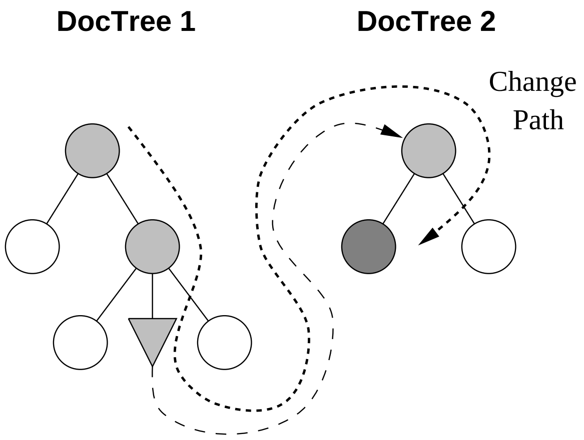


Figure 2.9: The compound document tree of Figure 2.5, after a change in a subdocument. The change site, in DocTree 2, is marked "local changes" (dark gray), as in Figure 2.8; the trail of "nested changes" (light gray) nodes extends up through the link into the root subdocument.

information. When there is a change in a token without such "history", clients must assume it has changed completely, and recompute all the information they derive from it. This is often acceptable: tokens in programs, for instance, are generally short, and full replacement is not costly. But when the content of a terminal is large and changes can be small, it is valuable for terminals to provide more detailed change reporting, so that clients may update only a fraction of the information corresponding to a whole terminal. One example is text paragraphs in natural-language documents. They may grow quite long; at the extreme, Sidney's *Defence of Poesie* (1595), a book of some 20,000 words, was originally printed without any paragraph breaks at all. But incremental performance in typing is essential. Accordingly, text paragraphs represented in Ensemble by a special "large string" node class, which can report changes in single characters or subranges. Thus, clients can update only those portions of the presentation that depend on the changed character, possibly changing only one line out of thousands.

Presentation Changes

A presentation tree may contain several individual presentations. To guide its client, the view, to the presentations that have changed, it is marked with change bits in the same fashion as the document tree. When a presentation has changed size or position, the old and new values are both recorded, so that the view may update onscreen window structure, possibly moving windows whose relative positions have changed, before calling on the individual renditions to update their windows.

Different presentations handle change reporting in different ways. With each one, though, our approach has been to make common changes efficient. In Ensemble today, by far the most common changes are single-character edits. Other changes, we find, are sufficiently rare that a relatively inefficient response, even including a full window redraw, is acceptable.

The publications on Proteus [42, 65] do not treat change processing or representation in detail. In practice, individual Proteus presentations address the common cases in ad-hoc ways. The Proteus text presentation, for instance, compiles a list of lines that are affected by a change. If there is no larger-scale change, redraw is restricted to updating these lines.

In the chapters on the line-based and tree-transformation presentation tools, we will discuss change reporting in greater detail. Here it is enough to note the diversity of the solutions. With the line-based tool, it has proven sufficient to list changed lines, so clients must redraw the whole line with each change. As we will see, the transformation tool provides a general change-reporting mechanism, using change bits like those on the document tree; but the details of change handling depend on how each specialized "medium" uses the tool's attribute mechanism. In the pretty-printer, for example, nodes are marked with size and parent-relative position; so renditions can track size and position changes in a top-down walk over the tree, in the same way that views track changes in the presentation tree.

Extras Changes

Change processing is also an important concern in the implementation of extras. When the user presses a cursor key, for example, Ensemble updates a parasite, changing the value of the extra in it. This change should be propagated efficiently to just those views that display the cursor, and it should be represented concisely enough that redrawing can be minimized.

To propagate extras changes, we use a very simple model, based on one data structure, an *extras bag*, little more than an unordered collection of extras. A parasite is itself an extras bag with just one extra; the boxes marked "Extras" in Figure 2.7 are extras bags; and each presentation in a presentation tree, and each rendition in a view tree, has an extras bag to hold the extras it must draw. Each extra in a bag is tagged with an "owner", the address of the parasite from which it was derived. The bag also has a simple "history" mechanism: when a cursor moves, for instance, this indicates that the old extra for that owner has been deleted, and a new extra inserted. Further, the bag has a `copy` method, taking another extras bag as an argument; this method reads the other extras bag's history, and repeats the same insertions and deletions. So when the user moves a cursor, it is easy to copy this change from the original parasite, to the extras bag at the parasite's drawing level, to the extras bag at the appropriate rendition or presentation.

Chapter 3

Line-Based Presentation

The first presentation tool we built in Ensemble was the "line-based" presentation system, specialized for presenting programs. This is a well-known presentation problem, with a long history of solutions. We will return to program presentation in Section 5.1, and discuss some of the literature in Section 7.3. With the "line-based" tool, though, we had two specific purposes in mind: first, to provide simple and adequate presentation services for the needs of the programming-language researchers in the Ensemble group, and second, to investigate approaches to the general engineering problems of presentations.

Line-based presentation is the mode programmers are used to, from the standard tty-style text display (deriving from line printers) to such editing tools as `vi` [47] and `emacs` [83]. The stream of text is segmented into lines delimited by special characters. Each line's appearance is determined by the characters it contains: the complete text is simply presented as a vertical concatenation of all the lines.

In Ensemble, the original line structure of a program is distributed among the terminals of its parse tree. (See Figure 3.1.) To present the lines, a presentation tool must logically reassemble them from these terminals. This is difficult, if not impossible, in a tool based on tree structure, like Proteus [42] or the tree-transformation techniques described later in this dissertation: in many programming languages, such as C and Pascal, the syntax tree of a document is essentially unrelated to its line structure. Newline markers are just one of a set of "whitespace" characters, treated during parsing as equivalent delimiters. Any number of newlines may intervene between any pair of tokens, and a comment, which Ensemble generally treats as a single whitespace token, can have any number of newlines within it. Thus the reassembly into lines requires close scrutiny of the content of the nodes, while tree-attribution systems generally attend only to their structure.

It may seem perverse to break up the lines into tokens, assemble the tokens into an unrelated tree structure, and then reassemble them logically into lines: but this strategy offers a number of advantages, in comparison both to a straightforward text editor and to existing tree-based presentations in Ensemble. In comparison to a text editor, it is easy for this tool to use the parse tree to guide its presentation decisions. For example, keywords and syntactic structures can be marked with special text attributes. Also, syntax errors, which Ensemble's parsers report as annotations on the tree, can be integrated into the lines. And in comparison to other Ensemble presentation tools, the line-based strategy saves space. Lines in programs are generally bounded in size, with few growing longer than 100 characters. This means that the entire presentation content of a line can be regenerated in a relatively short time. Thus the presentation can store just a small amount of information for each line, and regenerate the rest on the fly, whenever the line needs to be drawn or updated.

The design of the line-based presentation tool can be divided into two parts. The *framework* is a generic mechanism for handling lines and assembling them into a page, implemented as a family of C++ classes. The detailed layout and display of the lines are handled by individual line-based presentations, which are implemented by subclassing the generic mechanism.

3.1 The Framework

The most important element of the design is the representation of a line, implemented by the C++ class `LPLine`. It contains pointers to the beginning and end of the line's textual content, each represented as a *location*, the combination of a node and an integer offset. (The offset is necessary because a line may begin in the middle of a token, like the second line in Figure 3.1.) The line also contains records of the graphical height and width of the line, which enable it to be used for hit detection and overall size computation. Anything else, such as the internal layout of characters, is regenerated on demand.

The lines are stored in a special-purpose array, which lays them out vertically, with their left edges at $x = 0$. The array is designed to handle common display and update operations as fast as possible. The most common display operation is to redraw part of a window. To do this, the presentation uses the array to look up the set of lines that lie within a particular y range. The most common update operation is to locate and replace

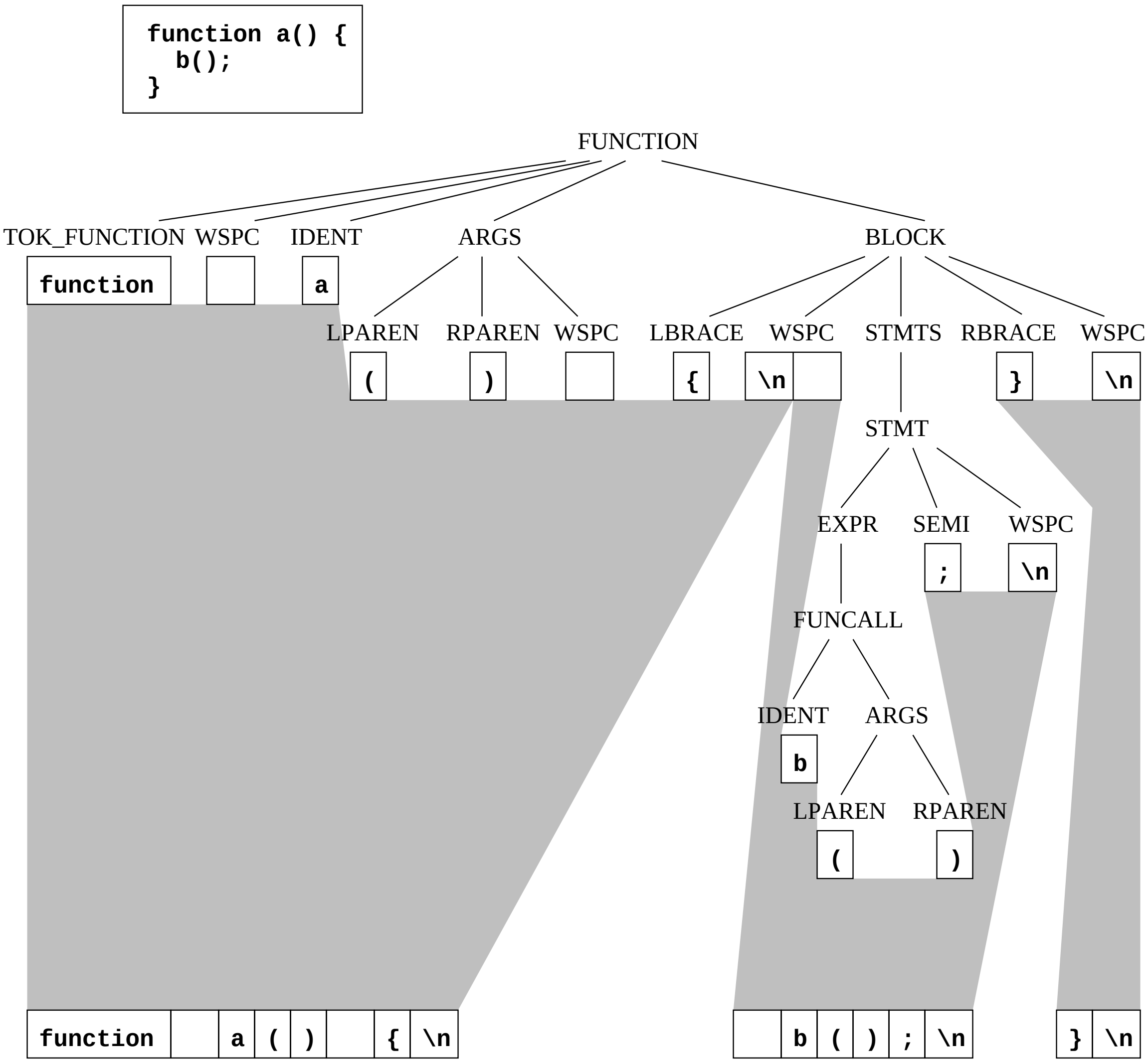


Figure 3.1: Creating a line-based display in Ensemble. The input document (top) is a program fragment of three lines. Its core representation in Ensemble is a parse tree (center); in this view, the tree includes whitespace nodes, which are not part of the abstract syntax. The line structure of the original is recovered by scanning the terminals in the tree (bottom); each line is a contiguous range of terminals.

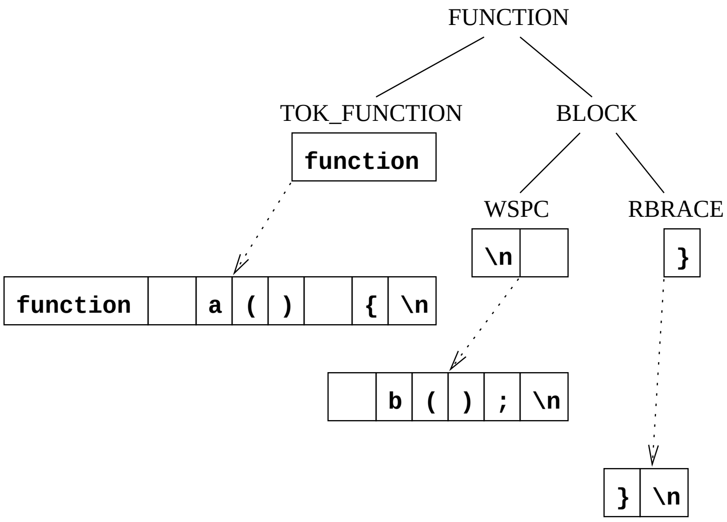


Figure 3.2: Shadow `LPLineTree` structure representing the line breakdown in Figure 3.1. Terminals contain the lines; nonterminals correspond to the input tree structure.

the lines affected by a change, using the change bits on the input tree (see Section 2.4.2).

To handle these operations in reasonable time, the "array" is implemented as a tree, called an `LPLineTree`. Terminals of the tree hold the lines, and each nonterminal holds summary information, describing the cumulative count and size of the lines below it. Thus a line can be found either by its array index or its y position, in time proportional to the depth of the tree. To make updates efficient, the tree structure simply mirrors the input document tree. Each line is stored in a terminal corresponding to the input node at which the line begins (thus, terminals containing comments may have many lines). The entire path from this input node to the root of the input tree is mirrored by nonterminal nodes. In practice, the LPLineTree contains about 5-10 percent as many nodes as the document tree. Figure 3.2 shows the shadow structure corresponding to the example in Figure 3.1.

This framework is intended to serve for many different line-based presentations. The basic Ensemble presentation functions are provided by a parent class `LinePres`. Each specific presentation is implemented as a subclass: the most important virtual method is `gen`, which scans the text of the input document, given a starting point, and returns a new line, complete with size information. Each specific presentation also provides its own subclass of `LPLine`, with a virtual `draw` method for the use of the rendition. Thus, each presentation determines how the document is segmented into lines, and how those lines are displayed.

The `LPLine::draw` method draws the whole line in the correct position on a window, plus any cursors or selections. When the user makes a change in a line, no matter how small, the whole line is redrawn. In principle, common changes could be drawn more efficiently: for example, a character insertion could be handled by moving the block of characters after it to the right, and drawing just that character in the gap. But we sacrifice efficiency in redraw time to save space. To compute such precise graphical change information would require the presentation to compare the line's graphical state before and after the change. And lines are supposed to have very little stored state; the line-based framework assumes that the detailed representation of a line's appearance is transient. One of the questions to be addressed in our conclusion is whether this trade of time for space is justified.

3.2 Individual Line-based Presentations

So far, two presentations have been implemented in the line-based framework. Both are in current use in Ensemble.

3.2.1 "Attributed" Line-based Presentation and ALPS

The "attributed" line-based presentation system provides a basic display of the text of the program, with lineation and whitespace taken literally from the document. In addition, it gives the designer some control over the appearance of the characters, using a simple inherited attribute system, and allows the display of error messages and some graphical tokens alongside the text. We begin by presenting the ALPS style language, and then discuss its implementation.

Language

The principal feature of a program in the ALPS language is a set of conditional rules, setting formatting attributes for some regions of the program. For example,

```
IDENT: bold = true;
```

means that all nodes of type IDENT will be displayed in bold type. If the type were a nonterminal symbol, the attribute value would be inherited by any descendants of the matching nodes, unless they override it in turn.

Name	Type	Default
FONT	String	`"courier"`
SIZE	Integer	16
BOLD	Boolean	**false**
ITALIC	Boolean	**false**
SPACE_ABOVE	Integer	0
SPACE_BELOW	Integer	0
TAB_WIDTH	Integer	8
FG	Color	black
BG	Color	white
SEL_BG	Color	turquoise
EMPH_SP	Boolean	**false**
ELIDE	Boolean	**false**

Figure 3.3: Complete set of attributes that can be set for a node in ALPS.

The rule consists of two parts, a condition and an action. The condition contains either a node type or an attribute condition, or possibly both. Thus `IDENT(ERROR)` matches all IDENT nodes with the ERROR attribute set; `(ERROR="parse")` matches any node with the given value for the ERROR attribute, etc. If a node matches more than one rule, the actions for all the rules are applied to it in the order found in the text.[1] If the rules conflict, providing multiple values for any attribute, the last value is used.

The action may contain any number of attribute settings. Figure 3.3 shows the complete list of attributes that may be set for a node.

SPACE_ABOVE and SPACE_BELOW specify extra vertical space, in pixels, to be added above or below the current line. TAB_WIDTH gives the width of a tab, in spaces of the current font. (If the font is proportional, the width of the letter 'n' is used instead, since space characters are often quite narrow.) Tab columns are measured from the left margin. SEL_BG is the background color used to indicate selection. EMPH_SP, if set **true**, "emphasizes" all whitespace characters, such as space, tab, and newline, by outlining them with visual elements such as the standard under-bracket ⊔. ELIDE replaces the yield of the subtree to which it applies with an ellipsis character.

There are also three global formatting variables, set with a similar syntax. Outside the context of an individual rule, BG sets the background color, and LEFT_MARGIN and RIGHT_MARGIN set margins. These values apply to the window as a whole.

[1]We will see that these semantics are the opposite of those in the transformation system described in Chapter 4, where only one match can be used.

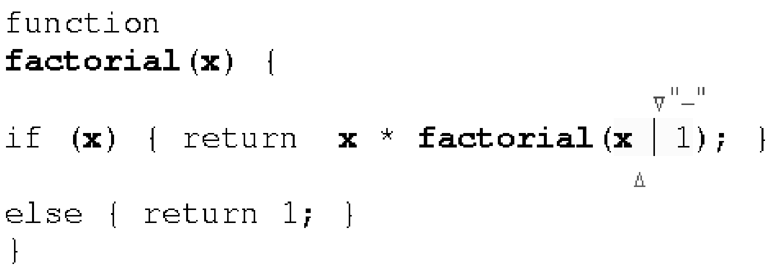


Figure 3.4: A simple program displayed in ALPS. The user has just deleted a mathematical operator and reparsed. The affected area is highlighted; the parser's "jam point" is indicated with an empty message underneath; and the deletion that caused the error is indicated with a message above.

All values may be specified using a general expression syntax, including the standard math operators. These operations also apply to colors. For instance,

```
(ERROR): bg = bg * 0.9 + "red" * 0.1;
```

resets the background color, mixing in 10% more red; if there are nested errors, indicated by repeated settings of the ERROR attribute on the input tree, they will be indicated by successively darker shades of red. The expression syntax also provides a `case` expression, illustrated in Figure 3.5.

The action may also specify "messages". These are special text insertions above or below a given character position in the current token. Figure 3.4, for instance, shows two messages and a background change associated with a single parse error in a program example. The syntax is `message_below(`*string*`, ...)`, with additional attributes to specify the text properties of the message and its offset in the current token. For example, the ALPS code that specifies the appearance of Figure 3.4 is given in Figure 3.5.

In addition to messages, there are several other actions that modify the given text: bracketing a range of text, or inserting a bitmap glyph or a literal character string at a given offset.

Implementation

Given an input tree, an ALPS schema defines presentation attribute values for every node. The set of values for a node is represented by a structure called `ALPattr`, containing a slot for each attribute. (ALPS has a typed object system used to represent all such values.)

```
default: size = 20;

message: fg = "red", font = "helvetica";

ultra_root: font = "courier";

IDENT: bold = true;

COMMENT: font = "times", fg = "brown";

(ERROR_JAMPOINT): message_below("", fg = "purple",
                                offset = val ERROR_JAMPOINT);

(ERROR_TXTDEL, !ERROR_SEVERITY): fg = "red", emph_sp = true,
                                 message_above(val ERROR_TXTDEL);

(ERROR_ISOLATION):
  bg = case (nesting_depth("ERROR_ISOLATION")) {
         1: "#ffffda";   // Yellow
         2: "#ffdaff";   // Magenta
         3: "#daffff";   // Cyan
         default: "LightGrey";
       };
```

Figure 3.5: ALPS specification used in presenting Figure 3.4. `ultra_root` is the symbol at the root of the parse tree.

The rules of a schema directly specify attribute *changes* associated with node types. For instance,

```
IDENT: bold = true;
```

says that whatever attributes have already been set for a node, if it's an IDENT, the value for the attribute `bold` is changed to **`true`**. Such attribute changes are represented by a structure called `ALPchange`, which also has one slot for each attribute. The difference is that the slots of an `ALPchange` may be empty, indicating no change in the value for the corresponding attribute. So the rule above is represented by an association of the token `IDENT` with an `ALPchange` in which the `bold` slot points to **`true`** and all other slots are empty.

In addition to attributes, a rule may specify messages, as illustrated in Figure 3.5. Also, it may insert extra text, a bitmap image, or highlighted bracket symbols, into the stream of characters.

Together, these rules define a set of attribute values and associated actions for each node in the program. For a given node, the set is defined procedurally, as follows: Start at the root node, with the default attribute values. Now, run through the list of rules in order: for each rule that applies to the current node, apply the corresponding `ALPchange` to the current attribute values. Repeat this for every node on the path from the root to the given node.

These values are not stored permanently. Rather, when it formats a line, the ALPS tool constructs a stack of `ALPattr`s, one for each node on the path from the root to the current terminal. This stack is initialized to reflect the path to the starting terminal of the line. The attribute values at the top of the stack correspond to the lowest node in the current path, and apply to all the characters in that node's yield. To move to the next node, the `ALPS` engine walks through the tree, taking the shortest path. When it moves up in the tree, it pops an entry off the stack; when it moves down, it pushes a new entry onto the stack, computing the new values as a difference from the parent. In all, it computes attributes exactly once for every node that is either in the line or an ancestor of a node in the line.

As it performs this "walk" over the terminal nodes in the line, it writes the formatted results into a structure called `ALPLineInfo`. This consists mostly of a list of "chunks", each one containing some displayable information and a set of attributes. The possible chunks are:

Text A piece of the original program text, usually a terminal token. This is by far the most common chunk type. The text is not copied, but indirectly referenced from the input tree.

Tab A tab from the program text. This requires special handling to determine the exact space value.

Bitmap An inserted bitmap, from the `bitmap` action. Used to mark a text position.

Bracket An open- or close-bracket character, generated by the `box` action.

InsText Inserted text, from the `text` action.

When the engine encounters message statements in the course of the walk, it adds them to a separate list in the `ALPLineInfo`, where they are marked with the appropriate attributes, and the current character offset within the line. Each chunk in the list contains enough information that its graphical size can be computed; so once the `ALPLineInfo` is completed (when the "walk" has reached the end of the line), the position of every element, and the final size of the whole line, can be computed.

This formatting method must be called once when each line is first formatted, so the `LPLineTree` can compute its sizes and positions correctly. It must also be called every time the line is to be drawn. The resulting `ALPLineInfo` is not immediately discarded, but cached, in case the same line is drawn again—and as we have seen, a line is drawn twice on an ordinary character insertion.

3.2.2 Everest

The other presentation system built within the line-based framework is Everest, by Marat Boshernitsan. Its language and implementation are based on ALPS, but it provides more complex layout within each line, using ideas from the pretty-printing literature [9, 78]. In addition to controlling such character attributes as font and color, it also uses the structure to select the points where lines begin and end, and to influence line-breaking decisions within the large "line".

Figure 3.6 shows part of a test program written in the SIMPLE language, displayed using ALPS. Figure 3.7 shows the same program displayed using Everest and the style file

```
// A relatively obfuscated program in SIMPLE language

var a,b,c; function foo(a, b){
// code comment
if (a<b)// statement comment
return(bar(a,c)+foo(b, c));else return
foo(a-1,b+1);a=b+c;// a comment
c=a+b;// another comment
if(a)return foo(a,bar(a+foo(b,b),c));if
(b){b=b+a;return a;}}
```

Figure 3.6: A test program in the SIMPLE language, displayed with ALPS.

`simple.ever`. Note that the lines are broken and indented much more readably. Everest is discussed fully in the technical report on the line-based tool [60].

3.3 Performance Evaluation and Conclusions

In the introduction, we proposed four measures of efficiency for presentation tools: refresh time, startup time, size, and change time. To evaluate the performance of the line-based presentation system, we collected a group of 13 Java [7] programs from the World-Wide Web (they were all indexed by Gamelan [34]). We displayed them with ALPS, using a simple schema file, and instrumented the system to record the four measures. We discuss them here out of order, to clarify the presentation.

In Figure 3.8 we show the size of each presentation as a function of the number of nodes in the program parse tree, the measure proposed in the introduction. This plot is roughly linear, about 10 bytes per node, though it seems to fall off a little with large documents, and there are some outliers. Figure 3.8 presents the same size measurements as a function of the number of lines in the text. This plot is more nearly linear, about 250 bytes per line. Evidently the memory consumption of the lines in the `LPLineTree`, which depend only on the text, dominates that of the nonterminals, which depend on the syntax.

With the line-based presentation tool, startup time is linear in program size, just like size: the whole program must be scanned before anything can be displayed. (In Section 6.2.4, we describe a strategy of partial formatting that speeds up start time for the

```
// A relatively obfuscated program in SIMPLE
 language

var a, b, c;

function foo(a, b)
{
    // code comment
    if (a < b)                          // statement comment
        return (bar(a, c)
          + foo(b, c));
    else
        return foo(a - 1,
          b + 1);
    a = b + c;                          // a comment
    c = a + b;                          // another comment
    if (a)
        return foo(a,
          bar(a + foo(b, b),
          c));
    if (b)
    {
        b = b + a;
        return a;
    }
}
```

Figure 3.7: The same program displayed with Everest, a tool developed within the "line-based" framework by Marat Boshernitsan.

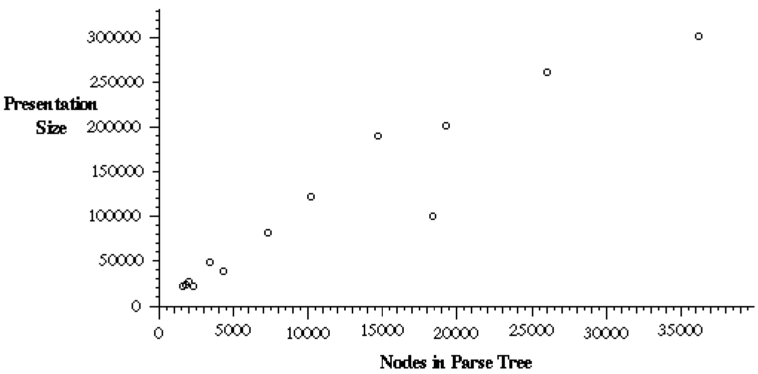


Figure 3.8: Size (in bytes) of ALPS presentations of a set of Java programs, presented as a function of the number of nodes in the parse tree.

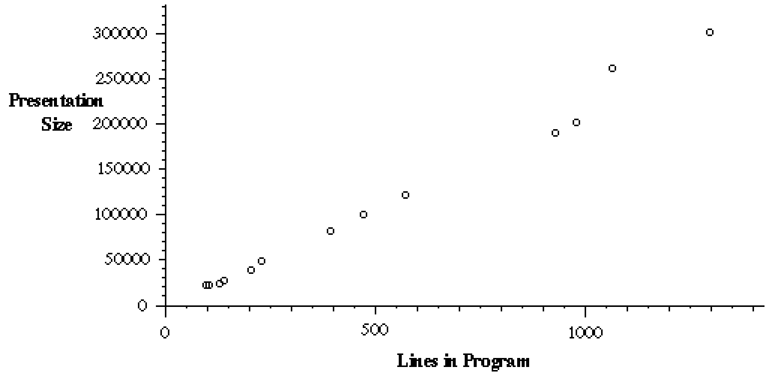


Figure 3.9: Size of the same presentations as in Figure 3.8, per line in the program text.

transformational tool. A similar strategy could be applied to this tool as well.)

Change time and refresh time are both asymptotically logarithmic for the line-based tool. The tree structure of the `LPLineTree` class, closely tied to the tree structure of the program, enables the affected lines to be found quickly, and changes to propagate cleanly up the tree. However, actual measurements are difficult to interpret; to see this, take the example of refresh time.

We measured refresh time with each of our test programs, using the same window size in all cases.[2] In each test run, we displayed window-fulls from the whole range of the program's length, and then averaged the time. Figure 3.10 shows the results expressed as a function of the number of lines in the program. While they fall within a fairly narrow range (from about 4.5 to 8.5 million cycles, or about 90 to 170 milliseconds on the test machine), the correlation to size is not clear. Since each window-full contains about the same number of lines, it occurred to us that refresh time might depend on the complexity of each line. So we replotted the same measurements, taking refresh time as a function of the number of nodes per line. This plot, given in Figure 3.11, is suggestive—the same points are at the extremes in both x and y dimensions—but still a long way from displaying any particular asymptotic behavior. We think the syntactic structure of programs simply varies so widely that it swamps any difference due to their size.

ALPS has become the standard presentation for programs in Ensemble. There are several reasons for this, mostly matters of ease of use. Users can bring up programs in ALPS without any schema at all. The minimal ALPS schema is very short, and rules

[2] Data for the largest program is missing, because the measurement tool ran out of memory.

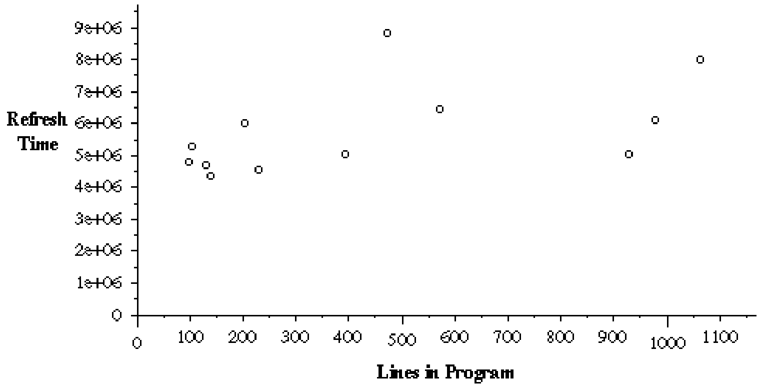


Figure 3.10: Refresh time, the time taken to redraw a window of a standard size, measured in machine cycles, as a function of the number of lines in the program.

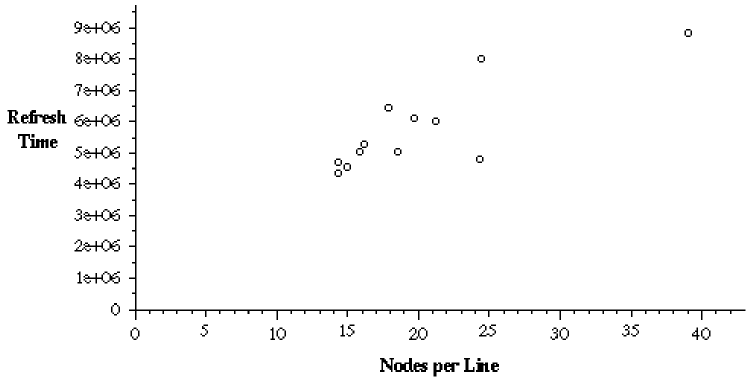


Figure 3.11: Refresh time as a function of the number of nodes per line.

can be added incrementally, with a simple syntax. Finally, if the user writes an ill-formed schema, the error is handled gracefully: the line number is indicated, and the presentation comes up, almost always in a readable state. Several non-specialists have written ALPS schemas: by contrast, other Ensemble presentation systems, including both Proteus and the tree-transformation tool described later in this dissertation, have been so difficult that only their authors and other presentation specialists have been able to program them.

Also, ALPS is efficient in practice. The asymptotic size performance of the line-based tool is linear, but the constant is quite small: the presentation takes about 10 bytes per input node, while the input tree itself is more than 100 bytes per node. (Proteus presentations are generally about as big as the input tree.) Finally, the special-purpose features of ALPS designed for error reporting (e.g. messages with offset) have been useful for demonstrating and debugging recent developments in incremental parsing and error handling.

It is also worth considering that the line-based presentation system is successful precisely because it is not designed as a general presentation tool: its controls are simple, extended only at a few points to handle specific needs of the users. We will return to this idea in the conclusion of the dissertation.

Chapter 4

Using Transformations to Define Appearance

In the introduction, we argued that presentation could be modeled as a transformation from document trees to layout trees. In this chapter, we present a language for expressing such transformations, with sufficient generality to handle all Ensemble document classes. The "language" of the output tree—its terminals and nonterminals and their layout semantics—is called a *medium*. For the rest of this chapter, we will work with the simple medium used in the examples of Chapter 1, consisting of text terminals and H and V (horizontal and vertical) nonterminals. The full-scale transformational formatting media of Ensemble are described in Chapter 5.

4.1 Writing Transformation Rules

There have been many languages designed to express tree transformations, especially for code generation in compilers. A review up through 1988 is available in the dissertation of Pelegrí-Llopart [68]. For this work, we chose to follow the model of Dora [17, 35], a transformation language developed by Charles Farnum at Berkeley for code generation research. The pretty-printer Ppml [45, 64] is based on a transformation language with a rather different syntax, but similar semantics. (We discuss Ppml in more detail in Section 7.7.)

A tree-transformation presentation style is expressed by a program called a *schema*. The schema falls into three parts: a *prologue*, a *body*, and an optional *epilogue*, separated by `%%` dividers, as in `lex` [55] and `yacc` [46]. The prologue begins with a line specifying the medium into which the document is being transformed.

The body consists of a list of transformation *rules*. Each rule specifies the transformation of an input tree fragment to an output tree fragment. It contains a *pattern* matching the input fragment, and an *action* constructing the output fragment. The complete transformation is generated by applying the rules to the whole input tree: that is, *tiling* it with matches of the patterns, and then composing the corresponding action fragments into a whole output tree.

The simplest rule converts one input node to one output node. For example,

```
FUNCTION -> "func";
```

converts any FUNCTION node in the input to an output node with the text "func". The pattern and action are separated by the token `->`, pronounced "yields" or "goes to"; the rule is terminated by a semicolon.

4.1.1 The Pattern

Tree structure in the pattern is indicated with a Lisp-like syntax, after Dora: inside nested brackets (`[]`), the first symbol is the parent of the rest. So the pattern

```
[MEMO TO FROM SUBJECT BODY]
```

matches the top level of the memo tree in Figure 1.1.

So far, our patterns have been made up of simple identifiers specifying node types. To make use of the content of the input tree, the input syntax provides *pattern variables*, elements that match input nodes and bind them to symbols. They are marked by an initial question mark (`?`). In their simplest form, variables match any type. For instance,

```
[MEMO ?a ?b ?c ?d]
```

matches any tree fragment with a MEMO node and four children, and binds the children to the symbols `a`, `b`, `c`, `d`. To restrict the pattern, type information may be added to the variables after a period, for example:

```
[MEMO ?to.TO ?from.FROM ?sub.SUBJECT ?body.BODY]
```

A question mark alone (`?`) matches a node of any type, but does not bind it to a variable. The root node of the input subtree is always bound to a variable `self`.

The pattern language also supports the Kleene star and plus (* and +) operators, familiar from regular expressions. Appended to a node or a subtree in the pattern, * matches zero or more instances of the subtree, and + matches one or more instances. In the terminology of Dora, these are "horizontal iterators." For example,

```
[MEMO TO FROM SUBJECT [BODY ?par.PARAGRAPH*]]
```

matches a memo with any number of paragraphs. The variable `par` is bound to a *list* of all the matching paragraphs, in order.

4.1.2 The Action

Actions are written with a similar syntax. For example, the notation

```
[V "Hello" "world"]
```

specifies an output tree fragment with a V node as its root and two children, both literal strings.

An action may use the variables bound by a pattern in two different ways. First, consider the case when the variable is bound to a single node. *Conversion* means that the node is translated into a corresponding output node, by a medium-specific mechanism. (This medium converts textual input nodes to textual output.) *Transformation* means that the subtree rooted at that node is transformed according to the rules in the schema, resulting in an output subtree. Either way, the variable use returns one output node or subtree. This result is substituted for the variable use in the output pattern. If the variable is bound to multiple nodes, because it was matched within a horizontal iterator, then the conversion or transformation will be performed once for each node, and the results will be inserted into the output in order.

A conversion use is indicated by a star (*) prefixed to a variable name. Consider this rule for a paragraph in a memo:

```
PARAGRAPH -> [H "-" *self];
```

This means that, for every matching paragraph, an H nonterminal will be inserted in the output tree, with two text children: one with a dash, and one with the text of the paragraph.

A transformation use is indicated by an exclamation point (!) prefixed to the variable. This rule, for example:

```
[MEMO ?to.TO ?from.FROM ?sub.SUBJECT ?body.BODY]
 -> [V *to *from *sub !body];
```

transforms the top level of a memo to a vertical arrangement of the text of the fixed nodes followed by the result of transforming the body. As with conversion, a transformation use yields one output node or subtree for each input node bound to the variable.

We have considered extending the language to allow actions to yield zero subtrees, or more than one. However, the restriction appears to have no worse consequence than that schemas sometimes require redundant interpolated nodes. (See Section 4.4 below.) The extension does not appear to be worth the extra difficulty in implementation.

4.1.3 Rule Evaluation Order

An input tree is transformed according to the first rule whose pattern matches, in the textual order of the schema.[1] So the order of rules can matter considerably; in general, programmers should put the most specific rule first, to catch special cases before common cases. For example, to show comments in red when they have the `FUNCTION_HEAD` attribute, and in blue otherwise, the user should write

```
COMMENT(FUNCTION_HEAD) -> *self(fg="red");

COMMENT -> *self(fg="blue");
```

There are two exceptions to the rule that the first match is used. The first is explicit: the language provides named evaluation contexts, like the contexts in `lex`. The context is indicated by a symbol and a colon (`:`). At a variable transformation site, the context is inserted before the `!`, to indicate that the subtree will be transformed in that context. A context is prefixed to a rule declaration, to indicate that the rule applies in the context. Thus `error:!ident` in an action means that the subtree bound to `ident` is to be transformed in the context called `error`. Similarly, the rule

```
error:IDENT -> [H "Error -- " *IDENT];
```

applies only in the `error` context. In evaluating `error:!ident`, Ensemble searches first for a matching rule in the `error` context, and only then, if that search fails, for a matching rule with no declared context.

[1] By contrast, in the ALPS schema language described in Section 3.2.1, every rule matching a node is used.

The second exception to the first-match rule is implicit. This case arises when a rule specifies a transformation of `self`, the root of the input subtree. For example, at the beginning of a schema, the user can write

```
?(ERROR) -> [V "Error" !self];
```

meaning that any node with an ERROR attribute will be prefixed with the string "Error", but otherwise presented as usual. If this transformation were carried out in the ordinary way, the same rule would apply again. To avoid this circularity, the second transformation is chosen from later rules in the schema.

4.2 Output Attributes

The output tree resulting from transformation is marked with two sets of attributes. *User-defined* attributes are set directly by the user or designer through the schema. *Derived* attributes are computed by low-level layout code, on the basis of user-defined attributes as well as the structure and content of the output tree. For example, the user may exercise control over layout with a user-defined justification attribute, specifying left, center, or right; the resulting positions of the nodes in the layout are derived.

Each medium specifies a set of user-defined attributes. Their values are set by assignment statements in the action: for example,

```
PARAGRAPH -> *self(FONT="times",size=16);
```

sets 16-point Times for paragraphs.

User-defined attributes are inherited: an attribute assignment applies to an output node and to all its descendants, unless they override the value. (A corollary is that any attribute may be set at any node, regardless of whether it is relevant to that node's formatting.) We adopted this policy because it worked well in ALPS; users seemed to be able to achieve what they wanted without excessive setting and unsetting of attributes.

Derived attributes, on the other hand, are synthesized, that is, computed for each node on the basis of its children. Originally we permitted complete freedom in the evaluation of derived attributes, allowing nodes to base them on attributes of ancestors and other nodes; but for reasons of performance (discussed further in Chapter 6), we adopted this restriction. In practice, we found only one case where this strict two-pass model was inadequate for representing a desired layout computation.

In addition to attribute assignments as above, we provide two variations, for notational convenience. Output nodes may take positional arguments; for instance, the text medium includes an "indent" node I, which takes its width as an argument, (`I(8)`), instead of an attribute (`I(indent=8)`). And they may take named arguments, which have the same syntax as user-defined attributes but are not inherited. The motivation for named arguments is that some nodes require detailed per-node controls that do not apply to other nodes, and for which inheritance is not useful; we will see examples in Section 5.3, on the graphing tool.

The prologue of the schema is used mainly to set default attribute values, those "inherited" by the root node of the output tree. Here, each attribute assignment is terminated with a semicolon (`;`), statement-fashion. Using the same syntax, the user also sets *global variables*, which are like user-defined attributes except that they cannot be reset within the tree; in the pretty-printing medium, for example, the right margin has just one value over the entire presentation, and it is set here.

The values assigned to attributes may be simple literal strings, numbers, or Boolean values. A special syntax is also provided for arithmetical expressions and variable references. Between the delimiters `$(` and `)`, identifiers can be used to reference values from three name-spaces: the variables bound by the pattern (returning the content of the input nodes, considered as a value), the inherited user-defined attributes, and any attributes attached to the input document. (No mechanism is provided for resolving name conflicts.) Within this context, the usual infix operators are provided for arithmetic.

4.3 Input Attributes

In Ensemble, document trees are marked with attributes for several purposes. For example, a syntax error is indicated by attributes on several nodes, indicating where the parser first encountered the error, which parts of the tree are inconsistent, etc. To reflect syntax errors, presentation schemas must be able to take account of input attributes. The transformation schema language provides two mechanisms for doing this.

First, nodes in patterns may be marked with input attributes, in parentheses at the end of the node syntax. `FUNCTION(ERROR)` matches all FUNCTION nodes with the ERROR attribute, and `FUNCTION(ERROR="type")` matches those with the error value `type`. This enables users to write rules that catch special attribute cases; because of the matching-order

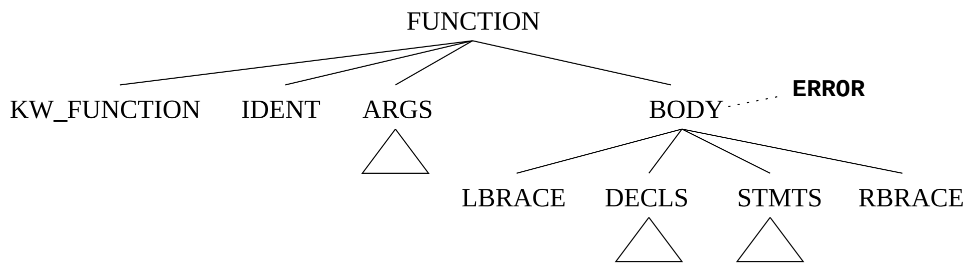


Figure 4.1: Part of the syntax tree of a function, with an error signaled by an attribute on the BODY node.

rule, such rules should precede un-attributed rules in the schema.

This turns out to be inadequate. Consider the rule

```
?(ERROR) -> [H "Error" !self];
```

intended to flag every node that has the ERROR attribute. It will not catch all such nodes: in particular, it will miss nodes that are covered by a pattern rooted at a higher node. Consider the syntax tree for a function, as illustrated in Figure 4.1. The BODY node has an ERROR attribute. If the pattern matching the function declaration is

```
[FUNCTION KW_FUNCTION ?name.IDENT ?args.ARGS
          [BODY LBRACE ?decls.DECLS ?stmts.STMTS RBRACE]]
```

then the BODY node will not be separately matched; the pattern-matcher will be used again on the declarations and statements below it, but the ERROR attribute on the body itself will go uncaught.

Accordingly, we added another mechanism for the same purpose, a "post-pass" to check for input attributes after each subtree has been matched. In the epilogue of the schema, the user can write rules to revise a tile, based on the input attributes of the nodes it covers. In these rules, the pattern consists only of an attribute qualifier, e.g. `(ERROR)`; the only variable available in the action is `self`, and it must be transformed. If such rules are present, every tile is checked to see if any of the nodes it covers matches any of these epilogue rules. If the pattern matches, the rule creates a tile that is inserted *above* the original in the output tree. Thus, its action may be used to alter the presentation attributes of the original tile:

```
(ERROR): !self(color = "red");
```

```
// Complete transformation schema for a memo

medium text;

font = "courier";
size = 16;

%%

[MEMO ?to.TO ?from.FROM ?sub.SUBJECT ?body.BODY]
-> [V(just=center, size=18)
     [H [V(just=right) "To:" "From:" "Subject:"]
        [V(just=left, italic=true)
          *to *from *sub]]
     !body];

[BODY ?pars.PARAGRAPH*]
-> [V(font="times", bold=true) *pars];
```

Figure 4.2: Full schema to transform a memo as in Figure 1.4.

or to insert new material:

```
(ERROR): [H "Error:" !self];
```

Either of these rules would catch the ERROR attribute in Figure 4.1. However, they would make no visual distinction between an ERROR attribute on the BODY, as in our example, and on the FUNCTION. It is not obvious how to achieve such sensitivity; in any case, this second mechanism has proven adequate for our uses. And while the post-pass may seem inefficient, we show in Section 6.1.2 that the time it actually consumes is small.

4.4 Summary

Now we can put all the pieces together into a complete transformation schema. Figure 4.2 shows a schema that transforms our sample memo into the layout shown in Figure 1.4. Figure 4.3 shows how it works, mapping input to output structure. The only difference from the original structure proposed in Figure 1.4 is an extra V node, interpolated at the body. This is a consequence of the restriction, noted above, that a rule must generate a single subtree; the extra node makes no difference to the visible layout.

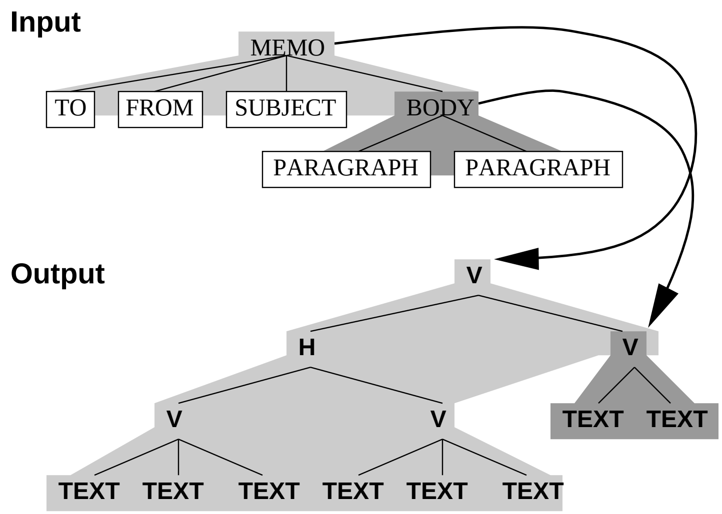


Figure 4.3: Result of applying the transformation specified in Figure 4.2 to the memo in Figure 1.1. Converted nodes are shown in boxes.

Chapter 5

Three Output Media

In the previous chapter, we illustrated the idea of presentation by tree transformation with examples drawn from a simple text medium. Here we present the full complexity of the actual media developed for Ensemble. These three media all share the same transformational mechanism; they differ widely, though, in they way they use the attribute mechanism to carry out formatting computations.

5.1 Pretty-Printing

The pretty-printing medium, primarily developed for the presentation of programs, is a superset of the text medium of Chapter 4. It is derived from Ppml [64], the pretty-printing system of Centaur [45]. (We discuss the differences of our work from Ppml in Chapter 7.)

5.1.1 Horizontal and Vertical Modes

As in the demonstration medium used in Chapter 4, the pretty-printer works by hierarchical nested layout. At each level, a parent groups its children, either in *horizontal mode*, running left to right, or in *vertical mode*, running top to bottom. There are two important controls over these modes: *spacing* and *justification*.

Spacing is illustrated in Figure 5.1. The user sets the space between nodes, which is uniform over the whole set of children; and further, a spacing *style*, which may have one of four values. Here, we lay out a set of nodes in vertical mode, once for each of the spacing styles, with the same space value. If the style is `gap`, then the interval is the gap between

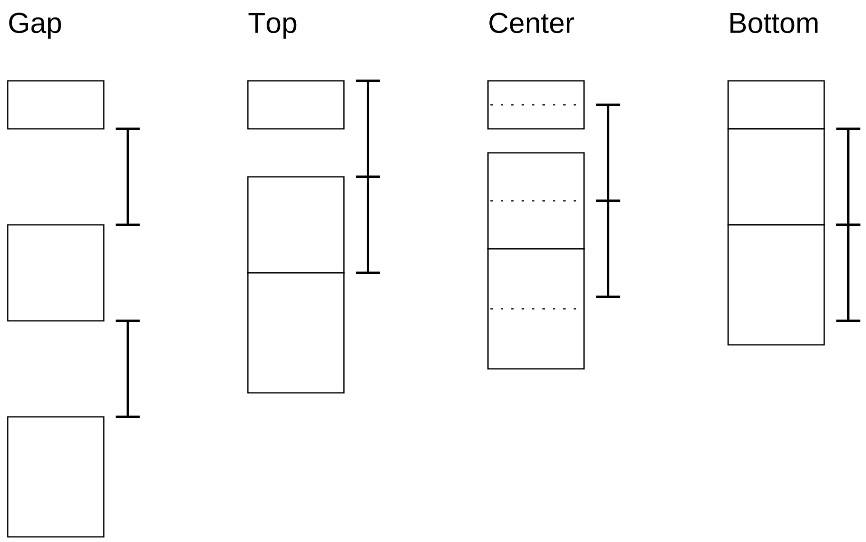


Figure 5.1: The four spacing styles of the pretty-printer's vertical mode. The space between nodes (indicated by the I-beams) is the same in each, but the spacing style controls how it is used to place the nodes.

the bottom of one child and the top of the next. If it is `top`, then it is the space from the top of one child to the top of the next. Similarly `center` spaces items by their centers, and `bottom` by their bottoms. Obviously, except with `gap` spacing, it is possible for objects to be too tall to fit within the given interval; in this case they abut, rather than overlapping. Horizontal spacing works similarly, with spacing styles `gap`, `left`, `right`, `center`.

In Figure 5.2, we illustrate justification, which controls the placement of children in the direction perpendicular to the main mode. In horizontal mode, vertical justification may be `top`, `center`, or `bottom`, with the usual semantics. Similarly, in vertical mode,

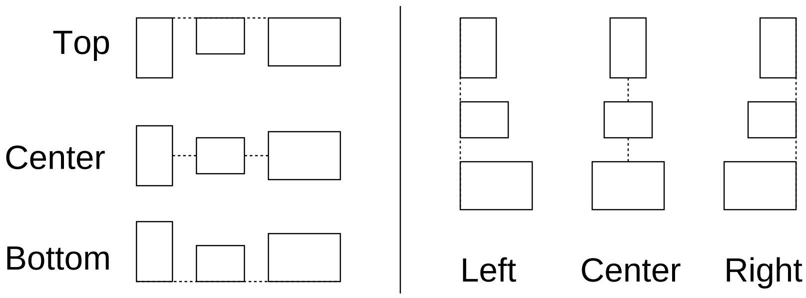


Figure 5.2: When objects are arranged in horizontal mode, their vertical position is controlled by justification: top, center, or bottom. Similarly, in vertical mode, horizontal justification may be either left, center, or right.

```
class LockedInt extends Object {
   private static int val_;
   LockedInt(int val) {
      val_ = val;
   }
   synchronized int get() {
      int val = val_;
      return val;
   }
   synchronized void set(int val) {
      val_ = val;
   }
   synchronized void inc() {
      val_++;
   }
}
```

Figure 5.3: Small excerpt from a Java program

horizontal justification may be `left`, `center`, or `right`.

The node types and user-specified attributes of the pretty-printing medium are summarized in Section A.1. The derived attributes are the graphical size of each node, and the relative positions of its children. Size is specified as height and width: layout is completely rectilinear. Positions are x and y values, relative to the upper-left-hand corner of the parent's bounding box. It might seem that these positions should be attributes of the children. However, computing them requires knowledge of the sizes of all the children and the formatting parameters of the parent. Thus, considered as attributes of the children, they are not synthesized.

5.1.2 Example of Pretty-printing

Figure 5.3 shows a simple fragment of a Java program, presented with the pretty-printing tool. The full presentation schema for the Java language has 195 rules, totaling more than 800 lines; in Figure 5.4 we show some of the rules that were applied in creating the figure.

To ease the creation of large schemas, we developed a simple automatic schema generator.[1] It scans the grammar of a programming language, and writes one rule in the presentation language for each production. The automatically generated rules use the vari-

[1] Much better heuristics are known: see for example van den Brand and Visser [90].

```
[classDeclaration ?mod.classModifierList
                  ?cl.class
                  ?name.simpleSymbol
                  ?x.extends
                  ?interf.interfaces
                  [classBlock ?lbrace."{"
                              ?comm1.COMMENT*
                              ?list.fieldList
                              ?rbrace."}"
                              ?comm2.COMMENT*]]
 -> [V [H !mod *cl I(8) !name !x !interf I(8) *lbrace]
       [H I(24) [V !comm1 !list]]
       *rbrace
       !comm2];

[fieldList ?a.field]
 -> !a;

[fieldList ?a.fieldList ?b.field]
 -> [V !a !b];

[field ?a.modifierList ?b.variableDeclaration]
 -> [H !a !b];

[field ?a.methodDeclaration]
 -> !a;
```

Figure 5.4: Some of the formatting rules used in generating Figure 5.3.

able names `a`, `b`, `c` .... Terminal nodes are converted, and nonterminals are transformed; the results are combined, in the same order, under a V node. While this does not immediately produce a usable schema, it gives the designer a good start: the whole program is visible (as a vertical column of tokens), and there is a syntactically correct rule for each structural feature of the language. To complete the schema, the designer gradually revises the rules to format these structures as desired. In Figure 5.4, several rules are still in their original form.

Note also how the schema recognizes recursive syntax for lists (`fieldList` in this case). Together, these rules translate a chained list of fields into a chained list of V nodes. This is somewhat inefficient, since the same appearance could be achieved by a single V node with all the fields as children. Dora [17, 35], the language on which our transformation syntax is modeled, provides *vertical iterators* (as opposed to such horizontal iterators as Kleene `+` and `*`) to handle this problem. Using vertical iterators, a single pattern could be written to match a chained list of any length, binding all the `field`s in the list to a single pattern variable. When this variable was used, the list would be expanded as a flat n-ary branch in the output.

However, this feature is not needed in Ensemble, because the language kernel provides a sequence notation for programming-language grammars. It would be easy to modify the Java grammar we use so that all the `field`s in a list were children of a single `fieldList` node. Then the limited horizontal iterators of the pattern language would be quite adequate.

We will return to this issue in Section 6.2, when we discuss efficiency.

5.1.3 The "Curly-Bracket Problem"

These Java examples also demonstrate what we have dubbed the "curly-bracket problem." This is a superficially minor matter that leads to interesting suggestions for how text presentation by tree transformation might be extended.

There are two standard conventions for formatting curly brackets (`{}`, enclosing blocks) in C and derived languages. Figure 5.3 observes one convention, putting the open-bracket at the end of the line preceding the block. As we see in Figure 5.4, this is achieved by writing rules to recognize the block structure together with its enclosing control structure, in order to rearrange the parts into a different hierarchy.

```
class LockedInt extends Object
{
   private static int val_;
   LockedInt(int val)
   {
      val_ = val;
   }
   synchronized int get()
   {
      int val = val_;
      return val;
   }
   synchronized void set(int val)
   {
      val_ = val;
   }
   synchronized void inc()
   {
      val_++;
   }
}
```

Figure 5.5: Alternate presentation of the Java fragment in Figure 5.3, following the other major curly-bracket convention.

```
[classDeclaration ?mod.classModifierList
                  ?cl.class
                  ?name.simpleSymbol
                  ?x.extends
                  ?interf.interfaces
                  [classBlock ?lbrace."{"
                              ?comm1.COMMENT*
                              ?list.fieldList
                              ?rbrace."}"
                              ?comm2.COMMENT*]]
 -> [V [H !mod *cl I(8) !name !x !interf]
       *lbrace
       [H I(24) [V !comm1 !list]]
       *rbrace
       !comm2];
```

Figure 5.6: Rewritten rule to present Java class declarations as in Figure 5.5.

Figure 5.5 observes the other convention, in which the open-bracket has a line to itself at the beginning of the block. (The original text source, an applet called Typodrome [1], was formatted this way.) To get this effect, the `classDeclaration` rule in the transformation schema is rewritten as in Figure 5.6. The only difference is in where the variable `lbrace`, matching the left curly bracket in the parse tree, is used in the action. This is an easy change; but unfortunately it needs to be repeated for every rule that matches a block. Our Java schema has 22 such rules, of which three are applied in formatting this code fragment—not only the class declaration, but the constructor and the methods. Changing 22 rules at once is not only tedious, but highly error-prone.

How can we modify the pretty-printing tool so that switching between the two styles is easy? The first solution we considered was to extend the transformation model, allowing several transformation passes, as in John Boyland's work on composing tree attributions for code generation [18]. Then each of the block rules in the first pass could create an intermediate block structure, something like this:

```
[H [... before ...]
   [BLOCK LBRACE [... content ...] RBRACE]
   [... after ...]]
```

In the second pass, a small group of rules could transform this into the desired style. For the "previous-line" style, the output would be

```
[H [H [... before ...] LBRACE]
   [... content ...]
   [H RBRACE [... after ...]]]
```

And for the "separate-line" style:

```
[H [... before ...]
   LBRACE
   [... content ...]
   RBRACE
   [... after ...]]
```

The most serious objection to this idea is that it complicates the programming model considerably, both for the user and the implementer. Not only does it add a second stage of transformation; but this second stage is a rewrite, rather than an atomic transformation.

Another, more attractive approach is to keep the present transformation technique and modify the computational strategy of the pretty-printing medium. Currently, we can think of each node as synthesizing a single rectangular layout for itself and all its descendants. Suppose that, in addition to this, each node could synthesize a "prologue" and an "epilogue", both also rectangular, but neither in any fixed relation to the main layout. Then the parent node could place all three of these elements separately. The H node, for example, could place the prologue of a child (if present) at the right-hand end of the previous child. Given these semantics, it would no longer be necessary for each rule matching a control structure with a block to match the block's brackets. Rather, all blocks could be handled with a single rule, creating a prologue and epilogue containing the brackets, and then the enclosing rule's H node would integrate them automatically. To switch between the two layout styles, the designer would only need to rewrite the block rule. To give the style of Figure 5.5, it would put the open bracket within its main body, and provide no prologue.

We will discuss other possible extensions of our text model in Section 8.2.

5.1.4 Comments

These Java examples also reveal a dirty little secret of pretty-printing: the difficulty of formatting comments. In almost all programming languages, comments are defined by a lexical syntax that is unrelated to the structure of the code around them. This practice derives from the batch model of program analysis, in which a lexer breaks the program text into tokens, discarding comments and whitespace, and then a parser discovers the structure of the resulting token stream. In an interactive software development environment (ISDE), though, discarding whitespace and comments is not an option: they must be preserved and displayed to the user.

In Ensemble, comments are stored as "extra" tree structure, effectively extending the grammar of the language. A comment or series of comments is represented as a sequence of tokens, paired with the preceding terminal symbol. As a result, our rules for matching the Java language must be extended to handle comments wherever they may appear. We have not done this consistently, instead adding comments to patterns as necessary for the programs in our test suite.

It would be cleaner to assign each comment to a "structural" node, that is, a node in the abstract syntax, and store the comment as an attribute of that node rather than a terminal in its own right. Then the Java schema could use rules based on input attributes (Section 4.3) to insert them in the output tree in a consistent manner. The criteria for deciding the node to which a comment should be attached, though, are unclear. van den Brand and Visser [90] discuss a solution to this problem, also in the context of a transformational pretty-printer derived from Ppml. However, they make the assumption that the textual order of the program (reading left to right, top to bottom) remains the same in the presentation as it was in the original text. This restriction is unacceptably severe.

Abstractly, the "comment problem" can be seen as the result of a conflict between two document models, competing within each program. Comments and whitespace assume a low-level text-based document model, while every other feature of modern programming languages assumes a hierarchical, "structure-based" model.

In short, handling comments is still an unsolved problem. The documentation comments of Java [7], explicitly associated with the declarations that follow them, are a welcome recognition that it can be solved by improvements in language design. But "free" comments remain, in Java and virtually all other languages.

5.1.5 Line-Breaking

When a horizontal sequence of objects is too wide for the available space, it must be broken into segments which can be stacked vertically. This process, known as *line-breaking*, has a long research history. To clarify the discussion, we divide it into two categories, depending on the source of the objects.

"Stream" line-breaking arises in formatting a text paragraph such as this one. In terms of a structured-document model like that of Ensemble, the paragraph is a homogeneous stream of characters within a single document node. For line-breaking, the stream is analyzed into a sequence of words and spaces, marked with possible break points. Line-breaking then becomes a problem of which of the break points to use. The best-known treatment of this problem in computer science is that of Knuth and Plass [53].

Like most pretty-printers, ours does not perform stream line-breaking. Apart from comments, the tokens with which it deals are small, and contain no whitespace to delimit words within them. Our pretty-printer displays multi-line comments correctly, in several lines within one text box, but their line breaks are inflexibly determined by the newline characters in the text.

"Structural" line-breaking arises when structurally discrete elements are grouped horizontally. Consider formatting expression trees in an ALGOL-like programming language. These are generally displayed in horizontal format, but when they grow too wide, they should be broken into several lines. This line-breaking step must take account of syntax, so the hierarchical structure of the expression remains comprehensible. Much of the pretty-printing literature is concerned with this form of line-breaking; generally, it is handled by unparsing the tree into a text stream marked with syntactic nesting information, and performing a variant of stream line-breaking on the result. While this is an effective solution, it limits the user's freedom to rearrange the output graphically. (We discuss this at greater length in Section 7.3.)

The Ensemble pretty-printing medium provides only limited structural line-breaking. The HV and HOV nonterminals are capable of formatting their children horizontally and/or vertically depending on the available space. However, the "available space" they use for formatting is determined solely by the global margin attributes; it does not take into account other context, such as the current level of indentation, which affects the actual space available. To do so would violate the strict attribution rules we have defined: one inheritance pass for the user-defined attributes, followed by one synthesis pass for the derived attributes (such as position and size).

The purpose of these attribution restrictions, as we will see in Chapter 6, is to achieve good incremental performance. Consider inserting a character into a large expression that was laid out with high-quality structural line-breaking. Change time wouls be proportional to the size of the expression, since a single insertion can cause a "ripple" of reformatting all the way to the end. However, Tim Wagner [92] argues that the size of expressions is bounded in practice. So perhaps, if the attribution restrictions were only violated within expressions, good empirical performance could be retained.

As we will see in Chapter 7, other interactive systems reflect the difficulty of line-breaking as well. Several, including Black's Pan Program Presenter [13] and Soiffer's MathScribe [82], simply omit it. (Soiffer argues a little defensively that line-breaking is less important for interactive graphical tools than for print, since a wide expression can be scrolled left and right in its window.) Structural line-breaking in interactive systems remains an open research question.

```
class LockedInt extends Object {
   private static int val_;
   LockedInt(int val) { [...] }
   synchronized int get() { [...] }
   synchronized void set(int val) { [...] }
   synchronized void inc() { [...] }
}
```

Figure 5.7: Alternate presentation of the Java fragment in Figure 5.3, with method bodies and other blocks elided.

```
[constructorDeclaration ?a.modifierList
                        ?b.simpleSymbol
                        ?c."("
                        ?d.optParameterList
                        ?e.")"
                        ?f.optThrows
                        ?g."{"
                        ?h.optConstructorStatements
                        ?i."}"
                        ?comm.COMMENT*]
 -> [V [H !a !b I(2) *c !d *e !f I(8) *g I(8) ELISION I(8) *i]
       !comm];
```

Figure 5.8: Rule to elide constructor declarations as in Figure 5.7. Note that variable `h`, the content of the constructor, is not used.

5.1.6 Extending the Program Presentation Model

Several extensions have been suggested for the presentation of programs in Ensemble.[2] Here we sketch briefly how they might be implemented within this pretty-printing tool.

Two of the suggestions have to do with *elision*, the controlled omission of parts of a program. Simple elision is easy to do with the pretty-printer. In Figure 5.7, we show the same Java example, presented with a schema in which method and constructor bodies are elided. This simply means that the actions in the rules don't use some of the content bound by the patterns. Figure 5.8 shows the constructor-declaration rule from this schema.

First, it would be nice to diversify the symbols used for an elision; currently it is

[2]Sue Graham, personal communication.

always represented by an ellipsis inside a rectangular box. The simplest way to do this is with an inherited attribute, say `ELISION_SYMBOL`. Any number of preset symbol values could be defined for this attribute; the formatting and display code would simply look them up in a table, choosing the appropriate sizes and bitmaps. For further flexibility, the attribute could also support string values, naming external bitmap files, like the `bitmap` action in the line-based presentation tool. Finally, once this wide range of symbols is available, it is not obvious that `ELISION` is the best name for this terminal. After all, it can be used anywhere, regardless of whether anything is being elided; perhaps the name should be changed to `GLYPH` or `BITMAP`.

Also note that elision can only be specified through the schema, using syntactic criteria. That is, it is possible to elide all method bodies in a Java file. But it is not possible to choose the elided areas interactively. It would be useful and convenient to be able to selectively "un-elide" structure, descending into detail in the manner of an outline editor.

One possible solution to this problem is to tie elision to attributes on the document node. Consider these rules, for example:

```
COMMENT(dont_elide) -> *self;
COMMENT -> ELISION;
```

An ordinary comment, without extra attributes, will miss the first rule and match the second, and thus be elided. But if the user sets the `dont_elide` attribute on the comment, it will match the first rule and be presented in full text.

Currently, there is no user interface to set attributes on nodes. This is a minor matter: a menu or other interface tool could be designed to select attributes for the currently selected node, when some key combination is pressed. More seriously, it is currently impossible to select an elided node. This is because selection requires a mapping from output tree to input tree. This mapping is mediated by variable uses within the rules; and an elision is not specified by a variable use. That is, if we write

```
[IF_STMT ?key.KEY_IF ?left."(" ?cond.EXPR ?right.")" ?stmt.STMT]
 -> ELISION;
```

this elides the if statement, but doesn't connect the elision to any of the variables on the left. The simplest solution is to connect the elision to the root of the match (the variable `self`). It is more useful, though, to permit the user to make the connection explicit, especially when the right side mixes variable uses and elision. In this case, for example, we might like to write

```
[IF_STMT ?key.KEY_IF ?left."(" ?cond.EXPR ?right.")" ?stmt.STMT]
 -> [V [H *key *left ELISION(cond) *right]
       ELISION(stmt)];
```

explicitly placing an elided condition and an elided statement in the output. This would permit the user to select one or the other elided subtree, by clicking on the corresponding elision symbol.

However, this solution creates further problems. It requires extensive duplication of rules, rewriting them to handle at least the cases with `dont_elide` and without, and possibly combinations of such cases when rules contain several elisions. A cleaner solution will involve a deeper change to the transformation model.

Currently each medium provides a method to "convert" an input node to an output node; this is how variable uses marked with `*` are handled. Suppose we added an `elide` method to the medium as well, returning an output node suitable for representing an elision. Suppose also that the transformation engine, when it encountered an input subtree with a certain attribute setting (perhaps with the attribute `elide` set to **true**), simply called this method rather than transforming the subtree. Then elision would be controlled by local attributes set on the document tree; mapping between elision nodes and input nodes would be clear; the pretty-printer, and all the other media, could control the appearance of an elision using any attributes or global variables; and style files could stand as they are. The price of this simplicity is a change affecting all transformation media.

5.2 Tree Display

Displaying the structure of a document as a tree has been a basic debugging and demonstration tool for Ensemble since its inception. Originally, the mechanism used was Realize [67], a general-purpose data-structure display tool. But with the tree-transformation mechanism, it became possible to handle tree display with only a modest amount of special-purpose code.

For Ensemble's purposes, there is no need for great flexibility in tree formatting. The root is always at the top; the vertical distance between nodes is fixed; and there is little variation in the appearance of individual nodes. The most important layout computation is determining the horizontal position of children below their parents. The goal is simple: to make the tree as compact as possible without obscuring the relationships between nodes.

As with the previous layout computation, this is accomplished by attribute synthesis. Each output node carries synthesized attributes representing the horizontal position of its children (relative to itself), and the shape of the subtree below it. The shape is recorded as a *contour*, a polygon outlining the right and left edges of the tree. To compute these attributes, a node first places each of its children so that their contours do not overlap, and then combines their contours, plus the outline of its own contents, into a single synthesized contour. This is essentially the "tidy-tree" algorithm of Moen [63], recast as an explicit attribute computation.

It is possible to write transformational rules in this medium, but there is really no point: the purpose of the medium is to show trees "as they are." The only important structural control is the selection of the *abstraction*, or level of detail of visible tree structure, which has the same values and semantics as in the pretty-printing medium (p. 122).

All the screen shots of tree structure in this dissertation, for example Figure 1.1 in the last chapter or Figure 5.10 below, were drawn using this tree medium.

Extending the Tree Tool

The most obviously useful extension of the tree medium would be to enable "horizontal" tree layout, with the root at the left, all its children in the next column, all its grandchildren in the next, etc. This is perfectly manageable in principle; but a lot of C++ code would have to be duplicated, and rewritten with the coordinates exchanged. The user might control the choice by a single global variable, perhaps `format_direction`.

It has been proposed that Ensemble should support documents that are general DAGs, not pure trees. For example, DAGs arise in GLR parsing [95], by the splitting and merging of multiple parse processes. Ensemble's parser and low-level tree mechanism already support this model.

Most likely, the DAG would appear to clients, such as a tree presentation, in the form of a "primary" tree structure, with "secondary" links across branches in the tree. These secondary links would probably be stored as attributes on the source and target nodes. If so, the tree medium could continue to base its tidy-tree layout on the primary parent-child relationships, and draw lines across the tree to indicate the secondary links.

Secondary links could be handled by extending the set of synthesized attributes attached to each display node, adding a list of unmatched links. If an input node is the source or target of a secondary link, a link record would be created, containing the parent, the child, the relative graphical position of the node, and a bit indicating whether the node was the parent or child. This record would be added to the output node's list, and then percolated up the tree by attribute synthesis. At each stage, its position field would be updated, to give the coordinates relative to the current node. This process would end at the lowest common ancestor (LCA) of the parent and child of the secondary link: here, records from both ends of the link would arrive at the same node, giving the correct relative position. Then the link could be drawn, and both nodes discarded from the list of unmatched links.

5.3 Graphs

The third transformation medium is a display tool for graphing numerical documents. It was mainly designed to demonstrate that the transformation tool can be used for computational strategies completely unlike the pretty-printer and tree media; but it has proven useful. (All the graphs in this dissertation were drawn with it.)

The most important layout operation in an ordinary two-dimensional graph is the coordinate mapping from data values to graphical x and y. If a graph is made up of dots connected by lines, then this mapping applies equally to the dots and the endpoints of the lines. If a graph presentation tool makes the designer specify the mapping redundantly, there is a serious danger that the specifications will be inconsistent. Our strategy is to defer the mapping as long as possible: to build the visual elements of the graph using a mix of data and graphical coordinates, percolate them up the output tree by attribute synthesis, and translate the coordinates of all objects at a single node, high in the tree, where the user can specify the parameters just once. By concentrating the scaling computation in one node, we hope to make schemas less error-prone.

5.3.1 Examples

Figure 5.9 shows a a set of numerical data derived from measurements of presentation size as a function of document size. For each of four documents, it gives a file name, a document size, and minimum, mean, and maximum presentation sizes under various conditions. As shown in Figure 5.10, the structure of this data set is very simple. Figure 5.11

File	Nodes	Min. size	Mean size	Max. size
`small.java`	569	345796	431701	502964
`subgraph.java`	2354	354320	597223	801472
`partial.java`	5931	409880	635467	865588
`graphAlgorithm.java`	28638	578052	1116045	1871476

Figure 5.9: A set of numerical data, presented as text with the pretty-printer medium.

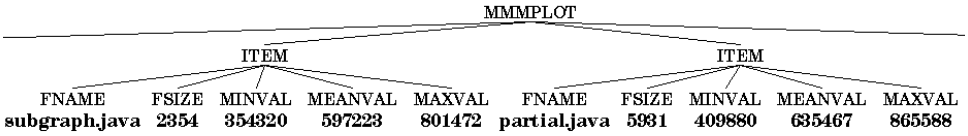


Figure 5.10: Part of the tree structure of the data set.

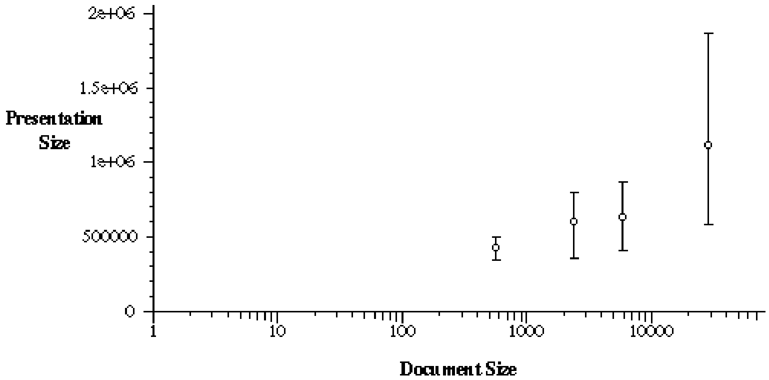


Figure 5.11: The same data, presented as a graph in "error-bar" style, on a logarithmic x projection.

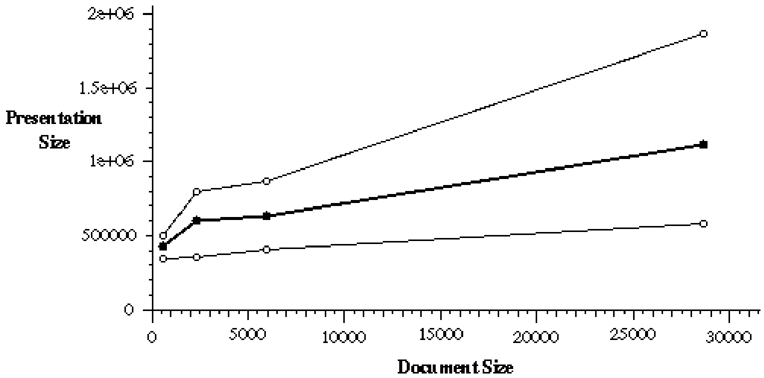


Figure 5.12: Another graph, derived from the same document with a different presentation schema. Note the different grouping of the data into lines.

```
medium graph;
font = "Times";
size = 12;
bold = true;
vert_just = center;
horiz_just = center;

%%

[MMMPLOT ?it.ITEM+]
-> [SEQUENCE
     "Presentation"(graph_x=36,graph_y=84)
     "Size"(graph_x=36,graph_y=100)
     [PLOT(x_axis=bottom, y_axis=left, x_mode=log,
           graph_x=100, graph_y=10, graph_width=400,
           graph_height=200, bold=false)
       !it]
     "Document Size"(graph_x=300, graph_y=248)
   ];

[ITEM ?fname ?fsize ?min ?mean ?max]
 -> [SEQUENCE(data_x=$(fsize))
       LINE(data_y=$(min), graph_x1=-2, graph_x2=3)
       LINE(data_y1=$(min), data_y2=$(max))
       LINE(data_y=$(max), graph_x1=-2, graph_x2=3)
       CIRCLE(data_y=$(mean), radius=2, fill="white")];
%%
```

Figure 5.13: Schema to present data as shown in Figure 5.11.

shows this information plotted in "I-beam" style, with the three values for a given document size connected in a cluster. The x axis is logarithmic. Figure 5.12 shows the same data plotted another way. This time, one compound line connects all the minima, another connects the means, and another the maxima; entries for one document are not visually grouped.

The difference between these two plots is determined by fairly simple graph schemas. (See Section A.3 for a listing of the node names and attributes they use.) Figure 5.13 gives the schema for Figure 5.11. This is fairly straightforward, but note the combination of data and graphical coordinates in each cluster. Figure 5.14 gives the schema for Figure 5.12. Here, the top pattern binds all the data clusters as a list. It transforms the whole list three

```
medium graph;
font = "Times";
size = 12;
bold = true;
vert_just = center;
horiz_just = center;
thickness = 1;

%%

[MMMPLOT ?it.ITEM+]
-> [SEQUENCE
     "Presentation"(graph_x=36,graph_y=84)
     "Size"(graph_x=36,graph_y=100)
     [PLOT(x_axis=bottom, y_axis=left,
           graph_x=100, graph_y=10, graph_width=400,
           graph_height=200, bold=false)
       [LINE(place=below) !min:it]
       [LINE(place=below,thickness=2) !mean:it]
       [LINE(place=below) !max:it]]
     "Document Size"(graph_x=300, graph_y=248)
   ];

// Plot just the min of an item
min:[ITEM ?fname ?fsize ?min ?mean ?max]
 -> [SEQUENCE(data_x=$(fsize), data_y=$(min))
      CIRCLE(radius=2, fill="white")
      REFERENCE];

// Plot just the mean of an item
mean:[ITEM ?fname ?fsize ?min ?mean ?max]
 -> [SEQUENCE(data_x=$(fsize), data_y=$(mean))
      CIRCLE(radius=2, fill="black")
      REFERENCE];

// Plot just the max of an item
max:[ITEM ?fname ?fsize ?min ?mean ?max]
 -> [SEQUENCE(data_x=$(fsize), data_y=$(max))
      CIRCLE(radius=2, fill="white")
      REFERENCE];
```

Figure 5.14: Schema to present data as shown in Figure 5.12.

times, in three different contexts: each makes circles and references for part of the data—the minima, means, and maxima in turn. The references are turned into LINEs set behind the circles.

Chapter 6

Implementation of the Transformation Tool

In this chapter, we show how the transformation tool has been implemented. First we describe the initial implementation, showing how the basic functionality of the system was achieved. Then we consider the four efficiencies described in the introduction, and show how the initial implementation was modified to achieve each one.

6.1 Initial Implementation

We begin with a brief enumeration of the elements of the design, and then address a series of special issues in more detail.

6.1.1 Elements of the Design

The foundation of the implementation is a simple common node class, providing n-ary tree structure and arbitrary attributes. Several different tree structures in the transformation module are derived from it.

A transformation is represented straightforwardly as an ordered list of rules. Each rule consists of two pattern fragments, one for the pattern and one for the action. We describe the pattern-matching mechanism in Section 6.1.2.

When an input tree is successfully matched against a rule, a *tile* is created. A tile consists of three parts. The *cover* is a tree fragment corresponding to the parts of the input tree covered by the pattern; if the pattern has iterators, the cover may have a somewhat different shape. The *template* is generated from the rule's action, based on the variable bindings; when there are iterators, variable uses may need to be expanded to multiple nodes in the template. Finally, the tile includes the actual *output* tree structure generated by the tile. This corresponds to the template in shape, but it is made of nodes from the output medium.

Consider again Figure 4.3, showing how a memo is transformed by the composition of two rules, one for the upper structure and one for the body. Here there are two tiles, one for each rule. The action of the top-level rule contains a "transformation use" of the variable bound to the body of the document; at this point, the two tiles, and their output tree structures, are joined. In general, these links form a tree of tiles, with a close relationship to the input tree. If tile A, at input node a, is an ancestor of tile B, at input node b, then a is an ancestor of b in the input tree.

The construction of the tile tree proceeds recursively, top-down, driven by the transformation uses of pattern variables. As it is constructed, the user-defined attributes from the rules are propagated down the growing output tree. When construction is complete, attribute synthesis begins, proceeding bottom-up.

Logically, a medium is a set of output nodes and their formatting semantics. In the implementation, the main function of the medium is to assist in the conversion of the output template of each tile to the actual node classes it provides.

6.1.2 Pattern Matching

In the initial implementation of the transformation tool, we used a very simple pattern-matching algorithm. To find the rule matching a given input node, this algorithm takes each rule in turn, and matches its pattern node by node against the input until it fails. The first rule that does not fail is the match. Iterators are handled by an eager strategy: if a subtree of the pattern is marked with `+` or `*`, the algorithm first matches it with as many input subtrees as possible. If the rest of the pattern subsequently fails, the algorithm "backs off" by one input subtree, and so on until the pattern succeeds or all combinations of iteration counts have been exhausted. This is clearly suboptimal; but before replacing it with a more sophisticated algorithm, we measured its actual cost.

In the worst case (a substantial Java program, presented with the standard Java

```
ERROR_TXTDEL
 -> [BOX [V(horiz_just=center)
           [H(size=12) "Deleted" I(8) $(ERROR_TXTDEL)]
           !self]];

ERROR_JAMPOINT
 -> [V(horiz_just=center) !self
      [H(size=12,fg="purple") I(4) "Jam at" I(8) $(ERROR_JAMPOINT)] I(4)];

ERROR_ISOLATION -> [BOX(fg="red") !self(fg=$(fg))];
```

Figure 6.1: Error presentation rules from the Java presentation schema used in our tests.

presentation schema, some 200 rules including many iterators), we found that the time taken by pattern matching was less than 13% of the presentation cost. So at present, changing the algorithm could not bring more than a 13% performance improvement. If other parts of the presentation system can be refined, increasing pattern-matching's share of presentation time to 50% or more, then it would be worthwhile to adopt one of the efficient algorithms from the literature [20, 22, 43].[1]

With the same Java program, we measured the time taken by the post-pass described in Section 4.3. In this phase, rules based on attributes of the input nodes are used to modify the transformation specified by the main body of the schema. The Java schema we used has three error-presentation rules in the post-pass, shown in Figure 6.1. The program had no errors, so the rules were never invoked; but still, after every tile was created, the nodes under its cover were checked for attributes, and the attributes were matched against the three error rules. (This is a realistic test: even when errors are present in a progarm, they are strictly localized by the history-sensitive parsing algorithm [95], so most tiles will have no error attributes.) In formatting the program, less than 2% of presentation time went to the post-pass.

[1] A related theoretical paper [85] suggests that a simple pattern-matching algorithm should be expected to be reasonably efficient. The authors investigate the average-case performance of a simpler pattern-matching problem, that of identifying all matches to a given pattern in a given tree. They conclude that "the *expected complexity* of sequential tree-matching"—that is, the algorithm we use—"is only *linear* in the size of the arguments." (Italics in original.)

6.1.3 Change Processing

As described in Section 2.4.2, changes to a document are represented by *change bits* on the tree. Every node on the path from the root of the tree to the change site is marked "nested changes"; the change site itself is marked "local changes". Because of the close relationship between the tree of tiles and the input tree, it is easy to follow the path of tiles covering the path of nested changes. This traversal stops when it reaches a tile whose cover includes a changed node. In general, at this point, the tile and all its descendants are simply regenerated. In the special case when the single changed node is used in a variable conversion—the most common case with single-character text edits—that single node is "reconverted", that is, either regenerated from scratch or reused with a small change. With large, relatively uncommon edits, such as block insertions of structure, crude regeneration is inefficient and often unnecessary; in the future, care should be taken to reuse as much of the transformation as possible.

The Proteus text medium [65] uses a large, complex text presentation node with line breaking for natural-language text. If this node were integrated into the tree-transformation tool, it would be possible to convey fine-grained updates to the view, capturing common cases that could be redrawn efficiently.

During change processing, nodes in the output tree are marked with "local changes" and "nested changes" bits, just like the input tree. The attribute-synthesis method is called for each affected node, in bottom-up order, bringing the entire tree into sync.

6.1.4 Cursors and Selections

Ensemble's general mechanism for handling cursors and selections ("extras") is described in Section 2.3. From the point of view of a presentation tool, the extras mechanism simply maintains lists of cursors and selections, to be displayed along with presentation information, in individual views. In this list, they are represented in document-oriented terms, using "locations" consisting of a node and optional media coordinates. To draw them, the transformation tool must convert these extras to display terms, using the transformations defined in the user's presentation schema to map document nodes to the corresponding output nodes. This mapping is essentially the same service already provided for change processing, where the problem is to locate the output node corresponding to a changed input node. Since the variables in a rule's pattern may be used any number of times in the action, a cursor location may correspond to many points in the output tree, or to none: the result is a possibly-empty set of locations in the output tree. (When an input node is mirrored in more than one place in the output, a cursor placed on any output image will appear on all. This takes a little getting used to, but there is no real alternative, since cursors denote a position in the document and not the presentation.)

The set of "output extras"—cursors and selections mapped onto the output tree—is passed as an argument to every drawing function provided by the medium. In the pretty-printing medium, drawing is delegated to the drawing methods of individual nodes, and this set is passed along with the other drawing arguments. Each node's drawing methods scan the set, looking for extras on the current node. A cursor on a nonterminal is drawn as a rectangular outline of its bounding box; a cursor on a text node is drawn as a vertical line between the appropriate pair of characters.

When the user clicks to place a cursor, the coordinates of the click must be mapped from two-dimensional presentation space to a location in the document. This procedure, known as hit detection, falls into two stages. First, the display coordinates are mapped to a node in the output tree, possibly with extra media coordinates such as a text offset. This mapping is performed by the output medium method, since it relies on the medium-specific encoding of position information on the output tree. Second, the output node is mapped to an input node, in the reverse of the mapping used when cursors are drawn. This mapping is performed by the core transformation engine, since it depends only on the transformation rules and not on properties of the medium. Both mappings may fail: some positions may not be covered by any output node, and some nodes in the output do not correspond to anything in the input.

Currently, cursors are implemented in the transformation tool, but selections are not. The only difficulty we anticipate in extending the implementation to handle selections is really a user-interface problem rather than a technical one: how to handle the contiguity of a selection. Since transformations reorder the tree (indeed, this is their purpose), a contiguous stretch of nodes in the document may be widely distributed in the output, and vice versa. So if the user clicks and drags across the window, is he selecting a range of the underlying document, or a range of the layout structure? If a selection is a contiguous range of the layout, the corresponding parts of the input document may be a number of separate ranges, making it difficult for the user to understand what he has selected. Further, the selected output may not correspond clearly to any input. Alternatively, if a click and drag is defined as a contiguous range of the input document, this will lead to similar difficulties, in reverse. For example, we might define the selection as the smallest contiguous input range covering all the structure corresponding to material in the indicated output range. This can lead to surprising behavior: a slight extension of the chosen visual range may unexpectedly select large portions of the document elsewhere on the window, and off the window as well.

6.1.5 Supporting Attribute Synthesis

Each medium may choose to represent synthesized attributes in whatever way is convenient. However, a general mechanism is also provided; the main service it provides is simple change-reporting. When an output node is created, an optional constructor argument is a list of the attributes it wishes to handle in this way. Attributes are named and indexed, and they may be marked either "per-node" or "per-child"; for example, child positions in the pretty-printing medium are stored as a per-child attribute of the parent. The number of per-child attributes is automatically kept consistent with the actual number of children. These attributes automatically record whether they have changed or not in a given incremental attribution pass. Thus, when a change propagates to a client rendition, the redraw code can update just those areas of the layout that have changed.

Incidentally, change processing gives another argument for making the position of a child, in the pretty-printer, an attribute of the parent. Consider a V node, with three children arranged in `gap` spacing style. If the user causes a change in the second child, increasing its height, the third child must move down. But the content of the third child's subtree is the same. If its relative position were one of its own attributes, it would unnecessarily be marked "changed", though its screen image would not need to be redrawn. Change markers belong on the second child, whose contents have indeed changed, and the parent, which has adjusted its own layout. This enables the rendition to update the third child's image by an efficient block-move operation, rather than redrawing it.

6.2 Coding for Efficiency

In the introduction, we described four categories of efficiency that are important for a presentation system in an interactive document environment. In this section, we examine the efficiency of the initial implementation of the transformation tool. Where its efficiency is inadequate, we show how it has been modified for better performance. Here we treat efficiency in asymptotic terms, as a function of N, the number of nodes in the document. We present measurements in Section 6.3 below.

We take the four categories out of order, to clarify the exposition. We draw our examples from the pretty-printing medium: this is where the transformation tool has been stressed most severely, since users frequently edit large programs, and since programs are "dense," having a high ratio of nodes to text.

6.2.1 Change Time

The change time problem falls into three stages. The first stage is updating the document, which falls outside the scope of this dissertation. As shown in the work of Tim Wagner [92], this stage is approximately logarithmic in document size.

The second part of the problem is propagating the change from the document to the presentation. The time taken to bring the presentation up to date depends on the depth of the change in the document tree, that is, the number of nodes on the path from the root to the change. Generally, this depth is proportional to the logarithm of the number of nodes in the tree, except in one important case: sequences.

Sequences of declarations or statements in programs can grow quite long. In parsers like Ensemble's, built by automatic parser generators like `bison` [30], these are traditionally represented with recursive productions in the grammar, for example:

```
DECL_LIST:      DECL
        |       DECL_LIST DECL
        ;
```

If the syntax tree represents the parse straightforwardly, with one nonterminal for every reduction, then a list of n declarations will become a chain of length $n + 1$. In the C language, which lacks nested functions, the top-level syntax of a file consists of a single list of declarations. Thus if a very large C file is represented by a parse tree with chained list productions, the search for any changed node will be dominated by the traversal of the chain of top-level declarations, with $O(N)$ performance.

With a hand-coded parser, this problem could be handled by making nonterminals like `DECL_LIST` n-ary nodes, collapsing the chain into a single parent with many children. With its automatically generated parser, Ensemble addresses the problem in a different way. It extends the language description syntax, representing sequences with an extended context-free grammar (ECFG) notation [54]. Thus, the DECL_LIST production could be specified simply as

```
DECL_LIST: DECL*;
```

Or, more expressively, the sequence notation could be used at a higher level, for instance

```
COMPOUND_STMT: '{' DECL* STMT* '}';
```

When sequences are identified in this way, the parser represents them internally as balanced binary trees, built of automatically generated nodes [94]. Clients may choose to view the sequences either as trees or as flat n-ary branches, using the abstraction mechanism (see Section 2.2.1). The time to find a change in a flat sequence is linear; the time to find it in a balanced sequence tree is logarithmic in the length of the sequence, preserving logarithmic change time overall. Currently, in the transformation tool, this choice is controlled by explicit syntax in the schema languages (see p. 122). As we show in Section 6.3, using balanced structure is much more efficient.

(In the future, the transformation tool should automatically view the input trees in the "raw", balanced abstraction, and also use balanced structure to represent sequences in its own output. This is not a trivial change. The transformation of flat to balanced tree structure affects not only the patterns and actions, but also the layout computation. Since derived (synthesized) layout attributes are presently implemented in handwritten C++ code, the attribution algorithms will probably have to be transformed by hand.)

The third part of the change time problem is propagating the change from the presentation to the rendition, i.e. the screen. It would be possible simply to redraw the entire content of the window on every change; as we will discuss in Section 6.2.2, this can be done in $O(\log N)$ time. While this is asymptotically logarithmic, it is not fast enough in practice to maintain the sense of interactive typing, and it can flicker badly. Our experience with the line-based presentation tool (Chapter 3) suggested that it is acceptable to redraw a whole line per keystroke. While the notion of a "line" is not well-defined in any of the media supported by the transformation tool, including the pretty-printer, we can translate this into a more general rule of thumb. A line in a line-based presentation generally has an area between 1000 and 10000 square pixels.

As an ad-hoc criterion to limit redraw time in the pretty-printer, we decided that any element that has changed or been moved horizontally might be redrawn; but that any element that has stayed still must be left alone, and any element that has moved vertically must be copied with a block operation. To detect these cases, we used the synthesized-attribute mechanism described in Section 6.1.5, which provides attribute change information. When the rendition client walks over the output tree, it can follow change bits to the nodes that have been reformatted, and then check whether the synthesized attributes have actually changed. If there is no change, it proceeds recursively to a lower level of the tree. When the first k children of a node have remained in place, and the rest have all moved up or down by the same increment, it updates the later children by a graphical block-move operation, rather than redrawing them.

After implementing these changes, we instrumented the pretty-printer to report the size of the area it redraws on each keystroke. In practice, while editing Java programs, we find that the median area redrawn is less than 5000 square pixels, and speed and appearance are acceptable.

6.2.2 Refresh Time

By "refresh time" we mean the time taken to redraw the contents of a window, when it is exposed by the window system, or when the user scrolls to a new location. There are three factors that affect this time. One is the number of visible elements inside the bounds of the window, and another is the time taken in drawing these elements individually. These two factors, to a first approximation, remain constant once the document has grown larger than a single window-full: in a text document, for instance, the variation in the number of "words" per "page", and in the size of the words, does not depend on document size.

The third factor is the number of nodes—whether visible or not—that are to be visited in the course of drawing the window. These nodes may be further subdivided into two categories, the necessary and the unnecessary. In the pretty-printing medium, the necessary nodes are the visible nodes plus all their descendants. This is because every node's position is expressed relative to its parent, so to draw a node correctly, one must visit each of its ancestors to compute its absolute position. The size of the necessary set, then, is bounded by the depth of the tree ($O(\log N)$) times the number of visible nodes (a constant, bounded in practice by the size of the screen).

The unnecessary nodes are those without visible descendants, but which are visited anyway. Now the drawing algorithm works top-down. When it reaches a nonterminal, it examines each child. If the child's bounding box overlaps the area to be redrawn, the child's draw method is recursively invoked. It is possible to construct "pathological" cases in which this simple criterion is wrong, that is, the node's bounding box may overlap the window though none of the node's terminal descendants do. But these pathological cases are very rare: in real program layout, most output tree constructions are very simple, and dominated by a long vertical layout of "lines". In other words, the top-down algorithm never pursues false leads deeper than one level into the tree. The unnecessary nodes, then, are limited in practice to siblings of necessary nodes, along paths to visible terminals. In practice, too, the branching factor of output trees is limited. So the number of unnecessary nodes can be bounded by the number of visible nodes (a constant) times the number of nodes on the path from the root ($O(\log N)$) times the maximum number of "unnecessary" siblings at each node on the path (a constant), or $O(\log N)$ overall.

6.2.3 Size

In the original implementation, the size of the output tree was linear in the size of the input tree, with a constant factor that varies somewhat between schemas: typically there are no more than twice as many output nodes as input nodes. To reduce the size, we chose to discard unused structure, replacing it on demand.

The granularity of this "paging" procedure is the tile. Periodically, the tile tree is scanned for unused material. Any tile that has not been touched since the last scan is reduced to a stub, and its descendants are discarded. Later, if navigation through an output node leads to this tile, it is regenerated transparently.

For paging to work well, the attribute computation for a node must never depend on information in another subtree that has been flushed. This is the most important reason that we restricted attribution to pure inheritance followed by pure synthesis.

In the best case, this policy will reduce the tree to just those tiles that contain nodes that are visited during the redraw of the current window contents. As we argue in Section 6.2.2 above, the number of nodes, and of tiles, in this set is $O(\log N)$.

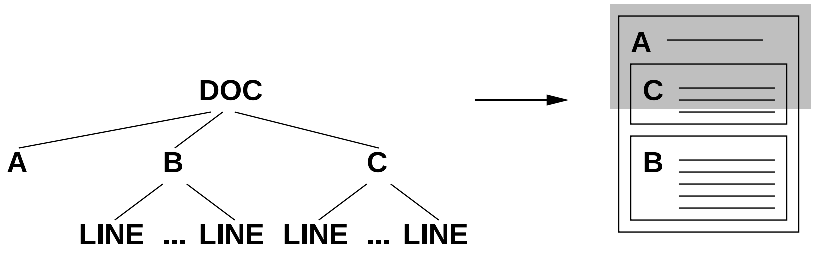
6.2.4 Startup Time

In the initial implementation of the transformation model, startup time—the time between loading a document and seeing it on the screen—is linear in document size. To make this time sublinear, it must be possible to format the output only partially, preparing just enough to fill the first window-full, and deferring the rest of the formatting until later.[2] To this end, we have added a `format` method to the medium interface, taking as an argument the dimensions of the minimum area to be formatted. The method returns a value indicating whether more formatting remains to be done. There is a default implementation of this method, which formats the tree completely and returns **false**; but real partial formatting has been implemented in the pretty-printing medium. In this medium, formatting was already driven by recursive calls to a layout method in each output node; to achieve partial formatting in practical cases, the only change necessary was to the layout method of the V node. The modified method scans just enough children to fill the required height; if this is less than the full count, it saves an index to the first unformatted child, so it can resume formatting again on a later pass.

This cuts the initial formatting costs to $O(\log N)$, since only the nodes necessary for covering the first window-full are formatted. But it does not address the costs of transformation: the entire output tree is still built before formatting can begin, at $O(N)$ cost. To avoid constructing unnecessary output, we take advantage of the transparent mechanism for constructing output on demand, which was already introduced to save space. When the presentation is built, its tile tree consists of one reduced tile at the root, with no output nodes or child tiles. Thereafter, structure is created on demand. If nodes that perform partial formatting take care, not only not to format unnecessary nodes, but not to touch them (i.e. request them through the tree navigation interface), the unnecessary nodes will not be constructed. This extra discipline has been added to the V node's formatting method. With this simple modification, presentation startup time becomes $O(\log N)$.

Once this first window-full has been constructed, formatted, and displayed, partial formatting continues in successive window-fulls until the layout is complete. The flushing mechanism described in Section 6.2.3 is active throughout this sequence. Thus, if a docu-

[2] Robert Wilensky has pointed out (personal communication) that many systems, e.g. PostScript viewers, begin drawing even before formatting is complete. In a sense, these systems have a "startup time" less than the time taken to format the first window-full; but it remains important to minimize the time before the user sees a correct display of the first screenful.




```
[DOC ?a ?b ?c] -> [V *a !c !b];
    [B ?line*] -> [V *line];
    [C ?line*] -> [V *line];
```

Figure 6.2: A simple text document: the desired presentation is a visual reordering of the content. The top-level presentation rule expresses this reordering by transformation syntax. The first window-full of the layout is indicated by a gray rectangle.

ment is long enough, structure will begin being discarded before the end is reached. In other words, the layout is never present all at once; yet by the end of the partial-formatting process, the parts that *are* present are correctly formatted, and missing parts can be restored transparently, without disturbing the rest of the tree.

We illustrate the progress of partial formatting in Figures 6.2 and 6.3. Note how the reading (including pattern-matching) of the input document is driven by the demand for the output tree, mediated by the specific transformation rules being used. In formatting the first window-full of this document, the transformation tool visits nodes A, B, and C, and the subtree below C, but it skips the subtree below B, which does not contribute to that part of the layout.

This highlights a general, somewhat abstract benefit of tree transformation as a presentation mechanism. If we consider the layout of Figure 6.2 in relationship to the document tree, it seems to lack locality: for example, A and C, though near on the screen, are far apart in the input tree. Yet the first window-full can be formatted relatively quickly, for two reasons. First, the output tree has good locality with respect to the graphical layout. Second, the relationship of the input tree to the output tree is expressed by formal rules, which enable the transformation tool to guide the traversal of the input document automatically, in response to the formatting of the output tree. In other words, even a presentation that is considerably reordered can take advantage of the benefits of locality, as long as its non-locality can be encapsulated in a formal transformation.

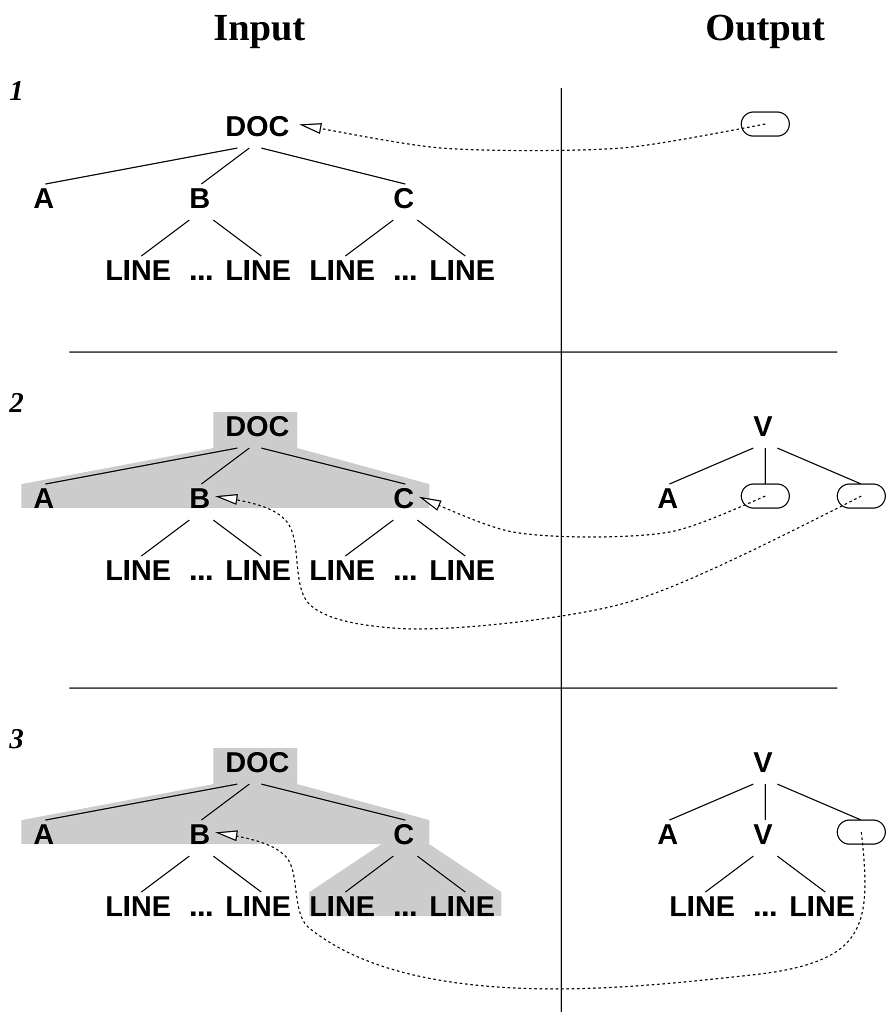


Figure 6.3: Stages of transformation in formatting the first window-full (gray area) of Figure 6.2. (1) At first, the only output is a placeholder, connected to the root node of the document. (2) A rendition client begins to examine the output. When the placeholder is first touched, the root node of the document is transformed, by application of the top-level rule, yielding one node of text and two more placeholders. (3) The client looks at the second child, and the C subtree is automatically transformed. Now the first window-full is ready to be displayed. Note that the B subtree, which does not contribute to the first window-full, has not been examined.

Partial Loading

As we mentioned in the introduction, part of the time the user experiences as startup is due to the initial loading of the document. In Ensemble today, a document is completely loaded before presentation begins. This means that *overall* startup time is still $O(N)$. (This is not only undesirable for ordinary user interactions, but unacceptable in the Internet environment, where the time between acquiring the beginning of a document and reaching the end may be minutes, and users expect to read a document as it comes in, possibly deciding not to load the rest.) To achieve sublinear loading time, it must be possible to begin formatting with only part of the document in place. There are several considerations that may make this more difficult in Ensemble than might at first appear. The first two affect Ensemble's general architecture, and the third affects the transformation tool.

First, different stages of a document's construction are currently recorded as distinct versions. This feature must be suppressed in the case of partial loading, to prevent the user from backing up to partially constructed versions that reflect only the history of the document's loading, not its editing.

Second, documents are often created bottom-up: this is true, for instance, of documents in ordinary Ensemble format and HTML. This means that the root node is not created until all its descendants are available—in other words, the tree structure is unavailable the entire document has been loaded. For partial loading, nonterminal nodes must be created and inserted into the tree, in a provisional or speculative form, *before* their children.

Third, transformation schemas generally assume the input document is complete and correct. To accommodate partial loading, the transformation must return an acceptable presentation of any partial form of the document, and then update it as the full document structure is filled in. To make the user responsible for this flexibility, by writing transformations to handle all possible partial productions from the source language, is unacceptable. A change must be made in the pattern-matcher, to enable it to find incomplete matches when full matches fail, and in the corresponding action-based output generator, to handle the incomplete variable bindings that will result.

```
SCHEMA sonseq;

MEDIA text;

sonseq: title epigraph sonnet+;
title = text;
epigraph: line+;
sonnet: number line+;
number = text;
line = text;
```

Figure 6.4: Structural schema (grammar) of the sonnet-sequence documents used for measurement.

6.3 Measuring Efficiency

The most important and demanding documents handled by the transformation tool are programs. A window-full of program text displays on the order of 1000 nodes from the parse tree, laid out by a correspondingly large output tree. However, it is difficult to use programs to measure the performance of the transformation tool as a function of document size. This is because performance is affected not only by the raw size of the document, but also by details of its structure: exactly how many nodes are visible, which rules govern their formatting, etc. Thus, with programs, plots of performance against document size look very noisy, often making it difficult to discern any trends at all, let alone the asymptotic behavior of the characteristic being measured. (We believe this is the reason for the inconclusive measurements reported in Section 3.3.)

Accordingly, we built a test set of simple natural-language documents, varying smoothly in size but uniform in structure. The documents are sonnet sequences, drawn from the sonnets of Shakespeare [80]. The grammar of the documents is shown in Figure 6.4. Each one consists of a title, an epigraph, and some number of sonnets. The internal structure of the sonnets is treated as an undifferentiated list of lines; a fuller grammar for print would recognize the last two lines of the Shakespearean sonnet as a distinct "couplet", traditionally indented a couple of ems to the right. (It might also recognize that there are fourteen lines in the standard sonnet, even though there are two exceptions in Shakespeare.) Figure 6.5 shows the sample documents we used, with their sizes, ranging from one sonnet up to the full set of 154.

File name	Size in nodes
`shake1.ens`	77
`shake5.ens`	257
`shake10.ens`	482
`shake15.ens`	707
`shake20.ens`	932
`shake33.ens`	1517
`shake50.ens`	2282
`shake75.ens`	3407
`shake100.ens`	4535
`shake154.ens`	6959

Figure 6.5: The sample documents used for these measurements. The file name gives the number of sonnets in each; the file size is the number of nodes in the in-core document tree.

In these tests, we used a uniform window size of 600 by 800 pixels, the default set by the current Ensemble installation. We used a simple presentation of the sonnets, omitting the epigraphs. Figure 6.6 shows `shake10.ens` displayed in this manner.

6.3.1 Size

We measured size under the most favorable circumstances: displaying and redisplaying the same window-full of the document, allowing as much as possible to be reclaimed by garbage collection. Figure 6.7 shows the results. Presentation size begins by climbing steeply with document size, then tails off. However, it does not tail off below a certain linear rate, about 80 or 90 bytes per document node.

We guessed that the problem might be the single flat list of sonnets. To test this assumption, we wrote an alternate presentation for the same document type. Using the `abstraction` variable (see p. 122), it specifies that the input tree for transformation should be the "raw" tree, in which the balanced binary trees underlying sequences are exposed. Accordingly, the presentation schema does not use any `+` or `*`: all the productions it recognizes have fixed arity. (The sequence productions have arity two.) The results of carrying out the same size measurement with this schema are shown in Figure 6.8. Now the logarithmic curve of document size is visible just as we hoped. In fact, beginning with the sequence of 20 sonnets (size 932), the growth of the presentation size is swamped by the inherent "noise" in the measurement (due, we speculate, to fine details of the binary tree structure). Note also that, in these measurements, the document is effectively larger than

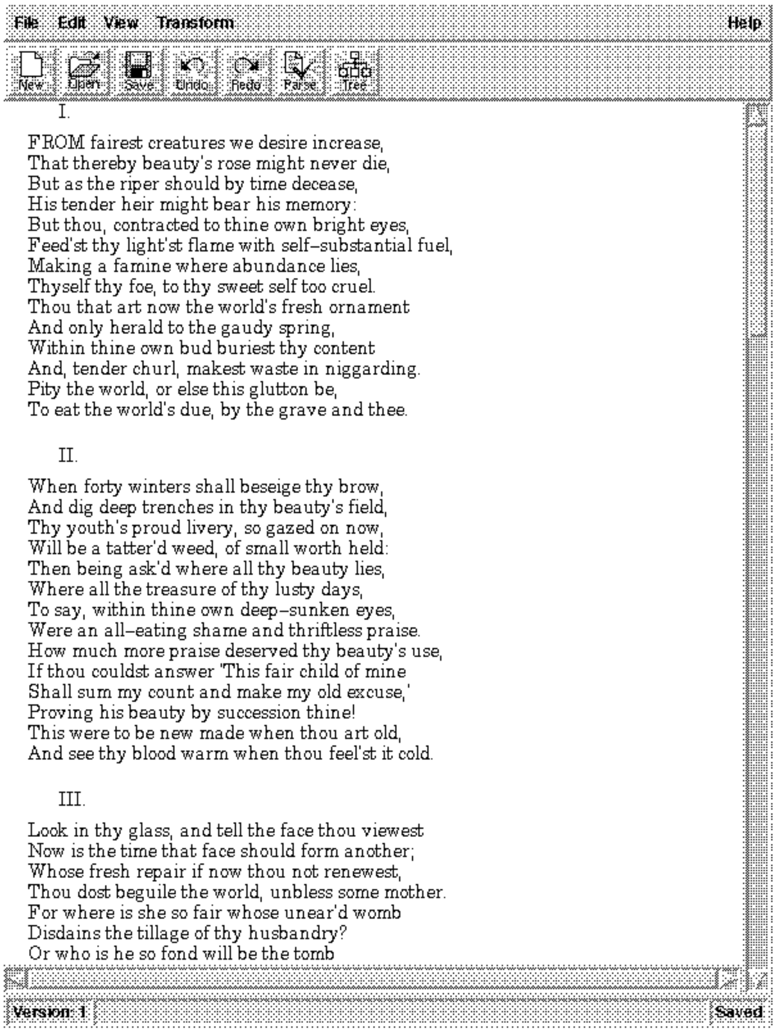


Figure 6.6: A sonnet sequence, displayed in the style used for these examples.

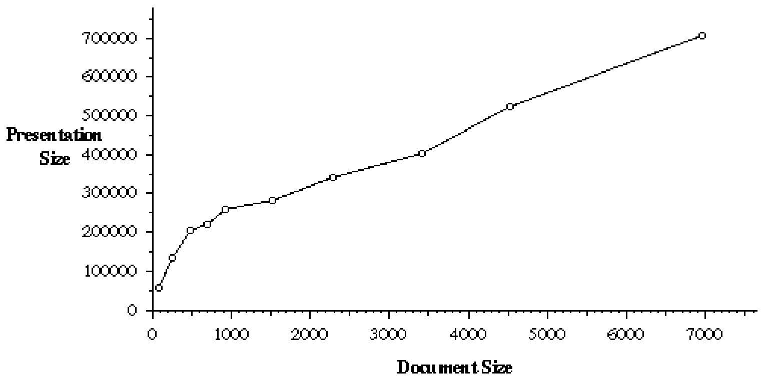


Figure 6.7: Presentation size as a function of document size, under optimal conditions, with a "flat" abstraction. Document size is in nodes, presentation size in bytes.

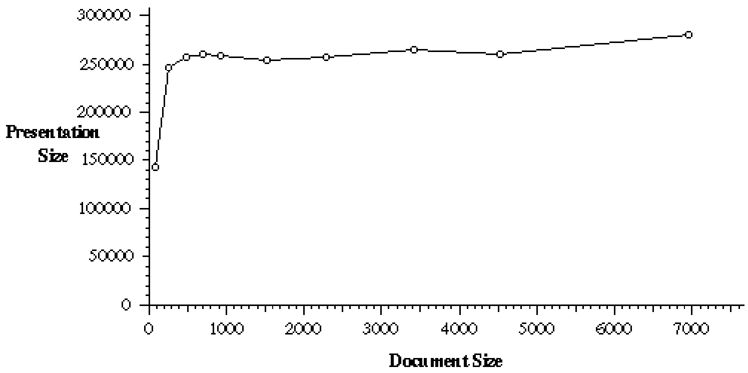


Figure 6.8: The same size measurements, repeated with a presentation schema using the binary-tree expansion of the sequence productions.

in the "linear" case, since it includes the binary sequence nodes in addition to the content nodes visible in the default abstraction.

We should also note that the performance of the "paging" scheme is often poor with programs. With "worst-case" loads—scrolling rapidly from top to bottom and back—more than half of the full tree is often present. Further, it can often happen that reconstruction of missing structure takes unacceptably long. One of our sample Java files, for instance, is about 1000 lines long, of which more than half is a single class declaration. When the user works in another part of the presentation, the structure corresponding to this large class is flushed. If the user then scrolls or pages to the beginning of the class declaration, Ensemble freezes for a long time—tens of seconds—while reconstructing the entire output

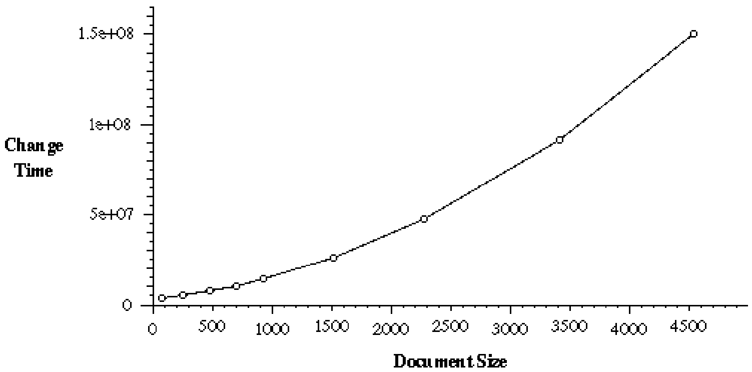


Figure 6.9: Change time, measured in machine cycles, for sonnet sequences presented with the "flat" abstraction. The time includes both presentation update and view update. Figures for the complete sonnet cycle (`shake154.ens`) are missing because the measurement tool failed.

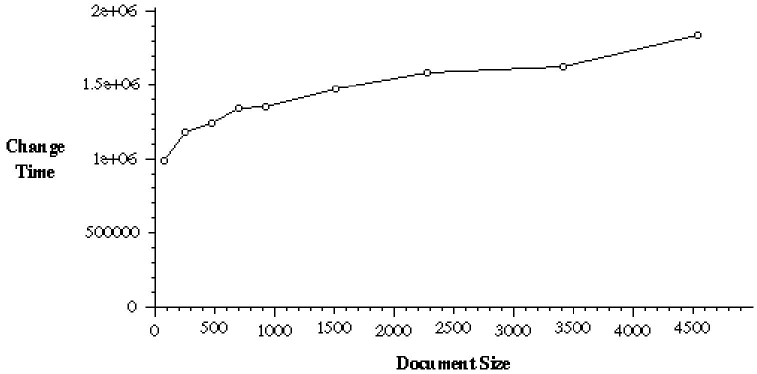


Figure 6.10: Change time, measured in machine cycles, for sonnet sequences presented with the binary sequence abstraction.

for the class. This problem could be solved by extending the partial-formatting mechanism to handle reconstruction as well, so that only the top of the class declaration was restored before drawing proceeded.

6.3.2 Change Time

We measured change time by opening up each document to the same place, the first sonnet, and deleting a word ("desire") in the first line. We used Quantify [70] to compute the number of machine cycles used to propagate these six document changes to the presentation and then to the view. (These measurements were taken on a 50-MHz SPARCstation II

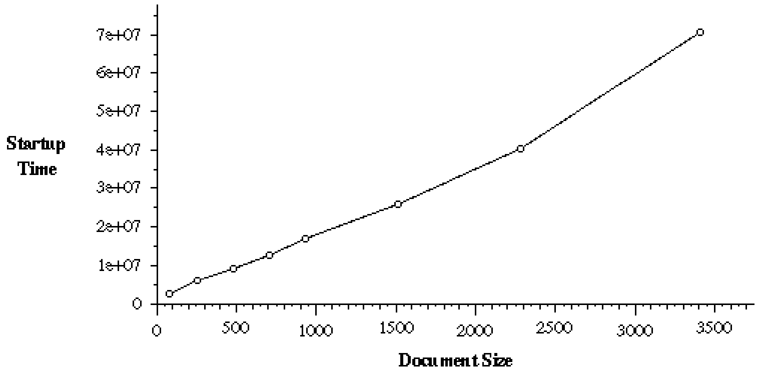


Figure 6.11: Startup time with the "flat" presentation. Again, the measurement tool failed with larger documents.

running SunOS 4.1.3.) Figure 6.9 shows the results for the "flat" presentation. Presentation and view update time both grow approximately linearly. Figure 6.10 shows the results with the "binary" presentation. In this case, both update times grow more and more slowly with document size.

6.3.3 Startup Time

To measure startup time, we instrumented the constructor of the master C++ class, which carries out the first partial formatting of the output tree. The results are shown in Figures 6.11 and 6.12. Startup time grows steadily with document size when the "flat" presentation is used. With the "binary" presentation, growth flattens off rapidly, even seeming to reverse itself.

6.3.4 Refresh Time

To evaluate refresh time performance, we displayed both the top and the bottom window-full of each document, measuring the time taken to fill in the window, and adding the two values. The results for these measurements are given in Figures 6.13 and 6.14. For both presentation structures, the performance is nearly uniform for all document sizes, with the "binary" presentation taking about twice as long as the "flat" one. Searching long lists of children to find the ones that fall within a bounding box is potentially time-consuming, but we use binary search to keep the performance logarithmic.

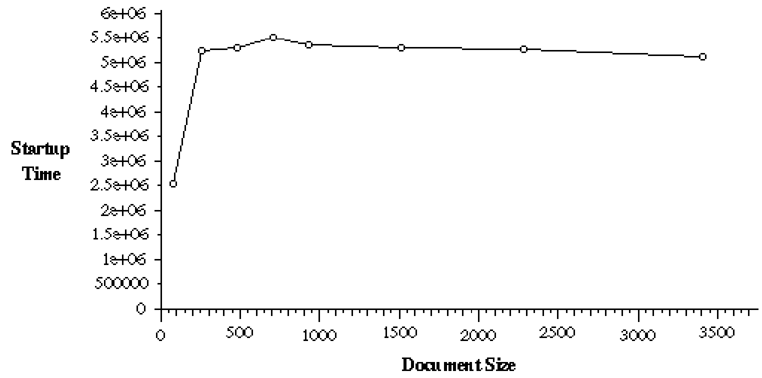


Figure 6.12: Startup time with the "binary" presentation.

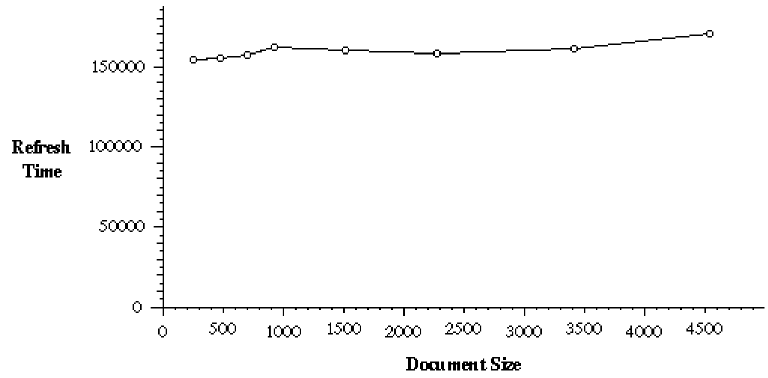


Figure 6.13: Refresh time measured with the "flat" presentation. The smallest document is omitted because it is smaller than one window; the largest is omitted because the measurement tool ran out of memory.

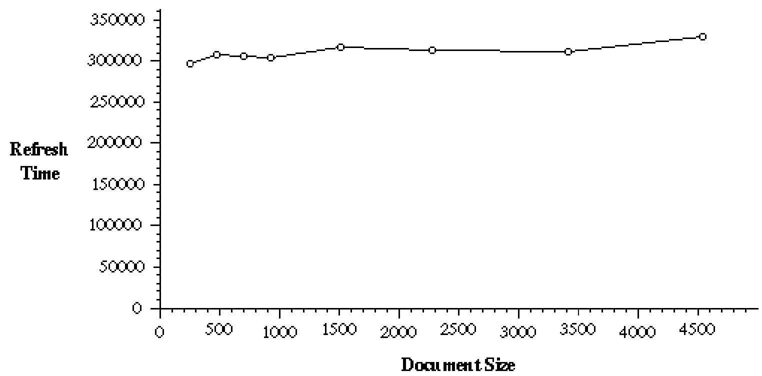


Figure 6.14: Refresh time measured with the "binary" presentation. The smallest document is omitted because it is smaller than one window; the largest is omitted because the measurement tool ran out of memory.

Core transformation engine	6674
Pretty-printer	2264
Tree display	1178
Graphing tool	2317
Total	**12433**

Figure 6.15: Lines of code in the current implementation, including all source files.

6.4 The Code

The current implementation of the transformation tool is in C++, plus specifications for the `bison` parser generator and the `flex` lexer generator. Figure 6.15 gives a breakdown of the lines of code in each of its various modules. Beyond this, there are a dozen or so schemas written in the transformation-schema language, for the three media.

Chapter 7

Related Work

In this chapter, we survey related work. We begin with tree transformation and presentation research, and progress to systems that use tree transformation for presentation in various ways.

7.1 Tree Transformation and Rewrite Systems

Hierarchical or tree structures arise naturally in many fields, and there is a long history of formal systems for describing their modification. In logic, the early work of Church on the lambda calculus [25] can be considered as a tree rewrite system; for a recent example, Ramakrishnan et al. have used formal transformations of proof trees to optimize logic queries [74]. Farther afield, the theoretical model of music developed by Heinrich Schenker in the early decades of this century [79] represents a composition as the result of a series of tree rewrites, beginning with a simple melodic fragment (the *Ursatz*). Frankel et al. [37] demonstrate the implementation of this model by explicit tree transformations in Lisp.

There is a substantial mathematical literature on tree transformations. A recent paper by Fülöp and Vagvölgyi [38], for example, presents a classification of the tree functions that can be computed by certain top-down transformations. An important motivating application for this work, since the beginning, has been compiler code generation. Through the 1960s, an important research goal was the automation of code generation by "syntax-directed compilation." In this model, an input string is parsed according to a context-free grammar. The resulting parse tree is transformed by rules applied to its nonterminals, reordering their nonterminal children and changing the terminals: the yield of the resulting tree is the output. Lewis and Stearns [56] and Aho and Ullman [4], among others, investigated the construction of automata to perform the conversion from input to output strings. But Rounds [76, 77] and Thatcher [88] began to consider trees as both the input and the output of interest. Our work has been influenced most by the use of such tree transductions in subsequent code-generation research.

Aho and Johnson [3] showed that instruction selection could be modeled by a rewrite system. An expression to be computed is represented as a tree; each machine instruction is represented by an expression-tree fragment describing the computation it performs. The generation of code, then, can be approached by seeking matches for the patterns in the tree. When a match is found, the appropriate instruction code is emitted, and that part of the expression is replaced with a single node representing the result of the instruction (typically a register). This process continues until the tree has been reduced to a single node. A numerical cost (typically time) is associated with each instruction; if the costs combine additively, the optimal instruction sequence can be found by dynamic programming. The output of interest is not the tree but the instruction sequence generated as a side effect. This approach is used by Aho et al. in `twig` [5], which automatically generates a code-generator from a machine description giving the pattern and cost for each instruction. Graham and Glanville [40] present a related code generator, in which the intermediate representation is linearized into prefix notation, and the pattern-matching is performed by parsing it.

Eduardo Pelegrí-Llopart [68] presents a class of transformation mechanisms called bottom-up rewrite systems (BURS), developing their mathematical properties and showing how to use them to construct a code generator. His dissertation also provides a useful review, classification, and theoretical discussion of other transformation systems. Charles Farnum [35] used tree transformations as the basis for a platform for prototyping compiler optimizations. The language Dora, developed in his dissertation and in a later paper with John Boyland [17], was the direct inspiration for our own transformation language.

OPTRAN, developed by Reinhard Wilhelm and others at the Universität des Saarlandes in Germany [58], is a batch-oriented compiler-construction system. It supports tree transformations written as sets of rewrite rules, modifying the tree in series. These transformations are used to carry out such optimizations as constant propagation. Since the tree is attributed, and the attributes may be read and modified by the transformation rules, maintaining the attribution of the trees incrementally is an important issue.

In the terms of Pelegrí-Llopart [68], what we have implemented is a *projection* system. The tree is analyzed into a *cover*,[1] a set of fragments matching the given patterns. Each fragment is transformed by the corresponding rule, and the results are composed into a separate output tree, by *parallel* composition, rather than the *serial* composition used in a rewrite system. He suggests using this model for the parallel maintenance of concrete and abstract syntax trees. It is appropriate for our application because it leaves the input tree intact, essential if the document is to persist, and because the cover makes it easy to maintain the transformation incrementally.

Tree transformation systems vary considerably in the form of patterns they support. In a *nonlinear* pattern, the same variable appears more than once; the value of the node bound to the variable must be the same in both places. In a *nonlocal* pattern, the input shape matching the pattern need not be a contiguous tree fragment. We have used only local, linear patterns. Pattern languages also vary in their support for *typing*. In one powerful form, a pattern may constrain the shape of the subtree rooted at the nodes it matches; such a pattern, for example, could recognize chains of even length. In a weaker form, patterns can allow variables to have nonzero arity. We do not support either of these forms, because we have not yet identified document presentation problems that require such structural analysis. In principle, Ensemble's existing semantic analysis facilities could be used to compute the same information. The variable "type" provided in our pattern language refers to a symbol name from the input document's grammar, not to the result of any pattern processing. Further, many varieties of *iterators*, notation for matching repeated structure, have been proposed for patterns. We provide the Kleene `*` and `+`, familiar from regular expressions; in Farnum's terms, these are *horizontal* iterators. Dora also provides *vertical* iterators for recognizing repeated nesting structure; we discuss the possibility of their use in Section 5.1.2 above. There are many more sophisticated features in the literature: for just one example, Boyland's APS [19] provides a mechanism to "factor out" common elements among patterns.

This brief survey by no means exhausts the variety of tree-transformation systems found in the literature. In one family, for example, the specification depends on a formal syntax of the output language. In the TT-grammars of Keller et al. [48], the specification

[1] Or "tile tree," in our terms. We use the term "cover" in a different sense: see Section 6.1.1.

consists of context-free grammars for the two tree languages, plus "production associations" and "symbol associations" between the grammars. These are compiled into transducers, written in an imperative programming language, to convert trees from one form to the other. And in the model of Giegerich and Schmal [39], source and target languages are related by a common core and a set of "derivors" from target to source; the transducer is generated by inversion of the derivors. In our application, though, the output trees have little interesting syntax. In most cases, output nonterminals can have any number of children, of any type, and the few constraints are semantic rather than syntactic. For example, a LINE nonterminal in the graphing output language requires REFERENCE items among the list of graphical objects collected from its children; but they may derive from nodes of several classes, scattered anywhere among its descendants. Since the input trees of programs and structured documents do have interesting syntax, an input-driven model such as projection makes better sense.

7.2 Attribute Grammars

Attribute grammars are an important formalism, relevant both to the attribution of Ensemble's output trees and to the transformation mechanism itself. They were introduced by Knuth [51] to represent the semantics of programming languages. In this original form, they consist of tree-attribution rules associated with the productions of the context-free grammar of the language. Each rule expresses the value of an attribute of one of the symbols in the production, in terms of the other values. Together, the rules specify a system of equations for attribute values over the entire tree. In the Synthesizer Generator of Reps and Teitelbaum [75], a language-based editor for programs, attribute grammars were used for type analysis and code generation. The difficulty with attribute grammars in an interactive setting lies in achieving good incremental performance. In the Synthesizer Generator, performance could be improved by restricting the attribute grammar to the class of *ordered* attribute grammars.[2] Alblas [6] shows that when a series of changes can be expressed as a formal tree transformation, they can be arranged into a series of alternating transformation and re-attribution phases, minimizing the number of node visits and recomputations. In both systems, an efficient evaluator was "compiled" from the attribute grammar.

[2]The Synthesizer Generator also supports transformations: but these are local rewrites carried out on user commands, not automatic, overall tree transductions.

By contrast, in the Ensemble transformation model, we restricted the form of attribute rules severely, so that changes propagate only directly up and down from a change site. This enabled us to define a fixed evaluation order regardless of the specified rules. Such restricted evaluation classes are acceptable for many of the forms of layout we encountered, but not for all—they fail, for instance, with "structural" line-breaking (Section 5.1.5).

Attribute grammars often include complex values, such as a binding contour or symbol table used to transmit variable definitions from the definition site to their use. Maintaining such aggregates incrementally is an important problem; in the simplest solution, the aggregate is treated as an atomic attribute, and a change to one definition in the symbol table requires reevaluation of all uses of variables in the scope. Analogous problems arise in document formatting: for instance, bibliographic citations rely on a similar aggregate of attributes, gathered in the bibliography and distributed throughout the document. The problem has recently been addressed by William Maddox [59], using the technique of "fibering" to maintain fine-grained dependencies efficiently.

The higher-order attribute grammars introduced by Swierstra and Vogt [86] extend the attribute model still further, allowing abstract syntax trees to be returned by semantic functions, passed as attribute values, and "grafted" into the current tree. This is a promising formalism for expressing syntactic/semantic constraints: one of the examples in the literature is a document representing a form, whose structure changes depending on the answers given (say whether the respondent is male or female). This is a realistic application, and currently beyond the capacities of Ensemble. Teitelbaum and Chapman [87] address some of the technical problems of implementing higher-order attribute grammars in an interactive environment.

7.3 Pretty-Printing

"Pretty-printing" has traditionally meant a cleaned-up presentation of a program, more readable and more consistent than the programmer's original source. It has generally been conceived as a batch service, using the syntax of the program as a guide to reformatting, changing linebreaks, whitespace and indentation but leaving the text otherwise intact.

The pretty-printing algorithm of Oppen [66] begins with the syntax tree of the program. Unparsing rules associated with the nodes in the tree emit text marked with newlines, conditional newlines and nested block structure—not necessarily nested in the same way as the syntax tree. This stream accumulates in a buffer, which is flushed either when an unconditional newline is emitted, or when the buffer fills. The indentation and linebreaks of the output are computed according to the nesting of the text in the buffer. Lisp pretty-printers, which go back at least to the sixties, work similarly. Richard Waters' XP [96], for example, later incorporated into the Common Lisp language specification [84], was the culmination of several generations of Lisp pretty-printers, and its layout mechanism is essentially the same as that of Oppen.

Some batch pretty-printers have extended the elements of the output text to include colors, text attributes, and graphical elements. Presotto and Joy's pretty-printer **`vgrind`** [69], for example, uses bold and italic fonts to distinguish syntactic elements. Baecker and Marcus [9] reconsider program layout extensively, evaluating the readability of programs presented in various styles.

Knuth and Plass [53] present the boxes-and-glue model of TEX primarily as a technique for line-breaking ordinary text, but they also apply it to programs. Pretty-printing then falls into two steps, similar to the unparsing and line-breaking stages of Oppen's algorithm. First, the program is converted to a stream of boxes and glue. The attributes of the text in the boxes, and the size, stretch, and break penalties of the glue items, are set according to the syntax of the program. Second, this stream is line-broken with the same dynamic programming algorithm used for ordinary text. It is not clear from the paper how the user can control the process.

The immediate ancestor of Ensemble at Berkeley was the Pan project, described by Ballance, Graham, and Van de Vanter [11]. Along with Ballance's work on interactive syntactic and semantic analysis [10] and Van De Vanter's work on user interface issues [89], it supported research on interactive pretty printing. The "Pan Program Presenter," developed by Christina Black [13], supported variable fonts, including proportional spacing and such attributes as bold, italic, and color. The user provided what were in effect unparsing specifications, constrained to traverse the syntax tree in preorder, and thus to emit the text in its original order. Format was controlled by obligatory linebreaks and indentation; line-breaking was not provided.

7.4 Conventional Style Mechanisms

In commercial word-processing systems, such as Microsoft Word [61] and Adobe FrameMaker [2], there is little separation between style and content. The author creates document text (and other media content), structure, and appearance all at once, working on the same document. A style mechanism is provided, in which names associated with regions of the document are mapped to sets of formatting parameters. This makes it easier to set a document's appearance and keep it consistent, both with itself and with predefined styles. But the names are not intrinsically related to the document's structure; the document is essentially a stream of text, and the set of names is arbitrarily extensible. Further, a document's set of styles is generally treated as part of its content. Thus it is difficult to apply different styles to a single document, and to use the style mechanism to transform document content.

A similar problem has arisen on the World-Wide Web, where HTML [12, 73] has become a *de facto* document and presentation standard. HTML is derived from SGML [41], a structured-document standard; but in HTML the structuring syntax is used to specify appearance directly. For example, "heading" tags are provided, as for headings of sections of text. But the notion of "section" is not actually supported by the language: any text anywhere may be marked with a heading tag. In practice, the tag actually means that the marked text should be displayed in a large font. As a result, it is difficult to discover the logical structure of HTML documents, or to transform it, and impossible to use logical structure to define appearance.

One current proposal for an HTML style mechanism is "cascading style sheets," developed by Lie and Bos and recently recommended by the W3 Consortium [57]. Styles in this format preserve the structure of HTML documents, augmenting them with such character attributes as fonts, size and color. Most of the contribution of the design is in negotiating the priorities of style information from various sources—entries in the original document, created by the author, and "cascades" of subsequent refinement. (See also the online bibliography of designs, proposals, and discussions of Web style sheet mechanisms, maintained by the W3 Consortium [91].)

7.5 Presentation by Tree Attribution

An important family of presentation systems for structured documents works by an attribution of the input tree, possibly modified by limited elaboration and elision. The principal examples of this family are Grif (now Thot) [71, 72], developed by Vincent Quint and others at INRIA in France, and Proteus [42, 65], developed by Ethan Munson in the Ensemble project at Berkeley.

These may be considered as transforming presentation systems, in which the transformation is severely constrained: any material that appears in both the input and the output must appear in the same order and with the same ancestor-descendant relationships. To enable flexible layout—in particular, *non-local* layout, in our sense (items near one another on screen may be far apart in the structure, and vice versa)—these systems provide a complex "navigational" syntax to specify remote attribute dependencies. For example,

```
BODY    { top = Parent.Left.bottom + 10; }
```

places the BODY node under the left sibling of its parent.

This notation is similar to an attribute grammar; in fact, since the form of the attribution rules is not constrained by the productions of the grammar, it is at least as powerful. Further, the rules are "interpreted" dynamically; the techniques for generating efficient evaluators by offline analysis, developed in the attribute-grammar literature, are not used. As a result, the system can make only weak guarantees about evaluation time. If the user types a character, the ensuing changes may propagate through the entire document, taking time linear in document size. Mittal's SHILPĒ [62] is an efficient reimplementation of Proteus, adding lazy and incremental evaluation. It avoids unnecessary reattribution, but it does not change the theoretical complexity, which is intrinsic to the model.

Some problems in document formatting are intrinsically hard to solve incrementally. In pagination, for example, Chamberlin showed that page numbering can require several passes to converge, and that it is possible to construct examples that do not converge at all [21]. But Grif and Proteus make even simple problems more complex than they should be. In Proteus, for example, the x position of a node is computed in absolute terms; in a vertical layout, a single change necessitates reevaluation through the rest of the document.

The flexibility of the attribute rules in these systems also makes it easy for the user to write attribute specifications that are circular or inconsistent. Such errors are easy to detect at runtime, so they need not affect performance; but they are hard for the user to diagnose.

Despite these problems, it should be admitted that there is at least one presentation problem which is a good deal easier to handle in tree-attribution systems than in our transformation model. This is the "curly-bracket problem," discussed in Section 5.1.3. Consider the problem in the context of the same Java formatting examples. To achieve the look of Figure 5.3, with the opening curly-bracket at the end of the "line", the rule for placing it would be

```
"{"     {
          bottom = parent.leftsib.bottom;
          left = parent.leftsib.right;
        }
```

while to place it as in Figure 5.5, with the opening curly-bracket on a "line" of its own, the rule would be

```
"{"     {
          top = parent.leftsib.bottom;
          left = parent.leftsib.left;
        }
```

and no other rule would need to change. The fields of the `classBlock`, for example, could be placed with

```
fieldList {
          top = leftsib.bottom;
          left = parent.parent.left + 36;
        }
```

for the correct result in either style. This works because curly-brackets, in most of the languages that support them, are used only for delimiting such "blocks". This constraint enables the user to make useful assumptions about the structure surrounding the node.

7.6 Document Transformation Systems

Several systems have applied tree transformations to the manipulation of structured documents. From the user's point of view, this can mean conversion between documents of different types, such as extracting the titles and abstracts of a series of articles to make a "digest". Or it can mean conversion between essentially equivalent types, such as two house formats for software manuals.

One such system is Scrimshaw, developed by Dennis Arnon [8]. It is designed to be used like the UNIX text-stream tools (`awk`, `sort`, etc.), as a command-line-driven filter acting on a stream of trees. Feng and Wakayama propose a system called Simon [36], in which transformations are represented by higher-order attribute grammars.

Both Scrimshaw and Simon provide features that go beyond structurally specified transformations. These include content-based queries, such as textual searches. Simon also provides attribution mechanisms for such tasks as resolving references. We discuss adding such services to Ensemble in Section 8.2.

7.7 Ppml

The most direct inspiration for our work has been Ppml, a pretty printer first developed by Morcos-Chounet and Conchon [64] and later integrated into the Centaur system at INRIA [14, 45]. Its basic mechanism is, like ours, a tree translation system driven by rules with tree patterns and variables. However, it works only with a single "medium," text with box layout, and while it is used interactively, little has been published on the details of its implementation.

A closer examination of the layout model of Ppml shows that, though it uses a hierarchical notation suggesting nested boxes, the real formatting model is closer to that of a conventional pretty-printer. The output tree is unparsed into a stream of tokens and whitespace separators. The nonterminals in the tree—V, H, etc.—specify the default separator for the subtree below them. The resulting stream is passed into a line buffer similar to the one used in Oppen's algorithm [66].

This formatting model has the great advantage of permitting good "structural" line-breaking. As we saw in Section 5.1.5, this remains a problem for our own pretty-printer medium. For ordinary pretty-printing, in which the output largely preserves left-right top-down textual order, and can be decomposed cleanly into "lines", the stream model is superior.

On the other hand, this model means that the user can't combine the nonterminals freely without unexpected effects. For example, consider this output tree structure, taken from the layout in Figure 1.4:

```
[H [V "To:" "From:" "Subject:"]
   [V "Mike" "Vance" "Paper draft"]]
```

In our pretty-printer, this produces the layout

```
To:       Mike
From:     Vance
Subject:  Paper draft
```

In Ppml, on the other hand, the same output tree structure is first unparsed into the token stream

```
"To:" <V> "From:" <V> "Subject:" <H> "Mike" <V> "Vance" <V>
```

where `<H>` and `<V>` are horizontal and vertical separators. This stream is passed into the line buffer, which emits it in the following arrangement:

```
To:
From:
Subject:  Mike
          Vance
          Paper draft
```

because the separators are treated as glue between neighboring tokens.

A paper by van den Brand and Visser [90] presents a pretty-printing system derived from Ppml, with a few variations. The chief contribution of their paper is in the automatic generation of formatters (style files or schemas, in our terms) from the programming-language grammar. Their ideas would work well in place of the simplistic schema generator described in Section 5.1.2 above.

7.8 DSSSL

SGML (Standard Generalized Markup Language) [41] is an ISO standard for the representation of structured documents. DSSSL (Document Style Semantics and Specification Language) [44] is a presentation model and style language for SGML, also recently adopted as a standard. Implementations of DSSSL are now becoming available: James Clark's Jade [27], a free implementation, is currently in its "second beta version," and a commercial implementation called Seng is forthcoming [29].

DSSSL contains two separate tree transformation passes. The first pass, called the "transformation process," is, in our terms, a document transformation system. Its purpose is to convert the input SGML document to an intermediate form, also expressed as an SGML document, before formatting. The second pass, called the "formatting process," is a transforming presentation system comparable to our contribution in this dissertation. In it, the document is transformed to a tree of *flow objects*, providing textual layout semantics.

To illustrate DSSSL's approach to specifying transformations, we take an example of the transformation process, provided by James Clark [26]. The input is a document whose structure includes HTML-like "definition lists." These lists are represented by DL nonterminals, with alternating children labeled DD (a list item) and DT (the corresponding descriptive text). In extended BNF, the lists can be described with the production

```
DL:      {DD DT}+;
```

The problem is to change the list structure, so that an intermediate item node is placed above each DD, DT pair and below the DL. The resulting structure can be notated

```
DL:      ITEM+;
ITEM:    DD DT;
```

Figure 7.1 shows Clark's solution. The transformation is expressed with rules similar to our own, embedded in Scheme [28], a dialect of Lisp. The rules are defined with the special form `=>`, which takes arguments comparable to the patterns and actions of our rules. The first argument is a *query* indicating the elements to which the transform applies. The second argument is a Scheme expression which is evaluated once for each node matching the query. It calls on a special set of functions to return a "subgrove spec", which is later read as instructions for building a fragment of the output tree. There is an optional third argument, a numerical priority to help disambiguate cases where several rules apply to one node.

The first rule in this example applies to each DT node, and builds a subgrove spec inserting the ITEM above it. The second rule applies to each DD node, and builds a subgrove spec transferring that node to the left, under the ITEM created by the DT rule.

This notation is complex, indirect, and difficult to read. By contrast, in a rewrite system such as Farnum's Dora [35], the same transformation could be expressed with a single rule:

```
;;; Insert item containing each dt, dd pair.

(=> (q-element 'dt)
    (create-sub (origin (current-node))
                (item-dt-subgrove-spec))
    1)

(define (item-dt-subgrove-spec)
  (subgrove-spec class: 'element
                 add: '((gi "item"))
                 label: 'ignore
                 children: (list (copy-current))))

(=> (q-element 'dd)
    (identity-transform-by-ipreced)
    1)

(define (identity-transform-by-ipreced)
  (create-follow (ipreced (current-node))
                 (copy-current)))

(=> (current-root)
    (create-root #f
                 (subgrove-spec subgrove:
                                (sgml-parse-prolog
                                  "dl-out.dtd"))))

(=> (document-instance)
    (default-transform))

(define (document-instance)
  (subgrove (node-property 'document-element (current-root))))
```

Figure 7.1: Expressing a minor structural conversion, adding intermediate ITEM nodes to an HTML-like list structure, in the DSSSL tree transformation sublanguage. Source: James Clark [26].

```
[DL ?pre* ?dd.DD ?dt.DT ?post*]
 -> [DL pre [ITEM dd dt] post];
```

According to the usual semantics of rewrite systems, this rule would be applied repeatedly until the pattern no longer matched, accomplishing the same transformation. In this case, the pattern notation is not only more compact, but also more precise: the transformation is only applied when the DD and DT nodes are found together, under a DL parent. Clark's solution does not perform any such structural checking.

The DSSSL formatting process converts the document tree to a tree of *flow objects*, including line-broken text and layout elements. It uses a syntax similar to the transformation process. Each rule consists of a query, defining the set of nodes to which it applies, and a *construct-expression*, corresponding to our action. The construct-expression is evaluated once for each node returned by the query; it returns a *sosofo*, or "specification of a sequence of flow objects." The construct-expression also directs the transformation of the children of the current node. Often, this is simply done with the function `process-children`, which returns the sequence of sosofos generated from the children, so they can be integrated in order into the current node's sosofo. But it is also possible to process the children out of order, using a variety of functions, such as `process-matching-children`, which takes only the children that match a given pattern. These functions enable the formatting process to reorder material from input to output.

In comparison to Ensemble, the flow objects created the DSSSL formatting process are much more sophisticated than any of the media we have developed. There is no technical reason, though, that they could not be adapted to Ensemble. This would be an interesting challenge, and would help to demonstrate the generality of our transformation model. It would require extending Ensemble's general model of presentation to include pagination, something that has long been discussed.

Flow objects perform "stream" line-breaking, but they do not provide a good solution to the "structural" line-breaking problem described in Section 5.1.5. It is possible to "pipe" the output of a series of flow objects into another flow object, where they can be line-broken. However, this higher flow object will treat the tokens as a uniform flow, ignoring their structure.

We believe our work provides a significantly better language for expressing structural transformations. In DSSSL, the transformation process and formatting process are both unnecessarily obscure. As we saw in discussing Clark's example, patterns with variables can help considerably in clarifying the relationship of input to output.[3]

On the other hand, DSSSL queries, in either transformation pass, can be based on content as well as structure. The Standard Document Query Language (SDQL), in which queries are formulated, provides word search and regular-expression functions. To add such features to Ensemble's transformation language would be an interesting experiment. They are not necessarily appropriate for Ensemble's purposes today: for one thing, supporting such queries or patterns in an interactive environment would require efficient incremental regular-expression processing, a complicated extension to the transformation model. Also, we suspect that such power is required more for document-transformation operations than for presentation. But by combining content queries and a pattern-based transformational language, we believe it would be possible to develop an alternative to DSSSL that is equally powerful but more compact and readable. And when a DSSSL transformation was expressed only as a pattern-based projection, it would be possible for a browser to implement it with the efficient techniques we have demonstrated here.

Finally, DSSSL is difficult to implement efficiently. In particular, content queries and imperative transformation specifications will sometimes lead to linear change-processing time. A somewhat restricted subset, DSSSL Online [15], intended for use in browsers and other interactive applications, retains the transformation model. There are no published results indicating the practical efficiency of Jade, Seng, or any DSSSL Online implementation. By comparison, the purely syntactic transformation model and restricted attribution model of Ensemble's transformation system allow a very efficient implementation, while still supporting useful layout functionality.

7.9 Formatting Mathematical Expressions

Mathematical expressions have a clear hierarchical structure, both logically and graphically. This has been exploited, for instance, in the languages and layout of Kernighan and Cherry's `eqn` [49], and of the math mode in Knuth's TEX [52]. There has also been a long series of interactive tools for editing mathematical expressions, some connected to computer algebra systems and some purely graphical. A survey of such systems and tools,

[3] It would be possible, of course, to write a pattern-matcher in DSSSL, whose Scheme base is quite powerful enough. But this would mean carrying a substantial piece of code around as a part of every DSSSL style file, and would be inefficient compared to a "native" pattern matcher.

up to about 1990, is available in the PhD dissertation of Neil Soiffer [82]. Soiffer's own contribution, MathScribe, is a general interactive front-end to computer algebra systems. Its design bears many interesting resemblances to our own work.

Like the Ensemble transformation tool, MathScribe can be described in two presentation phases: the transformation of an input tree to an output tree, and the layout computation performed over the output tree. The input to MathScribe is a textual mathematics representation, in the output format of a separate computer algebra system. This is parsed, to yield its intrinsic tree structure; then the transformation is applied.

The purpose of the transformation is to convert the expression from the input format to conventional mathematical notation. The input `Times[x, Power[y, -1]]`, for example, might be converted to xy^{-1}, $\frac{x}{y}$, or x/y. Structures that are already in readable form need not be touched. Because the input and output trees are in the same language, and may share structure, MathScribe sensibly uses a rewrite mechanism, rather than a translation, to model the conversion. In Ensemble, where document and presentation are kept distinct, the same computation could be modeled as a translation, at somewhat greater space cost.

The second phase, formatting the rewritten tree, proceeds in a nested-box attribute synthesis like that of our pretty-printer. This would be easy to implement using our attribute mechanisms.

One feature of MathScribe that could not be duplicated in Ensemble today is the sharing of common subexpressions. For some classes of expressions, this can apparently result in considerable space savings. It is possible that Ensemble could benefit from such sharing, but it can incur a considerable time cost. In particular, handling the exceptions to sharing, by a "copy-on-write" policy during user edits, can be laborious.

Interestingly, Soiffer made many of the same engineering decisions we did, though not necessarily for the same reasons. For example, he also separates a child's position (relative to its parent) from the attributes stored with the child itself. In MathScribe, the position is an attribute of a special node interpolated between the parent and the child, and the reason for the separation is sharing—a given node may be the child of many parents. In Ensemble, the position is an attribute of the parent, and the reasons are, first, the general rule that derived attributes must be synthesized, and second, a consideration of change processing.

Soiffer also faced and solved the problems of change processing in MathScribe, in terms of the specific attributes of his mathematics formatting model. They are completely subsumed by the more general methods presented here.

There are two areas in which Soiffer's treatment is more thorough than ours. First, he treats selection carefully. While we have provided for it in the general architecture (Section 2.3), we have worked out only sketchy plans for it in the transformation tool (Section 6.1.4). Second, he discusses the problems of structural line-breaking at some length—though, like us, he does not actually implement it.

Chapter 8

Conclusion

8.1 Our Contribution

In this dissertation, we have presented a general presentation mechanism for an interactive structured-document and programming-language environment. It works by a tree transformation, from the document to a separate tree, with related structure but composed of nodes in a special-purpose layout language. The user, or a designer, controls the appearance of the document by writing the rules of the transformation.

So far, we have implemented three media with this mechanism: a text medium mainly suitable for program presentation; a tool to display the tree structure of a document or a parsed program as a graphical tree; and a graphing tool to plot numerical data. Earlier research [82] shows that a similar mechanism can be used to display and edit mathematical expressions, making a fourth medium. And the mechanism is also flexible in another sense: particularly with text and graphing, the transformation rules enable the designer to specify widely varying appearances for a single document.

The mechanism has been implemented in the Ensemble document environment, using C++. We defined four important criteria of efficiency for an interactive presentation system, and tuned the implementation to meet each one.

Our survey of related systems and literature shows that there has been considerable work on tree transformation and on presentation, but that the application of transformations to presentations is just beginning. Ppml is a pretty-printer; it supports only text, and its layout is somewhat limited by a stream-oriented formatting model, derived from previous pretty-printers and specialized for certain line-breaking tasks. DSSSL is a presentation standard for textual SGML documents; it also supports only text at this point. While its layout model is rich, programming it is extremely complicated, and its interactive performance is unknown.

Thus, we go further than previous work, by showing that the transformation model is suitable to many media, and that it can be made quite efficient in practice.

8.2 Future Work

The most obvious direction in which this work should be extended is in the output media it supports. To begin with, there are two existing systems described in Chapter 7 that can be adapted to the Ensemble transformation model: the rich text model of DSSSL, with line-breaking and column layout, and the mathematical typesetting of systems like MathScribe. But further, as we found in developing the graphing medium (Section 5.3), the transformation model can also be applied to very different layout models. More presentation tools should be investigated, including multimedia applications such as audio and video, to explore the limits and possibilities of the model.

There is also a clear need for a simple object-oriented graphics medium in Ensemble, to let the user make figures by direct manipulation of circles, lines, rectangles, etc. Roy Goldman developed a graphics medium under Proteus [65], but the focus of his work was on developing interesting demonstrations of the Proteus attribution model, not on supplying basic drawing functionality. Object-oriented graphics documents in the tradition of MacDraw [31] have little structure apart from grouping, and this structure is directly manipulated by the user. But some graphical applications, such as drawing figures, exhibit repeated structure and call for a high degree of stylistic consistency. There is a history of figure-drawing languages [50, 97] in the literature; style specification by tree transformation may prove to be useful in this area as well.

Second, there are several problems that we have raised but left unsolved. In particular, the "structural" line-breaking described in Section 5.1.5 has not been implemented in the pretty-printing medium. There are sound technical reasons why it is impossible under the computational constraints of our current attribution scheme. However, the line-breaking feature is an important one, and some way must be found to solve the computational problem without undue violence to the model we have worked out. It is possible, for example, to process interactive typing with a cheap "emergency" update mode, and then to do correct line-breaking when time becomes available. (This applies to stream line-breaking as well. In Section 2.4.1, we discuss making the distinction between emergency and "convalescent" change-processing, at a larger architectural level.) Performance is not the only reason to make interactive update simple. If a paragraph or large expression reformats itself capriciously with every keystroke, this is visually distracting, and makes typing difficult.

Third, writing schemas for the transformation tool is hard. We have already seen some benefits from a very simple automatic tool for generating pretty-printing schemas (Section 5.1.2). And as we noted, van den Brand and Visser [90] developed a more sophisticated tool along the same lines. However, users would benefit from a graphical tool, with tree views of input fragments and the corresponding output, template displays of the graphical effect, and possibly also layout hints.

Fourth, even with an improved mechanism for designing schemas, a more detailed form of control over appearance is also needed. The choice of schemas can be a powerful mechanism, as we think this work has demonstrated, but it only supports display decisions that apply to the whole document. In Section 5.1.6, we discussed the possibility of adjusting presentation in detail using attributes attached directly to document nodes. But this mechanism has not been explored seriously: there is no graphical interface to set user attributes, and only a few test schemas have been written to take account of them. In extending this work, the idea must be given a fair trial. Does the user-attribute mechanism give adequate control? Or is it possible for the user to intervene more directly in the transformation process, perhaps specifying the transformation to be applied at a given node, overriding or selecting among the transformations described in the schema?

And a fifth area of interest for Ensemble is document transformation, in the sense of Section 7.6. Marat Boshernitsan has already implemented syntactic transformations between programming-language documents [16]. The document architecture and dependency web described in Chapter 2 turns out to handle the relationship well: a transformation object depends on the source document, and when notified of a change, correspondingly transforms the target document. A variety of services required for serious document processing, such as indexing and tables of contents, have been modeled as document transformations, using a combination of structural and semantic queries. It will be a challenge to integrate them into the interactive framework of Ensemble.

8.3 Valedictory

We set out to show that a presentation mechanism based on tree transformation could be flexible, powerful, and efficient. The examples in Chapter 5 demonstrated that it can be flexible enough to handle media as diverse as pretty-printing and graph layout, and powerful enough to express very different appearances for the same information. Chapter 6 showed that the implementation can be tuned to a high standard of efficiency. We believe tree transformation is a promising presentation model for many interactive applications.

Bibliography

[1] Absurd. Typodrome. `http://www.absurd.org/absurd/jb/typodrome`, January 1997. Java applet.

[2] Adobe Systems, San Jose, California. *Adobe FrameMaker*, 5.1.2 edition, 1997. Online information at `http://www.adobe.com/prodindex/framemaker/main.html`.

[3] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 22(3):488–501, July 1976.

[4] A. V. Aho and J. D. Ullman. Translations on a context-free grammar. In *ACM Conference on Theory of Computing*, Marina Del Rey, California, May 1969.

[5] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[6] Henk Alblas. Incremental attribute evaluation. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems: international summer school SAGA*, pages 215–233, Prague, Czechoslovakia, 1991. Springer-Verlag. Lecture Notes in Computer Science, vol. 545.

[7] Ken Arnold and James Gosling. *The Java™ Programming Language*. The Java™ Series. Addison-Wesley, Reading, Massachusetts, 1996.

[8] Dennis S. Arnon. Scrimshaw: A language for document queries and transformations. *Electronic Publishing*, 6(4):385–396, December 1993.

[9] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs*. Addison-Wesley, Reading, Massachusetts, 1990.

[10] Robert Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems.* PhD dissertation, University of California, Berkeley, 1989.

[11] Robert A. Ballance, Susan L. Graham, and Michael L. Van de Vanter. The Pan language-based editing system for integrated development environments. In *ACM SIGSOFT Fourth Symposium on Software Development Environments (SDE '90)*, pages 77–93, Irvine, California, 1990.

[12] Tim Berners-Lee and Dan Connolly. HTML 2.0 specification. RFC 1866, available at `http://www.w3.org/pub/WWW/MarkUp/html-spec/`, September 1995.

[13] Christina Black. PPP: The Pan Program Presenter. Master's thesis, University of California, Berkeley, 1990. Available as technical report UCB/CSD 90/589.

[14] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3)*, Boston, Massachusetts, 1988.

[15] Jon Bosak. DSSSL Online application profile. Online at `http://sunsite.unc.edu/pub/sun-info/standards/dsssl/dssslo/do960816.htm`, August 1996.

[16] Marat Boshernitsan. Design and implementation of tree-transformations in Ensemble. Final project, CS 264, University of California, Berkeley. Available online at `http://HTTP.CS.Berkeley.EDU/~maratb/cs264/paper.html`, December 1996.

[17] John Boyland, Charles Farnum, and Susan L. Graham. Attributed transformational code generation for dynamic compilers. In *Proceedings of the International Workshop on Code Generation*, Dagstuhl, Germany, 1992. Springer-Verlag.

[18] John Boyland and Susan L. Graham. Composing tree attributions. In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 375–388. ACM Press, January 1994.

[19] John Tang Boyland. *Descriptional Composition of Compiler Components.* PhD dissertation, University of California, Berkeley, 1996. Available as technical report UCB/CSD 96/916.

[20] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, November 1992.

[21] Donald D. Chamberlin. Document convergence in an interactive formatting system. *IBM Journal of Research and Development*, 31(1):58–72, January 1987.

[22] David R. Chase. An improvement to bottom-up tree pattern matching. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 168–177, New York, January 1987. ACM Press.

[23] Pehong Chen, John Coker, Michael A. Harrison, Jeffrey McCarrell, and Steven Procter. The VorTeX document preparation environment. In J. Désarménien, editor, *Second European Conference on TeX for Scientific Documentation*, pages 45–54, Strasbourg, France, June 1986.

[24] Pehong Chen and Michael A. Harrison. Multiple representation document development. *IEEE Computer*, 21(1):15–31, 1988.

[25] Alonzo Church. *The calculi of lambda-conversion*, volume 6 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, New Jersey, 1941.

[26] James Clark. Miscellaneous DSSSL information. Web site at `http://www.jclark.com/dsssl/`, 1995–1997.

[27] James Clark. Jade. Online information at `http://www.jclark.com/jade/`, 1997.

[28] William Clinger and Jonathan Rees, editors. *Revised⁴ Report on the Algorithmic Language Scheme*. Internet Scheme Repository, 1991. HTML at `http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_toc.html`.

[29] Copernican Solutions, Inc., Minneapolis, MN. *Seng*, 1997. Commercial DSSSL implementation. Information online at `http://www.copsol.com/products/seng/index.html`.

[30] Robert Corbett. `bison` release 1.24. Free Software Foundation, 1992.

[31] Mark Cutter. *MacDraw*. Apple Computer, Cupertino, California, 1.7 edition, 1985.

[32] Brian M. Dennis. EXL: The Ensemble extension language. Master's thesis, University of California, Berkeley, 1995.

[33] Brian M. Dennis, Roy Goldman, Susan L. Graham, Michael A. Harrison, William Maddox, Vance Maverick, Ethan V. Munson, and Tim A. Wagner. A document architecture for integrated software development. Department of Computer Science, University of California, Berkeley. Online at `ftp://ftp.cs.berkeley.edu/ucb/mhgs/maverick/ensemble.ps`, 1995.

[34] EarthWeb. Gamelan: The official directory for Java™. `http://www.gamelan.com/`, 1996–.

[35] Charles Farnum. *Pattern-Based Languages for Prototyping Compiler Optimizers.* PhD dissertation, University of California, Berkeley, December 1990. Available as technical report UCB/CSD 90/608.

[36] An Feng and Toshiro Wakayama. SIMON: A grammar-based transformation system for structured documents. *Electronic Publishing*, 6(4):361–372, December 1993.

[37] R. E. Frankel, S. J. Rosenschein, and S. W. Smoliar. Schenker's theory of tonal music — its explication through computational processes. *International Journal of Man-Machine Studies*, 10(2):121–138, 1978.

[38] Z. Fülöp and S. Vagvölgyi. A complete classification of deterministic root-to-frontier tree transformation classes. *Theoretical Computer Science*, 81(1):1–15, 1991.

[39] Robert Giegerich and Karl Schmal. Code selection techniques: Pattern matching, tree parsing, and inversion of derivors. In Harald Ganzinger, editor, *2nd European Symposium on Programming*, pages 247–268, Nancy, France, March 1988. Springer Verlag. Lecture Notes in Computer Science, vol. 300.

[40] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 231–240, Tucson, Arizona, 1978.

[41] Charles F. Goldfarb. *The SGML handbook.* Oxford University Press, New York, 1990.

[42] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proc. Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 130–138, Tyson's Corner, VA, December 1992. ACM Press.

[43] Christoph M. Hoffman and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[44] ISO Joint Technical Committee ISO/IEC JTC 1 (Information Technology). Information technology — processing languages — document style semantics and specification language (DSSSL). International Standard ISO/IEC 10179:1996(E), 1996. Available online at `ftp://ftp.ornl.gov/pub/sgml/WG8/DSSSL/`, in files `dsssl96b.pdf`, `dsssl96b.ps`, and `dsssl96b.ps.Z`.

[45] Ian Jacobs and Laurence Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA, Sophia-Antipolis, France, July 1992.

[46] Steve Johnson. *Yacc: Yet Another Compiler-Compiler*. Bell Laboratories, Murray Hill, New Jersey, 1978.

[47] William Joy. *vi*. University of California, Berkeley, 1986. UNIX User's Reference Manual, v. 4.3 Berkeley Software Distribution.

[48] S. E. Keller, J. A. Perkins, T. F. Payton, and S. P. Mardinly. Tree transformation techniques and experiences. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 190–201, Montreal, Canada, June 1984. Published as SIGPLAN Notices 19(6).

[49] Brian M. Kernighan and Lorinda L. Cherry. *Typesetting mathematics: user's guide*. Bell Laboratories, 1978.

[50] Brian W. Kernighan. PIC — a language for typesetting graphics. *Software–Practice and Experience*, 12:1–21, 1982.

[51] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[52] Donald E. Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1984.

[53] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, pages 1119–1184, 1981.

[54] Wilf R. LaLonde. Regular right part grammars and their parsers. *Communications of the ACM*, 20(10):731–740, 1977.

[55] Michael E. Lesk and Eric Schmidt. *LEX — Lexical Analyzer Generator*. Bell Laboratories, Murray Hill, New Jersey, 1980.

[56] P. M. Lewis and R. E. Stearns. Syntax directed transduction. In *Seventh IEEE Symposium on Switching and Automata Theory*, pages 21–35, Berkeley, California, October 1966.

[57] Håkon Lie and Bert Bos. Cascading style sheets, level 1. W3C Recommendation, online at `http://www.w3.org/pub/WWW/TR/REC-CSS1`, December 1996.

[58] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. OPTRAN — a language/system for the specification of program transformations: System overview and experiences. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation. 2nd CCHSC Workshop*, pages 52–65, Berlin, East Germany, October 1988. Springer-Verlag. Lecture Notes in Computer Science, vol. 371.

[59] William Maddox. *Incremental Static Semantic Analysis*. PhD dissertation, University of California, Berkeley, 1997. Available as technical report UCB/CSD 97/948.

[60] Vance Maverick and Marat Boshernitsan. "Line-based" program presentation in Ensemble. Technical report, in progress, 1997.

[61] Microsoft, Redmond, Washington. *Word 97*, 1997. Online information at `http://www.microsoft.com/word`.

[62] Alok Mittal. SHILPĒ: A presentation system for Ensemble. Master's thesis, University of California, Berkeley, 1995.

[63] Sven Moen. Drawing dynamic trees. *IEEE Software*, 7(4):21–28, July 1990.

[64] Elham Morcos-Chounet and Alain Conchon. PPML: A general formalism to specify pretty-printing. In *Proceedings of IFIP Congress*, pages 583–590, Dublin, Ireland, 1986. Elsevier North-Holland.

[65] Ethan V. Munson. *The Proteus Presentation System*. PhD dissertation, University of California, Berkeley, December 1994. Available as technical report UCB/CSD 94/833.

[66] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[67] John L. Pasalis. Realize: An interactive graphical data structure presentation system. Master's thesis, University of California, Berkeley, December 1992.

[68] Eduardo Pelegrí-Llopart. *Rewrite Systems, Pattern Matching, and Code Generation*. PhD dissertation, University of California, Berkeley, 1988.

[69] Dave Presotto and William Joy. *vgrind*. University of California, Berkeley, 1986. UNIX User's Reference Manual, v. 4.3 Berkeley Software Distribution.

[70] Pure Atria, Sunnyvale, California. *Quantify*, 2.1 edition, 1996.

[71] Vincent Quint. The languages of Grif. Available online at `ftp://ftp.imag.fr/pub/OPERA/doc/languages.ps.gz`, 1994. Translated by Ethan Munson.

[72] Vincent Quint and Irène Vatton. Grif: an interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation, Proceedings of the International Conference*, pages 200–213, Nottingham, April 1986. British Computer Society, Cambridge University Press.

[73] Dave Raggett. HTML 3.2 reference specification. World-Wide Web Consortium (W3C) Recommendation, online at `http://www.w3.org/pub/WWW/TR/REC-html32.html`, January 1997.

[74] Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey Ullman. Logical query optimization by proof-tree transformation. *Journal of Computer and System Sciences*, 47(1):222–248, 1993.

[75] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.

[76] William C. Rounds. *Trees, Transducers, and Transformations*. PhD dissertation, Stanford University, 1968.

[77] William C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4(3):257–287, 1970.

[78] Lisa F. Rubin. Syntax-directed prettyprinting — a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, 9(2):119–127, 1983.

[79] Heinrich Schenker. *Free Composition.* Longman, New York, 1979. Translation of *Der Freie Satz* (2nd ed., 1956).

[80] William Shakespeare. Collected works. World-Wide Web, address `http://www.gh.cs.su.oz.au/~matty/Shakespeare/Shakespeare.html`, 1997. The Project Gutenberg edition, reorganized by Matty Farrow.

[81] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, Reading, Massachusetts, second edition, 1992.

[82] Neil Soiffer. *The Design of a User Interface for Computer Algebra Systems.* PhD dissertation, University of California, Berkeley, April 1991.

[83] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, Massachusetts, sixth edition, March 1987.

[84] Guy L. Steele. *Common Lisp: The Language.* Digital Press, Bedford, Massachusetts, second edition, 1990.

[85] Jean-Marc Steyaert and Philippe Flajolet. Patterns and pattern-matching in trees: An analysis. *Information and Control*, 58:19–58, 1983.

[86] Doaitse Swierstra and Harald Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems: international summer school SAGA*, pages 256–296, Prague, Czechoslovakia, 1991. Springer-Verlag. Lecture Notes in Computer Science, vol. 545.

[87] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, White Plains, New York, June 1990.

[88] James W. Thatcher. Transformations and translations from the point of view of generalized finite automata theory. In *ACM Conference on Theory of Computing*, Marina Del Rey, California, May 1969.

[89] Michael Van de Vanter. *User Interaction in Language-Based Editing Systems.* PhD dissertation, University of California, Berkeley, 1992.

[90] Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering*, 5(1):1–41, January 1996.

[91] W3C Consortium. Web style sheets. Online at `http://www.w3.org/pub/WWW/Style/`, April 1997.

[92] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD dissertation, University of California, Berkeley, 1997.

[93] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. In *Fifth European Software Engineering Conference (ESEC'95)*, pages 205–218, Sitges, Spain, September 1995.

[94] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1996. Submitted.

[95] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997. To appear.

[96] Richard C. Waters. XP: A Common Lisp pretty printing system. AI Memo 1102a, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, March 1989.

[97] Christopher J. Van Wyk. A graphics typesetting language. In *ACM SIGPLAN Symposium on Text Manipulation*, pages 99–107, Portland, Oregon, June 1981.

Appendix A

Nodes and Attributes of the Transformation Media

In this section, we provide full listings of the nodes and user-specified attributes of the three transformation media.

A.1 Pretty-Printing

The chief nonterminals of the pretty-printing medium are as follows:

V Lay out children in vertical mode.

H Lay out children in horizontal mode.

HV Lay out children in horizontal mode. If they exceed the margins, break them into rows, and arrange the rows vertically.

HOV Lay out children in horizontal mode. If they exceed the margins, lay them out in vertical mode instead.

The terminals:

TEXT A text node. Usually generated by conversion from a text token in the original document, or from a literal string in the schema. Does not provide line-breaking.

I (for "indent"): Horizontal spacing, an invisible item with programmable width and zero height.

Name	Type	Default
`font`	String	"courier"
`size`	Integer	12
`italic`	Boolean	**false**
`bold`	Boolean	**false**
`fg`	String	"black"
`bg`	String	"white"
`vert_spacing`	Enumeration	`gap`
`vert_space`	Integer	0
`vert_just`	Enumeration	`bottom`
`horiz_spacing`	Enumeration	`gap`
`horiz_space`	Integer	0
`horiz_just`	Enumeration	`left`
`indent`	Integer	36
`linefeed`	Integer	12
`box_margin`	Integer	3
`box_weight`	Integer	1

Figure A.1: User-defined attributes of the pretty-printing medium.

L (for "linefeed"): Vertical spacing, an invisible item with zero width and programmable height.

ELISION An ellipsis marker to indicate elision of document content.

LINK A wrapper for a sub-presentation. Created only by conversion from a link in the original document.

Finally, a special nonterminal **BOX** wraps its children in a rectangular box.

User-Defined Attributes

The user-defined attributes of the pretty-printing medium may be divided into several groups. First is the set of ordinary text attributes: `font`, `size`, `italic`, `bold`, and the color settings `fg` and `bg`. Each TEXT node is formatted according to the single set of attributes that apply to it — there is no finer-grained formatting, e.g. highlighted ranges.

Another group is devoted to spacing control. `vert_space` sets the space value for vertical formatting, and `horiz_space` for horizontal spacing. `vert_spacing` takes the values `gap`, `top`, `center`, and `bottom`. `horiz_spacing` takes the values `gap`, `left`, `center`, `right`.

Name	Type	Default
left_margin	Integer	0
right_margin	Integer	200
window_bg	String	"white"
abstraction	Enumeration	(see text)

Figure A.2: Global variables of the pretty-printing medium.

Justification is controlled by horiz_just, an enumeration with the values left, center, and right; and vert_just, with the values top, center, and bottom.

The I (indent) and L (linefeed) nodes, creating blank horizontal and vertical space, take their default values from the attributes indent and linefeed respectively. These may be overridden locally by the explicit positional argument to the nodes.

Finally, the weight and spacing of the BOX node are governed by the attributes box_margin, the space in pixels between the inside edge of the box and its contents, and box_weight, the thickness in pixels of the line used to draw the box. The space between the outside edge of the box and other nodes is controlled by other means, such as the spacing controls of the vertical and horizontal formatting modes.

Global Variables

Figure A.2 gives the global variables of the pretty-printing medium. The left and right margins are set uniformly, in pixels, for the entire presentation. Similarly, there is a uniform window background color; this is global, as distinct from the inherited attribute bg, which applies only to the small rectangle immediately behind a node's text.

The fourth global variable, abstraction, is more complicated. As we noted in Chapter 2, an abstraction is a client's filter for looking at a document tree: this variable sets the abstraction through which the transformation client sees the document. Thus, the user controls how much of the fine detail of the document tree is actually used as input to the transformation process. The possible values are

raw_tree No filtering at all: expose all nodes, including the binary structure that implements sequences.

tty_prog Conceal structure nodes, but expose whitespace and comments. (So called because this abstraction retains all the characters that would be displayed in a tty-style

Name	Type	Default
`line_fg`	String	"black"
`change_fg`	String	"red"
`path_fg`	String	"blue"
`change_path_fg`	String	"purple"
`new_fg`	String	"green"
`window_bg`	String	"white"
`vert_space`	Integer	24
`show_text`	Boolean	**false**
`abstraction`	Enumeration	(see text)
`change_ref`	Enumeration	`previous` (see text)

Figure A.3: Global variables of the tree medium.

presentation of the text.)

struct_prog Filter out whitespace, but not comments.

bare_prog Filter out both comments and whitespace: the structure will conform to the nominal grammar.

text_doc Suitable for natural-language documents: leave out sequence nodes and the "ultra-root", "beginning of string", and "end-of-string" nodes used in program parsing.

If no value is specified, then the default abstraction is used: `text_doc` for natural-language documents, `tty_prog` for programs.

A.2 Tree Display

Figure A.3 gives the complete list of global variables available in the tree medium. Many of them control the way the medium shows the change state of the tree: whether nodes are new, changed, or on the path to changes. This is reflected in the color of the line between the node and its parent, so the user can quickly follow the path from the root to a change site. Apart from the actual color values used, the designer can change the "reference" version from which changes are computed. By default, this variable (`change_ref`) has the value `previous`, meaning that the reference is always the previous version. Previous in time, that is: the value `parent` means that the reference is the previous version in the document's edit

Name	Type	Default
`LABEL_FONT`	String	"courier"
`LABEL_SIZE`	Integer	12
`LABEL_BOLD`	Boolean	**false**
`LABEL_ITALIC`	Boolean	**false**
`TEXT_FONT`	String	"times"
`TEXT_SIZE`	Integer	12
`TEXT_BOLD`	Boolean	**false**
`TEXT_ITALIC`	Boolean	**false**

Figure A.4: User-defined attributes of the tree medium.

history. This may be different if changes are undone; `parent` means the change reference is the older version from which the current version was created, and `previous` means it is the version last visited, whether older or newer. And the value `synch` means the reference is the last version "synchronized", i.e. brought into consistency by parsing. With this value, changes will accumulate until the user parses the document, at which point they will be reset.

The Boolean global variable `show_text` governs whether the text content of terminal nodes is displayed. If it is displayed, it is editable with a text cursor. `vert_space` is the vertical spacing, in pixels, between successive levels of nodes in the tree.

The only inherited attributes are those governing the presentation of node labels and terminal text, listed in Figure A.4. Using these, a designer can alter the appearance of nodes of given type or structural context (though it must be said the demand for this feature is low).

A.3 Graphs

The node types provided by the graph medium are as follows. Each relies heavily on the named-argument mechanism for control. This is the way data from the input document influences the appearance of the result: data nodes are used as values for these arguments. The data nodes may be raw numbers or alphanumerically encoded.

CIRCLE A circle, optionally filled. Arguments: `radius`, graphical coordinates `graph_x`, `graph_y`, data coordinates `data_x`, `data_y`.

REFERENCE An invisible reference point, used to create lines further up the tree. Arguments: `graph_x`, `graph_y`, `data_x`, `data_y`.

LINE There are two ways to construct a line: either through named arguments or by synthesis from references below. If named arguments are used, the line is a single segment. The arguments are `data_x1`, `data_x2`, or `data_x` to set both at once; `data_y1`, `data_y2`, or `data_y` to set both at once; and similarly for graphical coordinates. If references are used, they are taken in order from the children, and not passed further up the tree. Generates a new graphical line object for the synthesized attribute list; the argument `place`, taking values `above` or `below`, governs whether the line is inserted at the beginning of the list (visually below what follows) or the end (visually above).

TEXT A text item. An alignment point is determined based on the justification attributes in Figure A.5, and this point is placed by the arguments `graph_x`, `graph_y`, `data_x`, `data_y`.

SEQUENCE A sequence of children. If present, the arguments `graph_x`, `graph_y`, `data_x`, `data_y` are used as a translation; thus if a sequence of objects share an x position, the fact can be compactly encoded.

PLOT The master node. Transforms the coordinates of the graphical items collected from its children, and creates axes to display them. It takes a long list of named arguments:

- `x_axis` Placement of the x axis: `bottom`, `top`, or a y data value, or omitted altogether.
- `y_axis` Placement of the y axis: `left`, `right`, or an x data value, or omitted altogether.
- `graph_x`, `graph_y` Graphical position of the plot (upper left-hand corner).
- `graph_height`, `graph_width` Graphical size of the plot.
- `data_min_x`, `data_min_y`, `data_max_x`, `data_max_y` Minima and maxima for display of the data range. If omitted, default to the actual minima and maxima, plus a margin.

The synthesized attributes of a node in the graph medium are simply an ordered list of lines, circles, text, and references. All these have mixed graphical and data coordinates. With exceptions noted above, each node gathers the attributes of its children and passes

Name	Type	Default
`font`	String	"courier"
`size`	Integer	12
`bold`	Boolean	**false**
`italic`	Boolean	**false**
`fg`	String	"black"
`fill`	String	(none)
`thickness`	Integer	
`vert_just`	Enumeration	**bottom**
`horiz_just`	Enumeration	**left**

Figure A.5: User-defined attributes of the graph medium.

the combined set up the tree. The PLOT node converts the data components to graphical values and adds the result to the original graphical coordinates. Further, it implements its own graphical contribution (axes, ticks, and labels) with these same synthesized attributes. The medium's `draw` method simply reads the collected attributes of the root node, whatever that is, and draws them to the screen, ignoring any data components.

Figure A.5 shows the user-defined attributes of the graph medium. The first four are the usual text attributes. Color control consists of a foreground and a fill color (no background). `thickness` is for lines. The positioning of text elements is governed by the two justification variables.

The only global variable is `window_bg`, for the window background color.