2-20-1986

SIGHT - A Tool for Building Multi-Media Structured-Document Interactive Editing and Formatting Applications

Robert F. Gordon Ph.D.
Molloy College, rfgordon@molloy.edu

George B. Leeman Jr

Christian L. Cesar

Mark A. Martin

RC 11720 (#52631) 2/20/86
Computer Science 54 pages

Research Report

SIGHT - A Tool for Building Multi-Media Structured-Document Interactive Editing and Formatting Applications

Christian L Cesar
Robert F Gordon
George B Leeman
Mark A Martin

IBM Thomas J.Watson Research Center
Yorktown Heights, New York 10588

IBM Research Division
San Jose · Yorktown · Zurich

Copies may be requested from:

IBM Thomas J. Watson Research Center
Distribution Services F-11 Stormytown
Post Office Box 218
Yorktown Heights, New York 10598

RC 11720 (#52631) 2/20/86
Computer Science 54 pages

SIGHT - A Tool for Building Multi-Media Structured-Document Interactive Editing and Formatting Applications

Christian L Cesar
Robert F Gordon
George B Leeman
Mark A Martin

IBM Thomas J.Watson Research Center
Yorktown Heights, New York 10588

Abstract:

SIGHT is a tool for building applications that edit and format multi-media structured documents. The media supported include text, line graphics, handwriting, images and audio. These information media are maintained in a single integrated hierarchical database.

The document architecture models documents as trees in which nodes can be shared, i.e., as directed acyclic graphs. For each document there is a logical (or abstract) represention tree and one or more physical (or layout) representation trees. A physical representation is the result of applying the formatter to a logical representation. Both trees are separate but share document content data. The physical representation is displayable and printable, but all editing effectively occurs in the logical representation.

Any number of document types can be supported. A document type is defined by the node types it can contain, by how these node types can be hierarchically organized, by what each node type can contain and by the format specifications used in formatting the document.

SIGHT provides applications a language to define new document types, a Core Editor, various specialized editors and a formatter. The Core Editor is further subdivided into a generic Tree Editor and a generic Node Editor. Both are not limited by document types but are sensitive to them. The Core Editor is the primary editing system. The specialized editors are called upon for media-specific editing and processing.

The most primitive level of SIGHT is an object management system, called PHOVIA, that supports network databases.

CONTENTS

List of Illustrations

1.0 Introduction

Office document preparation software has seen a qualitative improvement in the last decade or so. We have moved from batch formatters (e.g. Unix's NROFF/TROFF [Os76]) to integrated editor/formatters (e.g. Etude [Ha81]) and we are now on the road towards multimedia integration (e.g. TextFax [Ho82]). Concurrently we are seeing an ever growing concern with human factors accompanied by a concerted push towards greater interactivity through faster processors -- starting with the seminal work with advanced workstations at Xerox PARC [Th82] -- and better document rendition at the terminal through higher resolution displays -- as evidenced by the numerous composition workstations that have become available in the last few years (e.g. ViewTech and Texet, both reported in [Se84]). The reader is referred to Meyrowitz *et al.* [Me82] and Furuta *et al.* [Fu82] for a comprehensive survey of editing and formatting systems.

Although interactive systems are becoming more and more prevalent, batch formatters are still being improved and heavily used (e.g. IBM's Document Composition Facility; see [Ib84] for general information on DCF and further references). However, after formatters such as SCRIBE [Re80] showed that separation of the hard-to-write formatting specifications from the document contents greatly simplifies the creation of typeset documents, generic tags have become common place and are on their way to standardization. One of the nice consequences of generic tags is that they make the intrinsically hierarchical nature of documents more apparent despite the fact that tags and contents remain mixed within the flat space of a sequential text file and only the formatter -- and sometimes the user -- is capable of deciphering this hierarchy.

The next evolutionary step in separating format information from contents is the separation of tags from contents. Etude achieves it by making the intrinsic hierarchy of a document physically explicit. The document is no longer a continuous chain of characters -- some contents, some format information -- but a chain of content characters partitioned by a separate tree which contains the tags and associated format information. With Etude the document hierarchy no longer materializes only at formatting time but exists already at editing time. The editor cannot be simply character or line oriented; it must also understand tree structures.

Modeling documents as tree structures proves to have many advantages. In particular it makes integration of multi-mode and multi-media editing easier [Ho82]. The International Organization for Standardization (ISO) put its imprimatur on this kind of model when it was adopted for ISO's proposed Office Document Architecture (ODA) standard [Sm83].

In defining ODA, ISO made a bold move by adopting another important feature of Etude: keeping a separate hierarchical representation for the output of the formatter. Formatters will produce an "output" representation from the source "input" representation, but traditional batch formatters produce a representation that is meant only for previewing or printing and that has little structure besides what the previewing or typesetting equipment needs; moreover the output representation is independent of the input representation. Etude followed a different path. It borrowed the notion of boxes as found in TEX [Kn79] and built an output

representation in the form of a tree whose nodes correspond to the boxes that make the page layout of the document. This more abstract and geometrical output representation lends itself to easier changes and helps achieve some degree of device independence.

Though Etude separates input and output representations, they are related to each other principally to achieve Etude's foremost goal of integrating editing and formatting in lines similar to the work done at Xerox. The hallmark of this integration is letting the user work directly on the output representation of the document while the document is kept permanently formatted, i.e., a combination of the notions of What-You-See-Is-What-You-Get (WYSIWYG) and output editing (two notions that are often confused). The separation of the output and input representations is not a necessity for achieving this integration; however the two representations should be related if performance enhancement techniques such as incremental formatting are to be used. Other editors, such as POLITE [Pr81], have achieved editing and formatting integration by combining the input and output representation into one single representation, but at the cost of an inflexible data structure which makes support for multiple document types difficult.

Support for multiple document types is one of the reasons that underlies Etude's decision to separate the input from the output representation. In traditional batch formatters there is no clear notion of a document type. Indeed, contents and format information are intermixed, and the latter must appear with every single document even when it is of the same "type" as a previous one. Document types begin to materialize when format and other information common to various documents that look the same are extracted out of these documents into a common database and brought into action when a document of that type is to be created. This is what SCRIBE does with formatting specifications and is certainly one of the motivating forces behind the notion of generic tags. A SCRIBE document type is made of the formatting attributes that apply to specific tags in a document.

Etude takes the concept of document type a step further by also including the internal structure of the input representation as part of the document type definition. This is necessary if the editor is to maintain the well-formedness of the input representation tree structure. Given a document type, the shape of the tree and the nodes out of which the tree is built is an integral and important part of this document type definition.

The work we are about to describe is part of the evolutionary trend we have just sketched. Our aim is to build the next generation of editing and formatting systems. We are convinced that one of the critical dimensions of this next generation will be integration. In our view, integration is achieved for two components if the two components become one in the eyes of the end user. So, for example, it is not enough to allow graphics to be merged with text as many commercial systems do. True integration requires that graphics and text be created and manipulated concurrently in the same editing environment, on the same page of the displayed document. This is integration in the strongest sense of the word.

Two-dimensional visual modes -- text, line graphics, handwriting and images -- and audio are our primary target media for this form of strong integration. (Video is not in our immediate plans.) Secondary, but nonetheless important, targets for integration are specialized but basic uses of these media, such as tables, mathematics, spreadsheets and business graphics.

Our aim is an edit/formatting environment where these information media and their specialized uses can be created, edited, related and formatted concurrently in a unified manner. Some have called this "mixed mode editing" when applied to visual modes, "composite editing" and, more boldly, "universal editing".

Our document architecture, software architecture and various other design aspects are considerably influenced by the aforementioned software systems. We do differ however in one major way. Our purpose is to design and implement a core system and a set of tools which can then be used for building a finished system. One motivation behind this comes from the desire to perform user-interface experimentation. Human factors as regards office system software is a growing science, and we would like to benefit from the growing body of knowledge about what constitutes a good user-interface and contribute to this body of knowledge by making possible user-interface experimentation. We hope to achieve this by making our core system and our tools as complete as possible but user-interface independent.

In short, our work does not define what an editor/formatter application will be like for the end user, because we do not define what the user-interface will be. However, to gain insight into the requirements for our core system and tools, our approach has been to work simultaneously on the design of a family of WYSIWYG editor/formatters to be built on top of our system. This has provided us with feedback to help specify the functionality of our system. What we are interested in are the tools that are needed for building interactive, integrated, multi-media applications of which an editor/formatter is one of many, albeit the most important one for the office environment that we envisage. As said, these application-building tools are user-interface independent; they are meant for application writers. They present an interface that makes writing new applications and improving these with time relatively easy. With such tools we expect the application writer to concentrate most of his efforts on designing proper user-interfaces. We have grouped these tools and the core system out of which most, if not all, of our office applications will be built in the future under a single name, SIGHT -- an acronym for Sound, Image, (Line) Graphics, Handwriting and Text.

The principal components of SIGHT are (1) a document architecture -- which is very similar to ODA -- that supports lattice-structured documents and fairly modular document type definitions; (2) a software architecture -- which borrows much from the one used by Etude -- comprised of (a) an editing sub-system capable of editing trees and their contents based on the document type definition active at the time; (b) a formatting sub-system that (i) uses the editing sub-system and the same document type definition for building a tree-structured representation of the formatted document and (ii) is capable of formatting incrementally; (c) a presentation sub-system to help the application writer connect the contents of document trees with presentation devices (typically displays, printers, and speakers for output and keyboards, locators and microphones for input); and (d) an open-ended "specialty" sub-system to allow SIGHT to interface with media-specific programs (e.g., handwriting and voice recognition); and (3) a hierarchical object-oriented database, called PHOVIA, within which all document trees and their contents are created and manipulated and which takes care of memory and file management.

This report is structured according to those three major components. Chapter 2 exposes our document architecture. Chapter 3 gives a brief description of the internal organization

of the core system of SIGHT. Chapter 4 explains how document trees are stored within the PHOVIA database.

2.0 Document Architecture

In this section we explain SIGHT's document architecture, how document types are defined and how the document architecture can be made to pervade the operating system. The document architecture was created with office documents in mind but its generality does not constrain it to the office environment.

Interest in standards for office documents has been growing. These standards are expected to deal with document preparation and transmission. Of particular interest to us are standards that define the internal structure of documents. The International Organization for Standardization (ISO) in cooperation with various other standards bodies such as the Consultative Committee of International Telegraph and Telephone (CCITT) and the European Computer Manufacturers' Association (ECMA) have been involved in setting up a standard for "text structure" [Sm83]. "Text", in ISO's terminology, includes the various forms of digitally encoded information that a document can contain, in particular the various modes of visual media (character text, line art and images) and audio media (digitized sound).

Part of the proposed text structure standard is the Office Document Architecture, which we shall abbreviate to ODA. ODA defines an abstract document model whose central feature is the breaking down of a document into two structures: the logical and the layout. The logical structure reflects the organization of a document from the view of the author and is independent of the final rendition the document will take when displayed, printed or heard. The layout structure reflects the rendition of this document after formatting for some, possibly generic, device. The relationship between the logical and layout structures is given by layout directives that indicate how the first structure is to be mapped into the second. A recent document architecture proposal [Ho84] recommends that these two structures be related hierarchies of objects with neither structure being dominant.

ODA has other features which we shall allude to in the course of this report. ODA is an important reference point because the document architecture that we use is identical in several respects to ODA. On the other hand, our architecture stresses flexibility to an extent that appears to surpass ODA. Because of this and other aspects of our implementation, we have not made an effort to align our terminology too closely to the one used in ODA. We will present the terminological connections in the appropriate sections of this report.

2.1 Documents as tree structures

The inherently hierarchical nature of documents in general suggests that they be represented as tree structures. This choice is reinforced by our opinion, which is shared by others [Ho82,Ki84], that convenient editing of mixed-mode and multi-media documents requires that they have a well-structured internal representation. This same choice was also made by Ilson [Il80] but in the context of supporting an integrated editor/formatter, which is also one of our crucial requirements.

In a tree structured document representation, the root of the tree represents the entire document and deeper levels represent progressively smaller document parts. Though data of

different modes and media are kept isolated from each other deep within the tree, the tree structure allows these different modes and media to be put into close relationship (e.g., anchoring a figure to a word in a paragraph).

This notion of structuring documents as trees is commonly exemplified by listing the parts of a book: it contains chapters, which contain sections, which contain sub-sections, which contain paragraphs, which contain sentences, which contain words, which contain letters. This prototypical deep hierarchy yields trees with eight levels, each level containing only one type of node. Not all documents have such a clean hierarchy, however. A document model should handle any arbitrary complex trees. Levels in the tree may contain many different types of nodes (e.g., mixing paragraphs, figures and voice annotation), certain types of nodes may be found at different levels in the tree (e.g., a paragraph in a chapter section versus a paragraph within a figure) and branches in the tree may have different depth (e.g., chapters may have variable sub-section nesting). In short, document trees do not have to be balanced or homogeneous.

In our model, the data structures that underlie documents are trees. Trees are composed of nodes interconnected by two relationships: parent-child and sibling-sibling. Both are conceptually implemented as forward-backward pointer pairs so that travelling through a document can be done in any direction with equal ease. The parent-child relation is the basic tree structuring mechanism. A node may have any number of child nodes. Normally a node has only one parent node. (Node sharing will be discussed later.) The child nodes of a parent node are organized in the form of an ordered list, whether the order matters or not. Sibling-sibling relations interconnect these child nodes (Figure 1).

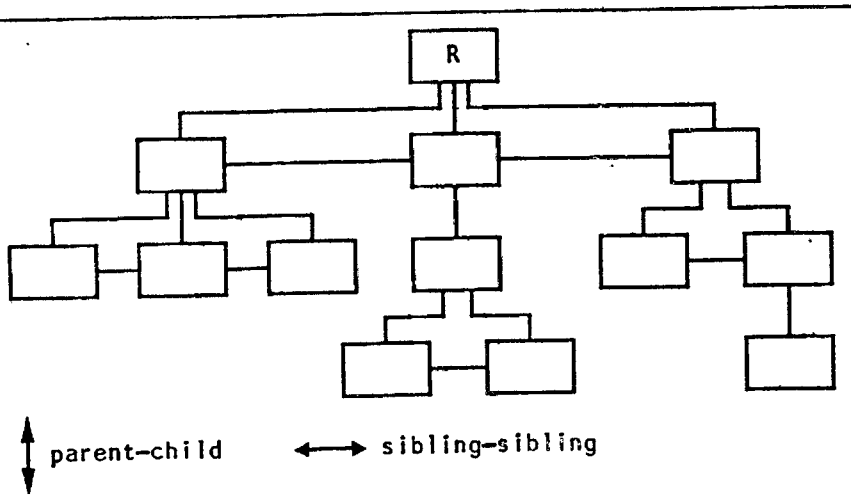

Figure 1. Base document model: The two relationships used for building document trees are the parent-child and sibling-sibling relationships. The first links nodes "hierarchically" and the latter connects nodes of a single parent "horizontally". This example tree shows that there are no connections between nodes of different subtrees except through ancestors that are siblings. In later figures we will not draw the sibling-sibling relationships.

Nodes are the only objects that exist physically. A node contains all the outgoing pointers that serve to define the tree structure -- to wit, pointers to parent, children and siblings -- and all other data. The latter include attributes and raw data. Raw data can be seen as the contents of a document and the attributes as all the properties that define how the document should be handled in terms of editing, formatting and presentation. These properties can take many different forms and serve many different purposes.

The most important attribute of a node is its type. All nodes must have a type. In principle there is no limitation to the number of nodes that can have the same type. How nodes of different types can relate and where they can appear in a document tree is dependent on the type of the document. How document types are defined is discussed later.

2.2 Node sharing

One of the important features of the model is that a node in a document tree can be shared, i.e., a node can have more than one parent. When one or more nodes are shared, the document is no longer a tree but a directed acyclic graph (Figure 2). Nonetheless we will use the term "tree" for both (pure) trees and those that contain shared nodes.

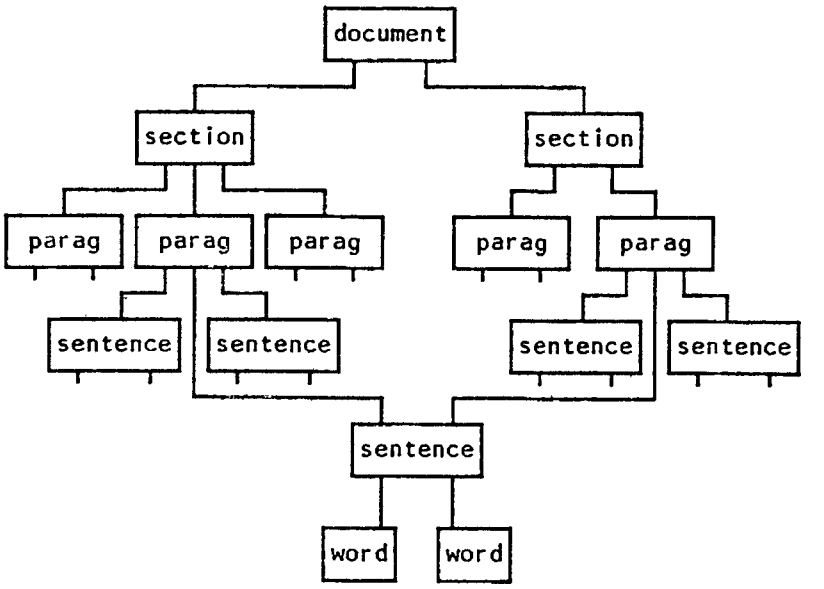

Figure 2. Node sharing: A document tree can contain shared nodes. In this sense the document tree is in effect a directed acyclic graph. In this example, a sentence is shared by two paragraphs.

There are no restrictions on the parents of a shared node. All may be at the same absolute level in the tree, in which case we talk of balanced sharing, or they may be found at different absolute levels, in which case the sharing is unbalanced. Absolute level is a measure of the distance of a node to the root (Figure 3 on page 8).

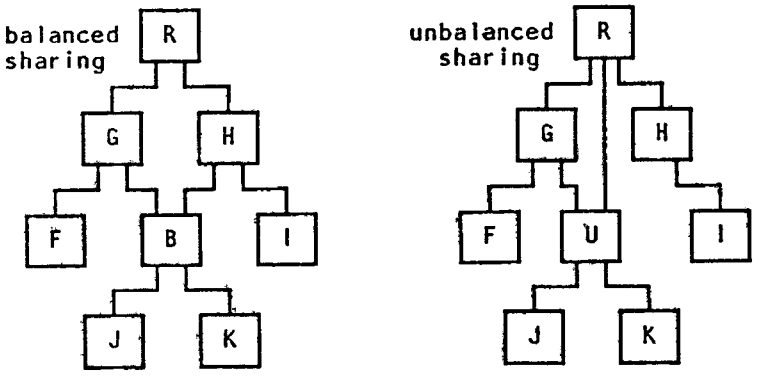

Figure 3. Balanced and unbalanced sharing: Sharing is balanced when the distance from the tree root to any shared node is independent of the path taken; it is unbalanced otherwise. Shared node B is two nodes away from the root whether the path is R-G-B or R-H-B. Shared node U is one or two nodes away from the root depending on the path taken, R-U or R-G-U.

Common uses for sharing are easy to find. Bibliographic references are natural candidates. The point where the reference is made is captured in a node that becomes the parent of the referent. The latter node is the one that carries all the bibliographic information. Many references to the same referent imply a shared referent.

Graphical objects are potential heavy users of sharing. Repeated copies of complex objects will become costly if they have to be replicated. If, in addition, this repetition is nested deeply, the cost could become prohibitive. Sharing is practically a necessity here.

Spreadsheets and their connection to data in other document elements (e.g., a number within some paragraph in the main text) can be implemented much more easily with node sharing. Sharing allows spreadsheet capabilities to spread beyond the confines of the spreadsheet proper. The dichotomy between the document as text and the spreadsheet as a self-contained separate computational environment disappears.

This notion of shared objects was proposed by Kimura and Shaw [Ki84]. Their document model allows, in addition to sharing, for links between nodes to be established independent of the hierarchical structure.

2.3 Document tree semantics

In this section we present two important aspects of document trees: attribute inheritance and the interpretation of node sharing. Attribute inheritance is an old notion. SCRIBE [Re80] implemented it in the form of environment nesting. Pertinent recent work is Andra [Gu84], a documentation preparation system that also uses attribute inheritance to radiate formatting information from shallow nodes (closest to the root) to deeper nodes in the tree. We use the same technique, except that the existence of shared nodes requires a slightly improved form

of attribute inheritance. Moreover, we can apply the technique not only to formatting information but to any information that is defined as inheritable.

2.3.1 Attribute inheritance

The critical difference between data and attributes in a node is that attributes are inheritable while data are not. An attribute value in a node applies to all its descendants but the value may be overriden at any level (Figure 4).

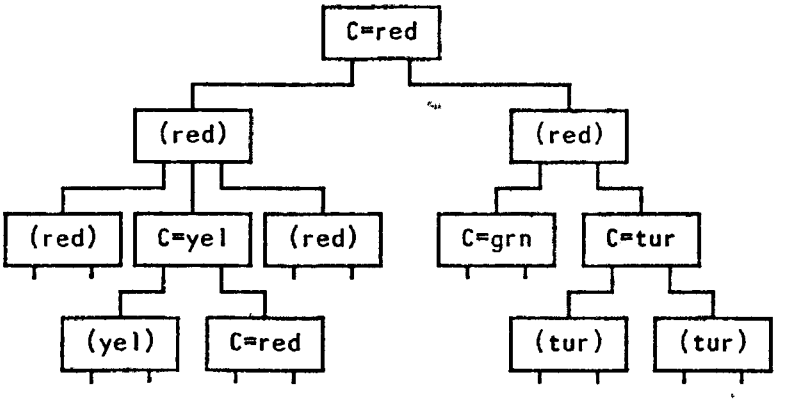

Figure 4. Attribute inheritance: Children inherit from their parents the attribute values that these parents carry or inherit from their own parents. If a node contains an attribute value, it overrides the value, if any, inherited from above. This local overriding value then becomes the new value of the attribute for all descendents of the node in question. In the example tree, a single attribute, C[olor], is shown for nodes that assign a value to it (nodes containing an assignment) and nodes that inherit its value from ancestors (nodes containing the inherited value in parentheses).

Attribute inheritance is desirable in view of the potentially high fan-out of nodes in a document tree (e.g., chapters often contain a large number of paragraphs). It allows attributes that are common to a certain level in a tree and have the same value to be coaslesced and migrated to higher levels. This avoids replicating the same attribute in all the nodes in a deeper level when a single copy of this attribute in a parent or higher ancestor suffices. It is enough then to remember that this attribute stored high in the tree is in effect applying to nodes at deeper levels (Figure 5 on page 10). In the figure, the Par_Width attribute of the *Paragraph* node type (a) is first migrated to the *Chapter* level (b) and then to the root (c). This reduces the amount of storage needed for attributes and their values and does not prevent the attribute from being overwritten at any deeper level (d).

While the inheritance mechanism is unambiguous for pure trees, it is often not so when the tree contains shared nodes. For such trees, static inheritance may be ambiguous for the shared nodes and all their descendants. Disambiguation is achieved through dynamic rather than static inheritance. By dynamic inheritance we mean that when a shared node is reached and attribute inheritance is calculated, the path taken to reach the node from above is

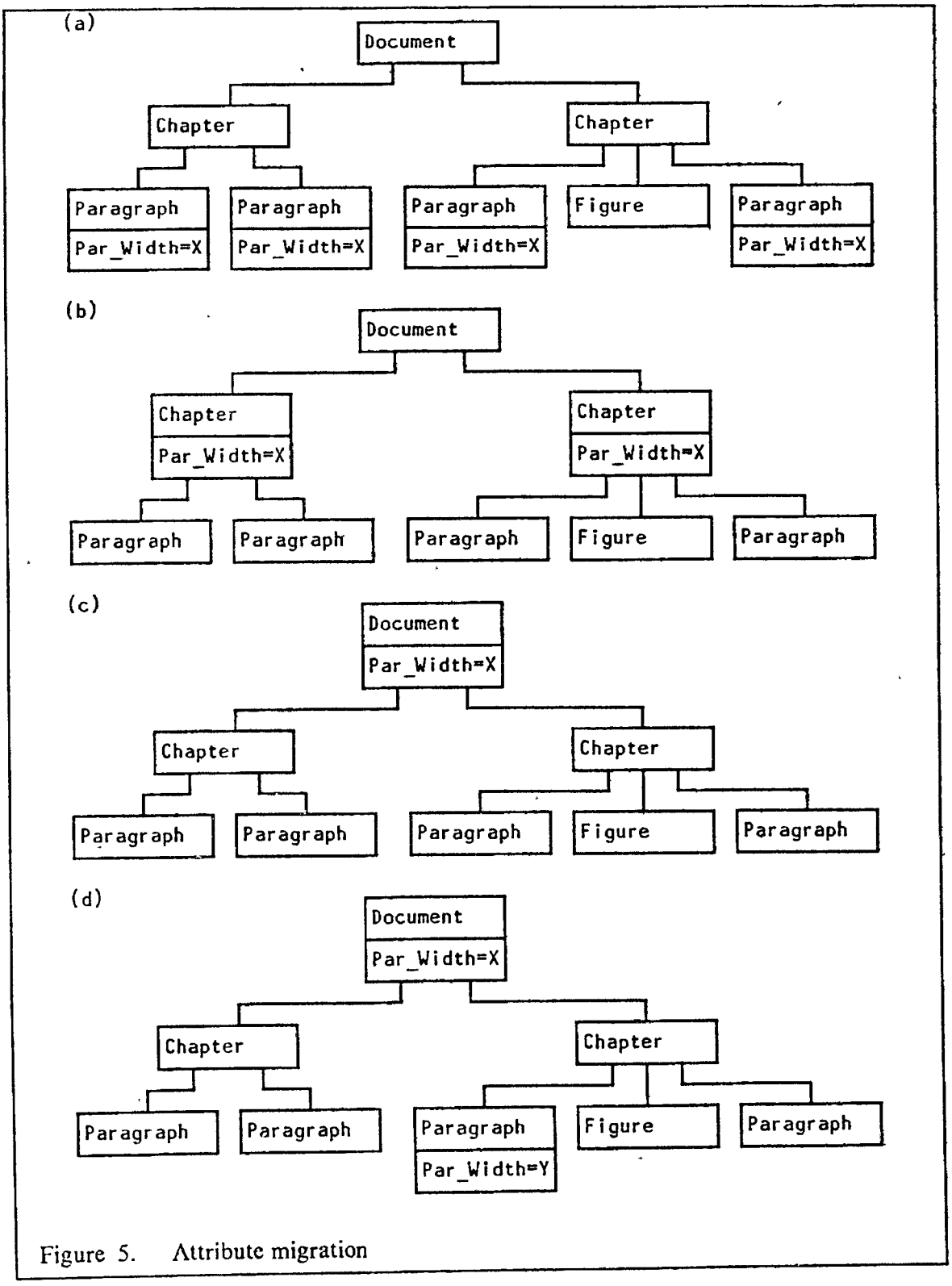

Figure 5. Attribute migration

normally known and it is along this path that attributes are inherited. If the same shared node is reached by a different path, a possibly completely different set of attribute values may apply to it (Figure 6 on page 11).

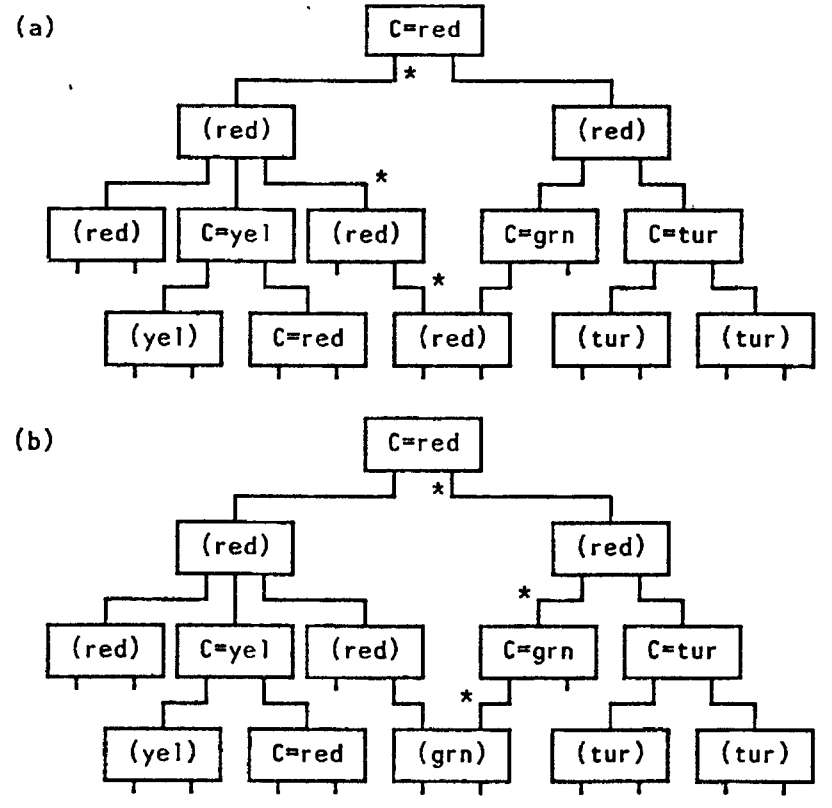

Figure 6. Attribute inheritance for shared nodes: Attribute inheritance is dynamic. This must be so because of node sharing. The attributes and their values inherited by a shared node may depend on the path taken to reach it. The shared node in the center of the example tree will have the C attribute equal to *red* if reached from the left (a) and *grn* if reached from the right (b).

In the rare cases where the path to a shared node can not be determined, inheritance starts from the highest level (i.e. closest to the root) for which the path is known. If the highest level is the shared node itself, it does not inherit anything. This is also true of nodes that are not shared because dynamic inheritance is the only form of inheritance supported. But situations where the path is not known are so rare and always under user control that we can usually think of having static inheritance with dynamic disambiguation at shared nodes.

2.3.2 Conceptual unraveling of shared nodes

The existence of sharing does not change the fundamental interpretation of a document as a pure tree. This interpretation is obtained by conceptually replicating every shared node until there are no more shared nodes (Figure 7 on page 12). The presence of sharing does not introduce new meanings into documents. The reasons for having sharing are, first, it saves space

since only one copy of a shared node need be stored; second, the editing of a shared node or any of its descendants assures that the changes are reflected immediately everywhere the shared node is used; third, it often saves time as a consequence of the previous reason; and fourth, the implementation of sharing as an extension of the parent-child mechanism rather than an ad-hoc ancillary mechanism, such as links, leads to a more uniform specification of the editing operations that can be performed on the document tree.

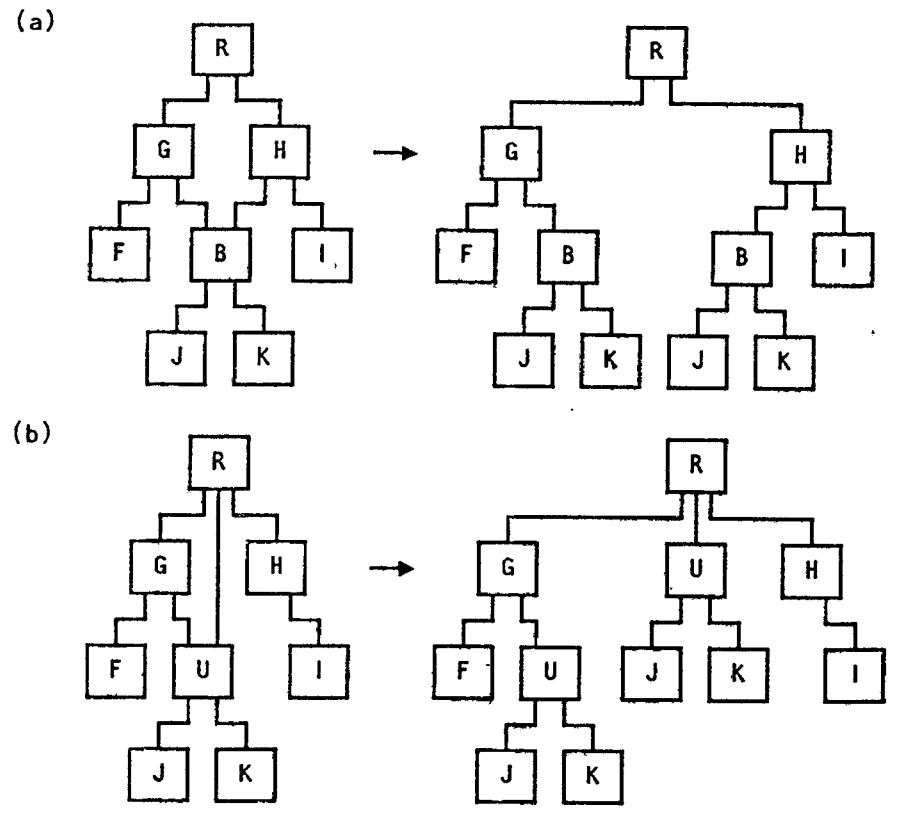

Figure 7. Conceptual unraveling of node sharing: The interpretation of a tree with shared nodes is that of a tree with all shared nodes unraveled. Example (a) unravels a balanced shared tree and example (b) unravels an unbalanced shared tree.

2.4 Document types

It is convenient to have the notion of a document type. In our model, a document type is a collection of descriptions and specifications. There is a logical description and a physical description (both descriptions and the formatting specifications that go with it will be defined more specifically in a later section). The logical description describes the document's logical structure that should exist during content creation and editing. The physical description describes the document's physical structure that should exist after formatting. Fundamentally, both descriptions describe tree structures. In this sense they are equivalent mechanisms with two different purposes. In this section we concentrate on this common mechanism. Any de-

scription, logical or physical, contains (1) a description of how a tree is to be hierarchically organized and what are its component node types, and (2) a description of what each node type can contain in terms of attributes and data. We call these the **tree description** and the **node type descriptions**, respectively. Though the following explanations apply equally to logical and physical descriptions, the examples are all meant for the former.

2.4.1 Describing the tree

2.4.1.1 Unconditional tree descriptions

Describing a document tree is defining what node types it can contain and how these node types can relate to each other. An extended BNF-like language is used for this purpose. Each statement of a tree description refers to a node type. The statement describes for this node type what other node types it can have as children. In other words, a statement's primary task is to describe valid parent-child relationships. For example,

Document → *Chapter*+
Chapter → *Section**

states that a *Document* node can be the parent of *Chapter* nodes and a *Chapter* node can be the parent of *Section* nodes. The language uses regular expression type notation for counting purposes. Thus, the "+" above indicates that the *Document* node may have one or more *Chapter* nodes as children and the "*" that a *Chapter* node must have zero or more *Section* nodes. "*" and "+" are shortcuts for a more comprehensive notation for specifying any arbitrary continuous range of values.

A statement's secondary task is to describe any relevant order between the children of a node. Thus

Section → *Paragraph* (*Paragraph* | *Figure*)*

states that a *Section* node must have at least one child (the oldest) and it must be of type *Paragraph*. Following this *Paragraph* node, there can be any combination of *Paragraph* and *Figure* nodes. The "|" is the usual OR of regular expressions.

Statements can be recursive. Thus

Section → *Section**

states that a *Section* node may have *Section* nodes as children. So defined, this nesting can be carried *ad infinitum*.

A node type may be described by more than one statement, as is the case for the two previous statements describing the *Section* node type. In such cases each statement is taken to mean an alternate description.

As an example, the tree description for the simple document type used in Figure 2 on page 7 could be

*Document → Section**
*Section → Paragraph**
*Paragraph → Sentence**
*Sentence → Word**

The extent to which one has to detail tree descriptions depends on what primitive node types are directly supported. As a minimum, a *Character* node type should be supported for text. The example assumes that a *Word* node type is directly supported. In practice, *Paragraph* node types should also be supported.

A tree description does not say anything about node sharing. Node sharing is always available. It is up to the application to decide when to share and when not to share.

2.4.1.2 Conditional tree descriptions

A tree description can contain conditional statements. A conditional statement is of the form:

```
| condition |  statement
```

If the condition is true, the statement is part of the tree description. If it is false, the statement is ignored.

The condition can be based on the structure of the tree (e.g., how many nodes of a certain type exist in the tree), the kinds of attributes and data that a node contains (e.g., how many authors does the document root contain) and the values of these attributes and data.

When a node type tree is described by more than one statement, be they unconditional or conditionally true, each statement is taken to mean an alternate description.

2.4.2 Describing the nodes

2.4.2.1 Unconditional node type descriptions

The tree description merely uses node type names. What each node type can contain is described by a node type description. There is one such description for each existing node type. A node type description defines what attributes and data can be contained in an instance of this node type. Such attributes and data can be defined as required or as optional. Required attributes and data are always assigned space when a new instance of the node type is created. Optional attributes and data are dynamically allocated as needed. Default values for attributes and data can be defined, in which case they are used for initializing a new instance of the node type (Figure 8 on page 15).

Node_Type:	**Document**
	Author*
	Date[1]
	Size[1]
	Owner[1] = "Worldwide Office Systems"

Figure 8. Unconditional node type description: A node type description contains a list of attributes and data an instance of this node type can contain. In this example the node type *Document* contains four items. The superscripts indicate the number of occurrences allowed for each item. Thus, the *Author* item can appear any number of times while *Date*, *Size* and *Owner* must appear once each. *Owner* has a default value assigned to it.

2.4.2.2 Conditional node type descriptions

Similarly to tree descriptions, node type descriptions can be conditional. Each attribute or datum declared to be part of a node, either required or optional, can be prefixed by a condition. If the condition is true, the attribute or datum is taken to be part of the node type description. If it is false, the attribute or datum is ignored (Figure 9).

Node_Type	**Document**
	Author*
	Date[1]
	Size[1]
|Date < 3/1/85|	Owner[1] = "National Business Computers"
|Date ≥ 3/1/85|	Owner[1] = "Worldwide Office Systems"

Figure 9. Conditional node type description: Items listed in a node type description can be conditionally included by prefixing them with a condition. In this example, *Owner* may have one of two values depending on the value of the *Data* item.

The condition follows the same pattern as the condition for tree description statements. In addition, conditions that relate to attribute and datum values can describe internal or external dependencies. An internal dependency refers to other attributes and data of the node to which the node type description is being applied. An external dependency refers to attributes and data of a different node.

The application of a conditional node type description is dynamic in the sense that for one instance of a node type the condition may not be satisfied while for another instance it may. It is typical then for a node type description that has a conditional item to have more than one condition regulating that item. In such case the conditions are normally mutually exclusive and cover the universe of possibilities so that one of the conditions will define the item.

2.5 The logical and physical documents

While SIGHT is aimed principally for WYSIWYG-type applications where the end-user sees and interacts with a single uniform document object, behind the curtains the application programmer must deal with two forms of document representation. The first form is the logical or abstract representation. The second is the physical or layout representation. The logical representation (LR) can be viewed as carrying the contents and organization of the document as given by the author. The physical representation (PR) is the physical embodiment of these contents for some device that can "play out" the media, be they visual or auditive. In SCRIBE's terminology [Un84], the LR is analogous to the *manuscript* and the PR is analogous to the *[printable] document*.

While the PR expresses what the end-user will see or hear, the LR is where all content data is and almost all content editing takes place. The PR is the result of formatting the LR into a viewable and printable form.

Given that the LR and the PR are separate and that document formatting is a mapping from LR to PR, an inverse mapping from PR to LR must be supported to help the application programmer map user interactions with the PR into actions on the LR. The typical cycle for applications that choose to use this document formatting model is thus (i) apply formatter to LR to create or update PR, (ii) PR seen and/or heard by end-user on output device, (iii) end-user interacting through input device somewhere within visible or audible portion of PR, (iv) input device location and action on PR mapped into location in LR, (v) LR edited and (vi) LR reformatted into PR (Figure 10 on page 17).

Applications are not forced to use this formatting model. An application may choose not to have a PR, preferring instead to combine everything into the LR and working the device drivers directly from and to it. But in so doing such an application foregoes the formatter provided with the system. In short, LR to LR formatting is not directly supported, LR to PR formatting is. Consequently our discussion is limited to the latter.

The existence of two different representations for documents is a notion that has been contemplated for some time. ODA also recognizes the same dichotomy (see [Ho84]). Their terminology is slightly different: our LR is their "logical structure" and our PR is their "layout structure"; but in both cases they are hierarchical. The fact that all content data in ODA must be within leaves (ODA's "basic objects") while in our case content data lacks this restriction is not significant; the two approaches are functionally equivalent.

2.5.1 The logical representation description

As we have seen, a tree representation needs a tree description and node type descriptions. The LR is a tree and therefore a document type includes an LR tree description and LR node type descriptions. A typical LR description will include node types such as *Chapter*, *Paragraph*, *Figure*, *Footnote*, and so on.

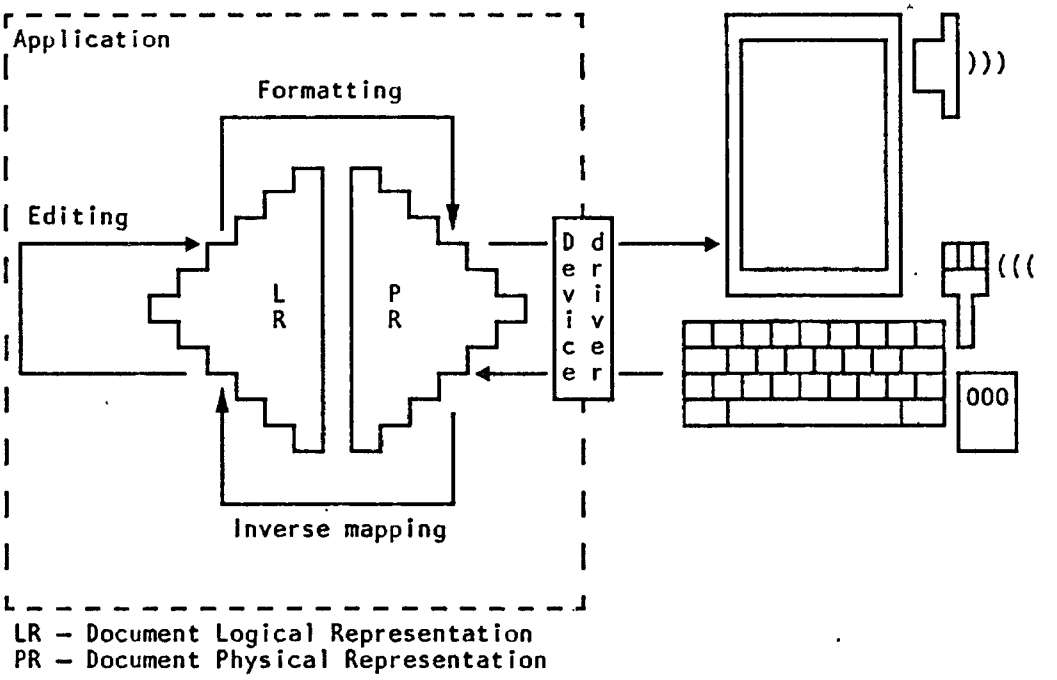

Figure 10. Edit/format cycle: In a multi-media terminal, the end user sees and hears the document and interacts with it through various input devices -- typically, a keyboard, a mouse and a speaker. Editing commands to the application responsible for the document in question are issued relative to the physical representation of the document which defines how it is seen and heard by the end user. The corresponding location in the logical representation is found by a SIGHT-supported inverse mapping. The editing command is then executed in the logical representation. If reformatting is deemed necessary and implicitly requested, the physical representation is appropriately updated. If this cycle is fast enough, the result of the editing command and any consequent reformatting is seen immediately after the editing command is issued. This is what is meant by real-time interactive editing/formatting. The inverse mapping can vary in difficulty. At one extreme there is direct raw data editing which, since they are normally shared by both representations, does not require any inverse mapping except for the purposes of finding where reformatting has to start within the logical representation (this usually entails finding the branch in the LR that leads to the node containing the raw data). At the other extreme there is editing of document parts generated by the formatter from scattered data in the LR. Not shown in the figure is the user-interface that decides how and where on the screen the document visible media are to be displayed and how the document auditive media are to played out through the loudspeaker. The user-interface is considered part of the outside layer of the application and is not shown here since SIGHT is meant to be user-interface independent.

2.5.2 The physical representation description

The PR is a tree and therefore a document type includes a PR tree description and PR node type descriptions. The PR tree description defines the nested box hierarchy which characterizes the desired document layout. Node types such as *Page*, *Column*, *Line* are often included. The PR node type descriptions define the information associated with these boxes. Typically such information includes dimension and positional values and visual properties (e.g. color) of the boxes. The PR node type descriptions often define default values for such information, thus fixing the skeleton layout of the document ahead of time.

The PR tree and node type descriptions delimit the scope of the formatter's work. The PR tree description tells the formatter what should and should not be included (e.g., through the presence or absence of a *Table_of_Contents* node type) and overall aspects of the layout (e.g., through the number of *Column* node_types under the *Page* node type). The PR node type descriptions tell the formatter what geometrical properties must be calculated (e.g. line breaks) and where they should be stored, and which do not have to be calculated unless it is a necessary step in the calculation of another node content data (e.g. word position within a line during line break calculation).

2.5.3 Formatting specifications

The formatter is node driven. Instructions for the formatter -- the formatting specifications -- are tied to nodes. In principle each node is supposed to carry the formatting specifications that the formatter needs to format the node and its descendants. But because formatting specifications tend not to change often among nodes of the same type, it is desirable to externalize such common formatting specifications while leaving the capability of nodes carrying their own private formatting specifications. Thus formatting specifications can be defined either for node types, in which case they are external to the logical representation of the document, or for node instances, in which case they are internal to the logical representation. External formatting specifications for a document type apply to all documents of that type. Internal formatting specifications apply only to the document that contains them. This allows the user to easily override the standard formatting that applies to a document type. (See Figure 11 on page 19.)

External specifications instruct the formatter on how to treat each node type in the logical representation and how to map this node type into some desired node type or arrangement of node types in the physical representation. With the specifications modularized according to node types, it is possible to share node type formatting specifications across document types. Thus, for example, the same formatting specifications for a node of type *Paragraph* can be shared across many document types.

Internal specifications are like external specifications but apply only to a particular instance of a node type, consequently only to the document that contains it. Internal specifications override the values given by the external specifications and define the formatting environment for the node it applies to and all its descendants. Internal specifications are only needed when a node type is to be treated differently than what is externally specified for this node type. These specifications are stored in the node in the form of (formatting) attributes, the same kinds that are used to build the external specifications.

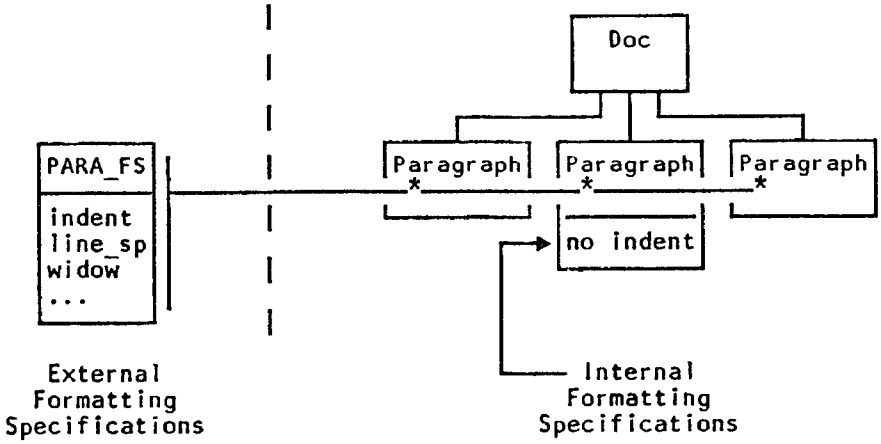

Figure 11. External and internal formatting specifications: The internal override the external for the purpose of local and usually temporary formatting changes. In this example, the external formatting specification tied to nodes of type *Paragraph* define a default indentation style that the author can override locally by redefining the corresponding attribute.

Because, like all other attributes, formatting attributes, internally or externally defined, are inheritable, it is not necessary that the formatting attribute meant to apply to a certain node type be carried by the formatting specification of that node type. It is enough that a higher node type (one that is always found as an ancestor of the node type in question) carry that information. This is not very significant for external formatting specifications since there is no space saving by such relocation of a formatting attribute, but it is useful for an internally defined formatting attribute that applies to all descendants with the stated node type. The principle at work here is to percolate internal formatting attributes towards the document tree root; and if it reaches this root and it is constant across documents, it probably should be externalized.

The inheritance of formatting attributes impacts shared nodes in the expected way. A shared node is potentially subject to different formatting specifications depending on the path used to reach it from the root. We see this as a plus. The same data can be formatted in completely different ways depending on where they appear in a document (Figure 12 on page 20).

2.5.4 Logical and physical document representations

Logical and physical representations are separate in our model. This is in contrast with other editor-formatters where the information about the logical structure of the unformatted document and the physical data that describes the formatted document are mixed together. (For example, POLITE [Pr81].)

The PR is in principle device independent though it is in practice created with a particular device in mind. It is possible to create a PR for an ideal device (e.g., one that has infinite re-

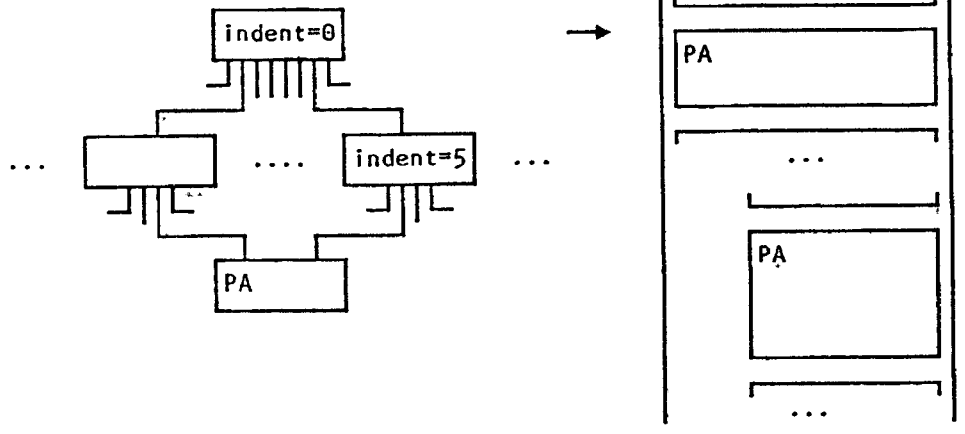

Figure 12. Formatting attribute inheritance for a shared node: A shared node may be subject to different formatting specifications depending on the path used to reach it. In this example, the shared paragraph (PA, in the figure) first appears in the document with an indentation set by an ancestor two levels higher. Its second appearance uses a positive indentation set by an ancestor one level higher.

solution) and let the device driver make the appropriate mapping to a finite resolution device. But the simpler the device driver, the more device specific the PR will be.

The difference between LRs and PRs is in the node types that each contains. A document's LR tree contains nodes that reflect logical components that are relevant to the author. Node types such as *Chapter*, *Section*, *Paragraph*, *Figure*, *Signature* are typical. A document's PR tree contains nodes that reflect the physical properties of the document as a visible and audible object. Node types such as *Page*, *Column*, *Line*, *Heading* and *Footing* are typical. From the editing point of view, there is little other difference between the two representations. They are both trees that can be manipulated in principle arbitrarily through the editor. From the end application's point of view however, there is a crucial difference between the two representations. The creation and editing of a LR is under the application's control while the creation and editing of a PR is normally the responsibility of the formatter. The end-user has only indirect control of the formatting process, a control that the user achieves through the creation and editing of formatting specifications that serve as guides to the formatter.

This however does not preclude the application from overriding the formatter and editing the PR directly. This is possible because the PR is like any other document tree. Of course it is then necessary to reconcile the actions the application may perform on the PR against the actions the formatter performs on this same representation. For the purpose of this paper, we assume that the formatter has exclusive access to PR of documents.

A document's LR and PR could be completely separate trees but for space reasons it is desirable that the deeper levels be shared, in particular raw data. One simple, albeit imprecise, way to visualize this sharing is by assuming the raw data arranged as a linear string that is

partitioned differently by the LR and PR. The LR partitions this string according to logical boundaries, e.g., where paragraphs start and end. The PR partitions it according to layout boundaries, e.g., where the line and page breaks are (Figure 13)

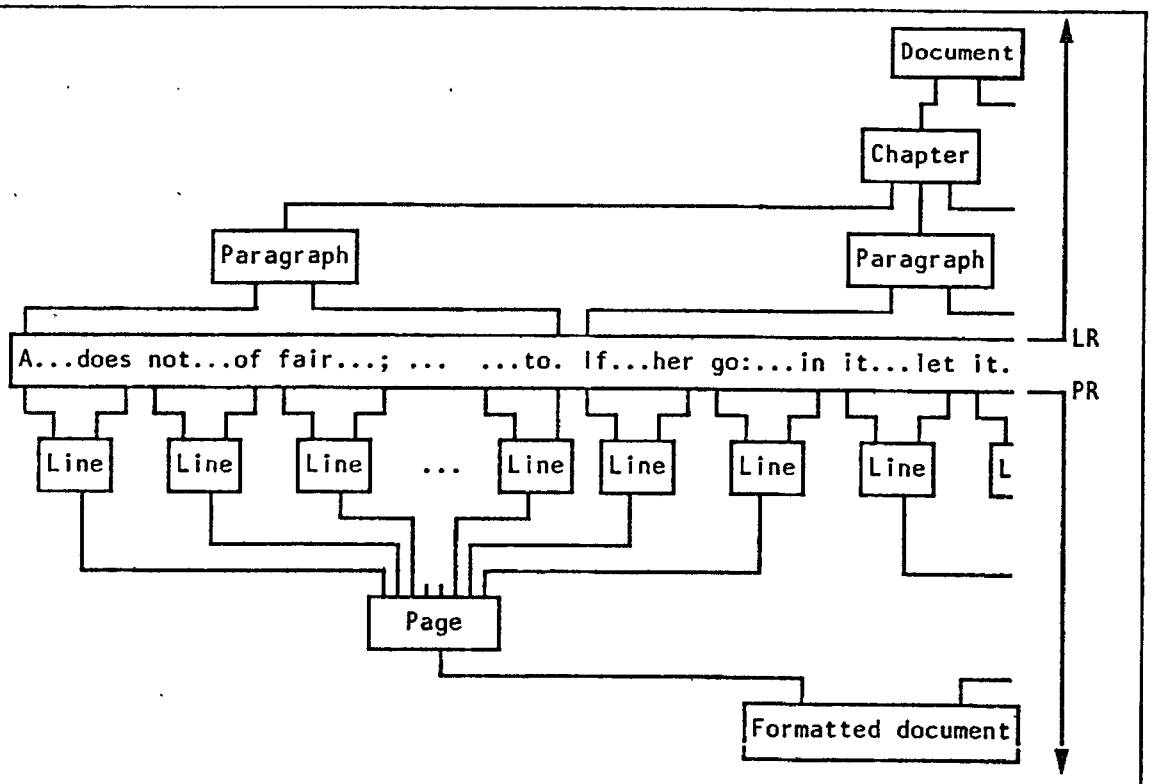

Figure 13. A conceptual view of raw text data sharing between LR and PR: In this simplified view, the LR partitions data according to logical components defined by the author while the PR partitions the same data according to physical components of the layout generated by the formatter.

The PR is often a simpler structure than the LR but, on the other hand, the former may contain additional subtrees that are not partioning the raw data but instead show how major components are organized. This is the case with the various front and back matter pages: table of contents, list of figures and tables, indices, etc. Our view is that these are physical rather than logical components of a document. Of course, the information that they carry is implicit in the LR. *Chapter* and *Section* nodes in the LR are ordered and partially define the table of contents. The same is true of *Figure* and *Table* which define the "list of" pages. These PR subtrees do share nodes with the LR. It should be easy to change the title of a chapter in the LR and have this change reflected immediately in the PR. By sharing the node that contains the text of the title (very likely it will be a node of type *Chapter*), the change in the LR is instantly reflected in the PR (Figure 14 on page 22).

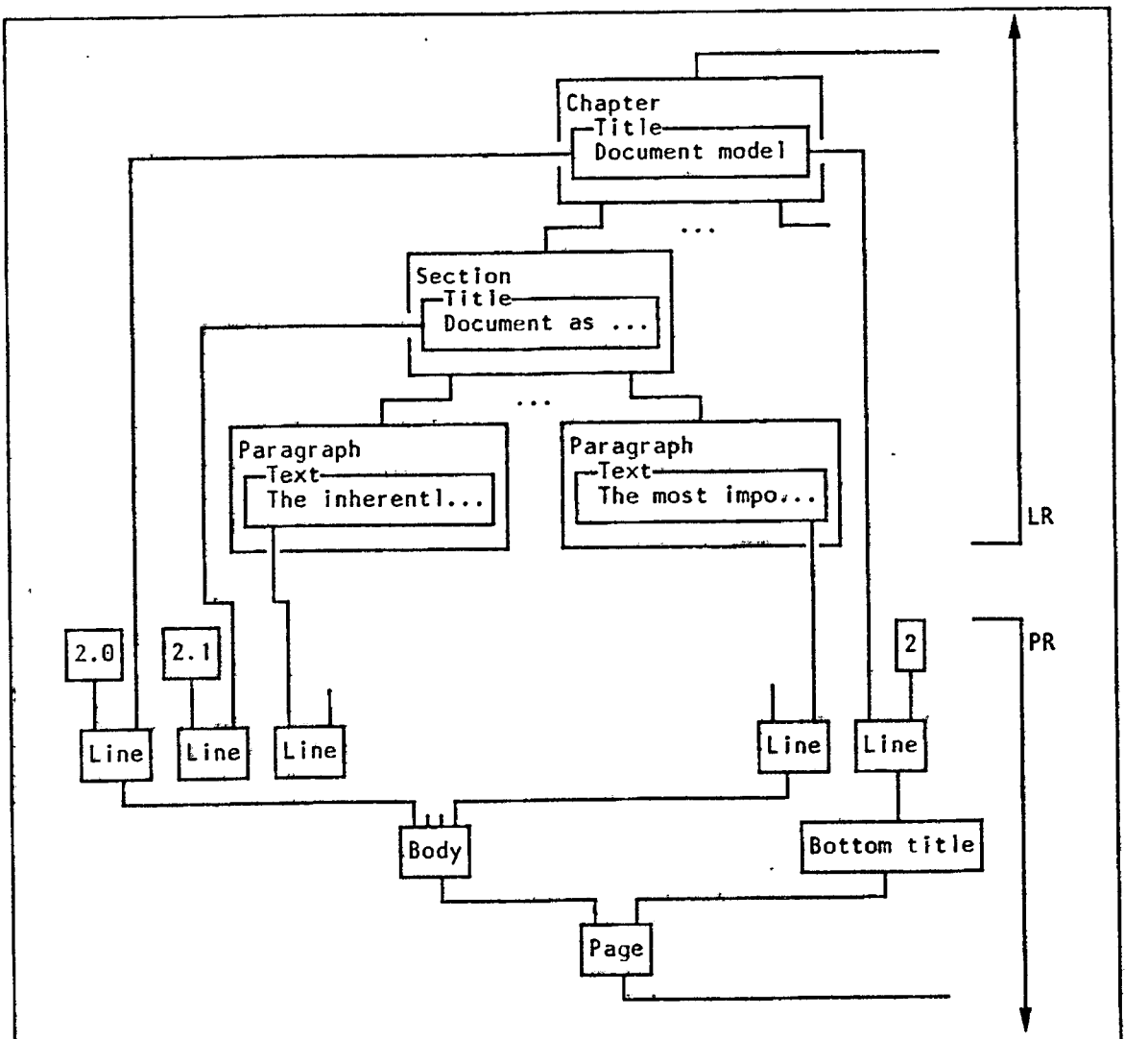

Figure 14. Content sharing between logical and physical representations: The PR not only has access to the data of LR leaves but also to any data in non-leaf LR nodes. In this example the titles of chapters and sections belong to the corresponding nodes. The formatter is instructed to paginate the document, each page laid out with an area for the body of the text and a bottom title that shows the current chapter title. The body in this case is simply a collection of lines. Title lines consist of a formatter generated chapter or section number followed by the title which is in the LR and is shared with the PR. The bottom line also shares the *Chapter* title data and appends to it a formatter generated page number.

2.5.5 Multiple physical representations from one logical representation

The separation of the LR from the PR allows us to have many PRs concurrently active for one LR (Figure 15 on page 23).

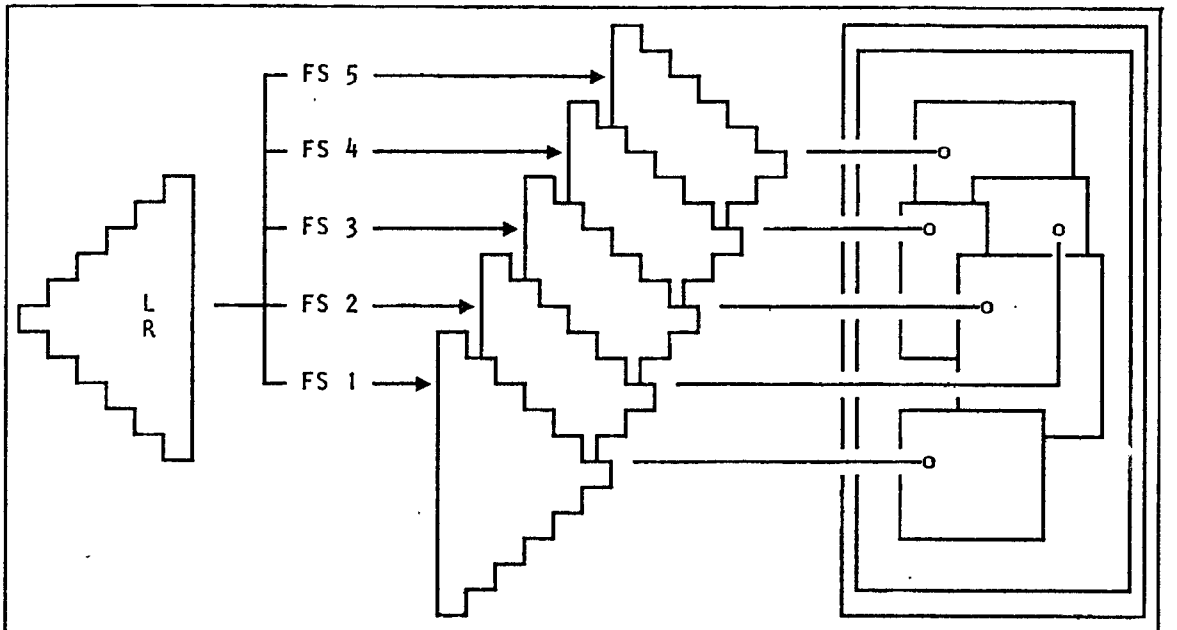

Figure 15. Multiple physical representations from one logical representation: The same document, as expressed by one LR, can be formatted in many different ways depending on the formatting specifications (FS) that are applied to the LR. Running the formatter on the LR for one FS produces one PR. Each PR can then be displayed in its own window on the display screen. It is also possible to display the same PR in many different windows, but this is a user-interface issue that is irrelevant for SIGHT.

The end-user conceptual model of a single representation document model carries over even if there are many PRs of one document. All of them still represent the same underlying logical document so that a change in one PR is immediately reflected in all other PRs because all changes are effectively performed on the LR that all these PRs share. Of course that implies that the end user conceptual model includes this notion of multiple PRs out of one LR. This is necessary if the user is to understand that he can move from one representation to another and that editing one means editing all. This can be conceptualized as applying different formatting specifications to the same document contents.

2.5.5.1 Activating multiple document types concurrently

As we have seen, each document tree, whether it is a LR or PR, requires a tree and node type descriptions. For each LR/PR pair there is also a formatting specification (FS) that describes how the LR is mapped to the PR. Consequently, in the case of multiple PRs from one LR, the LR and each PR has its own tree and node type descriptions. For each PR there is a formatting specification that describes how it is generated (Figure 16 on page 24).

Any of the components that constitute an overall document type -- the LR description, the PR description and the FS mapping from one to the other -- can be shared. The sharing of the

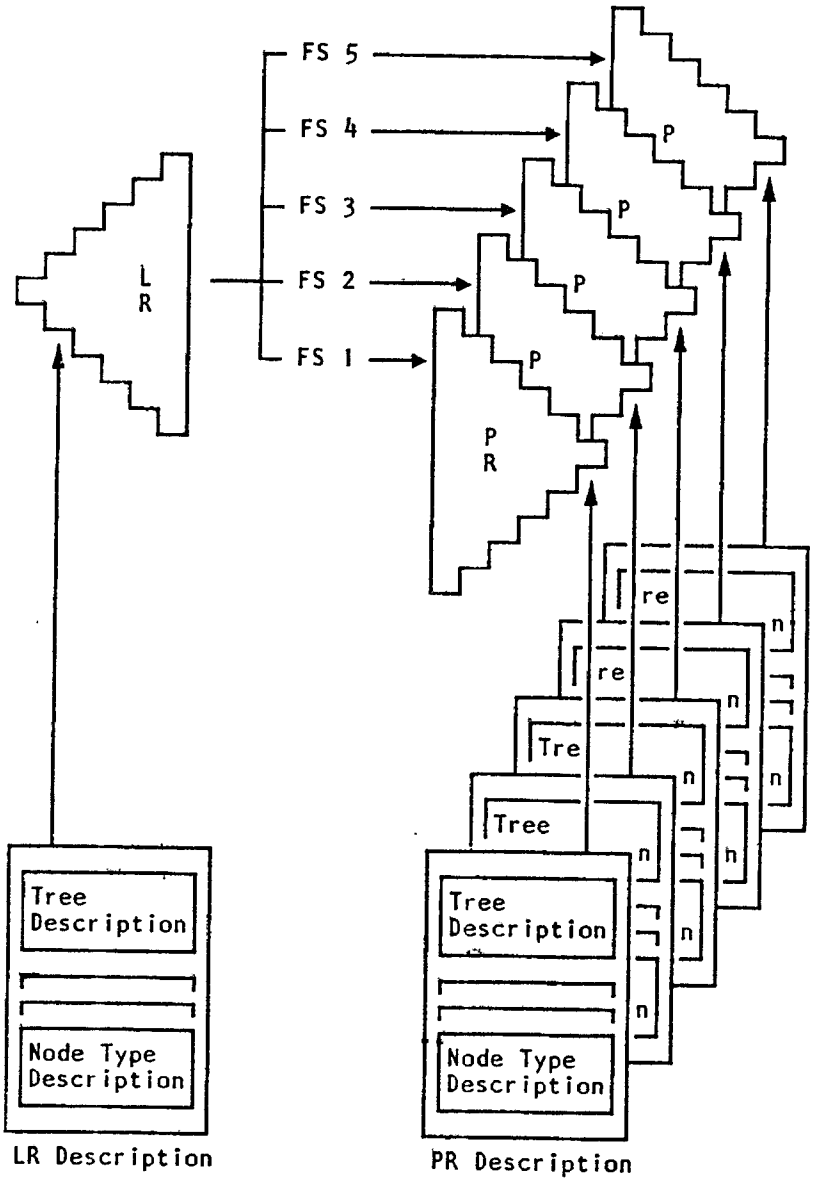

Figure 16. Multiple document types active at the same time: The overall characterization of a document type includes a set of LR descriptions (Tree and Node Types), a set of Formatting Specifications (FS) and a set of PR descriptions (Tree and Node Types). It is possible to share a LR description among several document types. One LR description can be used to maintain a document's logical representation while different FS/PR description pairs are used for different renditions of the document.

LR description has just been described. It is also possible to share a PR description or to share a FS mapping (Figure 17 on page 26). Using different FS's for the same PR description normally occurs when minor stylistic changes are desired, such as different fonts, different

form of highlighting, different paragraph styles, different page numbering, etc. without changing the overall layout structure of the formatted document. Using different PR descriptions for the same FS normally occurs when structural changes are desired without stylistic changes. A PR for a memo is quite different from a PR for a book, for example. One will not have a Table of Contents while the other will. But keeping the same FS for both gives a surface appearance of sameness, a style uniqueness that a publishing department may want to preserve across many different kinds of documents.

2.5.5.2 Physical representations viewed as formatted results of database queries

The LR that is subject to formatting has been selected from a database of documents. In our model, this LR is a subtree of a larger tree that contains all documents. The LR delimits the scope of the editing process. This delimitation process can continue within the LR for the purposes of formatting, that is, only a portion of the LR can be made visible to the formatter when generating a given PR. If multiple PRs are generated from one LR, it is possible for each LR-to-PR formatting process to look at different portions of the LR. These portions may or may not overlap. These formatting processes do not alter the basic structure of the LR nor its data and therefore can run independently without interfering with each other.

For example, a document containing text and a spreadsheet can be set up so that two differing PR's are presented to the user. The first PR presents the document as it will be printed. In this printed form, the author has chosen not to include the entire spreadsheet but only the row that shows totals. The second PR presents only the spreadsheet. The spreadsheet is presented in its entirety so that the author can edit the spreadsheet in the expected way. For the first PR, all the nodes that make up the spreadsheet except for the last row are excluded from the formatting process. For the second PR, all the nodes that make up the text are excluded.

This conceptual decomposition of a LR is normally defined from outside the LR (e.g. we externally define that all nodes of type A in the LR are included in the formatting process while nodes of type B are not). It is also possible to include/exclude nodes from the formatting process conditional to the contents of nodes within the LR (e.g. we externally define that all nodes of type A in the LR are included in the formatting process only if their size attribute contains a value greater than some given threshold).

This capability to select parts of a LR for generating PRs is a form database querying. In this view the LR is the active database and the PRs are the properly formatted results of queries into this database. For example, a LR for a phone book might include various related data such as name, office phone number, alternate phone number, recording machine availability, department number, location and computer logon id. One may be interested in seeing only names and office phone numbers. This query can be expressed through the PR tree and node type descriptions. If the underlying LR is changed, these PR descriptions are then used to check if the change requires reformatting the PR. In short a query in SIGHT is a live PR document that is always kept updated as the underlying database (the LR document) is changed. Document retrieval, editing, formatting and database querying are thus combined into one unified mechanism.

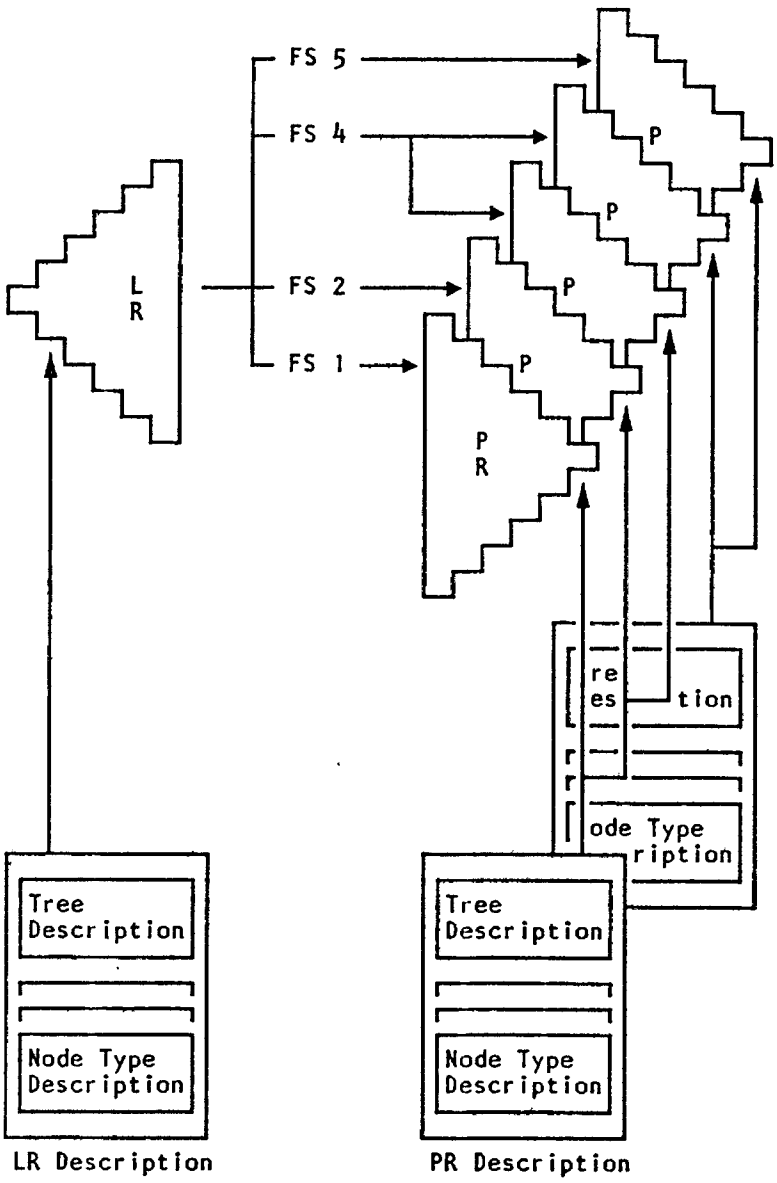

Figure 17. Sharing PR descriptions and formatting specifications: The FS and PR description of a document type are fairly independent and it is possible for a FS or a PR description to appear in any number of document types. There is no requirement that there be a one-to-one correspondence between FS's and PR descriptions. An FS could be usable with different PR descriptions and a PR description could be used with different FS's.

2.6 An extended document environment

If documents follow the above tree-structured model and they are grouped in hierarchically organized directory structures, one can implement the latter as an extention of the former

Figure 18). A directory at any level is seen just as another document albeit one with different properties. In this extended view, any node can be a root of a document. A document thus becomes a very general notion. What we think of traditionally as a document is simply a particular case of our more general document object; one that has as its root a node of a certain type.

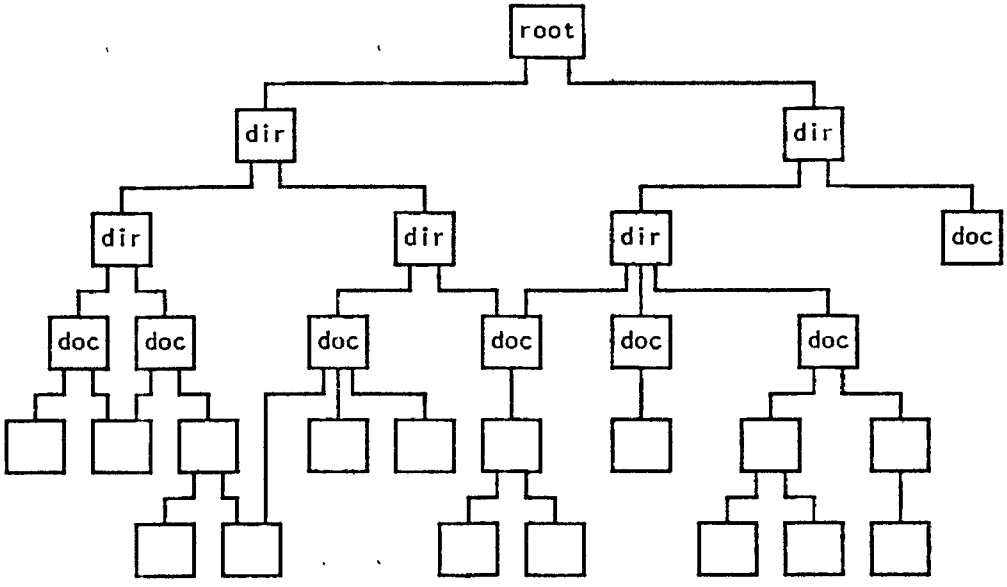

Figure 18. An extended document environment: All documents are subtrees of a hierarchical directory structure that has a single root ancestor called the primordial or master root. This example tree shows that extending the model to cover an operating system notion such as directories allows sharing to occur across documents and across directories.

The reason for extending the model this way is to apply it to not only the editing of documents in the traditional sense but also to the manipulation of higher level structures such as directories. There is little difference between moving a paragraph from under one chapter to another and moving a document from one directory to another. There is little difference between moving a subset of words from a paragraph to another and moving a subset of subdirectories from a directory to another. In our model all are tree editing operations. Tree editing thus becomes an all encompassing tool that applies to all levels of activity, ranging from text editing to tasks normally associated with the operating system. The same should be said of editing contents of nodes. Node editing must apply globally. Changing the name of a directory is not much different than changing the title of a chapter. Thus by keeping all information within one unified hierarchical data base, all operations over this data base are done through one uniform mechanism. What we achieve with this extended model then is uniformity.

A welcomed side effect of this extended model is that, because node sharing is supported

away with the notion of links (a la Unix), which are unsatisfactory as a means of keeping track of who is sharing whom. In our model a file (if we allow ourselves to use the term) can be in many directories at the same time. There is only one copy of this file and it knows who its parents are. Deleting this file from one directory does not necessarily mean deleting it from the other parent directories.

3.0 Document Editing, Formatting and Presentation

SIGHT's sofware architecture is similar to Etude's. Etude's user-interface is SIGHT's application and the editor/formatter/display complex is the same in both except that we prefer to call the last stage "presentation" rather than "display" given our support of multi-media. In this section we describe how the editing subsystem is organized, how the formatting subsystem is related to the editor and document types influence the editing and formatting processes, and we conclude with some remarks about document presentation.

3.1 Document editing

SIGHT's editing subsystem includes two major components: a general editor for trees and their contents which we call the Core (or Kernel) Editor and an open collection of specialized editors which we shall call, accordingly, Specialists.

3.1.1 The Core Editor

The Core Editor is a generic editor for trees. It is subdivided into two parts: a Tree Editor for manipulating trees without affecting the contents of nodes and a Node Editor for manipulating the contents of nodes without affecting where these nodes are in the tree. The Core Editor can in principle have access to any part of a tree. It is up to the application programmer to decide when to use the Core Editor or when to use a Specialist.

3.1.1.1 The Tree Editor

The Tree Editor is a collection of functions for creating and editing trees, for traveling and locating oneself within them and for making queries about the shape of a tree and one's whereabouts within it. Trees may or may not contain shared nodes. The Tree Editor's primary task is to manipulate parent-child and sibling-sibling relationships between nodes. It may read the contents of nodes but will not change them, except, of course, for the pointers that define the parent-child and sibling-sibling relationships and also, as a result, inheritable attributes.

The Tree Editor is controlled primarily by the application, but tree editing operations are constrained by the tree description in effect for the document tree type being edited at that moment. The Tree Editor will not create a tree that does not satisfy the tree description and will not edit a tree if that means it will no longer satisfy the tree description. When an editing command is issued from an application to the Tree Editor, it will check the effect of the editing command on the tree vis a vis the tree description. If the result is a valid tree, the command is executed. If the result is an invalid tree, the Tree Editor will attempt to "complete" the command so as to make it valid (see Figure 19 on page 30 for a simple example). Failing that, the command is not executed and the application is warned of the failure.

3.1.1.2 The Node Editor

The Node Editor is a collection of functions for creating, modifying and removing node attributes and data and for editing and retrieving their values. The Node Editor is not concerned with the parent-child and sibling-sibling relationships carried by nodes.

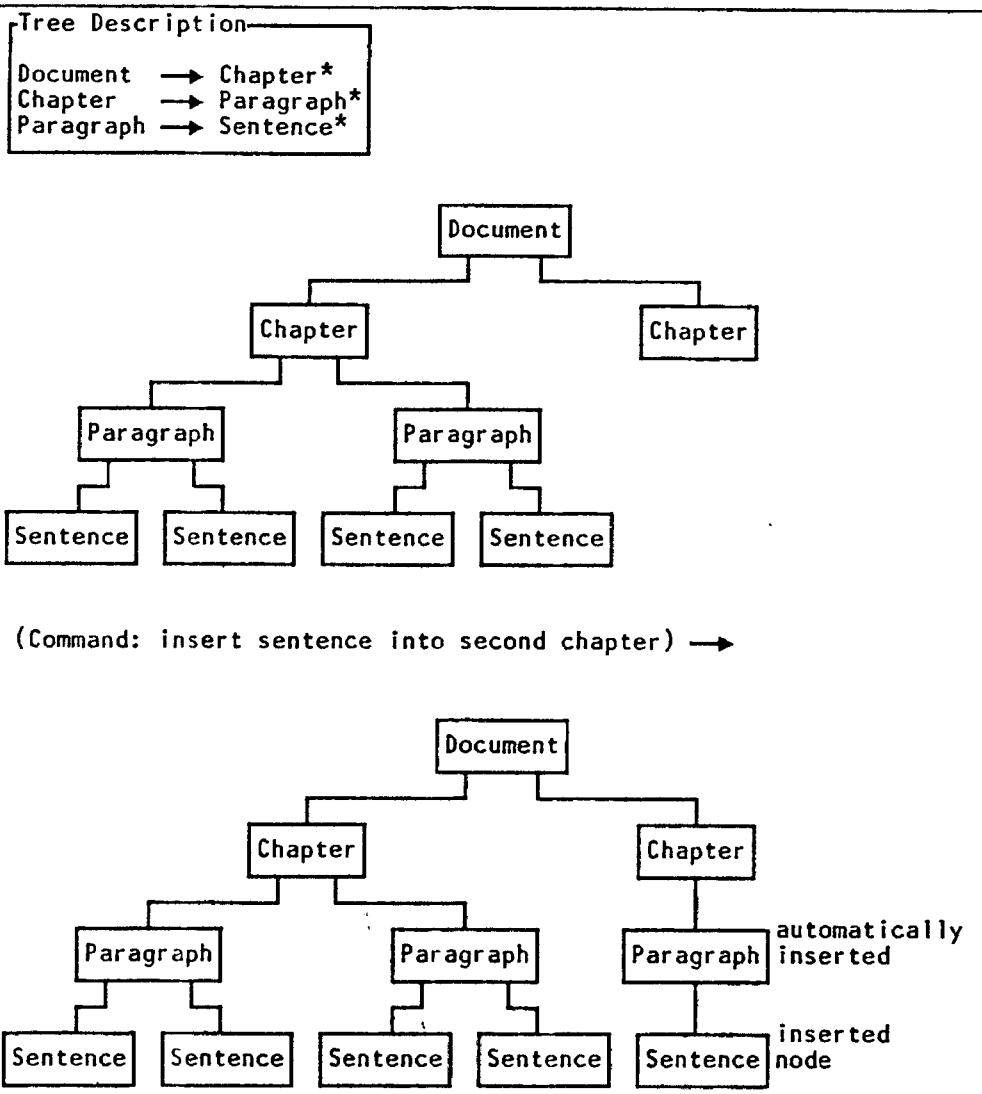

Figure 19. Tree Editor and tree description interaction: In this example a sentence is inserted into the empty second chapter of a document. According to the tree description for this document, a *Sentence* node can not be a direct child of a *Chapter* node, therefore the Tree Editor will automatically insert a *Paragraph* node between the previously childless *Chapter* node and the newly inserted *Sentence* node.

The Node Editor is controlled primarily by the application, but any node editing operation is constrained by the node type description in effect for the node being edited. The Node Editor will not create a node that does not satisfy the node type description and will not edit a node if that means it will no longer satisfy its corresponding node type description. When an editing command is issued from an application to the Node Editor, it will check the effect of the editing command on the node vis a vis its node type description. If the result is a valid node, the command is executed. If the result is an invalid node, the Node Editor will attempt to "com-

plete" the command so as to make it valid. Failing that, the command is not executed and the application is warned of the failure.

The Node Editor can call the Tree Editor. This is important, for example, when conditional node type descriptions have external dependencies whose values can only be reached by traveling within document tree structure, such traveling capability being exclusive to the Tree Editor.

3.1.2 The Specialists

Although the Core Editor is general enough to enable an application to edit any part of a document tree or any node contents, it nonetheless remains a generic program without much expert knowledge. Since we can not predict all the possible applications that may use SIGHT and the kinds of expert knowledge they may need, such expertise is relegated to ancillary program modules called specialists. These specialists are under control of the Core Editor which can at any time pass control to any of them for node-specific editing. In this sense specialists are extensions of the Core Editor. There is no limit to the number of specialists that can coexist with the Core Editor. With a simple interfacing protocol between the Core Editor and the specialists, new ones can be written without affecting the Core Editor and the others.

The kinds of specialists that are needed in an interactive document preparation facility include media specialists (text, line graphics, images, handwriting, audio and video) and specialists for mathematic formulae, tables, business graphics and spreadsheets. These specialists perform various input, output and internal functions specific to the type of data they are assigned to handle. For example, a text specialist is, at the simplest level, able to extract words and sentences out of an input string of characters and is, at more complex levels, able to indicate typographical errors and incorrect grammar. A line graphics specialist is able to take over from the Core Editor in manipulating document figure subtrees and possibly execute complex geometrical computations and perform drawing synthesis. The handwriting specialist is expected to perform recognition and plug the result into a node, likely of type *word*. The audio specialist is expected to do the same on input and, in addition, speech synthesis on output. The last two should also allow one to store handwriting and audio data without recognition.

The protocol between the Core Editor and a specialist is in essence very simple. The Core Editor passes a node to a specialist. This node defines the root of the tree to which the specialist is constrained. The specialist has then total control of this tree. A specialist may in turn call any other specialist or a new instantiation of the Core Editor to work on any deeper subtree of the tree under its control. Again the protocol simply involves passing a node that serves as the root of the tree to which the callee is to be constrained. This form of alternating recursive calls can be carried to any level of nesting, though in practice the nesting is almost never more than two or three. The Core Editor and the specialists can be instructed on when to pass control to each other. This will normally occur at specified node types.

The advantage of this organization is its modular construction. The Core Editor remains the generic editing mechanism available to all at any time. The specialists are the specific editors, selectively loaded only for applications that need their services.

The notion of having specialists, although not explicitly used by ISO's ODA, does not seem foreign to that architecture which can be gathered by ODA's notion of substructures within its basic objects. It appears that the scope of ODA does not give it access to these substructures, therefore requiring specialized editors to create and manipulate these substructures. If this assessment is correct, our model is very similar to ODA, except perhaps for the fact that our model is more flexible in that it can be given access to subsubstructures of substructures belonging to a specialized editor. This form of recursion does not appear to be supported by ODA.

3.1.3 The editing environment

Figure 20 shows how the constituents of the editing environment interact. (The numbers in parentheses below correspond to the numbers in the figure).

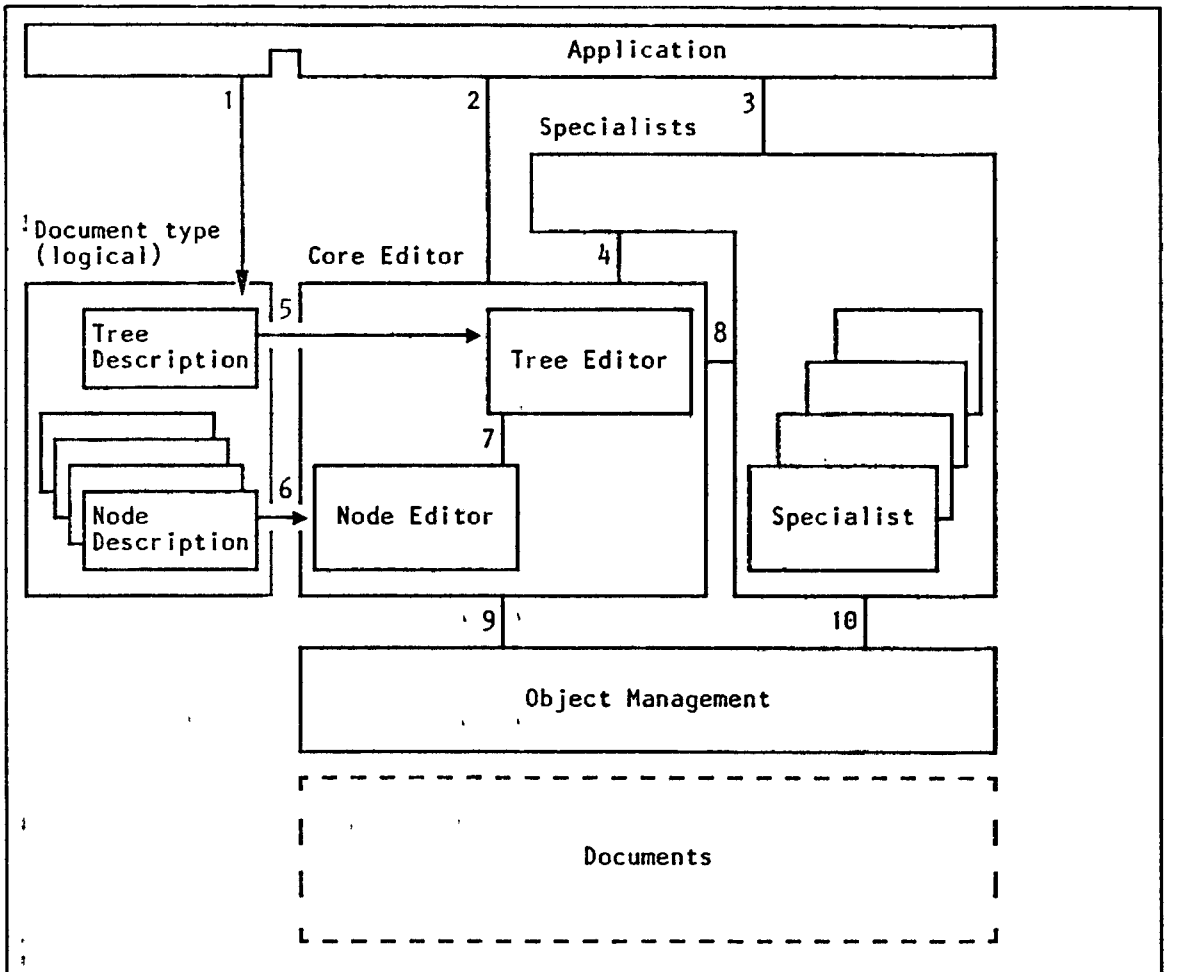

Figure 20. Tree-structured document editing organization

(1) An application that uses SIGHT is responsible for defining or selecting a pre-defined (logical) document type which will normally remain unchanged for the life of a document created under that type. As explained before, the document type, as far as the tree and node

contents are concerned, consists of a tree description and several node type descriptions. An application may support only one document type or it may support more than one. The root node of a document should normally tell the application to which document type it belongs. When the application reads in a pre-existing document from storage, its first task is to fetch the appropriate tree and node type descriptions from its document type data base. For each active document, there is an active document type; but for each document type, there may be more than one document.

(2) The application interacts with SIGHT by making calls to a Core Editor. These calls are essentially editing commands and queries issued against the document. The Core Editor executes the command or query and responds to the application on its success or failure in carrying out the command or with the answer to the query. The Core Editor is subdivided into a Tree Editor and Node Editor.

(3) The application can also make calls to Specialists. Specialists are mode or media specific extensions. Their main purpose is to perform functions that are not supported by the Core Editor, such as text analysis, mathematical computations, data compression and encription, handwriting and speech recognition, image processing, etc. As far as editing *per se*, they would be unnecessary given the generality of the Core Editor, but even then Specialists may have the advantage of being optimized for their supported mode or media and of affording a more appropriate interface to their individual tasks. Where one draws the line separating Core Editor from Specialists is an implementation decision, and often their functionality will overlap, leaving to the application the decision of which to use.

(4) Specialists can, similarly to applications, send document editing commands to the Core Editor. In this sense specialists act as intermediary applications. Thus, a spreadsheet specialist may do all the mathematical computations typical of this mode of data organization but may do all the equation editing with the aid of the Core Editor.

(5) The Tree Editor uses the currently active tree description to check on the validity of the document tree for all tree alteration commands. Only those commands that will not invalidate the document tree are allowed to execute.

(6) The Node Editor uses the currently active node type descriptions to direct how nodes should be edited. It can check when attributes and data can be added to or removed from a node without invalidating it. It can, to a certain extent, check whether values assigned to attributes and data are valid or within allowable bounds for a given node type.

(7) The Tree Editor and Node Editor can pass control to each other. For example, when the Tree Editor adds a new node to a tree, it can only create an empty node and attach it to a branch in the tree. It must then ask the Node Editor to fill in the node with the attributes and data, if any, that this node must contain according to the node type description assigned to the node. Alternatively, the Node Editor, after removing an attribute from a node, may find this node to be empty and then call the Tree Editor to remove the node from the tree.

(8) The Core Editor can pass control over to any specialist, and a specialist can pass control over to the Core Editor, recursively. This allows specialized trees (i.e. trees that are meant

by the application to be handled by a specialist) to be nested within generic trees (i.e. trees that are meant by the application to be handled by the Core Editor) and generic trees to be nested within these specialized trees to any level of nesting. As one travels deeper into these trees, control then passes back and forth between the Core Editor and specialists.

(9) The Core Editor is a collection of procedures that make use of object management procedures. While the object management subsystem implements a network data-base, the Core Editor enforces a hierarchical (lattice) organization.

(10) Specialists also have direct access to the object management procedures. They should, like the Core Editor, enforce a hierarchical (lattice) organization. If they don't, the application is responsible for isolating all cyclic structures from the Core Editor and give only the proper Specialist access to them.

3.2 Document formatting

Since we have targeted SIGHT to be used first in building an interactive editor-formatter, the formatter must execute in real-time. In principle, it should be possible to reformat a document quickly, or at least what is visible of it on the display, after every inserted letter or simple mouse-based editing. Because we do not expect office workstation technology to give us this kind of performance in the short run, various techniques are used to alleviate this computation bottleneck. Incremental formatting, where only selected parts of a document are subject to real-time formatting, and imperfect formatting, where local reformatting is accepted for display even though preceding material has not been reformatted, are two important techniques that improve responsiveness. Such techniques have been applied before, for example controlling the "degree of safety" for interactive formatting in the Janus formatter [Ch82]. If these and other secondary techniques are not enough for a given workstation, the reformatting granularity can always be raised to the word or even paragraph level.

3.2.1 The Formatter

The formatter is a priviledged application that exists immediately above the editing environment. Its purpose is to map LR's into PR's to be displayed, printed or heard. Since LR's and PR's are separate, the formatter reads the contents of a LR and creates or updates the associated PR; the LR structure and its data are not changed.

It is possible to have more than one PR for one LR, as seen in the previous chapter. The formatter can be called to maintain several PRs of a single LR concurrently. (We are assuming a single-user system, therefore there are no contention problems when editing one LR from many different PRs.) So it is possible to edit a document through one of the PRs and have the others updated concurrently as the underlying LR changes.

The formatter is a program that traverses the LR in traipse form (i.e., visit parent, visit children, visit parent again). It is essentially node type driven, that is, for each node visited there are certain actions taken according to the node's type. When the node is first visited, a format environment is established for the node and its descendants. When the node is visited again and exited, this format environment is closed.

The formatter may not visit all the nodes in a LR. Nodes or node types may be included in the LR but may be defined as protected from formatting (i.e., no format). The formatter is to take such a LR node and its subtree "as is". It must then be capable of calculating or be told of the physical dimensions of the object represented by the subtree. A node is created in the PR that carries these dimensions and points to the root node of the subtree in the LR. This technique is useful for graphic figures and annotations which, except for positioning and possibly zooming, do not normally require the services of a formatter.

Besides no-format, another reason for excluding nodes from the formatter is no-display. Certain nodes or node types may carry information that is not meant for inclusion into the document but only for use to infer other data that may appear in the document. A typical example of this are the mathematical equations that regulate the behavior of a spreadsheet. These equations will not show in the document, only their results. On the other hand, the end-user will very likely need to edit these equations. This can be done through another PR which is formatted with equation nodes included and all other data nodes excluded from the formatting process. In short, the end-user would be manipulating two PRs: one, the principal document and, two, a "document" showing the underlying mathematical structure of the spreadsheet(s) appearing in the principal document. Nodes excluded from one PR are included in the other and vice-versa. This constitutes an example of multiple document views by selective exclusion of LR nodes.

As the formatter traverses the LR while bypassing no-display nodes and the descendants of no-format nodes, it builds or updates the associated PR(s) also in traipse form. The model that underlies a PR is similar to Knuth's boxes [Kn79]. Each node in the PR corresponds to a box. The PR thus expresses a hierarchy of boxes and the data that go into the leaf boxes. The nodes contain mostly dimensional and positional information, visual properties (color, highlighting, reverse video, etc.) and media-dependent properties (e.g., font names).

Normally the formatter is the only program that creates and edits PR's. However, since a PR tree is like any other document tree, albeit built out of very different node types, the application can in principle bypass the formatter and edit a PR directly via the Core Editor and specialists. Of course a PR so edited will almost certainly differ from the PR produced by the formatter and, if the formatter is allowed to work on this PR again, it will destroy these editing changes and reestablish the PR as before. If this kind of editing is to remain uncorrected by the formatter, the formatter must be prevented from updating the affected part of the PR. This can be achieved by properly tagging the root nodes of the subtrees that are not to be updated.

The formatter has different responsibilities depending on the media being handled. In the case of text, the usual h&j (hyphenation and justification), pagination and font handling functions are present. In the case of line graphics, the formatter may be either instructed to take a figure "as is" from the LR (in which case the PR simply points to the figure subtree in the LR) or to change the style of the drawing (e.g., adding shadows to box outlines and making outlined arrows out of thin arrows in the original user's drawing) in which case a subtree with the altered drawing may have to be created and inserted in the PR.

The formatter can synthesize PR contents that did not exist explicitly in the LR. Tables of contents, lists and indices of various kinds are the most traditional of these synthesized contents. Less trivial and quite useful are business graphics that the formatter is capable of automatically generating, with the aid of a graphics specialist, and inserting in the proper place in the PR. The data for building such business graphics may come from spreadsheets maintained within the LR and which can themselves be targeted for formatting, say by selecting subsets of columns and rows and building these into tables to be inserted in the PR.

It is unnecessary to store a document's PR except for tagged nodes. A PR can be regenerated when the LR is loaded for editing or viewing. However, regenerating a PR takes time, and it is better for documents that are used frequently to have all their relevant PR's saved. The existence of a PR at all times also helps reduce the number of passes through a document. The PR intrinsically carries a record of the current values for cross-references. This is analogous to SCRIBE's "one pass formatter" which is obtained by saving, between runs of the formatter, an auxiliary file containing all the necessary information for resolving cross-referencing. In SIGHT the linkage between a LR and a PR achieves this purpose.

Though two passes are enough in most cases, it is possible to have a document so deviously arranged that an arbitrary number of passes are needed (Figure 21 on page 37). It is also possible to have oscillations, the prototypical case happening when two parts of a document influence each other in contrary ways, one part forcing a correction on the second which, when made, obviates the correction. *Ad hoc* approaches can detect such rare phenomena.

3.2.2 The formatting environment

Figure 22 on page 38 shows how the constituents of the formatting environment interact. (The numbers in parentheses below correspond to the numbers in the figure.)

(1,2) An application that uses SIGHT is responsible for defining or selecting pre-defined constituents of a document type. They are: (i) the tree and node type descriptions for both the logical and physical representations and (ii) the formatting specifications.

(3) The formatter is under control of the application. The application can instruct the formatter when, where and how much to format.

(4) The formatter uses the LR tree and node type descriptions in deciding how to traverse the LR, in selecting appropriate formatting specifications among those made available to the formatting process (6) and in determining the appropriate specialist to call for media specific tasks. For example, a node of type *Figure* may require intervention by the graphics specialist if the formatter decides the figure can not fit into a page and, consequently, must be scaled down or cropped.

(5) The PR tree and node type descriptions instruct the formatter on what kind of PR is desired and what results from the formatting process must be saved. For example, the PR tree description may instruct the formatter to include a table of contents and an index; the PR node type descriptions may instruct the formatter to keep only line break information and forget about relative word positions within a line.

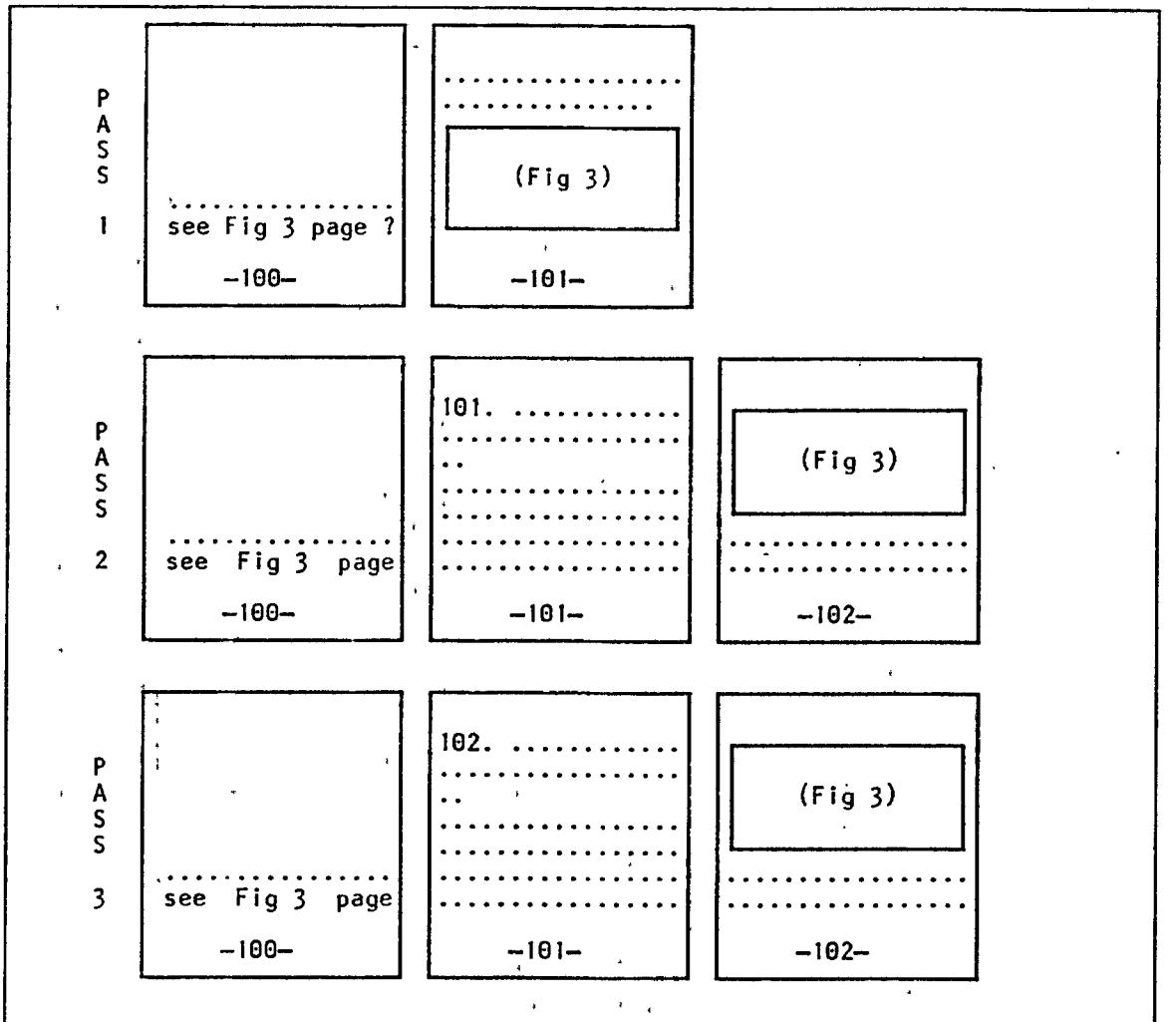

Figure 21. Document needing three pass formatting: It is possible albeit rare for a document to need more than two passes of the formatter. In this contrived example, an unresolved page reference for a figure allows the figure to be placed on the next page (101). During the second pass, the page value is obtained and inserted in the text causing the text to increase in length and the figure to be pushed to a subsequent page (102). The third pass finally gets the correct page number for the figure reference.

(6) The formatter uses the formatting specifications to make some, if not all, of the layout calculations, font choices, etc. The formatting specifications should be complete enough to cover all valid LR node types; in other words, every time the formatter traverses a node of a certain type, there should be a part of the formatting specification that says enough about that node type.

(7) The input to the formatter is an LR, which is traversed through the Core Editor.

(8) The formatting process maps the LR into a PR.

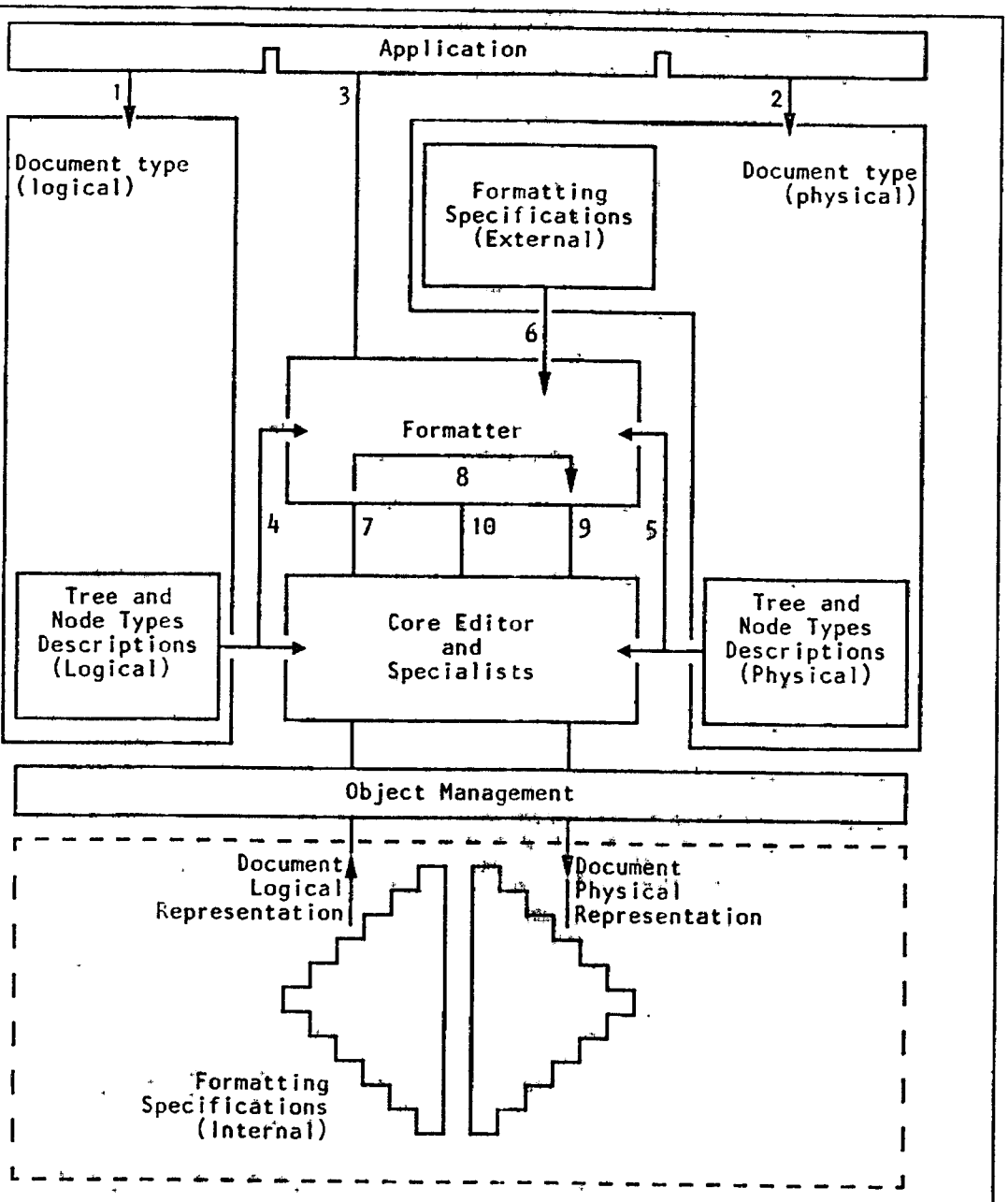

Figure 22. The formatting environment

(9) The output of the formatter is a PR, which is created or updated through the Core Editor and specialists.

(10) The formatter makes extensive use of the Core Editor for the purpose of incremental and imperfect formatting, both requiring the maintenance of markers that delimit areas that have been formatted, that are pending [re]formatting and that require additional passes of the formatter.

3.3 Document presentation

How documents should be presented for viewing is formally not a part of SIGHT, consequently our discussion of document presentation is limited to a few words.

Documents are seen and heard through their PR's. The formatter produces a PR that can be tailored for any particular device or for a generic device (say one with infinite resolution). The translation of the PR to a display, a printer or loudspeaker is done by a presentation specialist whose minimal qualifications include the ability to traverse the PR tree and read the contents of nodes for data and layout information. This specialist may issue commands directly to a device driver (for example, through a VDI or GKS interface) or may create an intermediate representation (e.g., a bitmap in memory). Because the sophistication of the presentation specialist may vary, the formatter can be instructed to produce a PR that contains the proper amount of detail needed by a given presentation specialist. If, for example, the presentation specialist is capable of justifying a single line, the PR does not have to go further than line breaks. If, on the other hand, the presentation specialist can only catenate letters into words, the PR must also give the position of every single word in a line.

Figure 23 on page 40 shows how the presentation specialist interacts with the rest of the system:

(1) As usual, the application controls the Core Editor.

(2) The application can communicate with a Presentation Specialist which interacts with input and output devices.

(3) Output to presentation devices (display, printer and loudspeaker) starts with the document PR which is scanned by the Core Editor under control of the Presentation Specialist.

(4) The scanning of the document PR is mediated by the active PR description.

(5) What parts of the document have to be scanned is determined by the Presentation Specialist.

(6) The scanned parts are farmed out to the appropriate device drivers.

(7) In the reverse direction, input devices (keyboard, locator and microphone) send their data to the Presentation Specialist.

(8) Input can be mapped via the Core Editor into a particular location within the PR.

(9) Having found the location within the PR, the Core Editor can be requested to map the location in the PR into a location in the LR.

(10)The LR description may help guide the inverse mapping from PR to LR.

(11)The Core Editor can be instructed to make editing changes at the location just found.

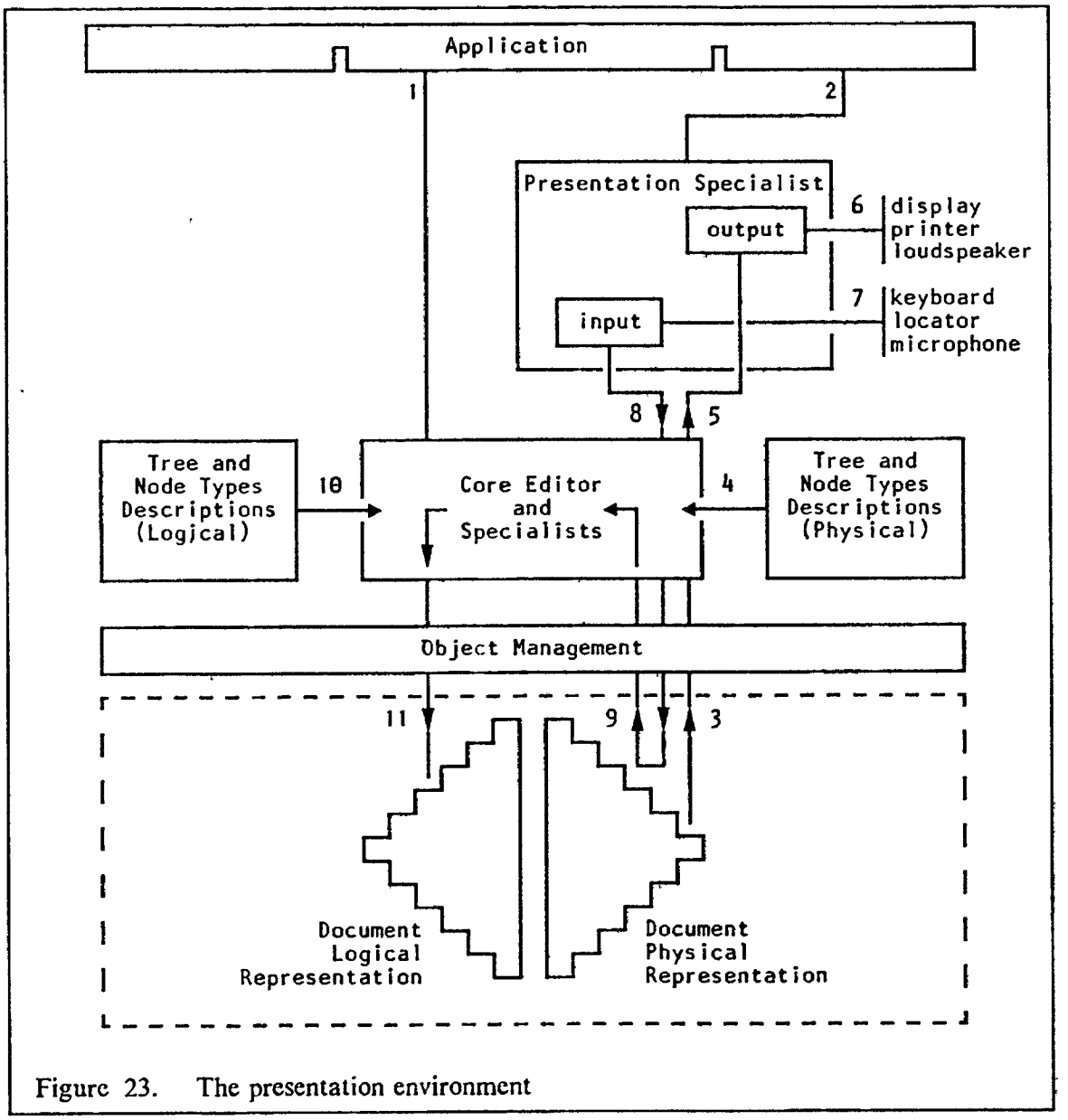

Figure 23. The presentation environment

Like all specialists, the presentation specialist has access to the Core Editor. Through it this specialist can maintain markers over the LR and PR to indicate, for example, what parts of a document are visible in what windows. In addition the Core Editor performs the critical inverse mapping from PR to LR. Thus, if the presentation specialist specifies a location within the PR (possibly the location of a key press or mouse button click), the Core Editor can pass to the application(s) owning the associated LR the corresponding location within the latter.

4.0 Document Tree Management

The object management component mentioned in the previous chapter is a network data base called PHOVIA. It is responsible for all aspects of memory management and provides to higher levels of SIGHT a collection of base functions for creating, manipulating and traveling around the network of data elements in the data base. SIGHT uses PHOVIA to structure an essentially hierarchical data base with sharing allowed and to make available to the application writer a more comprehensive set of functions for creating, manipulating, formatting and traveling over document trees. In this section, we describe how nodes are organized internally and how nodes are used to build trees.

4.1 Nodes as element lists

From the point of view of the document model explained in Chapter 2, there is only one kind of object out of which all documents trees are built: the node. From an implementation point of view, however, one must be able to handle the complicated internal organization of a node. Nodes must be able to grow and shrink as attributes are added or removed, as data are inserted or deleted and as they gain or lose children. Nodes are therefore dynamic structures that are best constructed out of more primitive objects. Consequently we have taken a two tier approach to object management. In the bottom tier we find objects we call **elements**. In the top tier we find the node objects. A node is constructed as a linked list of elements. Each element has an identifier type tag that indicates what kind of information the element carries. Certain element types carry pointers or fixed size data while others carry variable length data (Figure 24).

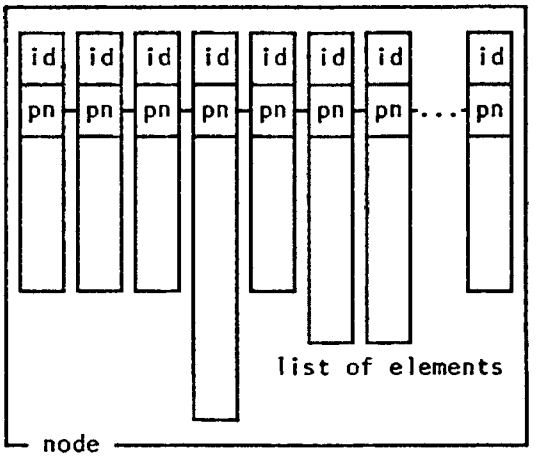

Figure 24. A node as an element list: A node is conceptually a doubly-linked circular list of elements. Each element has an identifier type, previous/next pointers and an id-type-dependent data field. (The previous/next pointers will not be drawn in the later figures.)

Given our document model, the most important element types are the ones that define parent and child relationships. The *Parent* and *Child* identifier types are the most basic pointer elements. If a node has one parent, the normal case, it will contain one *Parent* type element. If it has five children, it will contain five *Child* type elements. If the node is shared by more than one parent, it will contain one *Parent* type element for each parent node that shares it (Figure 25).

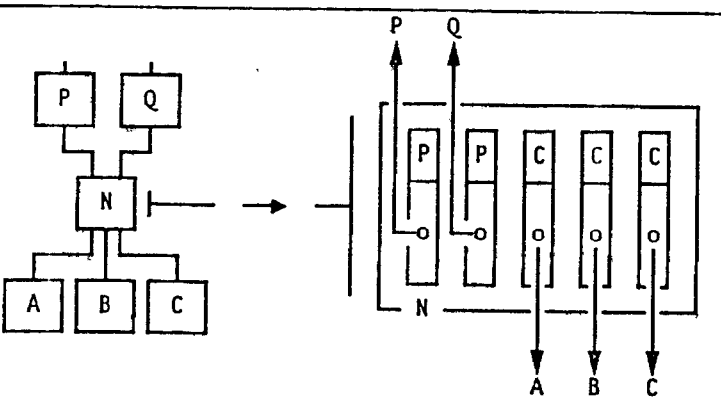

Figure 25. *Parent* and *Child* elements: A node contains one *Parent* element (P) for each of its parents and one *Child* element (C) for each of its children. A *Parent* element is a pointer to a parent node. A *Child* element is a pointer to a child node.

Next in importance are the element types that correspond to attributes. The *Attribute* identifier type is particular in that it is subject to the mechanism of inheritance described in Chapter 2.

The final and most generic class of element types -- *Datum* -- includes all data of fixed or variable size. Variable sized data elements carry their own length field. Fixed sized data elements, on the other hand, have their length defined outside in a table of element type definitions. The *Datum* type differs from the *Attribute* type only in that the former is not subject to inheritance.

While the parent and child relationships have their own specific element types, the sibling relationship is implicit and exists through the ordering of elements in a node. More specifically the ordering of a node's children is defined by the ordering among the child element pointers carried by the parent (Figure 26 on page 43).

The ordering of the children is thus not carried by the children but by the parents. This must be so if node sharing is to be supported. A node that is shared belongs to more than one sibling list and it would be cumbersome for it to keep track of its position in every one of these lists. This is better left to each parent node (Figure 27 on page 43).

In short, the order of elements in a node can and will, in the case of child pointer elements, have meaning. It is up to the application programmer to establish a convention on how ele-

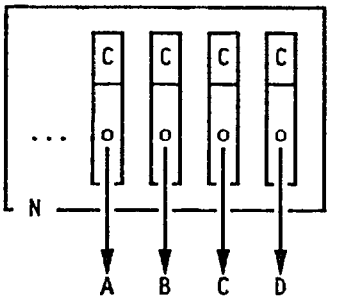

Figure 26. Children ordering: The order among children is defined within their parent. The order is the one defined over the respective *Child* type elements contained in the parent node. In this example, parent N has four children -- A the oldest and D the youngest. Changing the order among children is simply a matter of changing the sequence of *Child* elements within the parent; the child nodes are not affected in any way.

ments should be organized within a node and to assure that this convention is followed. The PHOVIA object management system does not enforce any convention.

4.1.1 How nodes are put together

The convention used by SIGHT groups all *Parent* elements together followed by all *Attribute* elements followed by intermixed *Child* and *Datum* elements (Figure 28 on page 44).

The reason for intermixing *Child* and *Datum* elements is to maintain the order among all the data that belong directly or indirectly to a node. The *Datum* elements in a parent node belong directly to it. The *Datum* elements in children and their descendants belong indirectly

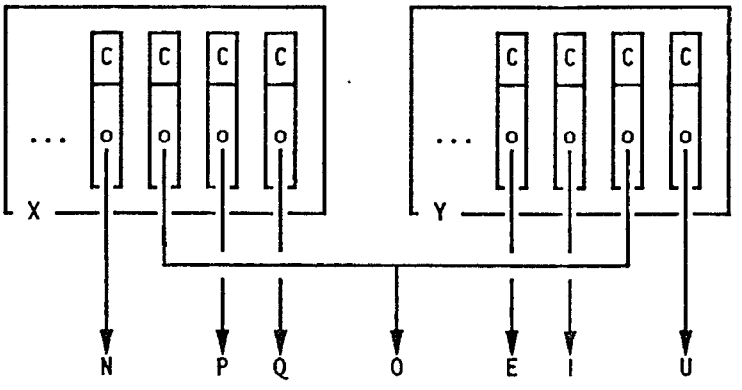

Figure 27. Children ordering under node sharing: Because children ordering is defined within parents, children can be freely shared without the children having to worry about who their surrounding siblings are. In this example, shared node O has immediate siblings N and P when taken to be a child of node X and siblings I and U when a child of node Y.

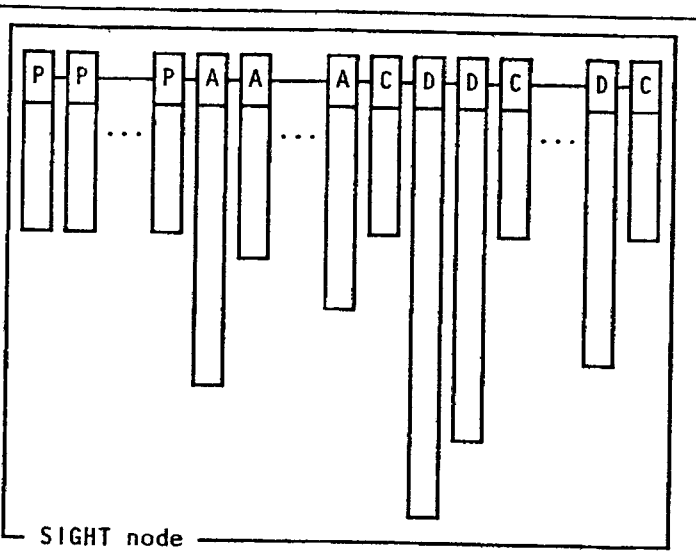

Figure 28. Element organization convention: all *Parent* elements are grouped together followed by all *Attribute* elements followed by intermixed *Datum* and *Child* elements.

to the parent node. But whatever levels the *Datum* elements find themselves in, when the document tree is traversed from left to right while moving up and down branches, the *Datum* elements must be conceptually strung together in sequence (Figure 29 on page 45).

4.1.2 Element type identifier aliases

PHOVIA provides for application-defined element types. It will keep track of their size and include them in the inheritance set if they are defined to behave as attributes. Applications can define their own "primitive" types. In addition element types can be defined as aliases of previously defined types, primitives or not. When an element type A is an alias of element type B, the properties of B apply also to A. Typically, an application will define new attribute element types -- e.g., *Color* -- as aliases of the primitive element type *Attribute*. It will define new data element types -- e.g., *Line Segment* -- as aliases of the primitive element type *Datum*. Finally, it may define new child element types -- e.g., *Word* -- as aliases of the primitive element type *Child*.

Aliasing to the element type *Child* serves to define node types. The disadvantage with this form of node type definition is that the type exists in a parent node rather than in the node to which it applies. On the other hand, it has the advantage that a shared node may have different types; each of its parents can look at the shared node in its own way (Figure 30 on page 46).

The alternative to this form of "implicit" node type definition is to have each node carry a *Node_Type* element which contains the node's type identification (Figure 31 on page 47). In this case the *Node_Type* element type is an alias of the *Datum* given that node types are not inheritable.

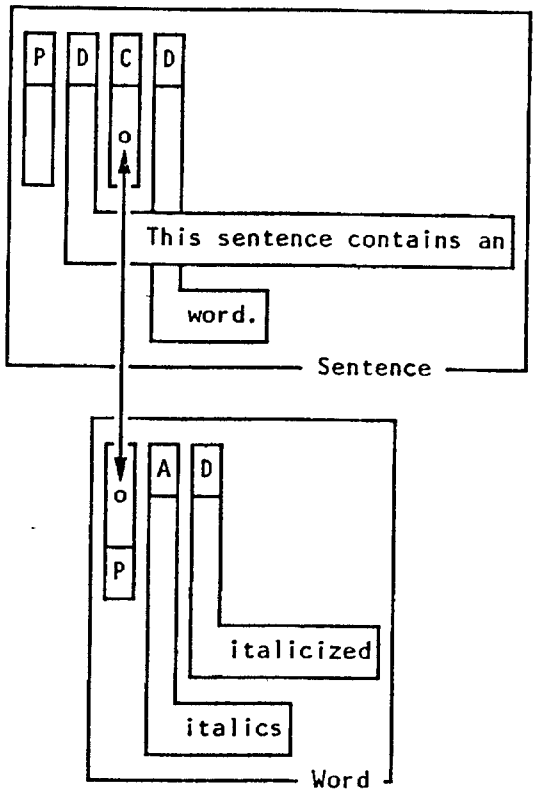

Figure 29. Concatenation of *Datum* elements: In this example, a sentence's text data was broken down into three pieces after one of its words was italicized. The first and last parts of the sentence are stored as data elements since the current attribute values apply to them. The word in the middle is made into a child node because a new attribute value applies to it.

Aliases can be cascaded to any arbitrary level. For example, one can define an element type *Point* which is an alias of the element type *Coordinate* which is an alias of the element type *Number_Pair*. Aliasing thus allows one to create levels of abstraction and encapsulate application-defined semantics in a hierarchical fashion.

Aliasing as described enforces a strict tree organization among element types. It is possible, however, to define a new element-type to be an alias of more than one already existing element-type. For example, the *Point* element type mentioned above will also be an alias of *Datum* (since it is not subject to inheritance). Use of this facility requires care since the semantics of the more primitive element types of which the new element-type is a sub-class may conflict. For example, an element cannot be both an alias of *Child* and *Parent*. PHOVIA will not check for conflicts of this sort. It is the application's responsibility to validate the consistency of the element type alias hierarchy.

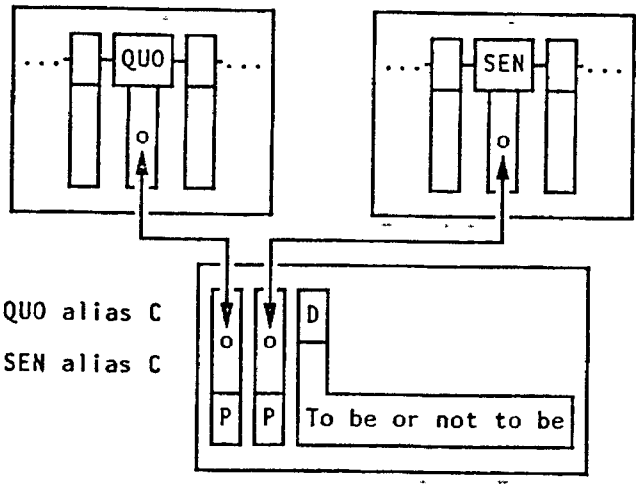

Figure 30. Implicit form of node type definition: Node types can be implicitly defined through element types. Using the aliasing facility, element types can be defined as alias of *Child* and these elements define the type of the node they point to. Because in this case the type of a child belongs to the parent, a shared node may have different types depending on the parent. In this example, element types *Quote* (QUO) and *Sentence* (SEN) are alias of *Child*. Two nodes look at a node containing some text in different ways; one looks at it as a quote and the other looks at it as a sentence.

4.1.3 Element type identifier table

Element types defined by applications are added to an element type table maintained by PHQVIA and which contains at the outset a small number of primitive types of which the *Parent*, *Child*, *Datum* and *Attribute* types are the most important. This table contains per element type (a) its unique identifier, (b) whether it is of fixed or variable length, (c) the field length for fixed length elements, (d) its aliases and (e) its name. The name of the element type is included, in part because one can not be sure that independent data bases, running on different workstations, will use the exact same identifiers for equivalent element types. If a document is transmitted from one independent data base to another, one also transmits the element type table used to create the document in the source data base. At the receiving end the document is displayed using the accompanying table or is changed to reflect the identifiers used by the receiving workstation. In the latter case, the accompanying table is discarded afterwards. Of course all this assumes that element type names used by the different data bases are the same. If not, an additional name to name equivalence mapping would have to be created among workstations desiring to exchange documents in order to bridge different naming conventions.

4.2 How trees are put together

Trees are built by connecting nodes via parent-child pointers. Each node has *Parent* elements that point to its parents. The only exception is the root node which does not have any parent.

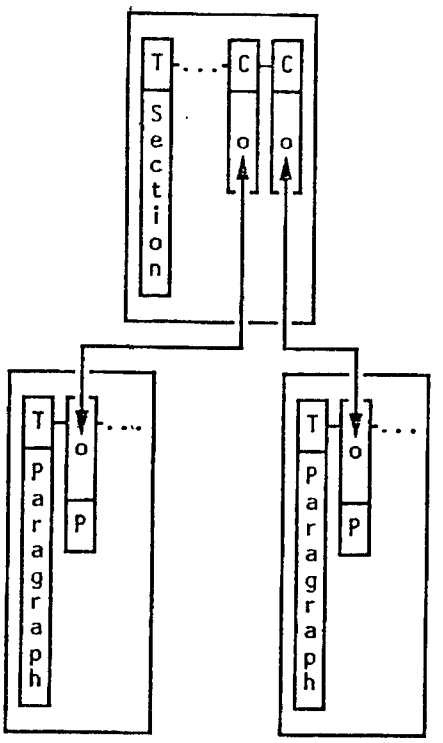

Figure 31. Explicit form of node type definition: Node types can be defined explicitly by having each node carry a *Node_Type* element. In this case, the node type is independent of what parents it has.

Each node that is not a leaf has *Child* elements that point to its children (Figure 32 on page 48).

The minimal cost for building a skeleton tree, i.e., one where nodes contain only *Parent* and *Child* elements can be calculated by counting the number of branches in the tree. A branch is defined here as the parent-child relationship that connects two nodes. In the example tree of Figure 32 on page 48 there are ten branches. Each branch implies a pair of elements, a *Child* element at the top extreme of the branch and a *Parent* element at the bottom extreme of the branch. The cost of a *Child* element is equal to that of a *Parent* element since both are pointer types. The cost of a pointer type element is the pointer itself (p bytes) plus the cost of the element identifier (i bytes) plus the inter-element linkage (n bytes -- n for neighbor). The cost for a branch is therefore $2(p+i+n)$. In the current implementation: p is four; i, two; and n, four. Consequently, twenty bytes are needed for a branch; thus 200 bytes for the example tree. It is more exact to give the cost in terms of branches rather than nodes because of node sharing. (In the case of a tree without shared nodes, it does not matter because branches and nodes are related by a trivial graph property -- branches equal nodes minus one.)

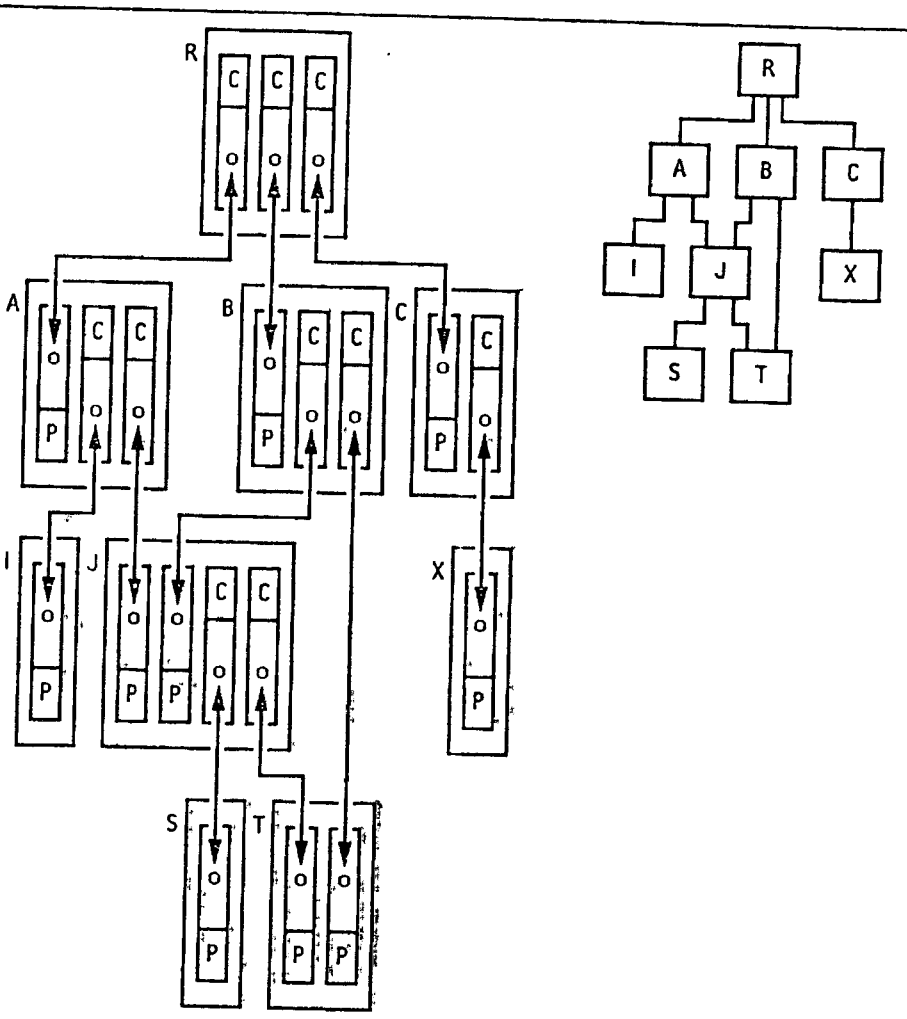

Figure 32. The use of *Parent* and *Child* elements in building trees: The drawing on the right shows how the user of SIGHT thinks of a tree and the drawing on the left shows the minimum number of elements to build that tree in PHOVIA.

4.3 Node element management

Node element management is principally responsible for allocating and deallocating elements in memory. Besides the usual initialization and termination functions through which the application accesses and releases the root of the master tree, the application is provided with the expected editing functions for inserting, deleting, changing and moving elements within and across nodes, for keeping track of one's location in a tree and traveling within it, for making various types of queries and for other miscellaneous actions.

Because the number of elements can be enormous for large size documents, document trees are often only partially loaded in main memory. To decrease the frequency of paging,

elements are clustered according to their proximity. As a minimum, elements of a node are clustered together so that the clustering problem need be analyzed only at the node level (for example see [Sc77]).

5.0 Conclusion

In Figure 2 on page 7 we gave an example of a document tree that partitions the document down to the word level. Some will argue that this is too fine a partition if the document storage requirement is to be kept relatively small and that one should not go below the paragraph level. Some will counter with the argument that a finer partition allows more detailed formatting (e.g. changing the font within a sentence) and that certain types of searches may be faster (e.g. getting the fourth word in the second sentence of a paragraph). In terms of our model it does not matter. It can support any hierarchy, deep or shallow, that the user chooses to define. Deep hierarchies have the advantage that they give greater and more convenient control over the finer structural parts of a document. Shallow hierarchies, on the other hand, need substantially less storage space. (Speed judgments are hard to make because time efficiency depends on the kind of operation that is performed on the tree.) A natural compromise is a shallow hierarchy that can be locally extended depthwise whenever different attribute values need be assigned to adjoining small structural parts of a document (as illustrated in Figure 29 on page 45). In the case of text, this probably means having paragraph nodes as leaves of the tree in general, but allowing a paragraph node to be refined into sentences, words and even characters. Another possible compromise is to assume a deep hierarchy that gets compacted from bottom up as long as attributes of adjoining nodes carry the same values and data carried by these nodes can be concatenated. The latter approach is the one taken by PHOVIA. The former approach is one that an application programmer can implement himself via proper document type definitions and use of the Core Editor and the appropriate specialists.

Though node sharing makes possible many useful functions, it does exact a price in added complexity to the Core Editor. Most editing functions provided by the Core Editor must be able to recognize when subtrees they are handling contain nodes that are shared by other subtrees. This is critical in functions such as tree deletion when one can not just sever the root of a subtree from the master tree. Whether the user has chosen to include the shared nodes in the deletion or exclude them, their existence requires careful scissoring of parent-child pointers.

A slightly different problem occurs with the copy tree operation. Should node sharing be preserved when a subtree is copied? Our approach to these kinds of problems is to make available to the application all identifiable and reasonable options. It is no surprise then that all this additional computation for handling sharing represents a performance degradation that, depending on the supporting hardware, can slow interactive editing. But, given the research nature of this project, efficiency is often sacrificed in favor of capability.

Our formatting model is characterized by the separation of the logical representation of a document from its physical representation(s). This does represent a space penalty even though the lower levels of these representations are normally shared. Maintaining separate logical and physical representations implies a considerable cost in processing time and memory space. On the other hand, having a document tree structure helps in implementing incremental and imperfect formatting because it is easy to apply these techniques to selected subtrees of a

logical representation and because the physical representation carries the information that is necessary for resolving cross-references.

PHOVIA has proven to be a flexible data structure, especially as regards the handling of shared nodes. The intrinsic symmetry between *Parent* and *Child* elements is perfectly suited for building lattices of nodes. Even cyclic structures can be built out of it, though in SIGHT this is never done.

SIGHT is implemented in the C language. Only editing subsystems have been implemented so far. These include the PHOVIA database, the Core Editor and the text and graphics specialists. A simple integrated text and graphics editor has been built using SIGHT for testing purposes. It illustrates the concurrent editing of text and figures and illustrates the ability to have shared objects (a shared object has all its instantiations on the screen change simultaneously when one instantiation is edited).

To achieve its full potential SIGHT will need to integrate specialized uses of the basic media. Spreadsheets, mathematical equations and 2-D animation are examples of these specialized uses. Some of them have been studied in detail but await implementation.

Through SIGHT we have attempted to create a sophisticated environment within which office applications can be built and share a common database of information. The application that we targeted more closely was an editor/formatter capable of handling all the expected information types that can be present in a document. This application still does not exist but the trend is clear. Full integration of text and graphics has just been achieved commercially as evidenced by the Interleaf software and the Texet CAP workstations. This was the most urgent integration to achieve. Mathematical equations will probably be next and someday spreadsheets will be manipulated directly from within documents. SIGHT has shown us that this is not a faraway vision, it is possible today if the proper resources are available.

References

[Ch82] Donald D. Chamberlin, James C. King, Donald R. Slutz, Stephen J. P. Todd and Bradford W. Wade (IBM Research Laboratory, San Jose), "Janus: An Interactive Document Formatter Based on Declarative Tags," *IBM System Journal*, Vol. 21, No. 3 (1982), pp. 250-271.

[Fu82] Richard Furuta, Jeffrey Scofield and Alan Shaw, "Document Formatting Systems: Survey, Concepts and Issues," *Computing Surveys*, Vol.14, No.3 (9/82), pp. 417-472.

[Gu84] Jürg Gutknecht and Werner Winiger (Institut für Informatik, ETH-Zentrum, Zürich), "Andra: The Document Preparation System of the Personal Workstation Lilith," *Software--Practice and Experience*, Vol.14 (1984), pp. 73-100.

[Ha81] M. Hammer, R. Ilson, T. Anderson, E. J. Gilbert, M. Good, B. Niamir, L. Rosenstein and S. Schoichet, "The Implementation of Etude, an Integrated and Interactive Document Production System," in *Proc. ACM SIGPLAN/SIGOA Conf. Text Manipulation* (Portland, Ore., June 8-10, 1981), ACM, NY, 1981, pp.137-146.

[Ho82] W. Horak (Siemens AG, Zentrale Aufgaben Informationstechnik - ZTI, Munich, Germany), "Experimental Text and Facsimile Integrated Workstation," *Proc. of the 1982 Int. Zurich Seminar on Digital Commun.*, pp.93-100.

[Ho84] W.Horak & G.Kroenert (Siemens AG, Corporate Laboratories for Information Technology, Munich, Germany) "An Object-Oriented Office Document Architecture Model for Processing and Interchange of Documents," *Second ACM-SIGOA Conference on Office Information Systems*, 25-27 June 1984, Toronto, Canada, pp. 152-160.

[Ib84] Document Composition Facility and Document Library Facility -- General Information, GH120-9158, IBM General Products Division, Tucson, Arizona 85744, January 1984.

[Il80] Richard Ilson, "An Integrated Approach to Formatted Document Production," Master Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (August 1980) 109 pp.

[Ki84] Gary D Kimura and Alan C Shaw (University of Washington), "The Structure of Abstract Document Objects," *Second ACM-SIGOA Conference on Office Information Systems*, Toronto, Canada, Vol.5, Nos.1-2 (25-27 June 1984), pp. 161-169.

[Kn79] Donald E.Knuth, **TEX and METAFONT - New Directions in Typesetting**, American Mathematical Society & Digital Press (1979).

[Me82] Norman Meyrowitz and Andries van Dam (Brown University) "Interactive Editing Systems" *Computing Surveys*, Vol.14, No.3 (September 1982), pp. 321-415.

[Os76] Joseph F. Ossanna (Bell Labs), NROFF/TROFF User's Manual, Unix documentation (October 1976), 33 pages.

[Pr81] J. M. Prager and S. A. Borkin (IBM Cambridge Scientific Center), "Personal On-Line Integrated Text Editor" IBM Cambridge Scientific Center, Cambridge, MA 02139 (September 1981), 12 pages.

[Re80] Brian K. Reid (Carnegie-Mellon University), "A High-Level Approach to Computer Document Formatting," *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, ACM (January 1980), pp. 24-31.

[Sc77] Mario Schkolnick (IBM Research Laboratory), "A Clustering Algorithm for Hierarchical Structures," *ACM Transactions on Database Systems*, Vol.2, No.1 (March 1977), pp. 27-44.

[Se84] *The Seybold Report on Publishing Systems*, Vol. 14, No. 6, 19 November 1984.

[Sm83] Joan M. Smith (National Computing Centre, Manchester, England) "Text Structuring," *Data Processing*, Vol.25, No.8 (October 1983), pp. 18-20.

[Th82] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull and D. R. Boggs, "Alto: A Personal Computer," in **Computer Structures: Principles and Examples**, edited by Daniel P. Siewiorek, C. Gordon Bell and Allen Newell, New York: McGraw-Hill, 1982, pp. 549-572.

[Un84] Unilogic, **SCRIBE Document Production System: User Manual**, Unilogic Ltd., Pittsburgh, PA (April 1984), 264 pages.