Document Formatting Systems: Survey, Concepts, and Issues

RICHARD FURUTA, JEFFREY SCOFIELD, AND ALAN SHAW

Department of Computer Science, University of Washington, Seattle, Washington 98195

Formatting systems are concerned with the physical layout of a document for hard- and soft-copy media. This paper characterizes the formatting problem and its relation to other aspects of document processing, describes and evaluates several representative and seminal systems, and discusses some issues and problems relevant to future systems. The emphasis is on topics related to the specification of document formats; these include the underlying document and processing models, functions performed by a formatter, the formatting language and user interface, variety of document objects, the integration of formatters with other document processing tasks, and implementation questions.

Categories and Subject Descriptors: H.4.1 **[Information Systems Applications]**: Office Automation—*word processing*; I.7.0 **[Text Processing]**: General; I.7.1 **[Text Processing]**: Text Editing; I.7.2 **[Text Processing]**: Document Preparation; K.2 **[Computing Milieux]**: History of Computing—*software*

General Terms: Algorithms, Design, Human Factors, Languages

Additional Key Words and Phrases: Formatters, editors, text manipulation

INTRODUCTION

Document preparation involves two principal tasks: defining the content and structure of a document, and generating the document from specifications of its appearance. The first task is typically called editing while the second, the subject of this paper, is known as formatting. More precisely, formatting is concerned with the layout of document objects on hard-copy media, usually paper, and various soft-copy devices, such as video displays.

While text processing, especially editing, has long been a major application of computers, it is only recently that particular attention has been given to formatting systems. The reason for this is a combination of technology and economics. Because of increasing costs of manually produced documents, decreasing costs of computers and storage, and the availability of high-quality, computer-controlled printers, typesetters, and display devices, it has become both feasible and worthwhile to use computer formatting systems for a wide variety of technical, business, and literary documents, such as letters, memos, invoices, brochures, reports, papers, and books. Many experimental and commercial systems have been developed for offices, laboratories, publishers, and, in fact, virtually any enterprise that uses written documents.

Our aims in this paper are to characterize the formatting problem and its relation to other aspects of document processing, to describe and evaluate several representative and seminal systems, and to discuss some issues and problems relevant to future systems. The emphasis is on topics related to the specification of document formats; these include the underlying document and

CONTENTS

processing models, functions performed by a formatter, the formatting language and user interface, variety of document objects, the integration of formatters with other document processing tasks, and implementation questions. A number of important related areas are not covered in any detail; for example, there is little discussion of font design, the characteristics of typical output devices, commercial typesetting programs, or particular applications such as newspaper production.

1. THE FORMATTING PROBLEM

1.1 Object Model of Documents

In order to discuss formatters and their functions and to distinguish formatting from other aspects of document preparation, it is convenient to use an object model of documents [SHAW80a], somewhat analogous to that in programming languages. The model introduces a uniform terminology which is useful when comparing and evaluating various systems and ideas, and it allows a more precise definition of terms such as editing, formatting, and viewing.

A *document* is an object composed of a hierarchy of more primitive objects. Each object is an *instance* of a *class* that defines the possible constituents and representations of the instances. Some typical document classes are business letters, papers for a particular journal or conference, theses, and programs in a given language; common lower level classes include such document components as sections, paragraphs, headings, footnotes, tables, equations, matrices, figures, polygons, and character fonts.

Objects are further classified as either *abstract* or *concrete*. To each abstract object, there corresponds one or more concrete objects. An abstract object is denoted by an identifier and the class to which the object belongs. One example could be the identifier "the" in the class *word*, indicating the abstract word object "the." Another abstract object may be the identifier "plus" in the class *operator*, denoting the operator for addition. We sometimes use the term *logical* object as an informal synonym for abstract object.

Concrete objects are defined over one or more two-dimensional *page spaces* and represent the possible formatted images of abstract objects. For example, a particular paragraph of a document, an abstract paragraph object, may be represented concretely in many different ways depending on font, hyphenation conventions, line length, and other concrete variables.

Example

The extended abstract for this paper [SHAW80b] has the logical objects (partially) defined and structured as follows:

⟨ExtendedAbstract⟩ = (⟨Header⟩, ⟨Body⟩, ⟨References⟩)
⟨Header⟩ = (⟨Title⟩, ⟨Authors⟩ ⟨Affiliation⟩)
⟨Body⟩ = ⟨Introduction⟩⟨Section 1⟩ ⟨Section 2⟩ ⟨Section 3⟩
⟨References⟩ = . . .
⋮
⟨Title⟩ = "Document Formatting Systems: Survey, Concepts, and Issues"

⟨Extended Abstract⟩ is an instance of the class of extended abstracts specified for a particular conference; similarly, ⟨Section 2⟩ is an instance of the class of sections. The

notation $(A, B, \ldots, F)$ denotes the unordered *set* of objects $A, B, \ldots, F$; and $A\,B \ldots F$ means the object *sequence* A followed by B followed by ... followed by F. Thus the ⟨Header⟩ consists of the object sequence ⟨Authors⟩⟨Affiliation⟩ and the object ⟨Title⟩. The two-dimensional representations of these abstract objects define the concrete objects of the document. In this case, one set of concrete objects appears in a technical report containing the extended abstract while another appears in a conference proceedings.

Document processing consists of executing various operations to define, manipulate, and view abstract and concrete objects. For this purpose, we distinguish between *ordered* and *unordered* objects. Many textual objects, such as paragraphs and words, are normally ordered, implying that we can speak of the first one, the last one, the next one, the preceding one, and so on. On the other hand, there are many objects that are more naturally treated as unordered for particular applications; these may include the elements of a figure or table, parts of mathematical equations, and pieces of unrelated text. In the ordered case, document processing involves working in order through a *sequence* of objects. In contrast, processing a set of unordered objects allows arbitrary selection of objects and even interleaving of the operations.

1.2 Editing, Formatting, and Viewing

Within the object model framework, we can consider the major operations of document processing as mappings from objects to objects. *Editing* operations are defined as mappings from either abstract to abstract objects or concrete to concrete objects. Conventional text editing operations map logical text objects to logical text objects; for example, a text insertion or deletion may be a mapping from strings to strings or from paragraphs to paragraphs. Also, editing operations on an already formatted document produce concrete objects from concrete objects. An example of this type of editing is interactively inserting or deleting text from an already formatted paragraph, thereby mapping concrete paragraphs to concrete paragraphs; interactive layout operations such as moving formatted text, tables, or figures around a document are also in this category.

Mappings from abstract objects to concrete objects are defined as *formatting* operations. Standard examples are transforming a logical character to its representation in a particular font, producing a two-dimensional word with possible hyphenation from a logical word, mapping a paragraph into a sequence of lines, and breaking an abstract document into pages. In the nontextual domain are mappings such as those that transform an abstract directed graph to a line drawing (e.g., producing flowcharts), operations for producing two-dimensional mathematical objects from a possibly one-dimensional (string) specification of an expression, and functions for constructing or laying out a table from a list of its entries.

An important part of an abstract to concrete object mapping is the page space domain of the concrete object. The constraints on page spaces are often the cause of complex interactions among formatting operations. For example, a paragraph-to-lines mapping may cause hyphenation in a word; and a paragraph-to-lines mapping may be modified because a section-to-pages operation leaves a first or last line of a paragraph on a page by itself (known as a *widow*). Different page spaces are possible, depending on the viewing medium and on the application. These include a sequence of identical rectangular areas or boxes, which correspond to conventional hard-copy pages; a rectangular box bounded, say, in the horizontal direction but unbounded vertically (typically viewed by vertical scrolling); and boxes that are unbounded in two or more directions, for example, full, half, or quarter planes that could be viewed by displaying small rectangular areas (*windows*) of the region.

It is useful to distinguish between formatting a document and displaying some part of the resulting concrete object. This leads to a definition of *viewing* mappings that produce hard-copy and soft-copy images from concrete objects. An example is a concrete formatted object, defined in some normalized coordinate system, that may be viewed on a display screen and on paper by two different viewing mappings. A

viewing mapping might also be the result of either windowing or scrolling some concrete object. The separation of formatting and viewing also permits a device-independent treatment of formatting. In the simplest case, our viewing mappings take the role of output device drivers.

To summarize, we have divided document processing operations into three types—editing, formatting, and viewing—depending on the domain and range objects.

1. Editing:
   Abstract objects → abstract objects,
   Concrete objects → concrete objects.
2. Formatting:
   Abstract objects → concrete objects.
3. Viewing:
   Concrete objects → output devices.

Many other kinds of operations, such as numbering figures, equations, or pages, correcting spelling, and indexing terms, deal directly with the objects resulting from either editing or formatting. Spelling correction and figure or equation numbering can be performed with abstract objects and, consequently, may be done before formatting; on the other hand, page numbering and automatic indexing require the concrete objects produced from formatting. One other important class of operations is *filing*. Like most computer systems, document-processing systems require facilities for storing and accessing files of abstract and concrete objects. While we acknowledge their importance, we, for the most part, ignore filing issues. We also do not discuss those applications that involve mappings from concrete to abstract objects, such as on-line character or sketch recognition.

1.3 Formatting Functions

Our study of a variety of abstract and corresponding concrete objects used in text, tables, mathematical equations, and figures has led to the following set of general formatting functions. At a more detailed level than our mapping definition, these functions describe what formatters do.

1. *Selection of Primitive Concrete Objects.* The usual selection task is the retrieval of particular characters within a specified *font*, where a font is a set of concrete character objects having the same size and style. Also included are variably sized symbols such as summation ($\sum$) for an arbitrary expression, special symbols such as a company logo, and atomic figure elements, for example, points, lines, curve segments, and filled-in areas.

2. *Horizontal and Vertical Placement of Objects.* Examples of horizontal placement are operations to indent, tab, flush, and center. Vertical placement occurs when skipping lines, starting a new paragraph or section, and placing equations, figures, and tables on a page. Some objects, such as subscripts, require explicit placement in both vertical and horizontal directions.

3. *Horizontal and Vertical Alignment.* By object alignment, we mean the horizontal or vertical placement of an object relative to some other object(s). Operations such as aligning equal signs in equations, centering a table entry, lining up decimal points, or "prettyprinting" a structured program fall into this category. Alignment can be viewed as a simple form of constraint satisfaction.

4. *Breakup of Abstract Objects into "Paged" Concrete Objects.* This function includes breakup of objects into lines and pages, with page header and footnote handling, and is the central task of most text formatters.

5. *Scaling.* Objects may be expanded or reduced in size to fit into an allocated space, to be compatible with other elements of the document or to improve their appearance.

These five general functions are often used in a cooperative and ordered manner. For example, alignment involves placement which requires selection of primitive objects, and the first three functions are performed before page breakup. It would be desirable to define these functions more precisely, for example, for systems design purposes, but much research remains to be done before this can be accomplished. Some ideas on how this could be approached have been given by Guttag and Horning [GUTT80], where algebraic axioms and predicate transformers are employed to specify the design of a display interface. For our purposes, the object model, formatting

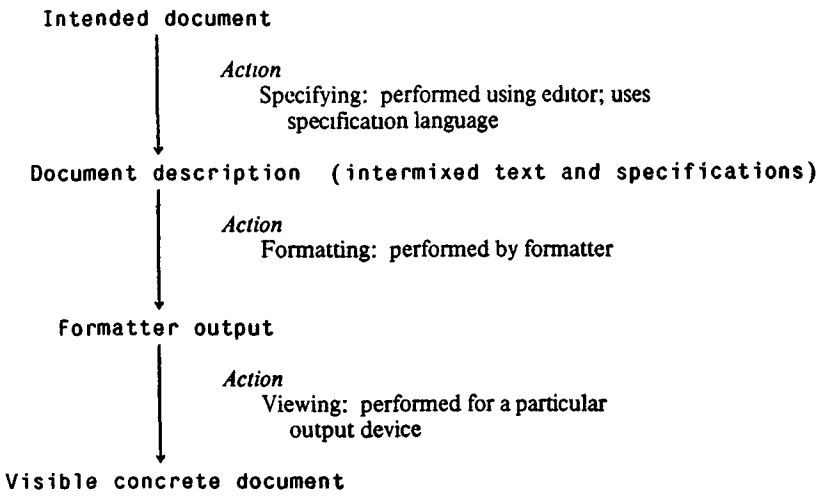


Figure 1. Steps in document processing.

definition, and list of formatting functions provide a useful framework for surveying past systems and discussing formatting issues and concepts.

2. REPRESENTATIVE AND SEMINAL SYSTEMS

In this section we discuss the history and evolution of document formatting by investigating some important original and representative systems.

It is useful to define some further terminology to describe the actions involved in document processing (see Figure 1). We call the mental image of the document the *intended document.* This is mapped, by an editing step which we call *specifying,* into a physical form consisting of intermixed specifications and text called the *document description.* The document description identifies the abstract objects of the document. The formatting and viewing mappings, as defined in Section 1.2, produce the *formatter output* and the *visible concrete document,* respectively. The visible concrete document is produced on a particular hard- or soft-copy display medium. Some formatters provide both a high-level specification language and also a lower level language for defining the meanings of new specifications. This lower level language is the *definition language.*

Although editing and formatting systems have been physically separated and developed individually for some time, formatting and viewing systems have typically been tied firmly together. In particular, in many systems the document description contains low-level information that requires a specific output device to be used to view the visible concrete object. There are two ways to separate formatting and viewing. The weaker of the two is provided by a *device-independent description* in which the same document description can be used (without change) to prepare formatter output for viewing on different devices. This is done either by rerunning the formatter after changing some parameters or by running different versions of the formatter. A stronger separation can be found in a few systems that also produce *device-independent formatter output.* Here, the same formatter output can be viewed to produce a visible concrete document on any of a number of different devices.

In selecting the systems to be discussed, we tried to pick those original early systems that were the first to present important ideas and which affected the designs of later systems; those systems that underwent a clear, clean, controlled, evolutionary development over the years; and those systems which to us represent present and future trends. When other factors were about the same, we preferred those systems with understandable, thorough descriptions in the open literature and those we have actually used. For purposes of presentation, these

systems have been divided into two groups: the *pure formatters* and the *integrated editor/formatters.*

Pure formatters accept a document description, previously prepared by a separate editing system. The formatter output may then be viewed, producing the visible concrete document. (Others sometimes call the pure formatters *document compilers* or *batch-mode formatters.*) Although many of the earliest formatters had an associated text editor, they are included in this class because the objects operated on by the editor and the formatter were logically disjoint. The editors in these early systems were provided out of necessity since general-purpose text editors were not common. This contrasts with the integrated nature of the editing and the formatting functions in the second group, the integrated editor/formatters.

Integrated editor/formatters allow one to view the visible concrete document while creating and modifying the document description, without leaving the editor/formatter. In other words, editing, formatting, and viewing are combined into one unified system. In the most general form, the user directly manipulates an exact representation of the visible concrete document. A form more closely resembling the pure formatters allows occasional viewing of the visible concrete document, and that only on request. (Integrated editor/formatters are also known as *document interpreters* or as *interactive formatters.*)

We describe the pure formatters in the next three sections, followed by a discussion of the integrated editor/formatters in Section 2.4. We found a few systems which, while not meeting the criteria for inclusion in the preceding categories, address unique problems or present ambitious solutions. A brief discussion of four of these systems appears in Section 2.5. Finally, a number of research laboratories are attempting to provide systems which combine the best features of the pure formatters with the best features of the integrated editor/formatters. Three of these projects are mentioned in Section 2.6.

2.1. The First-Generation Formatters

The first widely known pure formatters appeared in the 1960s. The available devices were quite limited: the output device was, at best, a simple typewriterlike printer, and the input was, at worst, from punched cards. The formatting functions provided were at a quite low, machine-language-like level of action. Most objects were related to the lines and pages of the document (the *format* of the document). Only a few objects associated with the document's logical *content* were supported: words and sometimes sentences and paragraphs. The formatting languages were fixed and nonextensible. The appearance of the formatting commands seemed quite ad hoc. For example, it is not always clear which commands took parameters or how many parameters were expected. However, defaults were provided for most unspecified parameters. Additionally, it was not possible to structure the document, for example, by applying scoping rules to definitions and collecting related sequences of commands into single units.

The document descriptions for these early systems consist of formatting commands and layout specifications intermixed with the text (i.e., the data). This form continues to be reflected in recent pure formatters. Two different styles emerge for distinguishing commands from text. The first is to differentiate distinct command lines and data lines by a particular character in column one of the command lines; the period, as popularized by the RUNOFF system, seems especially pervasive. The second style, used in the FORMAT system, is to precede the command by a reserved *escape* character. The end of the command is marked either by some delimiting character or, if the commands are of a uniform length, after the appropriate number of characters have been encountered. The early formatters also introduced the use of reserved characters to signal actions of a limited scope; for example, "¢" in FORMAT caused the subsequent character to be capitalized.

2.1.1 RUNOFF

RUNOFF, an early, influential formatter, appeared in 1964 on the Compatible Time Sharing System (CTSS) at M.I.T. [SALT65]. With its separate companion editor TYPSET, RUNOFF accepted a document description prepared on an uppercase/lowercase typewriterlike device with

CALL FOR PAPERS

The aim of this conference is to survey the state of the art of computer aids for document preparation.

Papers are solicited on

- Picture editing
- Text processing
- Algorithms and software for document preparation and other related topics

Detailed abstracts should not exceed five pages; they *must* be sent before October 31, 1980 to the Program Chairman. Selected authors will be notified by November 30.

Duration of one presentation will be of either 25 or 45 minutes.

Figure 2. A sample document. Document descriptions specifying this document are presented in later figures.

limited capabilities; it produced formatter output for viewing on the same device. This early version of RUNOFF had only eighteen primitive, low-level operations, all oriented to formatting the visible concrete document page. This orientation is especially apparent from a list of the available objects and the manipulations which could be performed: individual lines (center, break, undent, literal), collections of lines (set line length, initiate and terminate filling and justification, indent blocks of lines), arrangement of lines in vertical space (single space, double space, leave blank vertical space), pages (headers, paper length, begin new page, print page numbers), and files (append, that is, switch to the specified file for the rest of the input). All operations dealt with the physical format of the document; none dealt with the logical content of the document.

The visible concrete document to be produced is shown in Figure 2; the appearance of the input language is illustrated by the document description shown in Figure 3. Input lines are in one of two modes, command or text. It is not possible to change the meaning of special characters. In particular, commands cannot be signaled by any character other than the period. It is also not possible to modify the actions of particular commands (e.g., to turn off automatic breaking of lines). A command parameter, when present, is an integer or string literal; no more general expressions are provided.

Despite its inflexibility, cumbersome nature, limited functionality, and commands oriented to the output page, the early RUNOFF is an important system historically. It brought text formatting to the attention of many people. Elements of its design, particularly the two-mode form of input with separate text and command lines, have been adopted by many subsequent systems. RUNOFF has continued to develop over the years, increasing both in functionality and also in the range of objects provided.

2.1.2 FORMAT

FORMAT was developed for use on the IBM S/360 computer. The first published descriptions appeared in the late 1960s [BERN68, BERN69, EHRM71]. As with RUNOFF, a text editor was provided. Although physically in the same program as FORMAT, the editing functions were, again, logically distinct. FORMAT ran in a batch-processing environment. The document description was given entirely in uppercase on punched cards; characters not explicitly capitalized were automatically converted into lowercase. The visible con-

```
.center
CALL FOR PAPERS
.space 2
The aim of this conference is to survey the state of the art of
computer aids for document preparation.
.nojust
.space 1
Papers are solicited on
.space 1
.indent 10
.undent 2
- Picture editing
.space 1
.undent 2
- Text processing
.space 1
.undent 2
- Algorithms and software for document preparation and other
related topics.
.indent 0
.space 1
.adjust
Detailed abstracts should not exceed five pages; they must be
sent before October 31, 1980 to the Program Chairman.  Selected
authors will be notified by November 30.
.space 1
Duration of one presentation will be of either 25 or 45 minutes.
```

Figure 3. Document description for RUNOFF to produce the document of Figure 2. Command lines begin with a period (.). The other lines are text lines. Since there was no significant blank character in RUNOFF (unpaddable space character), ".nojust" is invoked before the itemized listing to prevent extra spaces from being inserted into the lines. Filling of lines continues. ".adjust" restores justification. ".indent" resets the left margin, ".undent" decreases the ".indent" for the one line following. The underlined word in the next to last paragraph would have to be produced by the editor (TYPSET) before running RUNOFF since RUNOFF did not have facilities for underlining.

crete document was viewed on a line printer with uppercase and lowercase letters.

Again, as with RUNOFF, many commands manipulate concrete page-oriented physical objects: groups of lines (filling, justification, defining length) and pages (breakup, numbering, defining height, multiple columnation, specifying headers and footers). Others, however, address more logical, content-oriented, objects: words (producing alphabetical listing of words used), phrases (underlining, centering, capitalizing, horizontal spacing between sentences), and paragraphs (indenting, placing blank lines between paragraphs, eliminating widows). Unlike the treatment of characters in RUNOFF, FORMAT's operations may apply to individual characters (specifying case, overprinting). Horizontal spacing commands (tabbing) as well as vertical spacing commands are also provided.

Figure 4 presents the document of Figure 2 specified for processing by FORMAT. Three types of commands are present. The *character-level commands* are reserved characters that appear in the text but have special meanings. *Phrase-level commands* are single letters that may be grouped together. A group of phrase-level commands is preceded by the escape character ")" and terminated by a blank. Some phrase-level commands specify a particular action (e.g., terminating the current line), and others act as toggles (i.e., the first use starts an action, the next terminates it). The third type of command, the *paragraph-level command,* does not cause immediate formatting actions but establishes values for the general attributes of the document, for example, the left margin position, the page length, or the meaning associated with special characters.

```
)V
INDENTATION OF COLUMNS ON LEFT AND RIGHT IS (0,0),(5,0)
PARAGRAPH INDENT IS 0 PRINT POSITIONS
SEPARATION LINES BETWEEN PARAGRAPHS ARE 1
TABS ARE SET AT RELATIVE COLUMN POSITIONS 5
NONTRIVIAL BLANK IS REPRESENTED BY SPECIAL CHARACTER 44 (#)
GO
)M¢ CALL FOR PAPERS )M¢LLP ¢THE AIM OF THIS CONFERENCE IS TO SURVEY THE STATE OF
 THE ART OF COMPUTER AIDS FOR DOCUMENT PREPARATION. )P ¢PAPERS ARE SOLICITED ON
)LLH2W1 ##- )T ¢PICTURE EDITING )HLLH2W1 ##- )T ¢TEXT PROCESSING )HLLH2W2 ##- )T
 ¢ALGORITHMS AND SOFTWARE FOR DOCUMENT PREPARATION AND OTHER RELATED TOPICS. )HP
 ¢DETAILED ABSTRACTS SHOULD NOT EXCEED FIVE PAGES; THEY )U MUST )U BE SENT BEFOR
E ¢OCTOBER 31, 1980 TO THE ¢PROGRAM ¢CHAIRMAN. ¢SELECTED AUTHORS WILL BE NOTIFIE
D BY ¢NOVEMBER 30. )P ¢DURATION OF ONE PRESENTATION WILL BE OF EITHER 25 OR 45 M
INUTES.
```

Figure 4. Document description for FORMAT to produce the document of Figure 2. The lines following the ")V" until the line containing GO are paragraph-level commands, defining global attributes which hold until they are reset. Each symbol within the text following the escape symbol ")"and preceding the next blank is a phrase-level command which has a more limited scope of action. Some, such as ")P", the begin paragraph command, and ")L", the terminate current line command, have an immediate effect. Others, such as ")M", center phrase, and ")¢", capitalize phrase, serve as toggles. The first appearance turns the action on, the next turns the action back off. Character-level commands, represented by special symbols (¢, capitalize next character, and #, significant blank, in this document) affect the next character only. Notice that ¢ is both a phrase-level and a character-level command. Input is expected to come from cards. Characters are converted to lower case unless a "capitalize" command is in effect. The end of a line has no special significance within the input.

No macro facility is provided and it is not possible to modify the actions of particular commands. It is possible to redefine the reserved characters that invoke various character-level commands. Arguments, when present, are literals. No expressions or variables are allowed.

The treatment of the document description as one long string of characters makes direct correction of the description extraordinarily difficult. For example, ending a word in column 80 of a card requires leaving column 1 of the next card blank. Therefore, one must use the associated editor to effect any changes. Further, the document description is difficult to read as it reflects so little of the structure of the document. Some rudimentary features are provided to help handle some of the more routine writing tasks, in particular, the paragraph-level command DICTIONARY which produces an alphabetized list of the words used in the document. In this paper, features of this kind are collectively known as *writer's workbench* features. (The term "writer's workbench" was inspired by E. Ivie's "Programmer's Workbench" [IVIE77, REID80a, CHER81].)

Clearly this is an early system, inflexible and low level in nature by today's standards; for example, to produce ten blank lines, one must enter ")LLLLLLLLLL". The style of input has been designed to use the entire punched card, not for readability or ease of entry.

But again, it incorporates design features which show up quite regularly in later systems. Most visible is the embedding of commands within the text and the use of an escape character to signal the switch from the text to the command mode. The use of reserved characters or strings to initiate certain fairly short-lived actions is common in later systems. Also significant is the provision of commands which manipulate logical objects (FORMAT's *paragraph* command) and commands which provide writer's workbench features. Both of these ideas are developed substantially by later pure formatters.

2.2 The First Structured Formatters

The late 1960s and early 1970s found the development of a new generation of formatters (the *first structured formatters*) based on lessons learned from using the

early first-generation ones. Superficially, the document description still looked the same; both of the systems we discuss in this section are certainly RUNOFF descendants. However, the functions performed increased both in number and in sophistication. Ideas were incorporated from other areas of computer science. Macros provided a way to collect commonly used sequences of commands, to define new commands, and to reflect the logical structure of the document in the input. Conditional control statements, general arithmetic expressions, string and integer variables, and block structuring were borrowed from programming languages, providing structure in the input representation of the document. Writer's workbench features were added to make the formatters easier to use for the writer of a document: sections were automatically numbered, tables of contents and indices were created during formatting of the document, footnotes were properly numbered and placed, and so forth. Kaiman [KAIM68] proposed an early system which anticipated many of the developments.

It is in these first structured formatters that we see the idea that document formatting is more than just taking a sequence of words and forming them into lines which are then moved around on a printed page. Instead, the document consists of logical objects (sentences, paragraphs, sections) and the purpose of the formatter is to allow the manipulation of these objects.

Low-level primitives were still found intermixed with this higher level view of documents. While higher level commands could be created from the lower levels by using the macro definition facility, the lower level primitives remained visible to the user. The inability to hide lower level primitives is still present in current formatters.

Another significant difference between the first-generation formatters and the first structured formatters was the increasing sophistication of the document processing environment and of the available output devices. Providing a means for creating text input was no longer considered a problem which needed to be solved by the formatter. Instead, a general-purpose text editor was assumed to exist to take over this function. However, the formatting package still included the facilities for handling the different output devices. Thus although editing had been separated from formatting, viewing and formatting were still contained in the same system package.

It is interesting to compare the first structured formatters with the commercial systems produced for the VideoComp[1] phototypesetter, originating with PAGE-1 [PIER72] in the middle 1960s. These commercial systems were derived from M. Barnett's earlier work at M.I.T. [BARN65]. Like the first structured formatters, PAGE-1 borrowed many ideas from programming languages. Further, both PAGE-1 and the first structured formatters provided more sophisticated features for mapping objects into page spaces. However, applying similar ideas in different environments produced substantially different systems. PAGE-1 was intended for use in commercial typesetting and emphasized the definition of the concrete objects needed to control the typesetter, rather than the abstract objects useful in document specification.

2.2.1 PUB

PUB was developed at the Stanford Artificial Intelligence Laboratory, starting in 1971, for use on the PDP-10[2] computer [TESL72]. Its designers called it a *document compiler*, illustrating the parallel between translating a document description into formatter output and compiling programming language statements into an executable form. Initially, the output could be viewed only on a standard video or hard-copy terminal, or on a line printer. Many other viewing devices were subsequently added.

PUB's commands manipulate the same kinds of low-level objects as FORMAT: lines, pages, words, phrases, sentences, and paragraphs. Several higher level objects are also provided: columns (multiple columns of text on a page), footnotes, and sections and subsections. Sections and subsections

[1] VideoComp is a trademark of Radio Corporation of America.

[2] PDP is a trademark of Digital Equipment Corporation.

are automatically numbered and contain a heading that can also be used to generate a table of contents. Individual characters or groups of characters can be overprinted to form new characters. The REQUIRE statement can be used to cause part of the input to be taken from another file.

PUB's designers made an attempt to classify the constituent parts of some of the objects. Paragraphs are defined to consist of three parts: the "crown," the "vest," and the "hem." The crown is the first line of the paragraph, and the vest is the remainder. The hem is the last line of the vest.

A page in PUB is made up of *areas*. Areas are of two types: those which continue across subsequent pages (type *text*) and those which exist on only one page, truncating their contents when they fill up (type *title*). An area must be given a name and may be positioned arbitrarily on the page. However, at least one of the areas on each page must be named *text*. By default, a page contains three areas: two of type *title*, named "heading" and "footing," and one of type *text*, named "text." The last line of a text area is used only to eliminate widows, otherwise it is left blank.

The formatting language is similar to RUNOFF in appearance; Figure 5 provides an example specification. Some symbols and sequences of symbols have special meaning within text lines, but most actions can be redefined to be associated with a different control character.

A macro facility is provided which allows grouping of commands, control characters, and text. Macros can have arguments and may be declared to be recursive.

A number of ALGOL-like features are provided in PUB, most adopted from the SAIL programming language [VANL73]. Most notable is that of block structuring. Portions of the manuscript may be grouped into *blocks*, bracketed by BEGIN and END statements. Document parameters set by declarations within a block revert to their original values at the termination of the block. Similarly, macros and variables defined within a block hold only for the duration of the block. Another kind of grouping, the *clump*, also is provided. Clumps are bracketed by START and END statements. The main difference between a block and a clump is that declarations made in a clump continue to hold after the clump is exited. Thus, clumps are used in defining compound statements which can change the global environment.

Variables may be defined and used in other commands. Constants may be string, decimal, or octal. A number of predefined variables provide information about the document being produced. For example, CHAR denotes the number of characters printed so far on the current line; LMARG denotes the current left margin, the value of which can be changed through assignment; and DATE denotes the present date. A complete set of arithmetic and logical operations are available to allow expressions to be formed from variables and constants. Special-purpose operators, such as the unary "↑", which capitalizes its string operand, are also defined. An **if ... then ... else** statement allows conditional compilation of parts of the manuscript.

Certain identifiers can be declared to be *counters*. The value associated with a counter can be incremented and printed in any of a number of ordinal number systems. Use of counters makes it possible to refer to section numbers and page numbers symbolically within the text. PUB replaces the symbolic name with the actual section number or page number.

A special form of macro, called a *response*, is triggered by specified character sequences in the text, by changes to particular counters, or by the filling up of an indicated area. The response can be used to print page headings, define character sequences to mark the beginning of paragraphs, and to provide many other useful functions.

A document can be divided into arbitrarily named *portions* which are then processed sequentially. Portions are used to collect the information needed to generate, for example, a table of contents or a set of end notes. The SEND command is used to send text and commands to a portion. When processing reaches a portion, the portion issues the RECEIVE command to retrieve the collected information which is then processed. Since the RECEIVE command will optionally sort the collected information using provided sort keys, por-

```
.TURN ON "↓", "_", "#"
.SINGLE SPACE
.INDENT 0
.PREFACE 1
.ONCE CENTER
CALL FOR PAPERS

The aim of this conference is to survey the state of the art of
computer aids for document preparation.

Papers are solicited on
.SKIP 1
.BEGIN INDENT 3, 5, 5 ; PREFACE 1 ;

-#Picture editing

-#Text processing

-#Algorithms and software for document preparation and other
related topics
.END
.SKIP 1

Detailed abstracts should not exceed five pages; they
↓_must_↓ be sent before October 31, 1980 to the Program
Chairman.  Selected authors will be notified by November 30.

Duration of one presentation will be of either 25 or 45 minutes.
```

Figure 5. Document description for PUB to produce the document of Figure 2. PUB uses Stanford's extended version of the ASCII character set. Command lines start with a period (.). Text lines do not. The period marks a command line, not the beginning of a command. Therefore, multiple commands can be placed on a single line, separated by semicolons if necessary to prevent ambiguity, or a single command could span several command lines. Commands could also be included in the text if surrounded by "{" and "}". Each paragraph starts with a blank line which causes a paragraph break (technically, the paragraph break also ends the preceding paragraph). ONCE is a special scoping command which applied to any command means the scope of the command is the following paragraph. Thus in this example, ONCE CENTER means that the next input line should be centered. This is a specialized scoping rule. More generally, definitions made following a BEGIN are in effect until the matching END. The # represents a significant blank, the ↓_ begins underlining, the _↓ ends it.

tions may also be used to implement indexes. A special portion called FOOT is defined by the system. Footnote text is sent to FOOT for placement at the bottom of the page.

The PUB language is perhaps as much a programming language as it is a document formatting language. Certainly the document formatting features are the ones most commonly used. The programming language constructs allow implementation of many additional features. However, the implementation of extensions through macros in PUB, as well as in later systems, means that extensions to the formatting language add to the available set of commands instead of replacing lower level commands with higher level commands. Consequently, lower level commands may interact with higher level commands in unexpected ways.

One significant contribution by PUB is the incorporation of block structuring. The document specification can more directly represent the relationships between abstract objects through the use of sequential and nested blocks. Additionally, the inclusion of programming language constructs adds to the ability to extend the formatting language. The extensive group of writer's workbench tools which have been developed using PUB's powerful set of constructs is another significant contribution.

2.2.2 NROFF

NROFF is the UNIX[3] operating system's formatter, intended to produce documents on various typewriterlike terminals [OSSA74]. This formatter was developed at Bell Laboratories during the early to mid-1970s on the PDP-11, and was derived from the earlier ROFF [THOM75] which itself was derived from RUNOFF.

In this section we discuss only "bare NROFF." The many macro packages, preprocessors, and postprocessors that have been developed for use with NROFF and the closely related TROFF (for phototypesetters) are discussed in Section 2.3.

The objects supported by NROFF commands are basically the same as those supported by PUB: lines, pages, words, phrases, sentences, and paragraphs. Overprinting of characters can be used to form new characters.

Programming-language-like features have been provided but are not as general as those in PUB. For example, NROFF provides *environments* which are similar to PUB's blocks in that they allow the collection of certain document parameters. It is possible to switch to a new environment in a push-down fashion and later to restore the previous environment. However, it is only possible to define three environments (numbered 0 through 2) and environments can only be pushed down to a maximum depth of ten. Additionally, only certain attributes of the documents are actually local to the environment; many attributes are global and not affected by environment switching. Environments are also not nested: undefined local attributes are given a default value, not the value of the previously entered environment. Therefore, the concept of environment switching is quite different from PUB's block structuring and less powerful as well. The same idea is applied to input files. Input can be obtained from multiple files, which can be pushed down upon each other to a maximum depth of five.

NROFF's substitutes for variables are called *number registers* and *strings*. The values in number registers and strings can be displayed in the text, modified, used in expressions (if numeric), or invoked as commands (if strings). Predefined numeric registers provide system information which can be included in the text (e.g., the current page number and the current date).

Macros can be defined and can be recursive. Up to nine parameters can be provided on macro invocation. Conditional control statements allow selective inclusion of input text lines. Built-in condition names allow testing for such cases as even or odd page number.

It is as interesting to notice what has not been explicitly provided in NROFF as it is to notice what has been provided. Not defined are facilities for handling page headings, page footings, multiple columns on a page, or footnotes. Instead, there are more general mechanisms called *traps* and *diversions*, which, when combined with macros, can be used to implement these facilities. Traps cause the invocation of a macro at a given spot on the output page and therefore can be used to generate page headings and page footings. Diversions cause formatted text to be diverted into a macro definition which can be invoked later as a command, causing the processed contents to be treated as input at that point. Essentially, diversion provides a mechanism for defining macros containing formatted text as the body of the macro. Diversion combined with traps can be used to implement footnotes. Adding in page positioning commands allows implementations of multiple columns on a page.

The formatting language itself consists of separate commands and text. Two groups of commands are provided. The first appears on separate lines and is distinguished from text by either "." or "'" at the beginning of the line. If "'" is used, a command which would normally terminate a text line will not perform the termination. This form resembles RUNOFF quite closely in appearance; details may be seen in Figure 6. The second group of commands are flagged within a text line with the escape character "\". This group provides the same kinds of functions as do the special characters used in PUB and FORMAT. User-defined commands of the first form are written as macros, possibly with parameters. User-defined

[3] UNIX is a trademark of Bell Laboratories.

```
.ll 70
.ce 1
CALL FOR PAPERS
.sp 1
The aim of this conference is to survey the state of the art of
computer aids for document preparation.
.sp 1
Papers are solicited on
.sp 1
.in +5
.ti -2
-\ Picture editing
.sp 1
.ti -2
-\ Text processing
.sp 1
.ti -2
-\ Algorithms and software for document preparation and other
related topics.
.in
.sp 1
Detailed abstracts should not exceed five pages; they
.ul 1
must
be sent before October 31, 1980 to the Program Chairman.
Selected authors will be notified by November 30.
.sp 1
Duration of one presentation will be of either 25 or 45
minutes.
```

Figure 6. Document description for NROFF to produce the document of Figure 2. Command lines begin with ".", the remaining lines are text lines. The escape character "\" is used to give the following character special meaning. "\ ", used here, is an unpaddable space character (significant blank). ".in +5" increases the current left margin by five characters; ".in" restores it to its previous value.

With the exception of the "\ " sequence, this simple example could also be processed successfully by ROFF, NROFF's predecessor. Other mechanisms existed in ROFF to provide unpaddable spaces.

macros or strings can redefine NROFF commands, previously defined macros, or previously defined strings by reusing the name. User-defined commands of the second form are stored as strings and cannot have parameters.

Bare NROFF is an extremely low-level and difficult language to use. Parts of the language seem unintended for human use. In fact, this is probably the case; many parts are used primarily by the formatter's preprocessors.

There is no denying the immense popularity of the UNIX document-processing system. As becomes clear in the next section, this popularity is largely due to the system's ability to evolve, providing facilities to meet changing needs and becoming more powerful and convenient to use. NROFF and TROFF are the bases for this ability to adapt. Their flexibility allows the implementation of many much more usable document-processing programs.

2.3 Structured Formatters with Many Objects

In this section, we discuss three of the most interesting and influential pure formatting systems in current use: Scribe,[4] TEX,[5] and that provided by the modern UNIX system. We call these systems "structured formatters with many objects" in recognition of the increased sophistication and flexibility of the systems, particularly with respect to definition of new logical objects within the document.

[4] Scribe is a trademark of Unilogic Ltd.

[5] TEX is a trademark of the American Mathematical Society.

Each of these systems has generated much interest and discussion. Efforts are being made to prepare computer-system-independent versions of each: two separate companies have formed to market different variants of Scribe; the American Mathematical Society is preparing a portable PASCAL implementation of TEX; and the entire UNIX operating system, not just the formatters, has been converted to run on several different computers.

The functionality of these systems has increased substantially from that of the earlier pure formatters. TEX and the UNIX formatting system can include complicated mathematical equations in their documents. Table specification in UNIX is particularly easy. Objects can be integrated, especially in the UNIX system, which allows inclusion of mathematical expressions in tables and, in a recent addition, inclusion in text of line drawings which can, in turn, contain text. Each of these systems can produce output for a variety of devices; Scribe provides device-independent description, TEX produces device-independent formatter output.

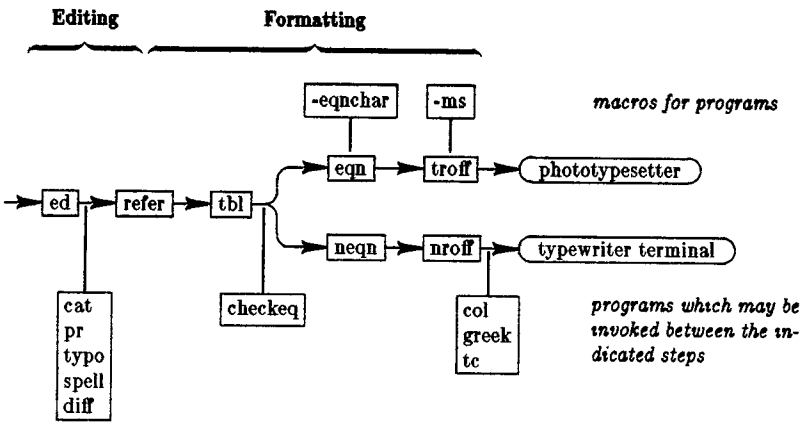
The philosophies behind the user interfaces of these systems differ greatly. The separation between TEX and Scribe is greatest, with the UNIX formatting system falling in between the two. The TEX user is viewed as being an author who wants to position objects exactingly on the printed page, producing a document with the finest possible appearance. Consequently, its emphasis is on the power and flexibility of the formatting language. It may be expected that TEX will become easier to use as new macro packages and preprocessors are developed. The Scribe user is viewed as an author who is more interested in easily specifying the abstract objects within his document, leaving the details of the appearance of objects to an expert who establishes definitions mapping the author's objects to the printed page. The emphasis is on simplicity in the input language and provision of writer's workbench tools.

Each system includes some interesting organizational and implementation details. TEX presents formatting as an optimization problem. Here, the purpose of line filling is not to fit text as densely as possible into an area, but to reduce undesirable effects such as excessive hyphenation and widows. Scribe makes an attempt to separate the content of the document from the formatting actions by using a mostly declarative language. Scribe also allows easy definitions of new environments through partial modification of existing environments. All definitions and global declarations must precede any text. Changes to the standard environments are therefore easy to detect during later modification of the document. The UNIX system is organized as a set of small programs which may be connected together in a variety of configurations. Its "building block" approach contrasts with that of TEX and of Scribe which are both implemented as large, monolithic programs.

2.3.1 *The UNIX Document-Formatting Tools*

The UNIX formatting system is a part of the larger collection of document-processing tools available within the UNIX operating system [KERN78]. Figure 7 summarizes the available tools. The formatting package, which has developed and grown substantially over the years, consists of the sibling formatters NROFF and TROFF, and of a number of macro packages, preprocessors, and postprocessors. The system is one of the first with nontrivial capabilities for formatting text, tables, mathematical equations, and, recently, line drawings. Indeed, TEX is the only other modern pure formatter with comparable formatting capabilities.

Before discussing the components of the UNIX formatting system, some general observations may be in order about the UNIX programming environment [RITC78]. The overall aim of this environment is to provide a powerful set of tools with a simple, and often extraordinarily terse, command language syntax. The expected user would seem to be an experienced professional, a person with frequent contact with the computer system. The easy interconnection of processes through the mechanism of *pipes*, which connect the output from one process to the input of another process, encourages development of systems that consist of a set of separate programs, each performing



- *ed*: line-oriented text editor
  - *cat*: list file without pagination
  - *pr*: list file and paginate for printing
  - *typo*: detect spelling errors using statistical analysis
  - *spell*: detect and attempt to correct spelling errors with dictionary
  - *diff*: compare files, generate troff commands to place marginal bars when differences found
- *refer*: generate bibliographic citations. Refer has its own separate subsystem for maintaining the bibliography data base file
- *tbl*: table formatter
- *eqn* and *neqn*: mathematical equation formatters
  - *checkeq*: make sure that equation is syntactically correct before passing it on to eqn or neqn
  - *eqnchar*. macro package specifying special characters normally unknown in troff
- *troff* and *nroff*: RUNOFF-like formatters
  - *ms*: macros for partial separation of content from format
  - *col*: convert nroff output to print on devices without reverse scrolling
  - *greek*: convert nroff output to print Greek characters on Teletype 37
  - *tc*: convert troff output to print on Tektronix 4024 DVST terminal

Figure 7. Some of the document-processing tools available on UNIX, version 7. (Teletype is a trademark of Teletype Corporation. Tektronix is a trademark of Tektronix, Inc.)

a single function. The intention is to provide a set of *software tools* [KERN76a]: programs that are continually improved by much trial, error, discussion, and redesign. When new requirements develop, the tendency is to produce a new program derived from the already existing one rather than to increase the functionality and complexity of the original. Creation of new software is preferred to modification of old since modification threatens to introduce weaknesses into previously stable parts of the system. This philosophy is reflected strongly in the organization of the formatting system as a set of distinct preprocessors and postprocessors to the central RUNOFF-like formatters. We also find different programs with similar or identical input languages producing output reflecting slightly differing requirements (e.g., EQN and NEQN for processing mathematical equations that produce input for TROFF and NROFF, respectively). We note in passing that this philosophy also seems to encourage development of an unusually wide variety of document analysis programs, such as programs for gathering statistics on word frequencies [MCMA78].

We shall now discuss several of the document-formatting tools available on the UNIX system. The most commonly used method for extension of a pure formatting language has been through macro definitions. We describe one UNIX macro package, the -ms macros, which makes a low-level attempt to provide an input language separating format from content. We also discuss four TROFF preprocessors: EQN, which formats mathematical expressions; TBL, a table formatter; REFER, which looks up bibliographic references and generates TROFF commands to produce a properly formatted citation within the text; and PIC, which allows string descriptions of line drawings.

2.3.1.1 TROFF/NROFF. In Section 2.2.2 we have discussed the functions available in NROFF, the UNIX formatter producing output for typewriterlike devices. Now we wish to discuss TROFF [OSSA76], which prepares output for phototypesetters.

The input languages accepted by TROFF and NROFF are nearly identical. Thus, all of the discussion about NROFF also applies to TROFF. TROFF must support additional functions since a phototypesetter has more capabilities than does even the most sophisticated typewriterlike printer. However, NROFF typically ignores those TROFF commands which it cannot carry out (for example, changing character sizes) and thus maintains input language compatibility. There are a few commands which are present only in TROFF or only in NROFF but, since it is possible to determine which formatter is being used while the formatting is going on and conditionally to include or exclude input lines, it is always possible to set up an input file which will be acceptable to both formatters. However, it will, at times, take quite a bit of work to set up this file. Thus, a weak form of device-independent description is provided by these formatters.

TROFF and NROFF have been modified over the years to support special functions needed by the various preprocessors. The documents which TROFF produces can be typographically very complex. Text, mathematical equations, tables, and line drawings can all be specified and combined. TROFF and NROFF may be considered to be useful primarily for implementation of higher level document-specification languages. Using TROFF directly to set complex documents is more complicated than almost anyone would wish. Macro packages and preprocessors are essential for effective use [KERN76b].

2.3.1.2 The -ms Macro Package. Let us now look at one of the available NROFF/TROFF macro packages, the -ms macros [LESK76b]. Objects supported here are simple but certainly higher level than those provided by the bare formatters. They include indented and unindented paragraphs, footnotes, section headings, indented outlines, and blocks of text which are to be kept together within a column of text. Commands are provided for changing fonts, for increasing or decreasing type point size, and for specifying the number of columns on the page.

We include one example document description, specified using the -ms macros, as Figure 8 (we later show descriptions of this same document in Scribe, Figure 16, and in GML, Figure 20). One interesting object used in the example is a document heading. It consists of document title, authors' names and affiliations, and document abstract. The positioning of this object with respect to neighboring objects varies with the type of document being produced. In fact, fields of this object can appear in more than one place in the final printed document. If "released paper format" (.RP) had been specified at the beginning of the -ms input file, a separate cover page containing the document header object and the current date would have been generated. The title and author information would have been repeated on the first page of the text. Thus abstract objects can be represented multiply in the concrete form of the paper and some abstract objects can be unordered with respect to their neighbors.

As the example illustrates, NROFF/TROFF commands are needed to augment the -ms macros with even simple text. Setting more complicated text requires that the values in registers used by macros be altered by commands in the text. The values within these registers can be device dependent, for example, the width of a col-

```
.TL
Extended Abstract
.br
Document Formatting Systems:  Survey, Concepts, and Issues*
.AU
Alan Shaw, Richard Furuta, and Jeffrey Scofield
.AI
Department of Computer Science
University of Washington
Seattle, WA  98195, U.S.A.
.AB
Formatting, the final part of the document preparation process, is
concerned with the physical layout of a document for hard and soft
copy media....  Our aims are to characterize the formatting problem
and its relation to other aspects of document processing, to
evaluate several representative and seminal systems, and to
describe some issues and problems relevant to future systems.
.AE
.FS
*This research was supported in part by the National Science
Foundation under grant number MCS-782685.
.FE
.NH
The Formatting Problem
.PP
In order to discuss formatters and their functions and to
distinguish formatting from other aspects of document
preparation, it is convenient to use an
.I
object
.R
model of documents [Shaw 80], somewhat analogous to that in
programming languages.
.PP
A document is an object...
.NH
Representative and Seminal Systems
.NH 2
Pure Formatters
.PP
Some typical first generation formatters...
```

Figure 8. A document description using the -ms macros. This figure shows input to either NROFF or TROFF using the -ms macros. The document specified is the first part of the extended abstract for this paper [SHAW80b]. The case of commands is significant. Uppercase commands are defined by the -ms macros. Lowercase commands are NROFF/TROFF commands. The title of the document is placed between the ".TL" and the ".AU" commands. The NROFF/TROFF command ".br" (break) was necessary to separate the text "Extended Abstract" from the rest of the title. Authors' names, between ".AU" and ".AI", and authors' address, between ".AI" and ".AB", follow. The text between ".AB" and ".AE" is the paper's abstract. The title, authors' names, authors' address, and abstract will be placed on the first page of the text, formatted properly based on conventions established within the macro package. For example, the title will be centered and underlined when the formatter is NROFF, centered and written in a larger point size in boldface when the formatter is TROFF. The footnote, located between ".FS" and ".FE", could not be placed in the header since the header information is treated specially. ".NH" defines a section heading which will be numbered ("1." and "2." in this example), ".NH 2" a subsection heading, also numbered (2.1. here), and so on. ".PP" defines the beginning of a paragraph. Text following ".I" is set in italics. ".R" restores the normal (roman) font.

$$\frac{1}{2\pi}\int_{-\infty}^{\sqrt{y}}\left(\sum_{k=1}^{n}\sin^2 x_k(t)\right)(f(t)+g(t))\,dt$$

Figure 9. A sample mathematical equation [KNUT79c, p. 91]. Figure 10 shows specification of this equation in EQN. Figure 18 shows this equation specified in TEX.

```
.EQ
1 over {2 pi} int from {- inf} to {sqrt y}
left ( sum from k=1 to n sin sup 2 x sub k (t) right )
left ( f(t) + g(t) right ) dt
.EN
```

Figure 10. The equation of Figure 9 specified in EQN. Text enclosed in brackets, "{" and "}", is *grouped* and syntactically treated as if it were a single unit. "sub" means subscript, "sup" means superscript, "left (" and "right)" bracket a group which is surrounded by parentheses large enough to enclose the group's contents. Notice that EQN will automatically set function names, for example "sin," in a roman font instead of in the italic font used for the other textual material in the finished equation.

umn or the spacing between lines of text. Some separation of format from content has been achieved, but the separation is not complete.

With UNIX macros, the entire underlying implementation language remains visible during formatting. Indeed, use of the underlying language may be necessary to achieve certain effects. The syntax of the newly defined commands is fixed by the semantics imposed by the formatter on macro invocations. Further, since macro commands are implemented by grouping commands from the base formatting language, the commands which can be implemented are limited by the functionality of the formatter itself. Another approach, which may be used to provide additional commands, is to create a "filter" program, one through which the input passes before reaching the base formatter. While still limited by the functionality of the base formatter, this approach allows definition of a syntax appropriate for the problem being solved, and allows hiding of parts of the base formatting language. This is the approach used with great success by the various NROFF/TROFF preprocessors.

2.3.1.3 EQN. EQN, a TROFF preprocessor (and the related NEQN, an NROFF preprocessor) provide a high-level declarative language for specifying mathematical equations within TROFF-prepared documents [KERN75].

Objects specified using the language are viewed as being enclosed in rectangular boxes. The language specifies the relationships between boxes. Thus, larger boxes are built from smaller boxes. An equation specification in EQN is quite aural in form. An EQN specification of the equation shown in Figure 9 is presented in Figure 10. As this example shows, the EQN specification is close to what would be recited by a person reading the equation from left to right.

The equation specification is delimited by the .EQ and .EN commands. Equations can also be specified within a text line if surrounded by defined delimiter characters. Reserved words are used within the specification to indicate relative positioning of the object-containing boxes, to specify symbols not present on the keyboard, and to identify parts of the equation requiring different typographic settings. Spacing of the equation description is not significant except where necessary to delimit a reserved word. Reserved words can be defined or redefined by the user through a limited macro definition facility.

EQN equations are easily included in tables and in text. It is possible to use TROFF strings within an equation specification, but not TROFF commands. However, this is

not really a limitation since the EQN language can be used to specify almost any desired equation.

The language is defined by a context-free grammar and implemented using a compiler-compiler. Some of the benefits of this approach are discussed in Section 3.3.

2.3.1.4 TBL. TBL, the UNIX system's table preprocessor, defines a simple, nonextensible, declarative language which allows specification of fairly complex tables [LESK76a]. The TBL language specifies rectangular tables with entries which may be numeric or textual (either short phrases or formatted blocks of text, possibly including mathematical equations). Any entry within the table may be enclosed with a box or separated from adjacent objects with either horizontal or vertical rules (these rules may be either double or single). In fact, the table itself may be enclosed with a box. Adjacent table entries (again, either horizontally or vertically) may be merged to form a single, larger entry. Certain low-level font and type size changes may be specified within this language.

The model of tables used in this system is an object which consists of a sequence of rows which are divided into columns. Row templates are used to describe the positioning of entries within the columns (or, as mentioned, within a sequence of the adjacent boxes defined by the row and column divisions). The templates used now differ from those in the original version of TBL, which used column templates.

The table of Figure 11 is specified by Figure 12. Table definitions in the TBL language consist of a line of global options, a sequence of line templates (also called the table format), and a sequence of lines defining the rows of the table. Blocks of text to be formatted over several lines can be included within the table as in

```
Column 1 information ⊕ T{
A multiline block of text to be
formatted by TROFF and
placed in the table as the second column
T} ⊕ Column 3 information
```

Columns are separated by the tab character, represented by ⊕. Table entries can include NROFF/TROFF commands. EQN equation specifications may also be included. Either EQN or TBL can be run first without altering the results, although efficiency considerations dictate that TBL be run before EQN. This flexibility is possible since TBL acts by generating NROFF/TROFF commands and macros; the actual calculation of the values needed for formatting the table is delayed until NROFF/TROFF is invoked. Most NROFF/TROFF commands work properly within the TBL definition, but some formatting commands that alter environmental attributes used by TBL will have unforeseen effects. With this exception, text and mathematical equations have been integrated with tables.

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Figure 11. A sample table [LESK76a, p. 7]. Figure 12 shows specification of this table in TBL. Figure 19 shows this table specified in TEX.

2.3.1.5 REFER. The function of the NROFF/TROFF preprocessor REFER [LESK78] is to retrieve a particular citation from a centrally maintained bibliography database given an imprecise form of the citation. When the citation is found, strings are generated which NROFF/TROFF macros (for example, -ms) use to print the complete reference and to insert the appropriate citation into the text. By default, citations are numeric. The -ms macros place the references in footnotes; the citation itself is the superscripted number that refers to the footnote. The appearance of the citation can be changed by redefining the macros. Also, some REFER options allow other citation and reference formats to be specified. REFER options can indicate that references are to be collected, not alphabetized, and then listed at the end of the text. Citations can be numeric, can consist of the senior author's last name and the date, or can consist of the first *n* initials of

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
Year⊕Price⊕Dividend
1971⊕41-54⊕$2.60
2⊕41-54⊕2.70
3⊕46-55⊕2.87
4⊕40-53⊕3.24
5⊕45-52⊕3.40
6⊕51-59⊕.95*
.TE
* (first quarter only)
```

Figure 12. Document description for TBL to produce the table of Figure 11 [LESK76a, p. 7]. The table begins with the ".TS" command and ends with the ".TE" command. Note that the final line is outside the body of the table itself. Global options, specified by keywords, are declared once, at the beginning of the table definition, between the ".TS" command, which marks the beginning of the table, and the terminating ";". The global option "allbox," used here, causes each entry in the table, and the table itself, to be enclosed within a box. Other global options can, for example, cause the table either to be centered within the available horizontal space or to expand in width to fill the available space.

Row templates, which follow the global options and are terminated with a ".", consist of codes which represent characteristics of column entries within the row. A single line of codes is given for each row in the table, one code per column. When there are more rows in the table than templates, the last template given holds for the remaining rows. Templates used here specify that an entry is to be centered within a field (c), that the previous entry in the row should span into the current field (s), and that numbers are to be aligned at their decimal points (n). Left justification, right justification, vertical spanning, and centering of blocks of left adjusted text can also be specified. Templates can also indicate that horizontal and vertical lines be drawn between entries, alter the font used, and so on. New templates can be defined in the middle of a table, changing the table's format.

The remainder of this table definition is data to be entered into the table using the formats defined by the templates. Extra columns in the text are ignored. Rows are entered one at a time with columns separated by the tab character, represented as ⊕ in this figure. Vertical and horizontal rulings between individual entries and rows, and vertical spanning may also be specified at this time, if desired.

the last name. Authors' names can be printed in the references with last name first or first name last.

The input document contains imprecise citations consisting of a sequence of key words and authors' names. A citation of the report defining REFER could look like this:

```
.[
Lesk inverted indices UNIX 1978
.]
```

REFER would find the complete citation in its database and generate the appropriate strings to allow the NROFF/TROFF macros to place a complete reference and proper citation into the text. A separate language is used within the database to specify entries, as shown in Figure 13.

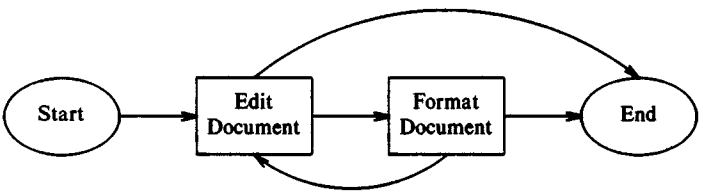
REFER is particularly interesting for two reasons. First, it serves a purpose quite different from the other NROFF/TROFF preprocessors. The other preprocessors all provide languages for describing new objects simply. REFER provides what we have called a "writer's workbench" tool. Second, the implementation is of interest. The central bibliography available at Bell Laboratories contains over 4000 entries. Searching this large central database for references would be prohibitively expensive without an efficient means. Inverted indices provide this efficiency. Briefly, an inverted file [KNUT75], which contains the inverted indices, is like a book index: the values of the attributes within the records are the lookup keys, and keys point to the records in which they are contained. The Bell Laboratories implementation uses a precomputed hash table for quick retrieval of the lookup keys.

2.3.1.6 PIC. A recent addition to the UNIX document-processing system is the picture specification language PIC [KERN81a, KERN82]. This one-dimensional string language, implemented as a TROFF preprocessor, provides a way to specify line drawings, possibly with enclosed text, equations, or other TROFF specified material, within typeset documents. Thus PIC adds an important class of figure objects to those expressible within the UNIX system and partially integrates the other mathematical and text objects with these figure objects.

PIC's primitive objects are the box, line, arrow, circle, ellipse, arc, and B-spline. Heavy reliance on associating symbolic names with objects, on describing positions relative to other objects, and on default values for object size, orientation, and other attributes, allows a simple, flexible, and

```
%T Document Formatting Systems:  Survey, Concepts, and Issues
[Extended Abstract]
%A A. C. Shaw
%A R. Furuta
%A J. Scofield
%R Technical Report 80-10-02
%D October 1980
%I Department of Computer Science, University of Washington
%C Seattle, WA
```

Figure 13. A REFER database entry. Each field of the entry is flagged by a two-symbol code: the character %, followed by a letter indicating what the field is (for example, "A" for author, "T" for title). See the references for a full listing of this entry [SHAW80b].




```
.PS
ellipse "Start"; arrow
B1:box "Edit" "Document"; arrow
B2:box "Format" "Document"; arrow
E2:ellipse "End"
arc -> cw from top of B1 to top of E2
arc -> cw from bottom of B2 to bottom of B1
.PE
```

Figure 14. An example using the PIC language. Specifications in PIC consist of the name of a primitive object (ellipse, box, and arc in this example), followed by optional specification of object attributes. Text contained in quotes is displayed, centered inside the object. Specifications are either written one per line or separated by semicolons. By default, the figure is assumed to grow from left to right. Names followed by a colon label the following object, allowing symbolic references to the location of the object. The position of the objects could also have been specified without labels by using ordinal values. For example, the first arc could have been described as

arc –> cw from top of 1st box to top of last ellipse

The two arcs specified here are drawn in a clockwise fashion (cw) and contain an arrowhead(–>).

high-level description of figures. It is, however, possible to specify the concrete attributes of the object in an exact, low-level manner. Objects may be scaled, translated, or placed with respect to an identified point in a figure. Invisible objects can be used to help in specifying figures. A macro facility is provided, allowing creation of a hierarchy of figure objects. Additionally, objects can be collected together into a *block* (syntactically represented by delimiting a sequence of specifications with "[" and "]" brackets). The block may then be manipulated as a single object. Figure 14 presents an example.

The approach taken to picture specification is similar to, but at a higher level than, that used in the Lawrence Livermore system, discussed in Section 2.5.2. IDEAL, another TROFF preprocessor for figure de-

scriptions using quite a different approach, is discussed in Section 2.5.3.

2.3.1.7 Discussion of the UNIX System. By any standards, the UNIX formatting system is a successful one. An informal evaluation conducted for *Physical Review Letters* compared UNIX composition to the typewriter composition method already in use at the journal for articles with a moderate amount of mathematical and tabular text [LESK77]. UNIX reduced the keyboard time needed to prepare the articles, averaging 2.4 times as fast as typewriter composition. Further, total estimated composition costs per page decreased by one third with UNIX.

As previously indicated, a large part of the success of this system is its unique ability to change incrementally to meet more sophisticated demands. Paradoxically, the system's highly modular design has also discouraged integration between different types of objects and between the languages used to describe these objects. Section 3.7.2 continues this point.

2.3.2 Scribe

We find a different approach to document processing in Scribe, developed in the late 1970s by B. Reid at Carnegie-Mellon University [REID80a, REID80b, REID80c, REID81]. In the pure formatters already discussed, the user of the system retains most of the responsibility for the appearance of the final printed document. In Scribe, this responsibility is given to the formatter. The emphasis in the document description language is on the logical content of the document, not on its physical format. Formatting details are determined by the system, varying for individual types of documents and for different output devices. Other systems permit their users to specify the logical structure of the document, most generally through use of commands defined using the system's macro facility, but the user can also specify the concrete form of the document directly through the low-level positioning commands which remain available for use. The Scribe user is required to specify the logical structure of the document. For the most part, lower level positioning commands have not been provided.

One result of this philosophy is that the object types which the Scribe system can handle are limited to those which can currently be described completely by their content, without resorting to use of concrete object positioning commands. In essence, this means that Scribe currently is restricted to fairly simple textual objects. There are no facilities, for example, for line drawings, complex tables, and complicated mathematical expressions. TEX and the UNIX document formatting system attempt to allow any object to be specified, although perhaps with great difficulty. Unlike these, Scribe contents itself with the easy specification of objects sufficient to provide many, but certainly not all, of the commonly needed document types.

Another result of this strong separation of content from format is that document descriptions stated using Scribe are highly portable. Not only can the same document description be used at different computer sites, but the same document description can be used to produce visible concrete documents for viewing on different devices; the necessary details are supplied by the Scribe system.

2.3.2.1 Environments and Commands. Declaring a Scribe document to be of a particular type (e.g., article, letter, thesis) specifies the attributes of the global environment for the document. An environment encloses document text, which itself may include a sequence of nested environments (e.g., italic phrase, or section heading). Another way of describing an environment is as a partial definition of the concrete attributes for the logical object it contains (e.g., left margin, or typeface). Unspecified attributes are generally inherited from the surrounding environment. Formatting, then, involves applying the environment attributes to the text contained within the environment. The definitions of the attributes of the environments provided for each individual document type are contained in the central *Scribe database.* The person responsible for maintaining the Scribe database at a particular site may add new document types to the database or remove old ones. Similarly, environment definitions in existing document types can be added, deleted, or modified. Importantly, this

```
@Style(indent=0,spacing=1,spread=1)
@Heading(CALL FOR PAPERS)

The aim of this conference is to survey the state of the art of
computer aids for document preparation.

Papers are solicited on
@Begin(Itemize)
Picture editing

Text processing

Algorithms and software for document preparation and other related
topics
@End(Itemize)

Detailed abstracts should not exceed five pages; they @i(must) be
sent before October 31, 1980 to the Program Chairman.  Selected
authors will be notified by November 30.

Duration of one presentation will be of either 25 or 45 minutes.
```

Figure 15. Document description for Scribe to produce the document of Figure 2. Environment and command keywords are preceded by the escape character @, which cannot be changed, and are optionally followed by delimited arguments or delimited text. The delimiters can be just about any matched pair of brackets. The position of keywords on the input line is not significant. Paragraphs are flagged with a blank line. The @Style command modifies global attributes of the document. If a @Style command is included, it must appear at the beginning of the input file before any text is encountered. Here, the "indent" argument specifies how much a paragraph should be indented, "spacing=1" means the document should be single spaced, "spread" indicates how many blank lines should be left between paragraphs. The "@i" environment contains a text string to be italicized. An equivalent way to specify any environment with delimited string argument is to use @Begin and @End command brackets. Thus the italicized string "@i(must)" could equivalently have been specified as "@Begin(i)must@End(i)".

means that the particular environments which are available within a particular document depend on the type of the document; different document types may provide different environments.

Text within an environment may be simply a string of characters, a paragraph containing a sequence of sentences, or a sequence of paragraphs. Text objects are available within all environments since these objects are defined internally by the Scribe compiler, not in the database.

A number of low-level features for creation of new objects are available, such as overprinting of characters and a macro facility. However, these features are not meant for the general user and are not described in the basic language tutorial [REID80b].

We present two examples of Scribe input. Figure 15 presents the same document already presented for the other pure formatters. Figure 16 is included to give an idea of how Scribe would be used to specify a more generally needed document type.

Keywords, preceded by the reserved escape symbol @ and followed, optionally, by a delimited argument, name either environments or commands. Environments have been discussed above. Commands differ from environments in three major ways. First, they are generally associated with a point in the document rather than with a region of the document. They may also associate information with that point in the document. For example, the command "@Label(*LabelName*)" marks a particular location in the document, saving the section number and output page number for retrieval in other parts of the document. Second, the actions associated with commands are *hard-wired* into the Scribe system. The actions cannot be modified as can the list of attributes associated with environments.

```
@Make(article)
@Center(Extended Abstract)
@Heading(Document Formatting Systems:  Survey, Concepts, and Issues@foot<This
research was supported in part by
the National Science Foundation under grant number MCS-7826285.>)
@Center(Alan Shaw, Richard Furuta, and Jeffrey Scofield)
@Center(Department of Computer Science
University of Washington
Seattle, Washington  98195, U.S.A.)

@PrefaceSection(Abstract)

Formatting, the final part of the document preparation process, is
concerned with the physical layout of a document for hard and soft
copy media....  Our aims are to characterize the formatting problem
and its relation to other aspects of document processing, to evaluate
several representative and seminal systems, and to describe some
issues and problems relevant to future systems.

@Section(The Formatting Problem)

In order to discuss formatters and their functions and to distinguish
formatting from other aspects of document preparation, it is
convenient to use an @i(object) model of documents@Cite(ShawModel),
somewhat analogous to that in programming languages.

A document is an object composed of a hierarchy of more primitive
objects....

@Section(Representative and Seminal Systems)

@SubSection(Pure Formatters)

Some typical first generation formatters...
```

Figure 16. A sectioned document specified in Scribe. This figure presents the same document segment as that specified in Figure 8. Documents of type "article" may be divided into numbered sections to three levels. (The first two, Section and SubSection, are shown here.) This example document consists of a title, authors' names and address, an abstract, and portions of the first few sections of the document. "@Cite(ShawModel)" refers to an entry in the bibliographic database for this document. In the printed document, a citation to the reference will be placed in the text at this point. Scribe will automatically generate a list of references and a table of contents for this document.

And third, many of the commands are procedural in nature. The environment mechanism is declarative.

A particularly useful feature of the Scribe system is its facility for defining and modifying environment definitions. These changes may be either global or local in scope. Global changes can only be specified at the beginning of the document specification, before any document text is encountered. The @Define command is used to define a new environment globally. For example, the command

@Define(Unfilled, Justification=no)

defines a new environment, named *Unfilled*, in which text is not justified. More usefully, @Define can be used to create new environments defined by *analogy* to existing environments: "The environment I want to specify is exactly the same as this existing environment except for these changes...." For example, the *Quotation* environment, used for displaying large quotations, narrows both its margins. The command

@Define(QuoteButFullRight=Quotation,
RightMargin=+0)

provides a new environment, named *QuoteButFullRight*, with the same attri-

butes as *Quotation* except for the right margin, which is not narrowed. Similarly, existing environments can be globally altered through the @Modify command which allows new values to be specified for attributes of an existing environment. The command

@Modify(Itemize,RightMargin=0)

would change the RightMargin attribute associated with the *itemize* environment in every place in which the environment was used within the document.

Local changes to an environment are made using the @Begin command. For example,

@Begin(quotation,RightMargin=0)

changes the RightMargin attribute for this use of *Quotation* only. Unfortunately, local environment changes can introduce subtle output device dependencies into the document description since many environmental attributes are defined in extremely device-dependent ways. We return to this subject again in Section 3.2.3.

2.3.2.2 Writer's Workbench Features. A major factor in the popularity of the Scribe system is its large number of writer's workbench tools. Many of these tools were also found in PUB and a few in the UNIX system. Scribe has a mechanism for collecting text during the processing of a document and then treating this derived text as input to the formatter at the end of the run. This mechanism is used, for example, to generate a table of contents or an index. Objects such as sections of a document, footnotes, or elements in an itemized list can be numbered automatically. A cross-reference facility allows symbolic reference to page numbers or section numbers of objects within the document. Scribe provides page layout aids which assist in placing footnotes and which will move a figure forward until enough blank space is found on a page to insert it. Similar aids are also available in most of the other pure formatters discussed. Additionally, individual document types may include useful options. For example, selecting the "Draft" option can cause diagnostic information related to the document description to be included within the visible concrete document.

A bibliography management facility is available. No central bibliography database is provided; each user maintains a personal bibliography containing specifications for a set of references. The form of entries in this database quite closely resembles that used in REFER (see Section 2.3.1.5 and Figure 13). In Scribe, the user defines a unique identifier for each entry in the bibliography. Citations within a document use this identifier to select the desired reference. The Scribe system fills in the text of the actual citation in an appropriate format for the document type. At the end of the document, a list of references is generated, also in an appropriate format for the document type. Citations and references can be generated in a number of different formats, such as those of the *Communications of the ACM* and *Information Processing Letters.* Unlike REFER, Scribe completely regenerates its internal bibliographic lookup tables each time they are needed.

Finally, Scribe includes some basic facilities for managing large documents. Large documents can be broken up into a number of different computer files and ordered into a tree structure. Global definitions made in the root apply to all nodes. Any subtree can be formatted separately without requiring that the entire document be processed. Auxiliary files are updated so page numbers, figure numbers, and cross references, for example, in other parts of the document will be correct the next time they are processed. This feature not only aids the individual author who is working on a large document, but also aids groups of authors who are each working on separate sections of a document. We discuss these facilities further in Section 3.5.2.

2.3.3 TEX

TEX was developed by D. Knuth of Stanford University in the late 1970s to provide high-quality typesetting of books containing much mathematical material [KNUT79a]. The system concentrates on the arrangement of objects in the visible concrete document. New and interesting algorithms have been developed for breaking paragraphs into lines, for collecting lines into pages, and for hyphenation—for the

tasks, in other words, which are normally performed automatically by a formatter. The specification language allows the description of extremely complex textual, tabular, and mathematical concrete objects.

The specification language appears to emphasize the expression of this wide range of objects rather than ease of use. Some ideas have been borrowed from other systems; most notably, the math mode language resembles EQN. However, the specification languages are more unified in TEX than in the UNIX system. In particular, features of TEX's math mode language, such as ellipses, are used in nonmathematical specifications more often than are features of EQN. On the other hand, the assessment by one of the implementors of the UNIX document-processing system is that TROFF is more powerful than TEX [KERN81b], since TEX does not provide page layout mechanisms as general as TROFF's traps, which can associate a macro invocation with a position on the output page. Another difference is that TEX does not include primitives for specifying line drawings as do the newer versions of TROFF [KERN81c].

2.3.3.1 The Boxes and Glue Model. Concrete objects are modeled as two-dimensional *boxes* connected to each other by *glue.* Boxes define the size of the object they contain and provide a reference point which is used to align the box with other boxes, either horizontally or vertically. Aligning two boxes produces an enclosing box. Typical box contents are characters, words, lines, paragraphs, and pages. Glue provides space between boxes. Glue has a natural size and may be stretched or compressed according to given constraints. Line justification may be thought of as pulling the objects on the ends of the line apart, or pushing them together, until the desired width has been reached. Since the glue put between sentences has more stretchability than the normal glue placed between words, line justification will cause more space to be added between sentences than between words. Similarly, a phrase can be centered within a line by placing glue of infinite stretchability at both ends of the phrase.

2.3.3.2 The Formatting Language. The formatting language uses control sequences preceded by a special escape character, usually \. Other characters also have special meaning, but all special characters can be redefined, including the escape character. Specifications are *grouped* if surrounded by set brackets, { and }. Definitions made within a group persist only until the end of the group, unless they are defined to be global. Thus one common use of groups is to specify a change with limited scope, say, a switch to an italic font. Because a group is treated as a unit, groups are also often used as arguments to commands. Both of these uses may be seen in the text example presented in Figure 17.

A separate math mode is provided for specification of mathematical equations. In-line mathematical equation specifications are delimited by the character $. Displayed mathematical specifications are delimited by $$. Figure 18 shows a specification for the displayed equation previously presented in Figure 9.

TEX's math mode language is similar to EQN's. The primary difference beween the languages is that TEX uses no reserved words; escape sequences are used instead. Special symbols are used, however, to specify some operations such as superscripting, invoked by ↑, and subscripting, invoked by ↓. Overall, TEX math mode and EQN seem quite close in their abilities to specify complicated equations. However, using TEX math mode requires more knowledge of typographic conventions than does EQN. In the example, the TEX math mode user must remember to type "\sin" since typographic convention indicates that function names are to be set in a different typeface from variable names.

A macro facility is included which allows definition of new commands. TEX macro definitions are somewhat unusual when compared to those of other formatting systems in that they offer a limited facility for defining new command syntaxes, similar to that provided in PUB by the response macros. As with the other systems, the body of the TEX macro definition includes a sequence of formatting language specifications that defines the macro's action. However, a definition also includes a pa-

```
\input basic % defines the standard macros, formatting parameters
\parskip 10pt
\parindent 0pt % no indentation
\def\yskip{\vskip3pt}
\def\textindent#1{\noindent
   \hbox to 19pt{\hskip0pt plus1000pt minus 1000pt#1 }\!}
\def\hang{\hangindent19pt}
\hsize 4in
\ctrline{\bf CALL FOR PAPERS}
\vskip 24pt
The aim of this conference is to survey the state of the art of
computer aids for document preparation.

Papers are solicited on
{\parskip 0pt
\par\yskip\textindent{$\bullet$}\hang Picture editing
\par\yskip\textindent{$\bullet$}\hang Text processing
\par\yskip\textindent{$\bullet$}\hang Algorithms and software for
document preparation and other related topics}

Detailed abstracts should not exceed five pages; they {\sl must} be
sent before October 31, 1980 to the Program Chairman.  Selected
authors will be notified by November 30.

Duration of one presentation will be of either 25 or 45 minutes.

\vfill % fill out rest of page with space
\end
```

Figure 17. Document description in TEX specifying the document of Figure 2. Text following a percent sign (%) is commentary and is ignored by TEX. The first seven lines of the specification establish macros and formatting parameters. "\parskip" defines the space which is to be left between paragraphs and "\parindent" the indentation at the beginning of each paragraph. The definitions of "\yskip", "\textindent", and "\hang" are adapted from Appendix E of the TEX reference manual [KNUT79c, p. 165]. "\yskip" will leave a small amount of vertical space. "\textindent" and "\hang" will be used in specifying lists of items flagged with a bullet in the left margin. "\hsize" establishes the document's line width.

The text of the document begins with line nine. Notice the difference in line nine in syntax between a group used as an argument to a command or macro, in this case as argument to "\ctrline" which centers the argument on the line, and a group used to limit the scope of a formatting parameter, here "\bf" which switches to a bold face font. "\vskip" specifies vertical blank space. "\noindent" inhibits indentation of the first line of the following paragraph. A blank line terminates the preceding paragraph, contributing its lines to the current page; the "\par" command could have been used instead. Notice that the measurements expressed in these specifications are stated in points (a point is 0.013837 inch) and therefore are highly oriented to the visible concrete document.

rameter pattern with embedded argument placeholders. When the macro is invoked, tokens in the invocation string are matched against tokens in the parameter pattern. Tokens corresponding to the embedded argument placeholders are substituted into the definition's body, which is then evaluated.

This macro facility is a powerful tool for simplifying and extending the specification language. Indeed, much of the "basic" language is implemented within a macro package, as may be noted in the example of Figure 17. The American Mathematical Society has sponsored creation of another macro package, called AMS-TEX,[6] designed to make specification of mathematical papers in TEX easier [SPIV80].

Tables are handled as text. Two commands of particular use in defining tables are "\halign" and "\valign". The group fol-

[6] AMS-TEX is a trademark of the American Mathematical Society.

```
$$ {1 \over 2\pi} \int\limitswitch↓{-\infty}↑{\sqrt y}
\bigglp \sum↓{k=1}↑n \sin↑2 x↓k(t) \biggrp
\biglp f(t)+g(t)\bigrp\,dt $$
```

Figure 18. The equation of Figure 9 specified in TEX [KNUT79c, p. 149]. This specification is extremely similar in form to that in EQN. See Figure 10 and the text for discussion. "\limitswitch" causes the limits to be placed above and below the integral sign. By default (in this example, the default would have been used if the specification had been ".. \int↓{−\infty}..."), limits are placed to the right of the integral sign. "\bigglp" and "\biggrp" are particular parenthesis characters somewhat larger than "\biglp" and "\bigrp", which themselves are slightly larger than the standard left and right parenthesis. Spaces have been added to improve readability, but only those separating control sequences from subsequent letters are actually required.

lowing a "\halign" contains, first, a (horizontal, hence the "h") row template, and then a sequence of row entries to be specified using the template. See Figure 19 for an example table specification. The "\valign" command performs much the same function except that a (vertical) column template is given and specifications are by column, not by row.

Again, the division of a TEX table specification into two parts, template and entries, is similar to the UNIX specification. Use of the formatting language to specify table templates rather than special characters, as in TBL, means that TEX's language is more general. However, TEX's table specifications are quite a bit more complex than are TBL table specifications, and the TBL language is easier to use. Once again, macro packages will undoubtedly be developed to make TEX table specification much simpler.

2.3.3.3 Line and Page Breakup. Among the most important contributions of TEX are its concrete document model and the algorithms used in the system's implementation. The TEX reference manual [KNUT79c] includes a description of the hyphenation routine, a list of the modes TEX gets into while processing a document, and a brief discussion of the methods used for breaking paragraphs into lines and for making lists of lines into pages. This last topic has been described in more detail in a later paper [KNUT81]. In essence, TEX tries to determine the "best" way to break each paragraph into lines, using a dynamic programming algorithm, where "best" means the way with the least hyphenation and with the glue settings that result in the least amount of "badness." The badness of a glue setting is high if the glue has to be stretched or compressed to a point close to its limits. The badness associated with a particular point can be affected manually by specifying a "penalty" for a break occurring at the point. If the penalty is negative, then the break is favored; if positive, then the break is discouraged. A similar algorithm is used in placing lists of lines onto pages. Here, TEX tries to avoid ending a page with a hyphenated line and tries to avoid isolated lines on the top or the bottom of a page.

2.4 Integrated Editor/Formatters

In this section we discuss those systems which combine the features of an interactive text editor with those of a document formatter, a group which we call the integrated editor/formatters. We do not discuss those systems which are primarily editors with a few formatting functions included. EMACS [STAL80, STAL81] is one of the more complicated systems of this kind.

These systems are divisible into two broad categories. In the first, represented here by QUIDS, the objects used in formatting have been integrated with those used in editing, but the editing and formatting functions have not been integrated. Only occasional viewing of the visible concrete document is permitted. In the second, both objects and functions have been integrated. Editing changes are shown directly on a representation of a visible concrete document. We describe four systems exhibiting this kind of integration: Bravo,

```
$$\vbox{\tabskip 0pt
\def\|{\vrule height 9.25pt depth 3pt}
\def\.{\hskip-10pt plus 10000000000pt}
\hrule
\hbox to 150pt{\|\.AT&T Common Stock\.\|}
\hrule
\halign to 150pt{#\tabskip 0pt plus 100pt
⊗\hfill#⊗#⊗\ctr{#}⊗#⊗\hfill#⊗#\tabskip 0pt\cr
\|⊗\.Year\.\hfill⊗\|⊗\.Price\.⊗\|⊗\.Dividend\.\hfill⊗\|\cr
\noalign{\hrule}
\|⊗1971⊗\|⊗41--54⊗\|⊗$\$2.60⊗\|\cr\noalign{\hrule}
\|⊗2⊗\|⊗41--54⊗\|⊗2.70⊗\|\cr\noalign{\hrule}
\|⊗3⊗\|⊗46--55⊗\|⊗2.87⊗\|\cr\noalign{\hrule}
\|⊗4⊗\|⊗40--53⊗\|⊗3.24⊗\|\cr\noalign{\hrule}
\|⊗5⊗\|⊗45--52⊗\|⊗3.40⊗\|\cr\noalign{\hrule}
\|⊗6⊗\|⊗51--59⊗\|⊗.95\spose*⊗\|\cr\noalign{\hrule}}
\vskip 3pt
\hbox{* (first quarter only)}}$$
```

Figure 19. Document description for TEX to produce the table of Figure 11 [KNUT79c, p. 108]. The designer of this table has decided that the table is to be 150 points wide (slightly under 2.1 inches). The table specification may be divided into four major parts. The first three lines provide some overall definitions of parameters and macros available within the specification. The next three lines specify the major heading of the table. The next eleven lines specify the body of the table. The final two lines specify the text which is to appear beneath the table as a footnote to the rightmost number in the last row of the table.

The group making up the table body contains two parts. The first defines a template which will be used in placing the seven column entries which make up each row (the bars adjacent to the three columns, specified as "\|", are considered to be separate column entries). Specifications for column entries are separated by the alignment tab, ⊗. The template ends with the first "\cr". Row specifications follow, each ended with "\cr". Row entries are separated by the ⊗. In essence, entries are substituted into the template, externally replacing the corresponding #. The "\noalign{\hrule}" which follows each row entry specifies the horizontal bar separating the rows in the visible concrete document.

Star, and Smalltalk, all developed by Xerox, and the Wang Word Processor. The use of multiple viewing windows and high-quality graphics devices by the Xerox systems is also of interest.

Two general observations may be made comparing the integrated editor/formatters to the pure formatters. The first is that the integrated editor/formatters tend to be more configuration dependent than do the pure formatters. The range of devices used by the different systems is quite wide, ranging from standard CRT terminals to bit-mapped displays with associated graphical input devices. Unlike the recent pure formatters, each system's design seems to be heavily influenced by the environment in which it operates. The second observation is that the sophistication of the objects used in these systems is less than that of those used in the pure formatters, especially when compared to those in the group we called the "pure formatters with many objects." None of the integrated editor/formatters provides objects at the abstract level used in Scribe or allows the careful control over the appearance of the visible concrete document provided by TEX. However, integrated editor/formatters are being developed which use these kinds of abstract and concrete objects. They are discussed in Section 2.6.

2.4.1 QUIDS

One of the first published descriptions of a system which combined editing functions with formatting functions in a unified manner was that of QUIDS (QUick Interactive Documentation System), designed and implemented in the mid-1970s at the University of London [COUL76]. QUIDS' editing functions are oriented to document text rather than to computer program text.

Consequently, the basic editing unit is the paragraph, not the line. Additionally, the system allows incremental viewing of the visible concrete document on request during preparation of the document description. The system integrates functions to edit, format, view, file, and print documents. However, the complexity of the object types permitted is quite limited. The system can only be used for very simple textual objects; no mathematics, line drawings, or other more complex objects can be represented or manipulated.

The model of the document employed within QUIDS is a sequence of abstract objects: paragraphs, tables, section and subsection headings, and associated titles. Most of these objects are logically ordered into a tree and assigned numbers based on the path from the root of the tree: section headings are numbered 0, 1, 2, 3, . . . ; subsection headings 1.0, 1.1, . . . ; and paragraphs 1.0.1, 1.0.2, and so on. These numbers are used within the system to identify the particular objects. Other objects represent nonsequential text, for example, page headings and footnotes. Low-level formatting parameters, such as those establishing the width of the margins, are also specified by commands and stored within the internal form of the document.

The system uses a standard CRT terminal as the interactive device. The QUIDS language consists of commands divided into three groups. Initially, the user is in "⟩" mode ("⟩" is the prompt displayed in this mode). The user types commands to edit an existing document (positioning within the document), file or print the document, or select one of the other modes. In "*" mode, commands can be entered to specify parameters for global options controlling the formatting of the document; for example, whether or not a title is printed at the top of each output page. In the "'" mode, the user enters a command to select one of the abstract object types (e.g., paragraph or section heading) and then enters the document text associated with the object. The "'" mode commands also specify the low-level formatting parameters mentioned above.

As might be noted from this discussion, commands are not selected from menus displayed on the screen. The user must remember what commands are possible and in which mode they may be used. Additionally, it is clear that a more sophisticated hardware configuration with a pointing device could be used to advantage. Significantly, the user does not manipulate a direct representation of the final document, but instead alters a logical representation of the document. Views of the final document are only presented when requested.

This simple, limited system has combined the editing and formatting functions and integrated the objects manipulated by each of these functions. However, a distinct separation of commands relating to each of these functions has also been retained; in particular, the "'" mode provides formatting commands and the "⟩" mode provides editing commands. Unfortunately, the formatting commands provided in this system are low-level in nature and oriented to the visible concrete document. The editing commands, however, operate at a higher level, representing the document as a structured set of ordered abstract objects. This separation contrasts strikingly to the approach taken in the Xerox systems, discussed below, in which editing and formatting have been more completely integrated.

2.4.2 *Alto, Bravo, and Star*

The Xerox Alto is a personal computer/workstation developed in 1973 [THAC79]. It includes an 8.5 by 11-inch bit-mapped display with a resolution of about 70 pixels per inch, a typewriter keyboard, and a positioning device called the mouse. Over the years, a number of important and influential document preparation systems have been developed which take advantage of the special input/output capabilities of this workstation.

One of the most influential of these systems is the integrated text editor/formatter Bravo [LAMP78]. This editor/formatter cleanly combines editing, formatting, viewing, filing, and production of hard-copy text documents. An option allows direct editing of an exact representation of the visible concrete document; the results of editing on the appearance of the visible concrete document are reflected immediately in the display. Multiple display windows are used to allow simultaneous manipulation of dif-

ferent documents or of different parts of the same document.

A limited number of object types are provided: characters, words, lines, paragraphs, and documents. Editing operations act either on individual objects or on a sequence of objects of one of these types. Objects are selected by positioning a cursor which is controlled by the mouse. Associated with character and paragraph objects are concrete attributes called *looks*. Looks are formatting properties that define the appearance of the object. Thus, character looks describe the character's font, its size, and its baseline (to allow superscripting and subscripting). Paragraph looks describe the shape of the text in the paragraph, for example, the margins, the space between lines, if the paragraph is justified or centered, and the default character looks for characters within the paragraph. Looks are not always visible; only their side effects, the visible concrete objects, are normally seen. Looks can be modified; modification of an object's looks alters the appearance or positioning of the object on the display.

An interesting idea, which is used in many commercial systems but has been implemented quite generally in Bravo, is the partial specification of document types using *forms* (templates). Forms are document skeletons with appropriate looks, headings, and other components already in place, and with textual indicators describing those fields that must be provided by the user. Forms are particularly useful for standard document types, such as business letters, interoffice memos, and technical reports, where much of the formatting information and some of the components (e.g., headers) are predefined. Creating a document of a particular type simply involves replacing the fields in the template with the actual document text; the retained looks in the form assure the proper formatting.

Bravo only provides for manipulation of simple text. Mathematical expressions, figures, and footnotes are not included, but there is provision for paging, page headers and footers, and up to two columns. There is little structuring of the objects in the text: looks are associated only with particular objects and are not related to each other. Thus making uniform changes to a document is difficult. In particular, changing the appearance of concrete objects associated with a particular abstract object type involves individual modification of each instance of the abstract object type. It is not generally possible, for instance, to change the font of all section headings in a document with one command; one must do each change individually.

Documents can contain more than just simple text, however, since a number of drawing packages are available that allow creation of figures to be merged into Bravo documents. Markup [NEWM78] adds both freehand drawings and figures constructed with straight lines. It can also be used to add text to drawings produced by other packages. Draw [BAUD78] is used for drawings which require precise placement of curves, as well as lines and text, within a figure.

Alto, Bravo, Markup, Draw, and other research systems developed in Xerox laboratories have provided the experimental basis for a number of commercial products. The most recent and interesting of these is the office workstation called Star, announced in 1981 [SEYB81, SMIT82]. This product is an integrated office system that provides document preparation, filing, electronic mail, and data-processing functions, all within a uniform command syntax and interpretation. Star's machine has many improvements over the Alto, such as a larger high-resolution bit-mapped display.

Document preparation in Star involves the direct manipulation of the visible concrete document, but with a wider range of objects than Bravo. Document objects include mathematics and line graphics with shading. Star also features multiple overlapped display windows, object *properties* which are a generalization of Bravo's looks, and pictorial symbols or *icons* for representing all system objects on the display screen. Examples of system objects represented by icons are documents, file folders, file drawers, in and out baskets, disks, printer devices, and object directories. A command typically involves the selection of an object (actually of the icon representing the object) with a mouse and the invocation of an operation by further object selections or keyboard entry.

2.4.3 Smalltalk

Smalltalk [GOLA76, GOLA83, INGA78, SHOC79, BYTE81] is neither a formatter nor an editor, but an interactive programming language and system based on object classes and instances, and on message passing. Developed and used in an experimental research setting originally on the Xerox Alto computer (described in the previous section), it has demonstrated the usefulness of class/instance language facilities in a number of editing, formatting, and related applications, and has been a productive test bed for interactive techniques on a bit-mapped display screen. This work has influenced several modern systems, such as Star and the systems presented in Section 2.6.

Editors for creating and modifying a wide variety of different objects, including text, freehand drawings, and character fonts, have been constructed. Formatting and viewing are integrated with editing: the resulting user interfaces deal with concrete objects, and the screen layouts are closely associated with the object class and instance definitions. The same language, Smalltalk, is used both for programming objects and for invoking them; that is, the interactive user language and the extender language are the same. A particularly useful systems feature is the subclass/superclass mechanism through which class attributes may be inherited. This permits a new class, a subclass, to be defined by modifying and extending a previously defined class, the superclass.

The user interface contains an interesting window package that permits the definition and use of any number of screen windows simultaneously; the "active" window may overlap inactive ones in screen space, analogous to a sloppy stack of sheets of paper. This feature appears particularly applicable to the document preparation environment. Several parts of the same document or several documents can be viewed simultaneously and processed, by being displayed in their own windows. These and other Smalltalk features have strongly influenced the design of Star's user interface.

One particularly interesting system written in Smalltalk is ThingLab [BORN79, BORN81], which can be used to manipulate simulated objects whose interactions are governed by constraints. For example, a rectangle containing text may be constrained so that the text completely fills the rectangle. If the user changes the width of the rectangle, the height is automatically adjusted and the text rejustified so that the text still completely fills the rectangle; similarly, a change in the amount of text will cause a corresponding change in the size of the rectangle. The constraints may be very general; each constraint description includes a number of methods that may be used to satisfy it. The system is also able to satisfy some circular constraints. Although ThingLab is not a document-processing system, its constraint techniques could be useful in future systems for expressing and solving some formatting problems.

2.4.4 The Wang Word Processor

The Wang Word Processor is one of many commercial formatting systems that became available in the late 1970s. It is a self-contained, multiuser system with a dedicated processor and peripherals. Its design stresses ease of use, achieved through an integrated editor/formatter with a simple set of commands. Editing operations are applied directly to a representation of the concrete document that is continuously displayed on a CRT device. Text and commands are entered by single keys on a keyboard or by selection from a menu, and hard-copy output is produced for a typewriter terminal or a phototypesetter.

Most commands are entered by single keystrokes, and reside in the document as special characters. For example, indentation is accomplished by a special indentation key that inserts an "indent" command character into the document. Other, more global formatting commands are applied to an entire section of the document, usually interactively under the control of the user. For example, the system handles pagination and hyphenation by displaying each division point and requesting the user to make a decision about the division to be made. In all cases, the effects of the commands are immediately visible in the concrete representation.

The Wang system deals with a small set of the most useful abstract objects: words,

phrases, paragraphs, and page headings. It allows operations on these objects such as hyphenation of words, centering or underlining of phrases, and filling or justification of paragraphs. A number of lower level commands are also available to allow the construction of other objects. For example, characters may be placed as superscripts or subscripts, and special commands may be used to control horizontal and vertical spacing. As usual, the system places all these objects into lines and pages.

In addition to operations that introduce local formatting actions, there is a set of concrete attributes associated with each page (expressed in a template language) that determines its global characteristics, such as line and page length, tab settings, and interline spacing.

The set of commands may be extended through a general macro facility called a *glossary* that allows sequences of keystrokes to be named and called when desired. This facility includes limited recursion and conditional statements, and hence is quite powerful. Glossary entries are created just as is any document, using the full power of the system. However, there is no way to alter the behaviors of the built-in commands or any notion of variables or expressions.

Although this system offers only modest formatting capabilities, it appears responsive and easy to use. The integration of formatting and editing may help somewhat to make up for the lack of more sophisticated features. For example, although the system cannot automatically determine how to hyphenate words, it can produce hyphenations fairly painlessly by performing them interactively.

On the other hand, the commands and objects of the system are all at a rather low level. The lack of a higher level structure makes it difficult to restructure the document automatically after it has been changed. For example, the addition of a few new phrases may require that the entire hyphenation process be repeated for long sections of the document. In Section 2.6 we describe systems that attempt to retain the flexibility of integrated editing and formatting, while also making document restructuring easier by maintaining information about the high-level structure of the document.

2.5 Other Systems

In this section we present four interesting, unrelated systems. KATIB/HATTAT, a pure formatter, formats and typesets documents in Arabic script, perhaps the most difficult alphabet to typeset. The TRIX/RED formatter is contained within an extensive document-processing system which allows composition of quite elaborate documents containing intermixed color graphics (figures), text, and mathematical equations. IDEAL, a TROFF preprocessor, is a language for textually describing two-dimensional figure objects using a system of simultaneous equations to define the relationships between significant points in the figure object. And GML, implemented as a macro package for a RUNOFF-like pure formatter, includes a high-level declarative document specification language and many writer's workbench tools.

2.5.1 KATIB and HATTAT

The programs KATIB and HATTAT, written in the mid-1970s by P. MacKay of the Department of Classics at the University of Washington, formatted and typeset documents using the Arabic and the Roman alphabets [MacK77]. Arabic script writing is extraordinarily complicated. The shape and size of each letter is highly context sensitive, depending not only on the surrounding letters but also on the entire word in which the letter appears. Thus, while there are only 29 separate letters in Arabic (and no differing uppercase and lowercase), a high-quality typesetting job requires that more than 900 separate symbols be used. As MacKay wrote in 1977 [MacK77]: "The normal policy of every Orientalist journal in North America, even if it will still consent to print Chinese, cuneiform, or hieroglyphic, is to refuse all Arabic script text. Arabic is not merely 'penalty copy,' it is prohibited copy."

MacKay's system consists of two separate programs. KATIB (which means "writer" or "scribe" in Arabic) performs the page-formatting functions. HATTAT ("calligrapher" in Arabic) specifies the pen-

strokes to be used in forming the characters which KATIB has placed on the page. Importantly, it is HATTAT which determines the actual shape of the individual characters. KATIB only estimates the size of the characters by using the average value of the possible forms. Clearly, formatting and viewing have been separated in this system. KATIB also handles details of intermixing Arabic text (written from right to left) with Roman text (written from left to right). Input is entered in the Roman alphabet, from left to right; it is not necessary to type English or Latin text backward. Arabic letters are represented phonetically. Output from HATTAT is then processed by a phototypesetter.

The formatting language is surprisingly general, but unfortunately assembly-language-like in appearance. The language is based on one designed by David Packard [PACK73] for a system which handled intermixed English, Latin, and Greek text. Numeric variables may be defined, assigned values, used (along with constants) in expressions, and tested in conditional statements. Synonyms (i.e., macros), each with a two-character name, may be defined and invoked within the text by preceding the name with a reserved escape character.

2 5.2 TRIX/RED

The document preparation system at Lawrence Livermore Laboratories [BEAT79], developed in the middle through late 1970s, produces visible concrete documents, in color, containing text, mathematical equations, and graphics for display on high-resolution output devices. The system, consisting of a group of separate programs, can be logically divided into four parts: TRIX, which contains a distinct text editor (TRIX/AC) and text formatter (TRIX/RED); PCOMP, which compiles picture descriptions written in the string language PICTURE, producing low-level graphics primitives; TV80LIB, a set of routines allowing applications programs to generate figures for inclusion in the document; and REDPP, which merges the various outputs of the preceding parts into a single output stream, directed to a specific output device. Thus, two different languages are defined, one for text, the other for pictures.

The document description processed by TRIX/RED is RUNOFF-like in appearance with separate text and command lines. In some cases, the argument for a command may be several lines long. A special delimiter line is used to mark the end of a multiline argument. Mathematical equations can be defined either in the picture language (by drawing them) or with TRIX/RED. When TRIX/RED is used, components of the equation are first defined, then combined into larger parts, and finally displayed. For example, a fraction could be produced by first defining the numerator and the denominator; then defining the fraction to be the numerator placed over the denominator, separated with a line; and then finally directing that the composite object be displayed. This is a crude form of nested boxes. Unfortunately, the language for specifying equations is quite cumbersome. A limited form of nesting of environments is available for low-level typographic directives (e.g., selecting fonts, or point sizes). A macro facility, permitting text and numeric arguments, is available to allow extension of the language.

The PICTURE language, processed by PCOMP, is a context-free language with reserved words. The language, while not very powerful, is able to describe a useful range of figures. Primitive objects are lines, circles, and other geometric forms. Their position is specified within a coordinate space. Other attributes, such as radius of a circle, can also be specified. Simple text-formatting operations are available within the language, so text objects can be included within a picture. All objects can be colored, filled in, rotated about an axis, and scaled. No control structures (e.g., iteration, conditionals, or macros) are available. PICTURE language statements can either be embedded in the file processed by TRIX/RED or maintained separately. However, since TRIX/RED and PCOMP are separate programs, integration of the output from the two is awkward: the user of the system must specify to TRIX/RED how much space the picture will take and cannot use TRIX/RED commands within PICTURE language input.

Figures and drawings can also be generated by applications programs through calls

to routines in TV80LIB. While PCOMP is not interactive, some of these applications programs are, so interactive picture editing is possible.

The final part of the package, REDPP, merges the output from TRIX/RED, PCOMP, and TV80LIB routines, producing an output file for display on a particular device. Formatting and viewing have been separated in this system. TRIX/RED performs the page formatting, producing a representation which is device independent. REDPP performs the viewing function.

Again, as with UNIX, the organization of this system into separate programs has both advantages and disadvantages. We discuss these further in Section 3.7.2. Still, this is an ambitious system, certainly one of the few to treat characters as picture objects which may be colored, rotated, and scaled.

2.5.3 IDEAL

C. Van Wyk has developed an interesting one-dimensional (string) language for specifying line drawings in a document [VANW80, VANW81]. In this language, an object class is defined in two parts, a declarative section and an instruction section. The declarations specify the relations, or constraints, that must hold among the points of the object. These relations lead to a system of simultaneous equations that the points must satisfy. The instruction section of the object definition gives instructions for connecting points and for drawing other objects by invoking or calling them. When an object is invoked, additional relations, equations, and instructions may be inserted in the call. These "parameters" further specify the equations and must result in a unique solution for the point variables. At this stage, the instruction part may be executed with the solution points, drawing, for example, points, lines, text, circles, and rectangles.

The language has been implemented in C as a TROFF preprocessor called IDEAL. While it requires perhaps too much mathematical sophistication for general use, the language may be practical with an interactive or less mathematical user interface. It is significant chiefly because of its methods for declaring and solving constraint equations.

2.5.4 GML

The Generalized Markup Language (GML) was developed by C. F. Goldfarb of IBM over a period of years in the early to mid-1970s. GML first became available for general use in 1978 and is now part of IBM's Document Composition Facility [IBM80a, IBM80b, GOLC81a, GOLC81b).

GML is a pure formatter, implemented using macros written for the SCRIPT formatter. SCRIPT is a RUNOFF-like pure formatter first developed in the late 1960s [MADN68, IBM80c]. GML provides high-level declarative specifications, called *tags*, which are associated with points in the document text. Figure 20 contains more information about the specification language. Notice that the commands of the underlying implementation language (SCRIPT) remain available for use.

GML also incorporates many desirable writer's workbench features, such as automatic numbering of list elements, chapters, and footnotes; symbolic referencing to page numbers or other numbers associated with parts of the document; and facilities for collecting and formatting information to be included in a table of contents and in an index.

2.6 Some Current Developments

We wish to present three experimental systems still under development and not yet completely specified to conclude our discussion of representative and seminal systems. All combine the idea of high-level declarative object specification, taken from some of the recent pure formatters, with the idea of continuous viewing of the visible concrete document, as in some of the integrated editor/formatters.

2.6.1 JANUS

JANUS [CHAM81, CHAM82] is an integrated editor/formatter under development at the IBM Research Laboratory in San Jose. JANUS uses a work station with keyboard, joystick, and two screens. One screen is used to show the specification of a document in a declarative specification language; the other shows the corresponding page of the visible concrete document as it would appear if printed. The two screens

```
:frontm.
:titlep.
.se tl = 'Document Formatting Systems:  Survey, Concepts, and Issues'
.se ea = 'Extended Abstract'
:title.&tl. [&ea.]:fnref refid=funds.
:fn id=funds.
This research was supported in part by the National Science Foundation
under grant number MCS-7826285.
:efn.
:author.Alan Shaw
:author.Richard Furuta
:author.Jeffrey Scofield
:address.
:aline.Department of Computer Science
:aline.University of Washington
:aline.Seattle, Washington  98195, U.S.A.
:eaddress.
:etitlep.
:abstract.
:p.Formatting, the final part of the document preparation process, is
concerned with the physical layout of a document for hard and soft
copy media....  Our aims are to characterize the formatting problem
and its relation to other aspects of document processing, to evaluate
several representative and seminal systems, and to describe some
issues and problems relevant to future systems.
:body.
:h2.The Formatting Problem
:p.In order to discuss formatters and their functions and to distinguish
formatting from other aspects of document preparation, it is
convenient to use an :hp1.object:ehp1. model of documents [Shaw 80],
somewhat analogous to that in programming languages.
:p.A document is an object composed of a hierarchy of more primitive
objects....
:h2.Representative and Seminal Systems
:h3.Pure Formatters
:p.Some typical first generation formatters...
```

Figure 20. GML specification to produce the sectioned document of Figure 16. The figure presents the same document segment as that given in Figures 8 and 16. This GML specification uses the "starter set" tags. Other sets would provide different tags. Tags begin with a colon and end with a period. They consist of a tag name followed by an optional list of attribute-value pairs. Attributes either provide additional information (e.g., a short form of the title) or provide formatting parameters. A text argument follows the period. Pairs of tags which are defined as delimiting a multiline argument are flagged with "." and ":e" respectively.

This document description consists of two major parts: the front matter, beginning with the ":frontm." tag, and the body, beginning with the ":body." tag. The body and the front matter are formatted differently. The front matter contains a title page which is delimited by ":titlep." and ":etitlep." tags. The SCRIPT ".se" command, used in the third and fourth lines of the description, associates the document text to the right of the "=" with the symbol name to the left. "&tl." retrieves the string associated with the symbol "tl". Note that the period in "&tl." is part of the specification and not part of the text. Symbols are used in this example since the argument to the ":title." tag must fit onto a single line. The ":fnref refid = funds." tag retrieves the number of the identified footnote, here the one tagged with ":fn id = funds." which contains an id attribute corresponding to that used in the footnote reference. The actual implementation of the "starter set" tags prohibits inclusion of footnotes within the front matter, although the implementation could be changed to permit them. ":p." tags a paragraph. ":h2." and ":h3.", used in the body, specify text for section and subsection headings.

can be thought of as being two separate fixed-size windows on different representations of the document. Editing is performed on the document description rather than on the representation of the final document. Published material does not indicate how JANUS will aid the correlation of information on the two screens. Correlation may be awkward unless the system's user interface is carefully designed.

The description language is closely related to GML, associating high-level, declarative tags with particular locations in the document. The current prototype implementation uses PASCAL language procedures to define new tags.

A JANUS document is specified as a collection of *galleys*. One galley might contain document text and another might contain footnote text. Points in different galleys are marked as corresponding to each other. This allows a footnote, for example, to be correlated with its reference in the body of the text. The correlation is used in placing material from different galleys on the same physical page. The actual placement of galley material onto physical pages is done using information contained in *page templates*. There may be a number of page templates associated with any particular document, for example, a title page template, a body page template, and a template for the appendixes. Each template indicates where the material from each of the galleys contributing to the page may be placed, and specifies certain "fixtures" such as page headings.

JANUS also allows the user to point to a particular object on the page (say, a figure) and drag it to a new location. The rest of the page is reformatted accordingly. This feature permits local overriding of placement decisions made by the formatter and is implemented by creating a special page template to represent the manually altered page. Consequently, a manually repositioned item on a page will remain in its new location, even if surrounding material is reformatted. Further discussion of issues raised by this feature is in Section 3.5.1.

2.6.2 Etude

Etude [GOOD81, HAMM81a, HAMM81b, ILSO80], an integrated editor/formatter being implemented at M.I.T., uses a bit-mapped terminal. A Scribe-like model of document structure is combined with an internal model based on the boxes and glue of TEX. A document page consists of a collection of page spaces. Objects placed in these page spaces are obtained from one or more *subdocuments*, a concept closely resembling JANUS galleys. As in Scribe, Etude document type definitions are collected into a database. The editing language uses English-like commands. Special keys are associated with the more commonly used commands. A help facility is provided and a menu of commands is produced on request.

The document is displayed using four windows. One window displays paginated text in final form. Associated with this window is another window containing format information. This information window is placed at the margin of the text window. The displayed format information corresponds to the adjacent line in the text window. The third window serves as an interaction window, displaying prompts and echoing typed input. The fourth window shows the system's status.

2.6.3 PEN

PEN [ALLE81], under development at Yale University, presents another possible organization for an integrated editor/formatter. It differs from JANUS and Etude in its scope and goals. Rather than build a complete prototype of a future system, PEN presents a smaller experimental test bed. Like Etude, PEN includes a Scribe-like hierarchical model to describe the abstract structure of a document and a TEX-like boxes and glue model to describe the relationships between concrete objects. Unlike JANUS and Etude, PEN's document model does not incorporate pagination, thus allowing for a simpler formulation.

One of the interesting aspects about a PEN document is its tree representation. For textual material, the internal modes of this tree represent a hierarchy of objects within the document, for example, chapter, section, and paragraph. Internal nodes are instances of a template for the object they represent. Further, each node's type in-

cludes a specification of the type and number of those nodes which can be its children, thus placing constraints on the legal relationships among objects in the tree. Leaves of the tree contain primitive objects. In the case of text, these primitive objects are expressed using the boxes and glue model. A Smalltalk-like model of object invocation is used. Editing and formatting operations on a node are carried out by asking the node to perform the operation. It is the node's responsibility to perform the operation in an appropriate fashion; the action associated with a particular operation varies depending on the node to which the operation is applied.

One portion of the formatting problem which has been investigated in more detail is the specification of mathematical formulas. PEN includes a specification language called PEN-MATH. Since PEN-MATH is based on APL, it allows concise specification of mathematical structures such as arrays and sequences. PEN views the objects described by a PEN-MATH specification as being entirely contained within a leaf of the tree (these objects are not directly incorporated into PEN's object hierarchy). However, PEN-MATH extends PEN's object-oriented structure. In particular, parameters (called *looks*) may be passed to an operator, altering the way in which the operator displays itself and its operands. Thus the specification for a multiplied by b, $a \times b$, may be displayed as $a \times b$, $a \cdot b$, or ab depending on the parameters passed to the operator.

3. ISSUES AND CONCEPTS

The previous sections have identified a number of issues and concepts that suggest ideas for further research and that should also be of use in the design and evaluation of formatting systems.

3.1 Document and Processing Models

A formatter is easier to understand and to design if it is based on a consistent model of documents and of the operations used in processing them. Current systems offer some interesting and useful models, but much development remains to be done.

3.1.1 Document Models

Because the notion of classes and instances is a powerful means of characterizing sets of related objects, a document model like the class/instance model described in Section 1.1 seems to be a natural choice. A further advantage is that this is an integrated model of abstract and concrete documents. Existing models have tended to be either concrete or abstract, but not both.

3.1.1.1 Abstract Models. The underlying form of the model presented in Section 1.1 is tree structured, as may be seen in the example class ⟨ExtendedAbstract⟩. However, the notion of ordered and unordered subtrees allows a flexibility of expression not present in strictly tree-structured models such as those of the XS-1 system [BURK80].

One limitation of any tree-structured model is that it cannot directly represent all the necessary relations among the objects in documents, since some violate the nesting restrictions of trees. Such relations are rather common, since parts of a document very often refer to other parts by name, by section number, or by page number.

A model of abstract documents that does not have this limitation is a generalized graph structure, such as the one used in the Hypertext Editing System (HES) [CARM69, VAND71], in the NLS system [ENGE68, ENGE73, VAND71], and in PIE [GOLI80, GOLI81]. By assigning particular structural meanings to the links in the graph, this model can be used to represent any relations among the objects of a document. For example, the relation between a footnote reference in the main text and the footnote to which it refers may be modeled in this way. This model could also be used to represent desired spatial relations among concrete objects.

Since trees are more comprehensible than general graphs, and since many documents are primarily tree structured, it may be more desirable to use a tree-structured model and to include general relations among objects as subsidiary information. For example, Scribe allows references between objects, although they are not di-

rectly included in Scribe's essentially tree-structured model.

It should be noted that neither HES, XS-1, nor PIE is chiefly a formatting system. In fact, no formatting system today offers a sufficiently explicit model for abstract documents. Scribe's notion of "document types" and PEN's tree structure come closest to this goal.

3.1.1.2 Concrete Models. A model for concrete documents must deal with two-dimensional components of the page. In general, the details of particular concrete primitives, such as characters, are hidden by considering them to reside inside simpler figures, such as rectangles or parallelograms. The model developed for EQN, with nested and juxtaposed rectangular boxes, has proved simple and very natural. The more refined model used in TEX, which places glue between the boxes, is the most complete model for concrete documents that has been implemented and tested.

Current experimental systems, such as Etude, JANUS, and PEN, may help to determine whether TEX's model can be made even more useful by being integrated into a higher level model for concrete documents. Etude and JANUS attempt to provide a high-level concrete document that consists of a set of related *galleys* to be placed into definable concrete page spaces. All three systems attempt to integrate this concrete model with a model for abstract documents.

3.1.2 Processing Models

Models for the processing of documents are rather diverse. The traditional processing method that was used in all early formatters accumulates characters into lines and pages, and takes appropriate action when these spaces threaten to overflow; these actions are taken immediately, without looking ahead into the document. This model has proved far too limited to offer flexible control over the appearance of the final document, largely because of this lack of look-ahead.

The syntax-driven model of the formatting process is based on the parsing of a context-free language. This model has been used for two slightly different purposes. In the first case, the syntax is that of an already existing language (in general, a programming language). This approach has been used to reformat programs into a more legible form, and has the characteristic that the input and output are syntactically identical strings. In the second case, the syntax is that of a language used to describe the objects of the document, and the resulting concrete output is very different from its input description. This approach is used in EQN and in the math mode of TEX. Although syntax-driven techniques are very powerful when applied to languages with easily described grammars, they cannot readily be applied to textual objects such as paragraphs, or other objects with less regular structure.

The processing model used in TEX for paragraph layout, described in Section 2.3.3, is the most satisfactory to date. However, since the use of penalty values to control the concrete appearance of objects is not based on a direct statement of the desired appearance of the document, the choice of penalty values appears to be a task requiring a fair amount of experience. A more direct way of specifying the desired appearance would be even more useful.

The use of constraints to specify desired properties of objects, as is done in IDEAL and in ThingLab, leads to a processing model in which the system attempts to satisfy all constraints simultaneously. The equation-solving technique used in IDEAL is quite powerful, although rather stringent restrictions must be placed on the types of equations allowed. Further, this technique is clearly limited to problem domains that are essentially numeric. The more flexible technique used in ThingLab includes equation solving as a particular case, but is much more general and may be used with constraints that are not numeric.

The development of document and processing models must be undertaken together, since they interact very strongly. For example, an attractive document model may be rejected if it cannot lead to good models for the processing of the document.

3.2 Formatting Functions

3.2.1 Kinds of Objects

Formatters must have facilities for dealing with many kinds of objects. Current sys-

tems provide a variety of textual objects, a few systems offer mathematical and tabular objects, and fewer yet offer pictorial objects.

There are many other useful specialized object types that need to be formatted, such as musical notation, chemical diagrams, chess positions, and crossword puzzle diagrams. In each case, it should be possible to take advantage of the structure of the objects to simplify their specification. One challenge for future general-purpose formatters will be to provide such new and useful object types, to allow users to create their own object types, and to define a uniform framework in which objects of all types may be used.

3.2.2 Composition of Objects

Documents consist of simple objects combined into more complex ones. For the most part, this structuring is conveniently done at the abstract level. That is, the user should describe the abstract object composition of the document, and the formatter should transform this structured abstract object into a structured concrete one.

There are two levels of definition that may be used in the construction of abstract documents. The first is the creation of new classes and subclasses. A user may want to define an entirely new document class such as "business letter," or may want to create a subclass by further specifying an existing class. An example of a subclass of the "business letter" class would be a "form letter" class with a fixed body. Instances of "form letter" would need to supply only a recipient, as the body would be supplied by the subclass. This notion of classes and subclasses is similar to that of Smalltalk.

The creation of new classes and subclasses is not allowed in all systems, and in systems where even a partial facility is provided (Scribe, TEX, TROFF), it is often so difficult that it is not intended for the casual user. In addition, few systems provide a mechanism for encapsulating the details of new classes. In many cases, the behavior of classes is controlled by global variables that may be inadvertently changed, either directly or through the use of conflicting low-level commands. Since the creation of new classes and subclasses is often the easiest and most natural way for a user to specify a desired document (or, even more so, a series of similar documents), ways of simplifying this task should be developed. This topic is discussed further in Section 3.3.3.

The second level of definition used in the construction of abstract documents is the definition of instances of an already defined class, for example, the definition of a particular paragraph and its body. All formatters allow this sort of definition, although it is not always thought of in the terms used here.

Structural information about classes of objects can be used by formatters to govern the composition of abstract objects and to simplify the user's task. Class information can be used, for example, to ensure that objects are correctly formed or, in interactive systems, to suggest prompts for parts of the structure to be entered next. Classes can also supply default values for attributes of their instances. This can substantially simplify the creation of new objects, allowing the user to concentrate on only the aspects of objects that distinguish them from the default for the class.

Some limited structural checking is performed by Scribe, which defines different kinds of object substructures depending on the type (i.e., the class) of the document being constructed. The PEN system has formalized this notion and performs even stronger checking. No other modern system offers functions of this sort. In TROFF, for example, abstract objects are most often bracketed by pairs of opening and closing commands, but TROFF does not check that such opening and closing brackets are properly nested, or that the enclosed material is of the correct type.

3.2.3 Abstract-to-Concrete Mappings

All present formatting systems severely limit the ways in which abstract-to-concrete mappings are performed for structured abstract objects. For example, most systems simply change a sequence of abstract objects into a corresponding sequence of concrete objects, and provide no control over the orderings of the objects and of their parts. This requires the user to specify completely the ordering of objects (such as bibliographic references), when it

may be more desirable to allow the formatter to choose the ordering. Recent systems have offered more control, but this is usually limited to a few special cases such as figures that may be reordered with respect to the surrounding text.

TROFF and TEX allow users to define new abstract objects, while TBL and EQN support a large number of built-in abstract objects. However, because none of these systems have a general way for users to express concrete attributes of objects, much of the desired control over their concrete representations must be built into their implementations from the beginning.

Scribe has more flexible mechanisms whereby the concrete attributes of its abstract objects, or "environments," may be modified at any time, and this modification may be local (to only a single instance of an object), or global (applying to all objects of the class). A limitation of this system is that it relies on the existence of a fixed universe of concrete attributes; there is no mechanism provided for extending the set of attributes.

3.2.4. Relations among Concrete Objects

Formatters should provide a means for expressing relations among concrete objects. For example, it should be possible to align designated parts of the concrete representation of nearby objects, to constrain the allowed distances between them (imposing either a minimum or a maximum distance, or both), or to specify objects whose size depends on the placement of nearby objects.

Only one system, TEX, permits constraints on distances between all types of objects, accomplished by means of glue specifications. Since there is usually glue between all pairs of adjacent objects, a very fine degree of control over the distances between them is possible. However, this scheme does not work for specifying distances between objects that are not directly adjacent, for specifying alignments of objects, or for making sizes of objects depend on other objects.

One result of the lack of a means to express relations between objects is that all recent formatters contain a number of rather low-level functions to allow concrete specifications. These facilities are similar to the strictly concrete control that was offered by earlier formatters such as FORMAT and RUNOFF. For example, EQN and TEX have many types of "space" characters that must be used to move parts of equations around when the provided structuring methods are not adequate. For the same reason, Scribe has low-level features for tabulation and a number of other positioning commands.

Controlling the concrete representations by low-level spacing commands is rather unsatisfactory for two reasons. First, it causes the user to be concerned with very low-level details when in fact often only high-level notions of alignment are involved. Second, use of these low-level spacing commands makes it very hard to change the document: a small change in one place will often require that all of the spaces be recalibrated. If the alignment constraints could be stated directly, this recalibration would not be necessary.

Another result is that common kinds of control over relations must often be built into formatters as special cases. For example, EQN and TEX provide special-purpose "alignment" operators; these allow designated parts of adjacent equations to be aligned but cannot be used to solve general two-dimensional alignment problems. As another example, both TEX and Scribe offer special predefined operations for dealing with footnotes, which require control over the maximum distance between concrete objects to guarantee that the footnote will be on the same page as its reference.

In each case these concrete functions violate the goal that the user be freed from low-level details. In many ways, this is similar to the dilemma faced in the design of higher level languages, where the attempt to eliminate low-level details restricts the programmer's control.

Two methods of solving this dilemma are found in recent systems designed for the formatting of graphics; these systems offer much greater flexibility in expressing relations among objects. The first of these, the PIC language, provides predefined names for the key parts of objects and ways of specifying how the named parts of objects are to be spatially related. To some extent,

it also allows the size of an object to depend on its relation to other objects.

The means for specifying these relations is very simple: objects are described in a sequence, and a point on each new object is placed either at an absolute location or in a described relationship to a point on an existing object. This simplicity also limits the complexity of relations that may be represented. No cyclic relations may be represented, and, with the exception of arc definitions, it is not possible to represent a relation between one object and several others. This structure may thus be somewhat cumbersome for very complex relations.

The IDEAL system allows even more flexible relations among graphical objects. In this system, relations among objects are represented by equations involving points on the objects, expressed as complex numbers. The system solves these equations to determine the actual points to be used. In this way, any desired relations among the points of the objects may be expressed, provided that the resulting equations admit a solution. However, a possible weakness is that equations are not always a natural means for representing desired relations.

Both of these systems have the drawback that relations among objects must be completely specified, although the use of defaults helps to reduce the problem. However, neither system provides a way of expressing general constraints on the concrete appearance of objects and allowing the system to choose the appearance that best satisfies these constraints. For example, the user cannot specify a range of allowable values instead of a single value. Experiments with methods for specifying and satisfying more general constraints, such as those of ThingLab, may provide insight into more flexible ways of expressing relations.

If formatters had better facilities for composing objects and more control over their concrete forms and relations, the number of primitive predefined objects could be decreased. For example, it would not be necessary to include a special kind of object, such as a figure, that may be reordered with respect to surrounding text, or a special command for aligning equal signs in adjacent equations. Formatters could provide only characters and line segments as primitives, and all of the normal objects could be built from them.

3.2.5 Page Spaces

Many characteristics of documents are best described as properties of the page spaces into which the document objects are placed, rather than as properties of the objects themselves. For example, pages are most often built from nested and juxtaposed spaces for page headers, footnotes, figures, and so forth. However, few formatters permit much control over the page spaces occupied by concrete objects; most do not allow these spaces to be either nested or juxtaposed, and require them to have a bounded rectangular shape. The lack of a general means for specifying such structures means that they must be treated awkwardly, as properties of document objects or as special cases.

Further, even more complex shapes are often required. For example, a page may be shaped like a rectangle with a smaller rectangle removed; the surrounding rectangle may contain text while the removed rectangle can be used for a figure. Even three-dimensional spaces could be considered—for example, layouts for transparent overlays could be designed in this manner. As the layout of pages becomes more and more complex, the task may become too complicated to be handled conveniently by a pure formatter. An integrated system may be needed, so that the user may see immediately the results of changes. The lack of sufficient control over page layout has meant that general-purpose formatters are not used for document types, such as magazines and newspapers, for which this control is essential.

Two of the experimental systems described above, JANUS and Etude, have proposed more flexible methods for specifying the nesting of page spaces; further research should attempt to provide even more control. If this research is successful, general-purpose formatters could be used for a number of layout problems for which more specialized programs have traditionally been used.

From the preceding discussion of objects, composition, abstract-to-concrete mappings,

relations, and page spaces, it is clear that even current formatters are very limited in the formatting functions that they offer. Some of the limitations are historical and are caused by former restrictions on output devices. Some are due to the lack of efficient algorithms for implementing the desired functions. Others, such as inadequate control over relations among concrete objects or the inability to format objects into complex page spaces, arise because it has proved difficult to design a language for expressing the desired functions.

3.3 Formatting Language

The usefulness of a formatting system is very dependent upon the formatting language used to specify documents. The document specification language must be able to express the structure and content of many different kinds of objects. Some means of controlling the abstract-to-concrete mapping of the objects is also required. Finally, the language may also allow the user to create new classes of objects.

Few systems give an explicit description of their formatting language. With the exception of EQN, no formatting language has even included its grammar in its published description, as do most modern programming languages. It is therefore often very difficult to determine whether an undesired feature of the concrete document is the result of an error in the formatting system or due to a misunderstanding of the syntax or semantics of the formatting language. In addition, the absence of a precise semantic description often makes it impossible to determine the effects of combinations of operations provided by the language.

Future formatting systems should provide more precise descriptions of the syntax and semantics of their formatting language. There is a great deal of benefit to be derived from the explicit use of a rich context-free language like that of EQN; such a language ensures great flexibility and internal consistency. Many other benefits of this approach are discussed in the original paper on EQN [KERN75].

Since formatting systems are used by a wider variety of people than conventional programming systems, it is important that the language be easy to use and to understand. It is also important, however, that the language be capable of describing any desired document. Much of the difficulty of designing a formatting language is caused by the conflict between these two goals.

3.3.1 Declarative Languages

One approach that has been used to make formatting languages easier to understand is to make them declarative rather than procedural. Since documents are essentially passive in nature and themselves perform no processing, this is a natural approach. It also allows the formatting process to be understood without a knowledge of programming concepts. With a declarative language, the document is viewed as a series of declarations that elaborate its structure and content. The abstract-to-concrete mapping is controlled by associating "properties" with the objects.

The power of a declarative language can be increased by using templates for describing parameterized structures, that is, structures of which part is constant and part is supplied later as an argument. Templates are a very natural means of specification, since the template language can be designed so that the template graphically resembles the class of structures that it encodes. Templates represent a natural method for extending declarative languages; in a well-designed language, named templates can be used in the same manner as structures that are built into the language.

Among the pure formatters, the declarative approach is taken most notably by Scribe, where concrete properties associated with text are defined by environments. Default properties supplied by Scribe's database may be supplemented or overridden in each case by the user. Scribe uses templates in some special cases, such as the representation of formats for dates and for numerical quantities, but does not really allow the language to be extended by means of templates.

Some integrated editor/formatters also offer what is essentially a declarative language. For example, both Bravo and Star define the appearance of objects by associ-

ating low-level properties with them. As described in Section 2.4.2, Bravo is conventionally used with a set of template files (forms) that simplify the process of creating new documents. A very similar facility is offered by PEN, where the templates are called "default instances" of objects.

The power and naturalness of templates is also demonstrated in both TEX and UNIX. Although their languages are chiefly procedural, both systems use a template language to specify table formats.

3.3.2 Procedural Languages

Many systems treat the formatting process as a series of operations to be applied to objects, much in the style of a traditional programming language. This has the advantage that the formatting language can be made extremely powerful. The inclusion in the language of only a small number of programming constructs can help to ensure that a user will be able to perform any function that is desired, since the formatting language can then presumably be used to calculate any computable function at all. This approach does have the corresponding disadvantage that a user unfamiliar with programming concepts will be unable to understand the more advanced features of the system.

In a purely functional system, the abstract-to-concrete mapping would be controlled only by the operations that were performed on the objects. In most systems, however, there is also a set of global variables that control this mapping. In addition, there is usually some way for particular values of the global variables to be associated with particular objects by entering a nested scope for the duration of the processing of the objects. Upon leaving the scope, the old values of the global variables are restored. This is the method used by PUB, TROFF, and TEX.

A procedural formatting language also allows the language itself to be extended easily through the definition of macros or procedures. If properly designed, the language can permit the extended operations to be used exactly as the built-in operations. In PUB, TROFF, and TEX, the user can define macros including recursion and conditional tests. PUB has an especially large number of programming constructs, including procedures and iteration in its later versions. These ensure that a user will be able to produce almost any desired document.

Formatters could benefit from even more ideas from conventional programming languages. Much could be gained by allowing variables and expressions of many types, including both traditional types such as integers and strings, and also abstract and concrete objects. In addition, such formatters should offer the kinds of debugging facilities that are provided by programming language systems. This is necessary because the increased power of procedural languages makes it harder to diagnose their failures.

As shown by TEX and UNIX, it is possible for a system to be a mixture of both declarative and procedural languages. One scheme, proposed for the JANUS system, provides a declarative language to describe particular documents, and a procedural language to define classes of documents by implementing the constructs of this declarative language. Many other organizations are possible. Again, there are similarities to the design of higher level programming languages, which most often consist of a mixture of declarations and executable statements.

3.3.3 Class Definitions

There are a variety of mechanisms for defining new classes. In a declarative language this may be done either, as in PEN, by using a template to represent a class of objects by means of a single, partially specified object, or, as in Scribe, by the simpler process of associating an environment name with a set of concrete properties.

In a procedural language, this may be done either by explicitly declaring classes as in SIMULA [BIRT79] or Smalltalk, or, as in most current formatting systems, by the simpler method of associating a procedure or macro with the new class. This procedure or macro is used to produce instances of the new class by calling or invoking it with appropriate arguments, most often a stream of text. The more explicit class/instance method has greater promise, however, since it permits more control over objects. For example, this method could be used to en-

sure that objects of a class were correctly structured.

Some systems, such as Scribe and JANUS, have separated the language used to describe classes from that used to describe instances of the classes. In both cases, this is done so that the general user need not be concerned with class definition. However, it also has the drawback that it may tend to draw too sharp a distinction between a class of objects and a single object. Usually there is a spectrum of objects in use, from a very generic class of "documents" at the top, through a number of more and more highly specified objects with fairly constant formats (such as "technical reports" or "newletters"), down to particular instances of documents (such as "Technical Report #37") at the most specific level. As described in Section 3.2.2, it is often as natural for a user to want to create a new subclass (a more specified form of an existing class) as it is to want to create a particular instance of a document. There is a danger, then, that too great a distinction between the languages used to describe classes and instances would make it difficult to create natural and economical descriptions of such a hierarchy of objects.

3.4 Integration of Objects

Systems that deal with a large number of different kinds of objects have often failed to provide a uniform framework for handling them. For example, the UNIX system offers separate languages and programs for its different clases of objects. It is worth investigating the advantages of a single language and set of commands for all these classes.

The UNIX system also places some limitations on the ways objects may be nested. For example, it is not possible to include one table as an entry in another, or to include graphical objects within a mathematical one. This is caused partly by the fact that the formatting processes communicate by one-directional pipes and are designed so that objects of one kind are all processed at once by a single program. This means that, for example, the results of formatting a table containing mathematics cannot easily be used as input to the formatting process for another mathematical object.

As another example, TEX integrates the formatting of textual, tabular, and mathematical objects into a single language. However, the integration is not complete because there are separate "modes" for handling text and mathematics. TEX thus fails to provide a single set of commands that is applicable to all objects.

Future research should attempt to find a single set of primitive operations (or properties, for declarative systems) that may be used in the creation and manipulation (or description) of objects of all types. This would ensure that objects might be nested in arbitrary combinations, and would reduce the amount of detail present in a formatting system based on these primitive operations. The Star system uses an especially small set of universal commands, and Smalltalk and its applications use a single mechanism for applying operations to all objects. These systems are thus rich sources of ideas for integrating objects.

3.5 Integration of Document-Processing Functions

In addition to integrating different types of objects into a single framework, systems should also attempt to integrate the many different functions performed in the preparation of a document. Systems in which these functions are not integrated require the use of a large number of unrelated environments. For example, one environment may be an editor, another may be the command interpreter of an operating or filing system, and a third may be the formatting system itself. Different commands and operations are used in each environment, and even the styles of interaction may be different for the different environments. There is usually a fairly large amount of mental effort and time required to move from one environment to another.

3.5.1 Integration of Editing and Formatting

The greatest gains in this area come from the integration of editing and formatting, as in the systems described in Section 2.4. In a system where these functions are not integrated, the document preparation pro-

cess is a cyclic activity of refining the document description, generating the resulting document, and finding flaws in the concrete appearance of the document. This process is repeated until the concrete appearance is satisfactory. An integrated editor/formatter reduces the effort of this task by making the generation of the concrete document a part of a single document creation procedure. In simple systems, the current concrete appearance of the document may be viewed on request.

This process may be carried even further, so that the formatting and viewing functions are carried out continuously, and the user may be considered to be applying operations directly to the finished document. This immediacy allows the document to be manipulated partly through the physically intuitive notions of moving objects around on a two-dimensional surface. It also reduces the amount of detail that a user must remember, since it is no longer necessary to be able to "predict" the system's actions when it is given a set of commands. Instead, the system's actions become immediately apparent.

The drawback of existing integrated editor/formatters is that the high-level structure of the document is not represented. Since the user manipulates only the concrete document, its abstract structure is obscured. This makes it difficult not only to manipulate logical entities as a unit, but also to generate several versions of the same document according to different formatting conventions.

Two experimental systems, JANUS and Etude, attempt to provide concrete and abstract information simultaneously. In these systems, the user may edit both the abstract structure of the document and its concrete format. Although similar in this respect, the two systems differ in their emphasis. In Etude, the user is expected to deal chiefly with the concrete form of the document, except when direct manipulation of its abstract structure is desired. In JANUS, on the other hand, the user is expected to be concerned chiefly with the abstract form of the document, perhaps checking its concrete appearance from time to time. Editing of the concrete document is intended only as a means of overriding the actions of the formatter. Although this is an attractive means of controlling the abstract-to-concrete mapping, it is of limited use unless the changes are also made a permanent part of the abstract document. Otherwise, there will be serious problems in maintaining consistency between the two versions of the document.

Integrated systems make it possible for programming to be done in an entirely new way. Rather than describe the desired actions symbolically, the user may actually carry out the actions, which are remembered by the system. The system may then be asked to repeat the actions at a later time. This technique is powerful, yet simple enough to be used by people with no knowledge of programming. It has been used in a number of experimental programming systems [SMIT75, CURR78], and work is going on to include it in the Star system [HALB81].

New methods for creating classes of objects can also be used in an integrated system. ThingLab, for example, implements an attractive technique that allows a user to define a class by constructing a particular instance of the class. This idea may be applied to document systems as well. For example, a class of form letters could be constructed by creating a single prototypical form letter. Instances of this class would specify different recipients but would otherwise be identical to the prototype.

Future systems could benefit from even further integration. For example, the creation of primitive graphical objects, special characters, and new character fonts may be made an integral part of the document preparation process.

3.5.2 *Integration of Other Functions*

In order to be most useful, document preparation systems must offer more than just editing and formatting functions. Even the earliest formatting systems attempted to provide a number of more general document preparation functions, such as the "dictionary" command of FORMAT. More of these writer's workbench facilities should be available, including detection and correction of spelling errors; generation of outlines, tables of contents, indices, and concordances; and citation of bibliographic references. A number of general resources,

such as dictionaries, thesauri, and manuals of writing style, would also be useful.

When a document becomes very large, changes are most often made only to part of the document; the other parts are unchanged or are changed only slightly. In this case it is much more efficient if the changed parts can be reformatted separately, without reformatting the entire document. The reformatting is complicated somewhat by the fact that parts of the document often refer by name, section number, or page number to other parts. This means that a change to one section may require changes (perhaps small ones) to the sections that refer to them.

Scribe solves these problems by allowing a document to be broken into a number of modules that may be formatted either separately or as a unit, and handles the references between these modules automatically. However, the user must state explicitly where the divisions into modules are to be made, and these divisions need not be related to the logical structure of the document.

The class/instance model outlined in Section 1.1 defines a document as being structured from a number of nested simpler objects such as paragraphs, sections, and so on. A document preparation system based on this model could use the high-level structure of the document, together with a knowledge of the references from one object to another, to determine automatically parts of the document to be formatted separately, and could allow many kinds of changes to be propagated through the entire document automatically.

There are also advantages in keeping historical versions of a single document and in maintaining families of related documents that have some parts in common. A system that understood these notions would allow the integration of a facility for comparing documents in order to determine their differences. This comparison could be used to determine the portions of a document that have changed since its previous version, or to capture the differences between the members of a family of documents. The PIE system has suggested a way of achieving these goals. PIE also allows the creation of alternate versions that may be maintained consistently in parallel. However, its ideas have not yet been tried in a formatting system.

Even more elaborate tools may be envisioned. For example, a system could help to maintain documents in a partial state of composition by maintaining an outline of parts not yet written, and could allow this outline to be easily fleshed out later. Improved facilities for allowing multiple authors to work on a document without conflict could be added. This might include, for example, a means of making comments on sections and a means of "locking" sections for exclusive access while they are being worked on. Recent programs in the UNIX system analyze aspects of the style and readability of documents [CHER81]; some potential also exists for the application of artificial intelligence and other techniques to the deeper analysis of document style and content.

3.6 User Interface

Every formatting system provides the user with a means of accessing the operations for creating, viewing, and modifying documents. The quality of the interface presented to the user may be judged in part by the following criteria:

- the amount of detail that the user must memorize in order to use the system;
- the amount of mental and physical effort that is required to perform common functions;
- the average number of errors made by the user, especially including errors from which recovery is difficult;
- the amount of time that the user is required to wait for the system to perform its functions, such as the time required for an integrated editor/formatter to update the contents of a screen, or the time required for a pure formatter to create a concrete document for viewing.

The overall design of a formatting system contributes a great deal to the quality of the interface. Ideally, a system should be based on a small number of powerful operations, so that it is simple enough to be easily understood (and hence memorized) by its intended users. Similarly, a system designed around a small number of orga-

nizing principles (such as a single context-free language) may be made very consistent. This allows the behavior of the system to be predicted easily and reduces the amount of memorization required. Finally, as stated in Section 3.5.1, a highly integrated editor/formatter may also reduce the amount of memorization required, since the user no longer must predict the behavior of such a system.

Another desirable feature of a system is the ability to provide access to a freely chosen subsystem oriented to a certain class of user or to a restricted class of problems. This has been referred to as *filtering* [GOLA79]. For example, a beginning user may learn only a very small set of commands, and this set may be increased as the user becomes more and more familiar with the system. At each stage, the user is able to access only commands that are well understood, thus reducing the possibility of error.

In recent years, a growing number of empirical studies of interactive systems have been performed, giving quantitative insight into the importance of the various aspects of user interfaces. Both the hardware and the software features of interfaces have been investigated [ACMC81, CARD78, CARD80, SHNE80]. Increasingly, research of this type is being used to assist the intuition of the user interface designer.

3.6.1 Software Improvements

Many software techniques have been developed to improve aspects of the interface, but most of these improvements require compromises in other areas. For example, using long identifiers as command names tends to reduce the amount of detail that the user must remember, since the command names may be made descriptive of their action. However, it tends to increase the amount of physical effort required for the user to enter a command, since it takes more keystrokes to enter a long name than a short one. Similarly, long command names make it easier to mistype a command, an error from which it is easy to recover. However, they also make it physically harder to type one command when another is intended, an error from which recovery may be more difficult.

As another example, prompting for portions of commands and data reduces the amount of detail that the user must memorize, but it increases the amount of data that must be emitted by the system, thus increasing the amount of time required for the system to perform its functions. The use of menus reduces the amount of detail that must be memorized, the number of errors in entering commands, and the amount of physical effort required to enter commands, but it also has the drawback that it increases the amount of time required for a system to perform display functions and that it requires the user to read more material between commands.

Multiple windows have been used to decrease the mental effort involved in switching from one context to another, since they allow a number of contexts to be maintained simultaneously. For example, they may contain menus, prompting information, or views of several different parts of a document. However, their use increases the amount of time required for display functions, and introduces the mental task of correlating the information found in different windows.

As the above examples illustrate, it is impossible to achieve simultaneously all of the desirable properties of an interface for all classes of users. However, one can do much better by taking into account the characteristics of the intended user. For example, a system may be intended for an expert user who may be expected to memorize all the details of a system. In this case, the design of the interface would probably emphasize a reduction of the physical effort required of the user. In other cases, the design of the interface may attempt to minimize the amount of detail to be memorized. In the most general case, a single system may offer a number of different user interfaces. This may be accomplished by implementing entirely separate interfaces; however, a more general and more consistent system will result from the use of filtering to provide access to well-defined subsystems.

3.6.2 Hardware Improvements

The user interface may also be improved through the use of hardware techniques,

such as high-resolution displays and graphical input devices, dedicated computers, large storage facilities, and high-bandwidth connections. For example, the time required for the system to perform display functions may be reduced by increasing the bandwidth between the processor and the display device. Higher bandwidth than that available under a large time-sharing system may be achieved on a single-user computer with a dedicated display. If the bandwidth is sufficiently high, many of the software techniques described above become practical, because there is little time spent waiting for the system.

Graphical input devices such as the mouse, light pen, and joystick substantially reduce the physical effort required to select and position objects. For example, they are used successfully for the selection of items from a menu. Bit-mapped displays and their extensions can allow the manipulation of many different types of objects, such as colored objects and halftone images.

3.7 Implementation

The utility of a practical document-processing system also depends significantly on its implementation. For example, an appealing model or formatting language may need to be rejected if it is impossible to build efficiently. Aside from choosing low-level data structures and algorithms, an implementation may also attempt to decompose the formatting problem into small independent pieces and to provide device independence. Another aspect of an implementation is the ease with which it fits into a larger system of which it is part. Each of these is now discussed in turn.

3.7.1 Data Structures and Algorithms

The published material of the earliest systems placed little emphasis on their data structures and algorithms, although they may often be deduced from the other information about the systems. For example, the document model used by HES is based rather directly on a data structure for representing segments of text linked into a graph structure, with pointers in the text that refer to locations in other text segments. Recently, a certain number of new algorithms and data structures have appeared. A notable example is the dynamic programming algorithm for paragraph layout in TEX, which is probably the first clearly described and nontrivial algorithm to be employed for this task.

The *sticky pointer* data structure [FISC79] provides a mechanism that can be used to associate pointers with textual data. In this scheme, the pointers are kept entirely separate from the data and point to a tree structure that in turn points to a linked list containing the text. The advantage of this structure is that it allows the data to change freely without requiring command updating of the pointers. Sticky pointers have already been used in the implementation of a text editor [ROBE81a, ROBE81b]. They may also be useful for an integrated editor/formatter, where the structure and properties of objects in the document could be represented separately from the objects themselves. Modifications to the document contents could be handled very quickly while keeping the structure up to date.

The familiar concept of an inverted file used in REFER also appears applicable to a number of other document-processing tasks that require a search of a set of textual objects that do not change frequently. For example, it could be used to locate a particular quotation in a large work.

In addition to these relatively low-level techniques, some interesting methods have been used for the implementation of formatting systems as a whole. For example, EQN, PIC, and IDEAL are constructed by means of a context-free language parser, generated using a compiler-compiler. In general, these and other preprocessors have proved very useful in systems that are broken down into separate programs.

3.7.2 Decomposition of the Formatting Problem

An implementation may be based on a single large program, as are TEX and Scribe, or factored into a number of smaller programs, as is done in the UNIX system. A factored system may be broken into small programs that process different types of objects, as under UNIX, or it may be broken into programs that handle different

parts of the formatting problem. For example, some systems use one program to format a document into a series of lines and a different program to place these lines onto pages.

One advantage of a factored approach is that each of the programs may be very simple and fairly easy to understand. Further, it is possible to set up a configuration of programs that is tailored to the complexity of the document to be formatted. This allows the overall system resource requirements for a particular document to be reduced. The disadvantage of this organization is that it may tend to multiply the number of languages used in the system (as is true under UNIX), and also it will multiply the number of programs to be maintained. There is also likely to be a certain amount of duplication of code among the separate programs, such as the code required to parse the formatting language.

The advantage of a single program is that it seems to make it easier to offer a single integrated system for handling all types of objects. As observed in Section 3.4, the use in UNIX of many small programs communicating by one-directional pipes tends to limit the possible nesting of different kinds of objects. Further, with a single program it is easier to provide a single language for describing documents. On the other hand, a single large program must be much more complicated, and also it cannot be tailored to the document.

One of the fundamental difficulties of the formatting problem is that the processing of objects depends not only on the objects themselves, but on the larger objects of which they are part. For example, the concrete appearance of a sentence is not known until the paragraph of which it is part is mapped into a set of lines; there is not enough information to format the sentence when taken by itself. However, the concrete appearance of an object (a paragraph in this example) clearly depends upon the objects from which it is composed. Therefore, neither a strictly top-down nor a strictly bottom-up algorithm can be used.

Besides complicating the formatting process itself, this fact has implications about the ways in which formatting systems may be broken down into small programs. That is, unless a certain amount of control over the concrete appearance is relinquished, it is not possible in general to break down the formatting problem into a number of completely independent programs that handle different objects. In the example above, it would not be possible to format the paragraph without formatting its component sentences or to format the sentences without formatting the paragraph of which they are part.

As another example, TEX exercises much more control than earlier formatters over the concrete appearance of paragraphs because it tries many different ways of placing words into lines. This interdependence could be carried even further. A system could try many different ways of placing words into both lines and paragraphs in order to choose the one that gives the pages the best appearance. For example, words could be formatted more tightly to eliminate a widow on the following page. In systems that separate the formatting of paragraphs into lines from the layout of lines onto pages, this would not be possible.

On the other hand, the resulting simplification makes it worth looking for cases in which the formatting problem can be decomposed. An example appears in the Arabic language system KATIB/HATTAT, where the placing of characters into lines is separated from the determination of the concrete forms of the individual characters, by using average values for the widths of characters. This method of decomposition may perhaps be extended to other cases where the different concrete forms of a given class of abstract objects do not show too much variation in size.

It should also be noted that decomposition can provide device independence. The viewing of a concrete document may be separated from the rest of the formatting process, allowing the same concrete document to be produced on a number of different devices without requiring that it be reformatted each time. This may be accomplished through a device-independent formatter output that is translated by the viewing process into low-level commands for controlling a particular device.

A decomposition of the formatting problem is achieved in the UNIX system by having the programs for separate objects perform a high-level "preformatting" func-

tion; almost all of the actual solutions to formatting problems are handled in the final pass through the low-level formatter TROFF. This approach, while very appealing, increases the amount of processing time required to produce a document from its description.

3.7.3 Interface to Host Environment

Many formatting systems today are designed as application programs under existing operating systems. One of the difficulties of this embedding occurs if it is desired to ensure that the formatting system could also function (without much change) under a different operating system. Independence of particular operating system features tends to increase the portability of the formatting system and of the abstract documents themselves. These can be very important goals, especially if a standardized means for exchange of documents is to be developed. At the same time, however, dependence upon a particular operating system often increases the usability and efficiency of a formatting system.

For example, Scribe is designed to execute in the environment of a generic operating system that makes only modest demands upon the specific operating system in which it is embedded and does not try to provide a sophisticated interface to any system functions [REID80c, Appendix B]. The central requirements are the ability to access files with names formed from short character strings and the ability to perform simple operations on these files, such as reading, writing, deleting, and determining date of creation.

Because Scribe's interface to the operating system is so simple, it is impossible for Scribe to provide a complete or integrated interface to its documents. It is necessary, for example, for the user to communicate with the command interpreter of the operating system to ask for a list of current files containing documents. Adding this function to Scribe would make the file interface more difficult to move to another system.

Scribe and the UNIX formatters have proved easy to move to new operating systems because they use only features, such as programs and text files, that are present in nearly all current systems. However, a formatting system that used interprocess communication more complex than that offered by files or pipes would be more difficult to make portable, because there is not as much agreement about how these facilities should be provided. Similarly, formatters that rely on specialized hardware not present in all operating systems could not easily be moved to systems not supporting this hardware.

4. CONCLUDING REMARKS

We have defined the nature of the formatting problem, surveyed some significant systems, and presented a number of concepts and outstanding issues and problems. Despite the impressive achievements in this fast-moving field, it is evident from our analysis in the preceding pages that much remains to be done before we can realize the potential inherent in computer, display, and printing technology—a hardware technology that makes it feasible, in principle, to specify, manipulate, and view the appearance of documents with an unprecedented degree of control, precision, flexibility, speed, and economy.

As they are developed further, formatting systems will remain a major part of such applications as publishing and word processing, but they will also become a major utility available in most general-purpose computer systems. Even more generally, a complete package of integrated editors, formatters, and other tools for computer-aided writing and reading of documents will be an important component of the computer system of the future.

ACKNOWLEDGMENTS

We are grateful to the following people for reading and offering helpful comments on various parts of the paper: A. Borning, D. Chamberlin, G. Coulouris, S. Johnston, B. Kernighan, J. King, D. Knuth, B. Lampson, M. Lesk, P. MacKay, H. Moll, B. Reid, J. Saltzer, L. Tesler, C. Van Wyk, and J. Zahorjan. We wish to acknowledge the assistance in obtaining certain references provided by G. Kimura, J. Saltzer, P. Samson, and L. Tesler. D. Knuth, G. Kimura, B. Kernighan, and B. Rice helped us obtain material used in the figures. Thanks are also due to B. Reid for his assistance in producing typeset versions of this paper. A. Goldberg and the referees made many constructive suggestions We used Scribe extensively in preparing

the many drafts of this paper. Figures 7 and 9 were produced using TEX and Figures 11 and 14 with the UNIX document production tools

This work was supported in part by the National Science Foundation under grants numbered MCS-7826285 and MCS-8004111. An extended abstract [SHAW80b] of this paper was presented at the International Conference on Research and Trends in Document Preparation Systems, held in Lausanne, Switzerland, in February 1981.

REFERENCES

ACMC — ACM COMPUTING SURVEYS. *Comput. Surv.* **13,** 1 (March 1981). Special Issue The Psychology of Human-Computer Interaction.

ALLE81 — ALLEN, T., NIX, R., AND PERLIS, A. "PEN: A hierarchical document editor." In *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 74–81. Also available as *SIGOA Newsletter* ACM **2,** 1&2 (Spring/Summer 1981), 74–81.

BARN65 — BARNETT, M. P. *Computer Typesetting: Experiments and Prospects.* The M.I.T. Press, Cambridge, Mass., 1965.

BAUD78 — BAUDELAIRE, P. C. "Draw Manual." In *Alto User's Handbook,* B. W. Lampson and E. A. Taft (Eds.). Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.

BEAT79 — BEATTY, J. C., CHIN, J. S., AND MOLL, H. F. "An interactive documentation system" In *SIGGRAPH '79 Proceedings, Computer Graphics* (ACM) **13,** 2 (Aug. 1979), 71–82.

BERN68 — BERNS, G. M The FORMAT program. *IEEE Trans. Eng Writ. Sp.* **EWS-11,** 2 (Aug. 1968), 85–91.

BERN69 — BERNS, G. M. Description of FORMAT, a text-processing program. *Commun. ACM* **12,** 3 (March 1969), 141–146.

BIRT79 — BIRTWISTLE, G. M., DAHL, O-J., MYHRHAUG, B., AND NYGAARD, K. *Simula Begin.* 2nd ed. Van Nostrand-Reinhold, New York, 1979.

BORN79 — BORNING, A. "ThingLab—A Constraint Oriented Simulation Laboratory." Ph.D. dissertation, Stanford Univ., Stanford, Calif., 1979. Available as Tech. Rep. SSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif., and as Tech. Rep. STAN-CS-79-746, Stanford Computer Science Dep., Stanford Univ., Stanford, Calif.

BORN81 — BORNING A. The programming language aspect of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Prog. Lang. Sys.* **3,** 4 (Oct. 1981), 353–387.

BURK80 — BURKHART, H., AND NIEVERGELT, J. "Structure-oriented editors." Berichte des Instituts fuer Informatik 38, Eidgenoessische Technische Hochschule Zuerich, Zurich, Switzerland, May, 1980.

BYTE81 — BYTE MAGAZINE *Byte* **6,** 8 (Aug. 1981). Special issue on Smalltalk.

CARD78 — CARD, S. K., ENGLISH, W. K., AND BURR, B. J. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics* **21** (1978), 601–613.

CARD80 — CARD, S. K., MORAN, T. P., AND NEWELL, A. The keystroke-level model for user performance time with interactive systems. *Commun. ACM* **23,** 7 (July 1980), 396–410.

CARM69 — CARMODY, S., GROSS, W., NELSON, T. E., RICE, D., AND VAN DAM, A. "A hypertext editing system for the /360." Center for Computer and Information Sciences, Brown Univ., Providence, R.I., March 1969. Also contained in *Pertinent Concepts in Computer Graphics,* M. Faiman and J. Nievergelt (Eds.). Univ. of Illinois, Urbana, Ill., 1969, pp. 291–330.

CHAM81 — CHAMBERLIN, D. C., KING, J. C , SLUTZ, D. R., TODD, S. J. P., AND WADE, B. W. "JANUS: An interactive system for document composition." In *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 82–91. Also available as *SIGOA Newsletter* (ACM) **2,** 1&2 (Spring/Summer 1981), 82–91. This report was also issued as IBM Computer Science Res. Rep. RJ3006 (37371), IBM Research Lab., San Jose, Calif., Dec. 1980.

CHAM82 — CHAMBERLIN, D. C., KING, J. C., SLUTZ, D. R., TODD, S. J. P., AND WADE, B. W. "JANUS: An interactive document formatter based on declarative tags." IBM Comp. Sci. Res. Rep. RJ3366 (40402), IBM Research Lab., San Jose, Calif., Jan. 1982.

CHER81 — CHERRY, L. "Computer aids for writers. In *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 61–67. Also available as *SIGOA Newsletter* (ACM) **2,** 1&2 (Spring/Summer 1982), 61–67.

COUL76 — COULOURIS, G. F., DURHAM, I., HUTCHINSON, J. R., PATEL, M. H., REEVES, T., AND WINDERBANK, D. G. The design and implementation of an interactive document editor. *Softw. Prac. Exper.* **6,** 2 (April–June 1976), 271–279.

CURR78 — CURRY, G. A. "Programming by Abstract Demonstration," Ph.D. dissertation, Univ. of Washington, Seattle, March 1978. Also issued as Tech. Rep. 78-03-02, Dep. of Computer Science, Univ. of Washington.

EHRM71 — EHRMAN, J. R., AND BERNS, G. M. "FORMAT, a text processing program."

SLAC Rep. 135, Stanfrd Linear Accelerator Center, July 1971.

ENGE68 ENGELBART, D. C., AND ENGLISH, W. K. "A research center for augmenting human intellect." In *Proc. Fall Jt. Computer Conf.*, vol. 33. AFIPS Press, Arlington, Va., 1968, pp. 395–410.

ENGE73 ENGELBART, D. C., WATSON, R. W., AND NORTON, J. C. "The augmented knowledge workshop." ARC Journal Accession Number 14724, Stanford Research Center, Menlo Park, Calif., March 1973. Paper presented at the National Computer Conference, June 1973.

FISC79 FISCHER, M. J., AND LADNER, R. E. "Data structures for efficient implementation of sticky pointers in text editors." Tech. Rep. 79-06-08, Dep. of Computer Science, Univ. of Washington, Seattle, June 1979.

GOLA76 GOLDBERG, A., AND KAY, A., Eds. "Smalltalk-72 Instruction Manual." Rep. SSL-76-6, Xerox Palo Alto Research Center, Palo Alto, Calif., March 1976.

GOLA79 GOLDBERG, A., AND ROBSON, D. "A metaphor for user interface design." In *Proc. of 12th Hawaii Int. Conf. Syst. Sci.*, vol. 1, University of Hawaii Press, Honolulu, 1979, pp. 148–157.

GOLA83 GOLDBERG, A., AND ROBSON, D. Smalltalk-80: *The Language and Its Implementation.* Addison-Wesley, Reading, Mass , 1983

GOLC81a GOLDFARB, C. F. "Use of an integrated text processing system in commercial textbook production." In *Abstracts of the Presented Papers, Int. Conf. Research and Trends in Document Preparation Systems* (Lausanne, Switzerland, Feb. 1981), Swiss Institutes of Technology, Lasanne and Zurich, pp. 121–122.

GOLC81b GOLDFARB, C. F. "A generalized approach to document markup." In *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 68–73. Also available as *SIGOA Newsletter* (ACM) **2,** 1&2 (Spring/Summer 1981), 68–73.

GOLI80 GOLDSTEIN, I. P., AND BOBROW, D. G. "A layered approach to software design." Rep. No. CSL-80-5, Xerox Palo Alto Research Center, Palo Alto, Calif., Dec. 1980.

GOLI81 GOLDSTEIN, I., AND BOBROW, D. "An experimental description-based programming environment· Four reports." Rep. CSL-81-3, Xerox Palo Alto Research Center, Palo Alto, Calif , March 1981.

GOOD81 GOOD, M. "An ease of use evaluation of an integrated editor and formatter," Tech. Rep. MIT/LCS/TR-266, M.I.T. Lab. for Computer Science, Cambridge, Mass., Nov. 1981. This is a revised version of Good's M.S. thesis, Aug. 1981.

GUTT80 GUTTAG, J., AND HORNING, J. J. "Formal specification as a design tool." In *Conf. Rec. 7th Ann. ACM Symp. Principles of Programming Languages* (Las Vegas, Nev., Jan. 1980), ACM, New York, 1980, pp. 251–261. Also issued as Rep. No. CSL-80-1, Xerox Palo Alto Research Center, Palo Alto, Calif., Jan. 1980.

HALB81 HALBERT, D. C. "An Example of Programming by Example." Master's thesis, Univ. of California, Berkeley, June 1981.

HAMM81a HAMMER, M., ILSON, R., ANDERSON, T., GILBERT, E. J., GOOD M., NIAMIR, B., ROSENSTEIN, L., AND SCHOICHET, S. "Etude: An integrated document processing system." Office Automation Group Memo OAM-028, M.I.T. Lab. for Computer Science, Cambridge, Mass., Feb. 1981. Presented at the 1981 Office Automation Conference, March 23–25, 1981.

HAMM81b HAMMER, M , ILSON, R., ANDERSON, T., GILBERT, E. J., GOOD, M., NIAMIR, B., ROSENSTEIN, L., AND SCHOICHET, S. "The implementation of Etude, an integrated and interactive document production system." *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 137–141. Also available as *SIGOA Newsletter* (ACM) **2,** 1, 2 (Spring/Summer 1981), 137–141. Previously issued as Office Automation Group Memo OAM-026, M.I.T. Lab. for Computer Science, Cambridge, Mass., Dec. 1980.

IBM80a *Document composition facility—Introduction to the generalized markup language: Using the starter set.* IBM, White Plains, N.Y., 1980. Order no. SH20-9186-0.

IBM80b *Document composition facility generalized markup language: Starter set reference.* IBM, White Plains, N.Y., 1980. Order no. SH20-9187-0.

IBM80c *Document composition facility: User's guide.* IBM, White Plains, N.Y., 1980. Order no. SH20-9161-1.

ILSO80 ILSON, R. "An integrated approach to formatted document production." Tech. Rep. MIT/LCS/TR-253, M.I T. Lab. for Computer Science, Cambridge, Mass., Aug. 1980.

INGA78 INGALLS, D. H. "The Smalltalk-76 programming system design and implementation." In *Conf. Rec. 5th Annual ACM Symp. Principles of Programming Languages* (Tucson, Ariz., Jan. 1978), ACM, New York, 1978, pp. 9–16.

IVIE77 IVIE, E. L. The programmer's workbench—A machine for software devel-

opment. *Commun. ACM* **20,** 10 (Oct. 1977), 746–753.

KAIM68 KAIMAN, A. Computer-aided publications editor. *IEEE Trans. Eng. Wr. Sp.* **EWS-11,** 2 (Aug. 1968), 65–75.

KERN75 KERNIGHAN, B. W., AND CHERRY, L. L. A system for typesetting mathematics. *Commun. ACM* **18,** 3 (March 1975), 151–157. Also available as Computer Science Tech. Rep. 17, Bell Laboratories, Murray Hill, N.J. (rev. April 1977).

KERN76a KERNIGHAN, B. W., AND PLAUGER, P. L. *Software Tools.* Addison-Wesley, Reading, Mass., 1976.

KERN76b KERNIGHAN, B. W. "A TROFF tutorial." Internal Memo, Bell Laboratories, Murray Hill, N.J., Aug. 1976. In *Documents for Use with the Phototypesetter,* version 7.

KERN78 KERNIGHAN, B. W., LESK, M. E., AND OSSANNA, J. F., JR. UNIX time-sharing system: Document preparation. *Bell Syst. Tech. J.* **57,** 6 (July–Aug. 1978), 2115–2135.

KERN81a KERNIGHAN, B. W. "PIC—A crude graphics language for typesetting," Computer Science Tech. Rep. 85, Bell Laboratories, Murray Hill, N.J., Jan. 1981.

KERN81b KERNIGHAN, B. W. Review of 'TEX and METAFONT: New directions in typesetting,' *Comp. Rev.* **22** (July 1981), 299–301. Review 38,151.

KERN81c KERNIGHAN, B. W. "A typesetter-independent TROFF." Computer Science Tech. Rep. 97, Bell Laboratories, Murray Hill, N.J., 1981.

KERN82 KERNIGHAN, B. W. PIC—A language for typesetting graphics. *Softw. Prac. Exper.* **12,** 1 (Jan. 1982), 1–21. A preliminary version of this paper appeared in the *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), and *SIGOA Newsletter* (ACM) **2,** 1&2 (Spring/Summer 1981).

KNUT75 KNUTH, D. E. *Sorting and Searching. The Art of Computer Programming,* vol. 3. Addison-Wesley, Reading, Mass., 1975, sect. 6.5, pp. 552–557.

KNUT79a KNUTH, D. E. *TEX and Metafont. New Directions in Typesetting.* Digital Press and the American Mathematical Society, Bedford, Mass., and Providence, R.I., 1979.

KNUT79b KNUTH, D. E. "Mathematical typography." In *TEX and Metafont: New Directions in Typesetting,* part 1. Digital Press and the American Mathematical Society, Bedford, Mass., and Providence, R.I., 1979.

KNUT79c KNUTH, D. E. "TEX, a system for technical text." In *TEX and Metafont: New Directions in Typesetting,* part 2. Digital Press and the American Mathematical Society, Bedford, Mass., and Providence, R.I., 1979.

KNUT81 KNUTH, D. E. AND PLASS, M. F. Breaking paragraphs into lines. *Softw. Prac. Exper.* **11,** 11 (Nov. 1981), 1119–1184. Also issued as Tech. Rep. STAN-CS-80-828, Stanford Dep. of Computer Science, Stanford, Calif., Nov. 1980.

LAMP78 LAMPSON, B. W. "Bravo manual." In *Alto User's Handbook,* B. W. Lampson and E. A. Taft (Eds.). Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.

LESK76a LESK, M. E. "Tbl—A program to format tables." Computer Science Tech. Rep. 49, Bell Laboratories, Murray Hill, N.J., Sept. 1976.

LESK76b LESK, M. E. "Typing documents on the UNIX system: Using the -ms macros with TROFF and NROFF." Internal Memo, Bell Laboratories, Oct. 1976. In *Documents for Use With the Phototypesetter,* version 7.

LESK77 LESK, M. E., AND KERNIGHAN, B. W. "Computer typesetting of technical journals on UNIX." In *Proc. Nat. Comp. Conf.* **46** (1977), 879–888. Also available as Computer Science Tech. Rep. 44, Bell Laboratories, Murray Hill, N.J., June 1976.

LESK78 LESK, M. E. "Some applications of inverted indexes on the UNIX System," Computing Science Tech. Rep. 69, Bell Laboratories, Murray Hill, N.J., June 1978.

MACK77 MACKAY, P. A. Setting Arabic with a computer. *Scholarly Publishing* **8,** 2 (Jan. 1977), 142–150.

MADN68 MADNICK, S. E., AND MOULTON, A. SCRIPT: An on-line manuscript processing system. *IEEE Trans. Eng. Writ. Sp.* **EWS-11,** 2 (Aug. 1968), 92–100.

MCMA78 MCMAHON, L. E., CHERRY, L. L., AND MORRIS, R. UNIX time-sharing system: Statistical text processing. *Bell Syst. Tech. J.* **57,** 6 (July–Aug. 1978), 2137–2154.

NEWM78 NEWMAN, W. M. "Markup user's manual." In *Alto User's Handbook,* B. W. Lampson and E. A. Taft (Eds.), Computer Science Lab., Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.

OSSA74 OSSANNA, J. F. "NROFF user's manual," 2nd ed., Internal Doc., Bell Laboratories, Sept. 1974.

OSSA76 OSSANNA, J. F. "NROFF/TROFF user's manual," Computer Science Tech. Rep. 54, Bell Laboratories, Murray Hill, N.J., Oct. 1976.

PACK73 PACKARD, D. W. "Can scholars publish their own books?" *Scholarly Publishing* **5,** 1 (Oct. 1973), 65–74.

PIER72 PIERSON, J. *Computer Composition*

Using PAGE-1. Wiley-Interscience, New York, N.Y., 1972.

REID80a REID, B. K. "A high-level approach to computer document formatting." In *Conf. Rec. 7th Annual ACM Symp. on Principles of Programming Languages* (Las Vegas, Nev., Jan. 1980), ACM, New York, 1980, pp. 24-31.

REID80b REID, B. K., AND WALKER, J. H. *SCRIBE Introductory User's Manual,* 3rd ed., preliminary draft. Unilogic, Pittsburgh, 1980.

REID80c REID, B. K. "Scribe: A Document Specification Language and Its Compiler," Ph.D. dissertation, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., Oct. 1980. Also issued as Tech. Rep. CMU-CS-81-100.

REID81 REID, B. K. "The Scribe document specification language and its compiler." In *Abstracts of the Presented Papers, Int. Conf. Research and Trends in Document Preparation Systems* (Lausanne, Switzerland, Feb. 1981), Swiss Institutes of Technology, Lausanne and Zurich, pp. 59-62.

RITC78 RITCHIE, D. M. UNIX time sharing system: A retrospective. *Bell Syst. Tech. J.* **57,** 6 (July-Aug. 1978), 1947-1969.

ROBE81a ROBERTSON, K. "ESP, a direct access editor: ESP user's guide," Tech. Note 134, Computer Science Lab., Univ. of Washington, Seattle, April 1981.

ROBE81b ROBERTSON, K. R. "ESP: A Direct Access Editor." Master's thesis, Univ. of Washington, Seattle, 1981.

SALT65 SALTZER, J. "Manuscript typing and editing· TYPSET, RUNOFF." In *The Compatible Time-Sharing System: A programmer's guide.* 2nd ed., P. A. Crisman (Ed.). The M.I.T. Press, Cambridge, Mass., 1965, sec. AH.9.01.

SEYB81 SEYBOLD, J. "Xerox's 'Star'," *The Seybold Report* **10,** 16 (April 27, 1981).

SHAW80a SHAW, A. C. "A model for document preparation systems," Tech. Rep. 80-04-02, Dep of Computer Science, Univ. of Washington, Seattle, April 1980.

SHAW80b SHAW, A., FURUTA, R., AND SCOFIELD, J. "Document formatting systems: Survey, concepts, and issues (Extended Abstract)," Tech. Rep. 80-10-02, Dep. of Comp. Sci., Univ. of Washington, Seattle, Oct. 1980. Also available in the *Abstracts of the Presented Papers, Int. Conf. Research and Trends in Document Preparation Systems* (Lausanne, Switzerland, Feb. 1981), Swiss Institutes of Technology, Lausanne and Zurich, pp. 47-52.

SHNE80 SHNEIDERMAN, B. *Software Psychology.* Winthrop, Cambridge, Mass., 1980.

SHOC79 SHOCH, J. F "An overview of the programming language Smalltalk-72," *SIGPLAN Notices* (ACM) **14,** 9 (Sept. 1979), 64-73.

SMIT75 SMITH, D. C. "PYGMALION: A Creative Programming Environment," Ph.D. dissertation, Stanford Univ., Stanford, Calif., June 1975. Also issued as Stanford Artificial Intelligence Lab. Memo AIM-260 and as Computer Science Dep. Rep. STAN-CS-75-499.

SMIT82 SMITH, D. C., IRBY, C., KIMBALL, R., AND VERPLANK, B. Designing the Star user interface. *Byte* **7,** 4 (April 1982), 242-282.

SPIV80 SPIVAK, M. *The Joy of TEX: A gourmet guide to typesetting technical text by computer,* Version -1. American Mathematical Society, Providence, R.I., 1980.

STAL80 STALLMAN, R. M. "EMACS manual for TENEX users." AI Memo 555, M.I.T. Artificial Intelligence Lab., Cambridge, Mass., Sept. 1980.

STAL81 STALLMAN, R. M. "EMACS, the extensible, customizable self-documenting display editor." In *Proc. ACM SIGPLAN SIGOA Symp. on Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 147-156. Also available as *SIGOA Newsletter* (ACM) **2,** 1&2 (Spring/Summer 1981), 147-156. This report is a revised version of AI Memo 519, M.I.T. Artificial Intelligence Lab., Cambridge, Mass., June 1979.

TESL72 TESLER, L. "PUB: The document compiler," Operating Note 70. Stanford Artificial Intelligence Project, Stanford, Calif., Sept. 1972.

THAC79 THACKER, C. P., MCCREIGHT, E. M., LAMPSON, B. W., SPROULL, R. F., AND BOGGS, D. R. "Alto. A personal computer," Tech. Rep. CSL-79-11, Xerox Palo Alto Research Center, Palo Alto, Calif., Aug. 1979.

THOM75 THOMPSON, K., AND RITCHIE, D. M. *UNIX Programmer's Manual,* 6th ed. Bell Telephone Laboratories, 1975, entry ROFF(1).

VAND71 VAN DAM, A., AND RICE, D. E. On-line text editing: A survey, *ACM Comput. Surv.* **3,** 3 (Sept. 1971), 93-114.

VANL73 VANLEHN, K. A. "SAIL user manual." Rep. STAN-CS-73-373, Stanford Dep. of Comp. Sci., Stanford, Calif., July 1973. Also issued as Stanford Artificial Intelligence Lab. Memo AIM-204.

VANW80 VAN WYK, C. J. "A Language for Typesetting Graphics," Ph.D. dissertation, Stanford Univ., Stanford, Calif., June 1980.

VANW81 VAN WYK, C. J. A graphics typesetting language. In *Proc. ACM SIGPLAN SIGOA Symp. on Text Manipulation, SIGPLAN Notices* (ACM) **16,** 6 (June 1981), 99-107. Also available as *SIGOA Newsletter* (ACM) **2,** 1, 2 (Spring/Summer 1981), 99-107.

Received November 1981, final revision accepted May 1982.