CMU-CS-81-100

Scribe:
A Document Specification Language and its Compiler

Brian K. Reid

October 1980

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at Carnegie-Mellon University

The author was supported by a Computer Science Department Research Assistantship while a graduate student, and gratefully acknowledges the numerous funding agencies, including the Defense Advanced Research Projects Agency, the Rome Air Development Center, and Army Research, which at various times funded that assistantship.

Support for the CMU Computer Science Department research facility, in which this work was performed, was provided by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551. The Xerographic printer on which this document was printed, and the workstations at which the diagrams were produced, were donated by the Xerox Corporation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, expressed or implied, of the funding agencies, the U.S. Government, Carnegie-Mellon University, or the author's advisor or thesis readers.

to Loretta
who saw me through it all

Abstract

It has become commonplace to use computers to edit and format documents, taking advantage of the machines' computational abilities and storage capacity to relieve the tedium of manual editing and composition. A distressing side effect of this computerization of a previously manual craft is that the responsibility for the appearance of the finished document, which was once handled by production editors, proofreaders, graphic designers, and typographers, is in the hands of the writer instead of the production staff.

In this thesis I describe the design and implementation of a computer system for the production of documents, in which the separation of form and content is achieved. A writer prepares manuscript text that contains no mention of specific format; this manuscript text, represented in a document specification language, is processed by a compiler into a finished document. The compiler draws on a database of format specifications that have been prepared by a graphic designer, producing a document that contains the author's text in the designer's format.

To simplify the knowledge representation task in the document design database, the document preparation task was parameterized into approximately one hundred independent variables, and the formatting compiler is controlled by changing the values of those variables. The content of the document design database is primarily tables of variable names and the values to be assigned to them.

To enable substantial feedback from actual users for validating the design, parameterization, and general utility of such an approach, the resulting computer system was built as a production-quality program and documented as a piece of software rather than as an experiment. Released under the name *Scribe*, it has been used as production software at several dozen laboratories. It is therefore possible to report on its effectiveness as well as its design and construction. I conclude with a critical retrospective on the project's basic principles, its implementation, and its overall strengths and weaknesses as compared both to existing alternatives and to an envisioned ideal.

A Linguistic Note

It is customary in scholarly writing to avoid the use of the first person, usually by using the passive voice. A sentence such as

> "I did not get the same results as did Smith when I performed the distillation experiment."

is often transformed into something like

> "The distillation experiment did not yield the same results as when Smith performed it."

In an attempt to recover some of the clarity of the original sentence, a common trick is to rephrase it in the third person:

> "The author's results at performing the distillation experiment were not the same as Smith's."

This thesis is about writing, publishing, and printing. I must frequently refer to "the writer" or "the author", not in an attempt to escape the first person for the third, but to talk about the writer who is using the computer system that I describe, or to differentiate an author from an editor or a proofreader in a discussion of information flow. As a further complication, the word "editor" used in the context of computer science normally refers not to a human being but to a computer program that changes text. Furthermore, following the dictum of current style conventions, the active voice is used [42, p. 13]. I have therefore adopted the following cast of characters in this thesis:

I, me:	Brian K. Reid
the author:	Someone who has produced a written manuscript
the writer:	Same as *the author*
editor:	A copy editor; a human being
text editor:	A computer program to change text

Table of Contents

PART I:
Introduction 1

1. Prior Work 5

2. Goals and Principles 9

2.1 Language Goals 11
2.1.1 Portability 11
2.1.2 Nonprocedurality 12
2.1.3 Domain 14
2.2 Compiler Goals 14
2.2.1 Quality 15
2.2.2 Clerical support 15
2.2.3 Mutability and Definition by Analogy 15
2.3 Documentation Goals 16

3. Typography and Formatting 19

3.1 Letter Placement and Spacing in Text 20
3.1.1 Letter spacing and kerning 20
3.1.2 Ligatures 24
3.1.3 Diacritical Marks 24
3.2 Lineation and Word Placement 27
3.2.1 Word Spacing and Justification 27
3.2.2 Paragraphing 28
3.2.3 Hyphenation 30
3.3 Tabular and Display Material 31
3.4 Page Layout 32

PART II:
Design and Implementation 35

4. The Document Specification Language 39

4.1 Rationale 39
4.2 Syntax 41

4.3 Language Abstract 42
4.3.1 Environments 42
4.3.2 Document Types 43
4.3.3 Commands 45
4.3.4 Declarations 45
4.4 Character Sets and Font Variations 46
4.5 Language Examples 48

5. The Environment Mechanism 53

5.1 Environment Entry and Exit 53
5.2 Types 54
5.3 Dynamic State Parameters 55
5.4 Static State Parameters 56
5.5 Pattern Templates 56
5.6 Definition by Analogy 58
5.7 An Illustrated Example 58

6. The Database 61

6.1 Device Data 61
6.2 Font Data 64
6.3 Document Format Definition Data 64
6.4 Libraries 70

7. A Writer's Workbench 71

7.1 Derived Text 71
7.2 Bookkeeping and Numbering 72
7.2.1 Cross Referencing 72
7.2.2 Indexing 73
7.3 Document Management 75
7.3.1 Division into Parts 76
7.3.2 Separate Compilation 77
7.3.3 Document Analysis Aids 78
7.3.4 Draft Editions 78
7.4 Database Retrieval 80
7.5 Summary and Prospectus 82

8. The Compiler 83

8.1 Overall Organization 83
8.2 Information Flow 85
8.3 The Auxiliary File Mechanism 86
8.4 Data Structures and Data Flow 87
8.4.1 Low-level data Types 87
8.4.1.1 Simple Types 87

8.4.1.2 Records and Storage Management 89
8.4.1.3 Strings 89
8.4.1.4 Association Lists 90
8.4.2 High-Level Data Structures 91
8.4.2.1 Manuscript Files 91
8.4.2.2 Fonts 91
8.4.2.3 Environments 92
8.4.2.4 Text Buffers 93
8.4.2.5 Symbol Table 94
8.4.2.6 Dictionaries 94
8.5 Parsing and Error Reporting 95
8.6 Formatting and Justification 95
8.6.1 Word Assembly 95
8.6.2 Line Assembly 97
8.6.3 Box and Page Assembly 97
8.6.4 Hyphenation 99
8.6.5 Footnotes 100
8.6.6 Floating, Grouping, and Page Break Control 100

PART III: Results, Conclusions, and Future Directions 103

9. An Evaluation of the System 105

9.1 Chronology 105
9.2 Evolution of the Compiler 106
9.3 Evolution of the Manuscript Language 107
9.3.1 Evolution of the Databases 108

10. Critical Retrospective 111

10.1 Language Goals 111
10.1.1 General Language Issues 111
10.1.2 Portability 114
10.1.3 Domain 116
10.2 Compiler Goals 117
10.3 Documentation Goals 118

References 121

Acknowledgments 127

Glossary 129

Appendix A. The State Parameters 133

A.1 Dynamic State Parameters 133

A.2 Static State Parameters 139

Appendix B. Compiler Implementation Details 143

B.1 The Generic Operating System Interface 143
B.1.1 The File System 144
B.1.1.1 Open for Text Input 144
B.1.1.2 Open For Text Output 144
B.1.1.3 Check For Text Input 145
B.1.1.4 Check For Text Output 145
B.1.1.5 Open Unique Text Output 145
B.1.1.6 Close File 145
B.1.1.7 Close and Delete 145
B.1.1.8 Rewind 146
B.1.1.9 Read Text Character 146
B.1.1.10 Write Text Character 146
B.1.2 Address Space Management 146
B.1.3 Environment Inquiry 146
B.1.3.1 Determine Date 146
B.1.3.2 Determine Time 147
B.1.3.3 Determine File Date 147
B.1.3.4 Determine File Time 147
B.1.3.5 Determine User Name 147
B.2 The Generic Device Interface 147

List of Figures

Figure 1: Information flow in a traditional publishing operation. 2
Figure 2: Ideal information flow in an automated publishing operation. 2
Figure 3: Information flow in a typical computerized publishing operation. 3
Figure 4: Type slug, showing protruding kerns. 21
Figure 5: Mechanical (top) and visual (bottom) spacing of the same text. 21
Figure 6: Derivation of kerning lists from spacing matrix. 23
Figure 7: A ligature character. 25
Figure 8: Variations in accent marks of letters within a font. 25
Figure 9: Paragraph with "rivers" of white space. 29
Figure 10: Unusual paragraphing styles. 29
Figure 11: Various schemes for marking text. 40
Figure 12: Font environments in the basic language. 44
Figure 13: Paragraph environments in the basic language. 44
Figure 14: Simple Scribe manuscript. 49
Figure 15: Document produced from manuscript in Figure 14. 50
Figure 16: An elaborate scribe manuscript. 51
Figure 17: Document produced from manuscript shown in Figure 16. 52
Figure 18: Manuscript used for the example in Figure 19. 59
Figure 19: State vector changes during environment processing. 59
Figure 20: Device definition for a photocomposer (part one). 62
Figure 21: A font family definition (Times Roman 10). 65
Figure 22: Sample device font (Times Italic Bold). 65
Figure 23: Document format definition for a business letterhead. 67
Figure 24: Document format definition for CMU thesis. 68
Figure 25: Twenty basic rules for indexers, from Collison [11]. 74
Figure 26: Decomposition of a document into a file tree. 76
Figure 27: Sample document directory. 79
Figure 28: Sample cross-reference summary. 79
Figure 29: Conceptual structure of the compiler. 84
Figure 30: Code space distribution. 84
Figure 31: Scribe data flow paths. 85
Figure 32: Major data flow paths within the compiler. 88

Figure 33: Document specification language grammar. 96
Figure 34: Word tokens, showing bounding and spacing boxes. 98
Figure 35: Use of bounding and spacing boxes in line assembly. 98

Part I

Introduction

Throughout history the reproduction of written material has been a craft requiring an enormous amount of tedious handiwork and a certain amount of intelligence and artistic sense. Beginning with Gutenberg's automation of the process of shaping the letters, various technological advances have reduced the tedious portions of the printer's art, but few inroads have been made into its more cerebral parts. This thesis describes a research project into automating those parts of the printing process that have traditionally required too much skill or artistry to be properly mechanized.

The production of modern-day printed material follows an information flow similar to that shown in Figure 1. An author types his work in rough form, and submits it to an editor. The editor marks various changes, and submits the marked manuscript to a typesetter, who produces typeset galleys. These galleys are then proofread against the original and possibly returned to the typesetter for error correction, and finally passed to the page makeup staff, who cut the galleys into page-size pieces, placing figures and footnotes and adding page numbers and other "running head" material. If the book is to have an index, then page proofs are hurriedly sent to an indexer, who produces the index for the book from the page proofs. The index is then rushed off to be typeset and made up into pages and added to the end of the book, which is then printed.

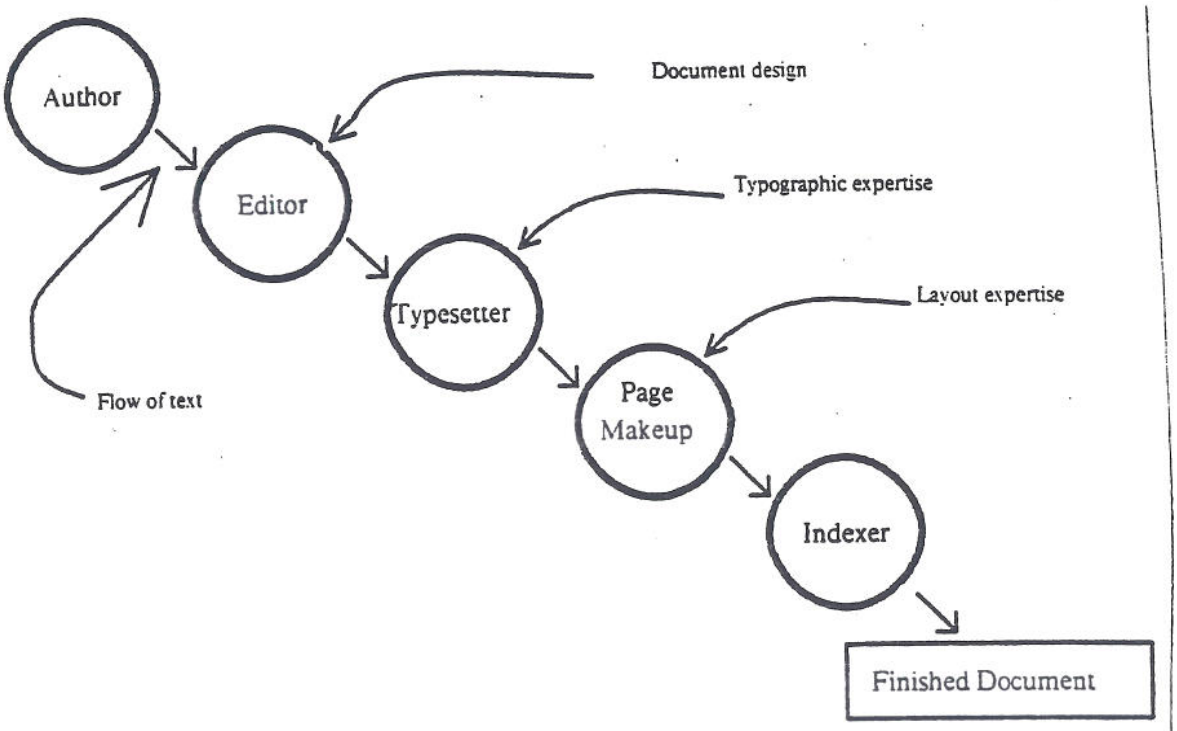


Figure 1: Information flow in a traditional publishing operation.

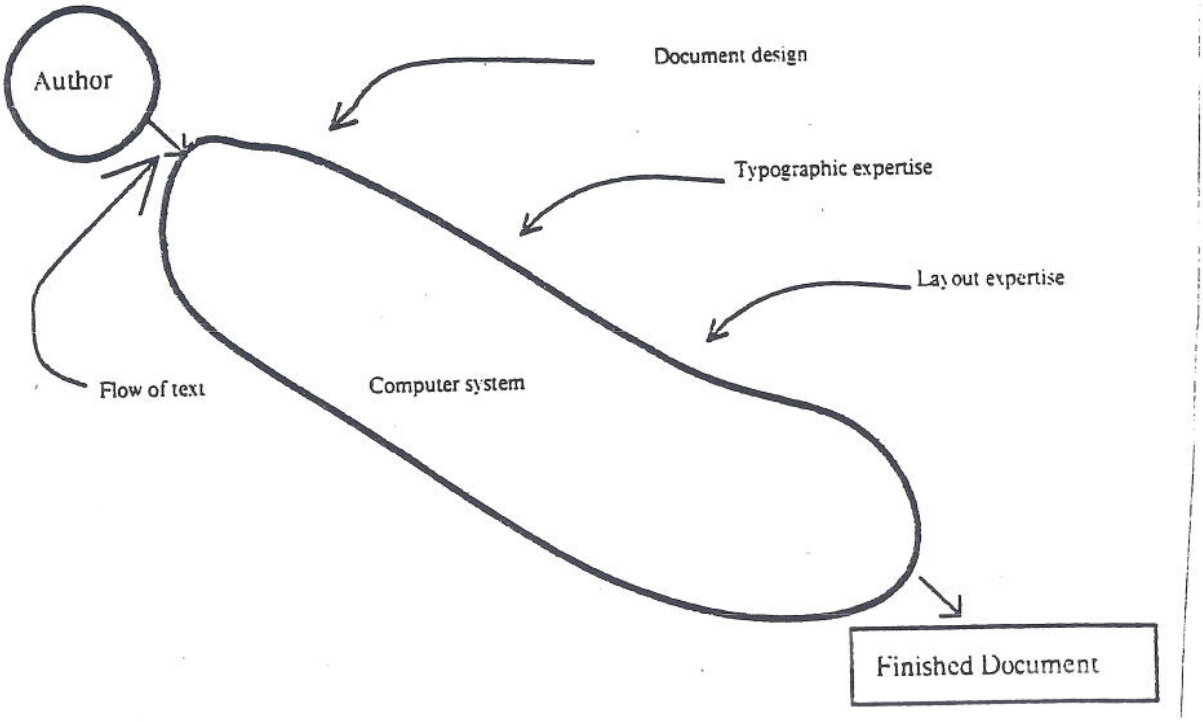


Figure 2: Ideal information flow in an automated publishing operation.

There are numerous sources of cost and delay in this production scheme. When the text is passed from each person to the next, errors and misunderstandings inevitably occur. Various tedious aspects of the page layout, such as footnote placement and cross-reference resolution, need to be completely redone if small changes to the text cause pagination to change. The index cannot reasonably be produced until the book is completely finished and all of the pagination decided.

We would like to be able to perform all of the tedious production work with a computer, so that the flow of work would be as shown in Figure 2. In comparing Figures 1 and 2, note that they differ only in the substitution of a computer system for several of the currently-manual processing steps.

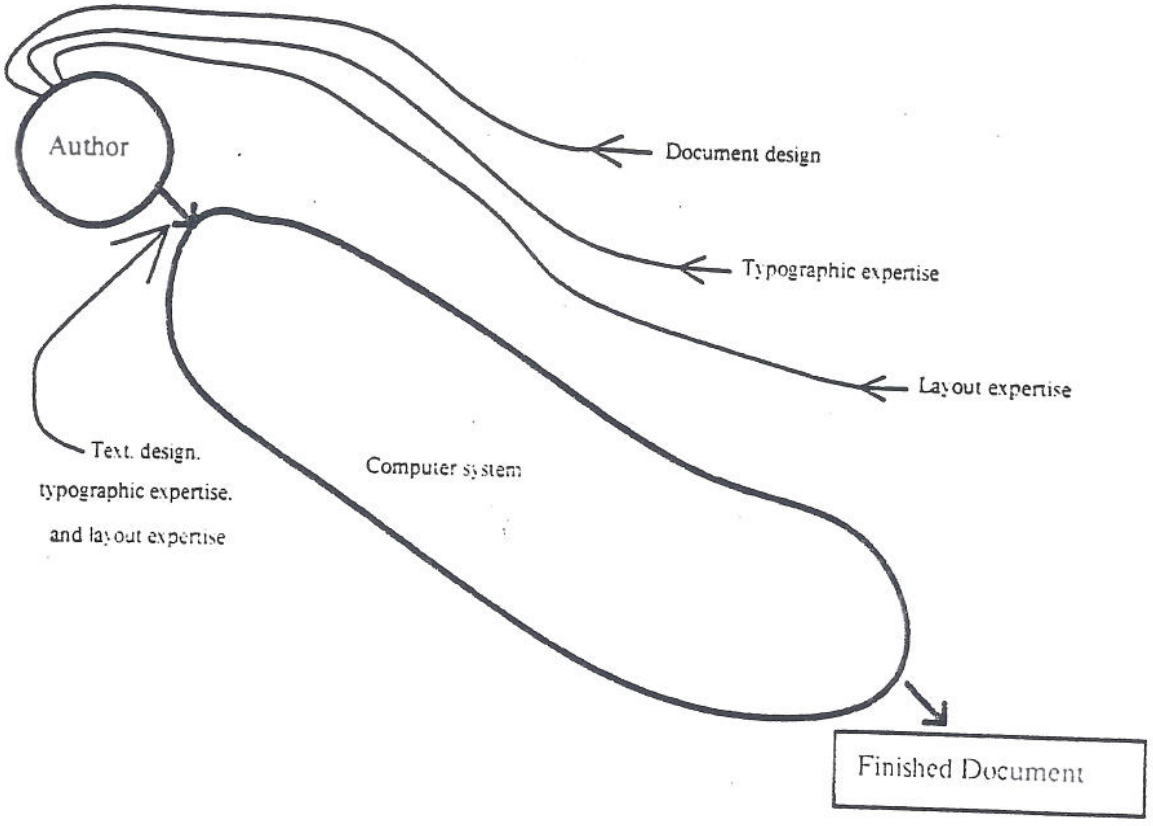


Figure 3: Information flow in a typical computerized publishing operation.

Previous attempts at complete computerization of the printing process, while technologically successful, have lead to a disruption of the traditional flow of information and expertise. Figure 3, for example, shows the flow of information and sources of expertise in a typical computerized publication operation. The author is now responsible for essentially all of the final appearance of the document, since the control codes that determine the appearance of the finished document are

intermixed with the author's text, and often typed by the author himself. While many authors enjoy this involvement in the physical and artistic aspects of the printing of their work, not all are interested or qualified [15].

The Scribe document specification language was designed to permit writers to prepare text in a relatively informal manuscript form that contains little or no typographic information. This language is processed by the Scribe compiler, which supplies all of the missing typographic detail to produce the final document. This first part of the thesis is devoted to a discussion of the ideas behind the language design and the principles behind the compiler design, and to the problems that need to be addressed by any document preparation system, whether automated or manual. Chapter 2 details the goals for the Scribe language and compiler. Chapter 1 sketches the prior work in computer document production. Chapter 3 discusses the issues raised and problems to be solved in document formatting.

Chapter 1
Prior Work

The early applications of computers to document formatting were concerned either with computer control of commercial typesetting equipment or with crude monofont formatting for a line printer. Very little of the pioneering work was recorded in the literature, but one can get a sense of the goals indirectly from the tone and intended audience of the instruction manuals.

The earliest text-formatting program known to me is the *Print* program completed in 1959 at Johns Hopkins University by R. P. Rich. It ran on an IBM 1401 computer, and produced output for an all-uppercase line printer [38]. Interestingly enough, it was not designed explicitly as a document preparation program, but as an information retrieval aid for a simple database system—it obviated the need for the textual data being stored to be in any particular format.

In 1963 Barnett, Moss, Luce, and Kelley reported the successful completion of a computer-controlled typesetting system that operated an optical photocomposer. The input commands in their formatting language corresponded to the physical capabilities of the typesetting machine: there were commands to change fonts, change magnification, position text, and so forth [4, 5]. Also in 1963, a formatting program for the IBM 7090 called *Text90* became available. Produced by G. Burns, it formatted text for a line printer, and with special print trains was capable of generating mixed case and special symbol output from punched-card input.

These two programs, one representing the point of view of commercial typesetting and the other the point of view of a software documentation writer, were the opposite ends of the spectrum in terms of their goals. Barnett *et al.*'s program placed typographic quality as the foremost goal, requiring the user to learn the nuances of the typesetting machine and to communicate with it in a language that is by modern standards unintelligible. Text90 placed simplicity of input as a high-priority goal, and since it could not achieve quality typography on its output line printer, it almost totally ignored questions of typography, concentrating instead on control and simplicity. The designers of all formatting programs must steer a compromise path between these fundamentally conflicting goals of simplicity and power, and in

studying these and later programs it is worthwhile to note the compromises of simplicity that were made in the interests of power and the compromises of power that were made in the name of simplicity.

Manuscript conventions used in Text90 have carried over into many similar computer text formatters. J. Saltzer's *Runoff* program developed at MIT for the CTSS system, which was operable by 1966, is the direct ancestor of most of the next generation of formatters such as *Roff, TRoff, Script, Pub,* and *Text360* [20, 34, 43]. The formatting programs in this family all used the input language convention that a flag character in the first position of an input record denoted a command and other lines were text. The early programs in this family were restricted to monofont printing devices; TRoff and Pub later provided multiple fonts and character sizes. While the basic command structure of these programs was device-oriented and tedious, some of them provided a macro facility that allowed an ambitious user to produce high-level commands by macro combinations of the existing commands.

Barnett's work at MIT led to the development by P. Justus of the *Page-1* formatting system at RCA, as part of the development effort for the RCA VideoComp photocomposition system [22, 35]. Justus later produced *Page-2*, a successor to Page-1. RCA sold its interests in the VideoComp hardware and software to Information International Inc. (III), who continued the software development on Page-2. Bell Labs' TRoff and III's Page-2 are currently the most widely used programmable photocomposition languages. A successor to Page-2, named *Page-III*, has recently been released.

In 1965, M. V. Mathews and J. E. Miller of Bell Labs reported a system for editing and typesetting that involved a high-resolution oscilloscope with a camera mounted in front of it [27]. Although it used a display tube, which in current technology is associated with interactive systems, the Mathews and Miller system was a batch system. It was similar in philosophy to the Barnett program, but did not have the commercial-grade typesetting machine available as an output device. High-quality mathematical and oriental-language typesetting was achieved by A. V. Hershey, of the Naval Weapons Laboratory (Dahlgren), who produced both a typesetting system and a typeface design system that could handle calligraphy and oriental languages as well as normal type [19].

Until about 1975, the trend in document preparation programs was towards increasing programmability by macros or interpreted commands. Essentially all were compiler-model programs, which is to say that they operated on a prepared manuscript file to produce the output file, with no interaction with the user. (The *Quids* interactive documentation system developed at Queen Mary College by Coulouris *et al.* is a notable exception [12].) The DPS program developed at the University of Maryland by K. Sibbald in 1973 epitomizes the algorithmic

approach [40]. Its manuscript language was imbedded in an interpreted programming language similar in style to Snobol; almost every user-level command was microprogrammed in this interpreted language. Other notable algorithmic systems are the Script family of programs [20], and the Texture system developed by M. Gorlick *et al.* at the University of British Columbia [14].

These early formatting programs had the common property that they all processed a low-level device-dependent input language. The user needed to modify the manuscript file to format for a different device, and needed to be aware of the detailed properties of the printing device if he wanted to use them. In 1975 the first high-level formatting system was reported by B. Kernighan and L. Cherry [23]. Their EQN system for typesetting mathematics processed a high-level machine-independent language into a formatted mathematical expression, regardless of the particular printing device used. EQN was actually implemented as a preprocessor to TRoff, but that fact was essentially invisible to a casual user. The concept behind the EQN system—a high-level problem-domain language with a processor that handles all of the device-dependent details—is one of the major concepts embraced by the work reported in this thesis.

The Yorktown Mathematical Formula Processor, developed by N. Badre at the T. J. Watson research center, is extremely similar to EQN in concept and implementation [3]. Another high-level system conceptually similar to EQN was the Generalized Markup Language (GML) developed starting about 1970 by C. Goldfarb at the IBM Cambridge Scientific Laboratory, and first available in 1978 [13]. It is a modification to the basic IBM Script system that allows automatic database retrieval of appropriate macro definitions according to the printing device selected.

Also reported in 1975 by B. Lampson was the Bravo system [25]. Although its only description is a user's manual that has never been published, the Bravo work has strongly influenced the design of text editing and formatting programs [33]. One expects that as computer hardware capable of supporting such systems becomes generally available, its influence will be more obvious. Bravo is a display-oriented formatting editor, running on a raster-display graphics terminal capable of displaying an entire page of text in actual size. The essence of Bravo is the maintenance on the screen of a faithful image of the finished document with all fonts, spacing, and letter sizes current on the screen. As the text is changed, the display is quickly updated to reflect that change. Unfortunately, the size of Bravo's video screen (8 inches wide), the resolution with which dots can be displayed on it (80 per inch), and the useful resolution of the pointing device used to select letters on it (about 0.05 inches) led to an implementation of Bravo in which the screen is only a crude approximation of the final output; in particular, line breaks do not appear in the

output as they did on the screen. As a result, Bravo is nearly useless for high-precision formatting.

As computer-controlled typesetting matured, more attention was turned to the quality of the typesetting. N. E. Wiseman, C. I. O. Campbell, and J. Harradine developed a book-production system at the University of Cambridge for the Cambridge University Press; it was reported in 1978 and is in production at the University Press [48].

In 1978, D. E. Knuth of Stanford University described his landmark TEX (Tau Epsilon Chi) system [24]. TEX was designed to give a writer the ability to produce technical manuscripts of the highest quality. Intended primarily for the production of books and other high-quality manuscripts containing large amounts of mathematics, it incorporates and expands upon many of the fundamental ideas of EQN in a formatting program that takes typographic quality seriously. The resulting system is very successful; and has proven to be extremely powerful in the hands of expert users. Many of the algorithms used internally by TEX for line breaking, hyphenation, page layout, and justification are notable improvements over the classical algorithms used in essentially all prior work as well as in Scribe. These algorithms are mentioned briefly in appropriate places in Part II of this thesis.

While TEX is the asymptotic case of a system that is willing to sacrifice in the interest of quality of the finished product, programs at the other asymptote—systems that sacrifice everything in the interests of simplicity—have been in use for some time in the publishing industry. Usually called *idiot text* systems in the printing industry, they process raw text that contains no commands at all, to produce galley files for commercial typesetting systems. These galley files must then be manually edited to override mistakes made by the idiot text system, but the bulk of the work—input of the actual text characters—does not need to be repeated. None of these systems is described in the literature.

Chapter 2
Goals and Principles

The ideal text formatting system is a good secretary. He can be given rough handwritten manuscript text and from it produce a polished document in appropriate form. A near-perfect separation of content and form is achieved: the writer provides only the text and the secretary performs all of the formatting, though possibly the secretary is assisted by clues or remarks placed in the text by the writer.

The fundamental goal of the research reported in this thesis was the design, construction, and documentation of a computer document preparation system that offers the same level of support for a writer that a good secretary does. Ideally, the writer would provide only text, and the computer system would correct spelling and grammar and perform all of the formatting.

The methodology used was to design a document specification language that embodied the kinds of information that a writer might reasonably be willing to convey to a secretary, and then to devise a compiler capable of compiling that language into an actual finished document. In the course of designing the compiler, it was found necessary to incorporate into it various specialized knowledge about typing and formatting as well as a more general mechanism for adding new knowledge to the compiler. The overall development strategy was to preserve the simplicity and domain of the specification language regardless of the complexity needed in the compiler to compile that language properly.

Since the overall project goal involved the construction of a working compiler for release to actual users, various subsidiary goals for that construction were adopted. Most of these goals amount to good engineering practice rather than innovation, and would be equally applicable to other compiler-like programs. Some of the goals specific to the document production task were motivated by negative experience with earlier document production systems.

After suitable reflection on the aggregate of project goals, design principles, and physical limitations on the available editing and printing devices, the following outline was set for the entire Scribe effort:

- Design a document-specification language of documents that frees the author from the need to specify any output format details but encourages him to identify and label the components of a document.

- Design and implement a compiler to process that language into finished documents. The compiler is to provide all of the details of formatting that were omitted from the manuscript.

- Design a knowledge representation and retrieval mechanism for the storage of these format details that will permit the compiler to be made to produce a wide range of document formats with reasonable efficiency and grace, and that will permit users of the system to define their own document formats or modify existing formats.

- Determine how to teach the system to novices, and write an introductory manual that presents the material properly. Since the system differs from existing similar systems in concept and not just in detail, the manual should also be able to present the material to people who are experienced users of other systems. Two different approaches might be needed for these two different audiences.

Users rarely perceive a system in terms of its separate design components, but will instead see and use it as a monolithic whole. Various goals were therefore set for the whole Scribe system, as understood and used by its user community. These goals flavored the design of the language, the structure of the compiler, and the organization of the manual. Although these goals were set as guides for the implementation, they are actually goals for the user's perception and style of use of the finished system.

The remainder of this chapter is devoted to more detailed discussion of those goals, principles, and beliefs that together motivated the design of the Scribe language and shaped the implementation of its compiler. Chapter 10 reviews these same goals, with commentary and analysis of how realistic they were and how well the finished system managed to meet them.

2.1 Language Goals

The Scribe document specification language is the language in which manuscript files are prepared. The compiler produces finished documents by processing files in this language. We want the document specification language to be able to serve both as the input to the compiler and as a communication language for the transmission of documents from one site to another. Furthermore, the language is to be nonprocedural, which is to say that it should direct the final result of the compiler without regard to the details of processing needed to achieve that result.

Nonprocedurality means that "statements" in the document specification language should be viewed not as imperative commands to the compiler, but as goals for the compiler. Furthermore, since the substantive part of a manuscript file is its text, the specification language is best viewed as a commentary on that text, as a set of labels or annotations marking sections of it. These labels can be very abstract: by combining the role label with specific goal information from the database, the compiler can determine the necessary or appropriate concrete action.

The document specification language must be able both to label regions of the text, as for example "this is a chapter heading", and to mark specific points within the text, as for example "a footnote reference goes here". The compiler or human reader must be able to distinguish unambiguously the text from the text labels. More conventional document production systems—publishing houses, for example—use visual methods, such as colored pencils or marginal notations, to distinguish text from labels. Since our language must be representable as a linear stream of characters, there must be some way of distinguishing text characters from label characters.

2.1.1 Portability

If the manuscript form of a document is not tied, explicitly or implicitly, to a particular printing device, we say that it is device-portable. If it contains nothing that ties it to a particular computer, then we say that it is site-portable. The mention of specific margins or amounts of spacing between lines or the mention of specific fonts, for example, will make a document be dependent on a particular printing device; the mention of file system directory names or "library" files not part of the manuscript will make it be dependent on a particular computer site.

If a manuscript file is both device-portable and site-portable, then it can be transmitted to another site as a means of communicating the document without the sender and receiver needing to agree on compatible manuscript conventions. The

receiver can compile it locally into a document, using whatever printing device is convenient.

We therefore require that the document specification language used in manuscript files be completely site-portable and device-portable, in order that it can be used as a document communication language as well as a document specification language. The necessary device-dependent details must be filled in by the compiler, which must therefore be sophisticated enough to generate the concrete device-specific document from an abstract device-independent manuscript.

There are two different interpretations of the notion of device portability. The first might be called "imitation of the ideal", and the second might be called "making do with the resources at hand". The first approach, imitation of the ideal, embraces the notion that there is one true output format for a document, namely the one that would be produced by a typesetter with an unlimited supply of fonts. Lesser printing devices are just an imperfect imitation of this ideal format, and one achieves portability by imitating the ideal format as closely as possible on the printing device at hand. The second approach, making do, assumes that the user is not interested in printing an imitation of a typeset document on some lesser machine, but rather in producing something that is maximally readable and attractive on the printing device at hand. The design goal for Scribe was to produce the best output for each kind of printing device, rather than to imitate the ideal.

2.1.2 Nonprocedurality

If the primitives provided by any system, whether digital computer or bank teller machine, coffee percolator or kitchen stove, do not directly fulfill the needs of the user, then he must synthesize the desired behavior by combining the primitives provided into patterns that yield the desired effect. Systems that are designed to be general-purpose, such as digital computers and kitchen stoves, typically provide low-level primitives that must be combined into higher-level functions before they are of any direct value to the end user. Systems that are designed to be special-purpose, such as automatic bank teller terminals and coffee percolators, provide the functionality needed by the intended user as direct system primitives.

There is clearly a continuum of possibilities between purely procedural and purely nonprocedural systems. If a system can be used directly, without synthesis, to solve the problem at hand simply by our describing to it the desired effect, then we call it purely nonprocedural. A vending machine is a pure nonprocedural system in the domain of food distribution: the desired result (candy bar, peanuts, gum, ice cream) is communicated to the system by way of its specialized keyboard, and the

mechanism within it delivers up the candy bar by means invisible to the user. The details of the algorithm used by the machine to locate and deliver the candy bar vary with its storage allocation schemes and implementation quirks. Their varying effects are sometimes discernible by an alert user in terms of delays or noises, but the result is normally the desired food item.

If a system cannot be used directly to solve the problem at hand by our just describing the goal to it, then it is at least partly procedural in that problem domain. Sometimes a system can be lured into solving a problem by giving it a series of sub-goals, each of which it is able to achieve, and the sequence of which will yield the desired effect. For example, there is rarely a key marked "tea with cream" on a beverage vending machine, though there is one marked "tea" and another marked "extra cream". By depressing first the "extra cream" key and then the "tea" key, tea with cream can be had. This is a simple procedure requiring little strategy and little knowledge of the internals of the machine in order to achieve a goal that is closely related to the domain that the designer intended for the machine.

Sometimes considerable strategy and knowledge of the implementation of a system can be used to coerce it into solving a problem substantially outside its originally intended domain. For example, a certain ice cream vending machine can often be used to get exact change for bus fare, assuming that a supply of quarters is available (bus fare is sixty cents), and that a possibly-borrowed "seed nickel" is available. The ice cream machine is designed to sell ice cream at a price not to exceed fifty cents. Its coin accepter will accept fifty cents in any form, and will then stop accepting new coins. If the coin release button is pushed while fifty cents or less is in the coin accepter, then all of the original coins will be returned. However, if a single nickel is placed in the machine, followed by two quarters, the second quarter will exceed the fifty-cent retention threshold of the coin accepter. Rather than retaining the second quarter in the coin accepter, the vending machine will drop it irretrievably into the coin box, and record its amount in a register. An attempt to insert a third quarter will be rejected, since the accepter is now over the fifty cent threshold. If the coin release button is now depressed, the ice cream machine will return the original nickel, the original first quarter, and five nickels from some internal supply. This process can be repeated indefinitely until the machine runs out of nickels or the would-be bus rider runs out of quarters.

Although the change-making example is relatively far-fetched, it is a good example of a system that is intended to be purely nonprocedural in a fixed domain being used procedurally to solve a problem radically outside that domain.

We require the language used to specify documents to be nonprocedural in the document specification domain, i.e., that a writer must not have to synthesize needed functionality from the primitives at hand, but should be able to use them

directly. This implies specialization: though suited for the specification of many documents, this language might not be appropriate for general computational purposes, or even for the specification of certain kinds of documents such as airplane tickets or road maps.

2.1.3 Domain

The scope or problem domain of a low-level procedural system is not well-defined—it can be used to solve those classes of problems for which its users are willing to synthesize solutions. There is generally a "kernel" domain that corresponds to the problems that the system designer had in mind when designing the primitives, but it is rare to see the use of a successful low-level procedural system restricted as its designer intended. The domain of a higher-level, more nonprocedural system is much more sharply defined.

Document formatting tasks are particularly hard to characterize, since their only common property is that they include marks on paper. A crossword puzzle is a document, and so is a display advertisement, but the algorithms executed to produce them and the criteria for success are completely different.

Scribe was designed to be able to handle the vast majority of the document preparation tasks found in a computer science research environment: academic papers, instruction manuals, homework assignments, an occasional textbook, Chinese recipes, business letters, and so forth. There was a conscious decision *not* to make it completely general so that it could be adapted to the production of all documents, but rather to assume a reasonably fixed domain and then try to characterize (and later parameterize) that domain. I considered it far more interesting to be able to do a really good job of producing 95% of the documents that people wanted than to be merely able to produce anything.

2.2 Compiler Goals

The Scribe compiler is to serve two purposes: to compile the author's specification into a document, and to provide document management and bookkeeping assistance to the author during document development.

2.2.1 Quality

In order to attract and keep users, the compiler must have a production-quality aura about it. This includes robust recovery from abject errors in the manuscript, responsible and accurate diagnostics phrased not in the compiler's terms but in the user's terms, and enough speed and reliability that people can actually use it. Nevertheless, the prototype compiler developed during this research work, even though it was expected to be released as software within Carnegie-Mellon's Computer Science Department, did not have ambitious goals with respect to speed or workmanship. It was instead to be organized in such a way as to permit maximum flexibility, encouraging experimentation with the language.

2.2.2 Clerical support

Much writing, especially technical and expository writing, requires a great deal of clerical support. Technical material is normally cross-referenced and indexed. Documents contain glossaries, bibliographies, tables of figures, or other derived text. Documents often contain fixed or boilerplate material that is assembled from various sources; it is useful to be able to postpone that data retrieval as long as possible in order that the most recent version be used, and that the manuscript file not contain an obsolete copy of the text.

We want the Scribe compiler to take on as many of these clerical support tasks as possible, both to free the author for more important work and to ensure the accuracy of the finished product. In extreme cases, the actual manuscript might contain no text except a title; the remainder of the document would be assembled by the compiler by appropriate database retrieval.

2.2.3 Mutability and Definition by Analogy

The *mutability* of a system is its ability to sustain gracefully various changes in its behavior. Many high-level computer systems permit the user to extend the system or redefine components of it by supplying a complete definition or redefinition of the procedure that implements them. The mutation of a system by reprogramming requires that one understand its primitives and be able to synthesize the desired new behavior by appropriate procedural combinations of those primitives, which is precisely the same set of skills needed to program it in the first place.

We require that the user be able to make *incremental modifications* or *definitions by analogy*; the user skills required to make such a mutation must be proportional to

the complexity of the change and not the complexity of the resulting changed object. Since the user is not expected to be able to program, mutation by reprogramming is not an acceptable method. The target users for this document preparation system certainly should not be expected to learn an elaborate definition language in order to be able to make small changes to the system behavior.

An incremental modification is a request to "change the definition of X so that the z property of its behavior is now q_z instead of p_z; leave all other facets of its behavior alone." A definition by analogy is a request to "define Y to be just like X, except that its z property is q_z instead of p_z." Less formally, an incremental modification is a specification for change to the definition of some standard compiler function that specifies only those characteristics of it that should differ. The parts of the compiler function that are not mentioned one way or the other in the change request are left untouched. Typically, the compiler's database would contain a definition for some relatively complex entity. Rather than providing a complete redefinition of the entity, the user specifies in his manuscript file an incremental modification that modifies that entity for the duration of the compiler run.

2.3 Documentation Goals

The user documentation is an essential part of any system design, but it is too often left until after the construction is completed. A comprehensive tutorial manual was an integral part of the system design of Scribe, and it ultimately played a crucial role in the evolution of the design of the system.

As compiler development and language changes progressed, the *User's Manual* was updated in parallel, though not necessarily on a daily basis. Any proposed modification to the specification language that was not easily documented, or whose documentation would not fit harmoniously into the existing manual, was rejected for that reason alone. The manual represents the view of the system seen by the user, and any complexity of the system that generated complexity in the manual, with resulting complexity in the user's mental model of how the system works, was considered a compromise of the design integrity of the system and therefore a bad idea.

Faithfulness in the maintenance of the manual during periods of system design activity, without resorting to "fine print" detailing the exceptional cases, is the best single control against the design evolving into the baroque morass of details and "features" that befalls many systems as they mature.

The user's manual for a system is an informal specification of its behavior. While valuable as a tool for preventing the design from becoming unmanageable, it is rare

to find a situation in which the implementation of a system does not force changes in its specification. One reason for this is that the informality of the user-manual level of specification often masks inconsistencies in the design. The use of a more formal specification scheme as part of the design process, as suggested by J. Guttag and J. J. Horning [18], could substantially improve the effectiveness of this sort of watchdog methodology. The design work on the Scribe project was completed before I became aware of the work of Guttag and Horning, else I would have attempted to use their methodology.

The specific goals for the user documentation were to produce three distinct documents aimed at different audiences. The *User's Manual* was to be a tutorial that made no assumptions about the background of the reader other than that he could use a computer and a text editor. The *User's Manual* was intended to be read front-to-back by a beginning user. The *Pocket Reference* was to be a summary of the information contained in the *User's Manual*, bound in such a way that it will fit in a pocket, and organized alphabetically by function. The third manual was to be the *Expert's Manual*, an advanced manual containing information that expert users and system maintainers will need in order to add to the database.

Chapter 3
Typography and Formatting

The preeminent English typographer Stanley Morison defined *typography* as "the art of rightly disposing printing material in accordance with specific purpose; of so arranging the letters, distributing the space, and controlling the type as to aid to the maximum the reader's comprehension of the text. Typography is the efficient means to an essentially utilitarian and only accidentally aesthetic end, for enjoyment of patterns is rarely the reader's chief aim" [31, p. 1]. A good typographer strives to produce documents that are both beautiful and legible. Where the two conflict, he must normally choose legibility.

Many illuminated manuscripts are beautiful at the expense of legibility, and many mass-distribution publications such as newspapers are legible without being noted for their beauty. Numerous studies have been published of the legibility of written material, each reaching a slightly different conclusion.

For example, in a classic textbook on typography and graphic design, Arthur Turnbull has concluded that readers find most legible that which is most familiar to them, and that all other factors are secondary [44]. Morison insists that "The typography of books requires an obedience to convention which is almost absolute" and "for a new font to be successful, it has to be so good that only very few people recognize its novelty" [31, p. 7]. S. H. Steinberg muses that "A book which, in some way or other, is 'different', ceases to be a book and becomes a collector's piece or museum exhibit, to be looked at, perhaps admired, but certainly left unread" [41, p. 28].

As typographic skill is transferred from the artists who devised it to the craftsmen, apprentices, and machines who will be performing it, that which was once just the artist's taste and judgment must be codified as rules, for the benefit of those not gifted with an artist's instincts. Various typographic traditions have evolved into numerous standards of correct practice; most of them are expressed as positive or negative constraints on the finished document. For example, one standard for the factoring of lines into pages requires that the last line of a paragraph not appear by itself at the top of a page [1].

Not all of the published rules are consistent with one another. A textbook for printers published in 1915 specifies that the inter-word spacing be reduced by 15 percent when the last letter of the first word and the first letter of the second word both have ascenders or descenders, e.g., between "shall be" or "and probably." [2, p. 40]. A recent monograph on typographic design specifies that in precisely the same circumstance, the inter-word spacing be *expanded* by the same amount [7, p. 33].

Printers have nevertheless traditionally been loath to reduce their artistic principles to a set of simple constraints; indeed, one reference work for printers explains:

> "Owing largely to the conservative ideas prevalent among printers in general, it is somewhat difficult to lay down hard and fast rules." [2, p. 39]

Even if the rules cannot be made hard and fast, they must at least be made rigorous and consistent, as specific constraints, before they can be used to guide directly any formatting program. It doesn't really matter which set of rules is used, but there needs to be some set.

This chapter is a discussion of the major and interesting traditions for the typography of Western languages, with consideration given to the constraints that those traditions place on computer programs engaged in typography. Where appropriate, data structures or algorithms appropriate for their implementation are discussed. Various terms from typography and printing are used without much explanation; the reader is referred to the glossary on page 129 for their definitions.

3.1 Letter Placement and Spacing in Text

The requirements on individual letter positioning and spacing are relatively insensitive to context, requiring at most the consideration of a small amount of context near the letters in question.

3.1.1 Letter spacing and kerning

Classical type fonts were designed around the idea that each letter was on a rectangular slug, and the width of that slug determined the width of the letter. The width was thus always the same for any letter, regardless of the context in which it was used. Figure 4 shows a drawing of one such rectangular type slug. The semicircular notch at the bottom of the body helps the typesetter more easily detect upside-down letters. Some type faces, such as italic, are slanted enough that parts of the letter needed to protrude beyond the edges of the slug. These protrusions, are called *kerns* [44, p. 58]. When a type slug having a kerned letter is placed next to

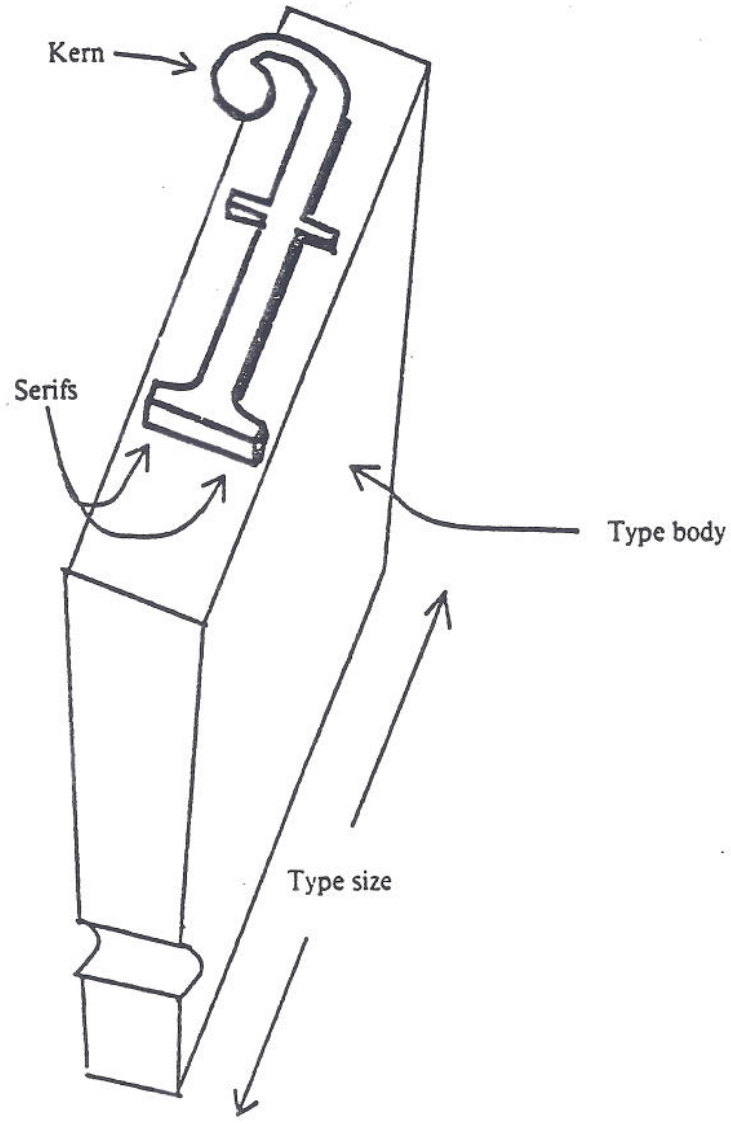


Figure 4: Type slug, showing protruding kerns.

Variable

Variable

Figure 5: Mechanical (top) and visual (bottom) spacing of the same text.

another slug, the kern overlaps the body of the second slug, providing a closer spacing than could otherwise be achieved.

When letters are not stored on rectangular slugs and are thus free of mechanical constraints on spacing, better spacing can be achieved. The term "mechanical spacing" refers to letters that have been spaced exactly as if they came from type slugs. See, for example, Figure 5. The words in the top row have been set according to a simple mechanical spacing, while the words in the bottom row have been set according to a more complicated "visual" spacing algorithm, in which the spacing between letters is dependent on those letters. Similarly, the amount of space to be left after a period depends on the capital letter starting the next sentence: if it is a *T* or a *Y* or an *A*, for example, the amount of space to be left after the period is reduced by a certain amount [44, p. 59].

If letters are mechanically spaced, with the amount of space given to each letter independent of its context, then a simple table of widths is sufficient to represent a font. If the actual printed width of a letter differs from the amount of space it is to be given on the page, as for example the script letter *j*, then a separate table of spacing increments is needed.

For the compiler to be able to implement non-mechanical spacing, the context that determines spacing must be bounded, and preferably fixed. For all practical purposes in body-sized text, it is sufficient to compute the space between two letters without considering any letters but those two. Regardless of how this distance is derived, it can be stored in a matrix indexed by letter pairs and used to determine the spacing. Although the complete $n \times n$ spacing matrix would be large (n for most fonts is one or two hundred), it is sufficiently regular that much more space-efficient encodings of the information can be used.

By subtracting the modal (most frequent) element from each row of the spacing matrix, and placing that modal element into the corresponding element of a vector, then the spacing matrix is reduced to a space-adjusting matrix or *kerning matrix*, and the vector so derived becomes a table of widths. The kerning matrix will be relatively sparse, and it will contain regularities based on the equivalence classes of the left and right edges of letters: the spacing between a left-hand letter and those with vertical edges (b, B, P, N, h, etc.) will normally be the same. The sparse kerning matrix can then be represented with short lists of kerning values by equivalence class of the letter's left edge, attached to the width vector. Figure 6 shows the various steps in this derivation.

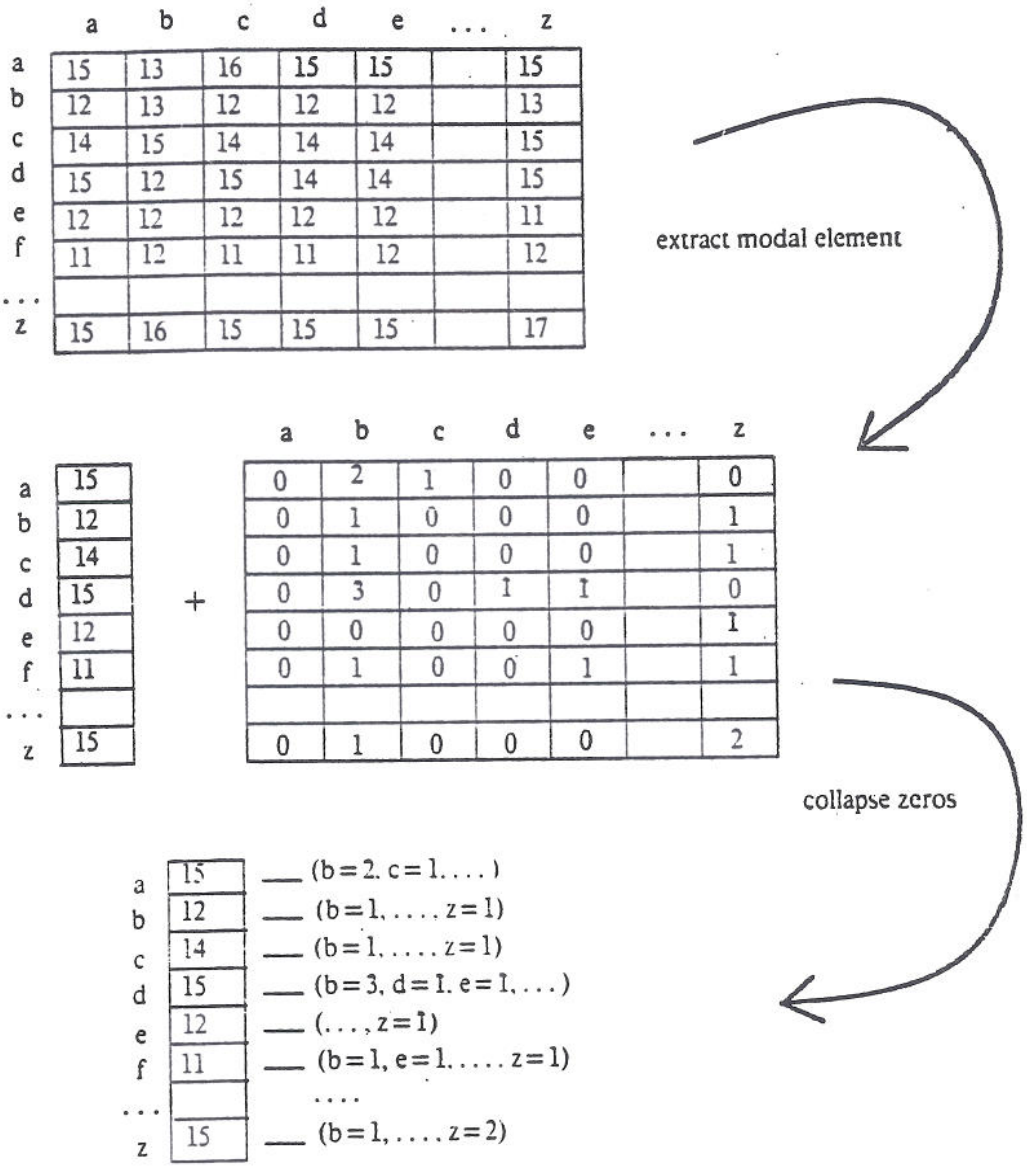


Figure 6: Derivation of kerning lists from spacing matrix.

3.1.2 Ligatures

In certain fonts, notably script letters and body fonts with serifs (Figure 4 shows serifs), the very shape of letters changes when they are used in certain groups. For example, when the letter *i* is used following the letter *f* in a Roman alphabet with serifs, it is customary to omit the dot over the *i*, letting the dot that is part of the top of the *f* serve that purpose: "*fi*". This shape-change is normally handled by designing a *ligature* character, which is a single character that prints in place of two or more letters. Figure 7 shows the type slugs for several ligature characters.

In modern Roman-alphabet fonts only the combinations "*ff*", "*fl*", "*fi*", "*ffi*", and "*ffl*" remain as ligatures, but in the early days of printing, there were hundreds of ligature characters. A Greek font developed about 1495 by Aldus Manutius (the most commercially successful printer of his era) had more than 1300 ligature characters in addition to the 50 or 60 ordinary alphabetic characters [29, p. 280]. Part of the job of a font designer was to decide which letterspacing could be handled with kerning, and which actually required ligation. Current typographers consider the Aldine Greek font to be a black mark on the record of an otherwise artistic designer.

For the compiler to be able to process ligatures, the information kept about each font must include a list of the ligated letter combinations and the printing sequence in the font that will generate the appropriate substitute character. The information available about each ligature must include its manuscript key ("ff" to generate *ff*, for example), the width of the resulting ligature character (not generally equal to the sum of the widths of the ligated characters), and the device-dependent code sequence that will actually cause the character to be printed.

3.1.3 Diacritical Marks

Most Roman languages have diacritical marks, or accents, that can be applied to letters to indicate a change in pronunciation, to indicate stress, and so forth. Most of them go above the letters that they mark:

Ä é Ö Û è Å ñ

but others go below the letters or even through them:

ç Ø ¢

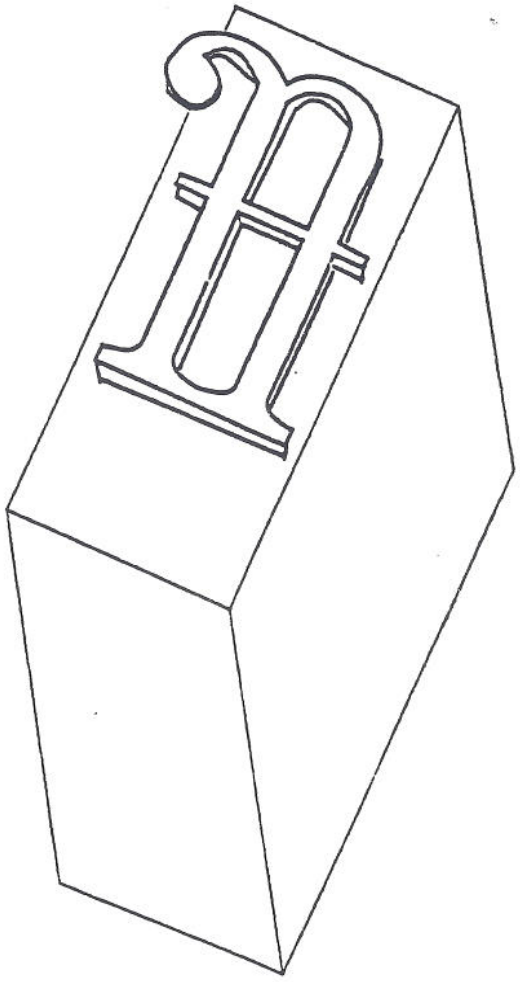


Figure 7: A ligature character.

Figure 8: Variations in accent marks of letters within a font.

Although English as commonly written no longer needs any diacritical marks, mathematical notation is rich with them, and many special-purpose applications such as pronunciation guides rely heavily on diacriticals.

A detailed examination of the accenting process will show that it is more intricate than the simple superposition of two characters. Figure 8 shows five different letters in the same font and point size that have been marked with a circumflex. Note that no two of the circumflex marks have quite the same position, and not all of them have the same size. We must consider the application of a diacritical mark to a letter or pair of letters as a function that takes into account the size, shape, and darkness of the letter in deciding how to accent it.

For the compiler to be able to accent letters properly, a considerable amount of information must be available to it. The horizontal position of an accent mark is determined both by the angle of the major axis of the character, the height of the character, and the center point of the accent mark itself.

Accent characters of size, darkness, and style appropriate to the font being accented must be used; the accent characters must either be part of the font, or else the font must have, for each kind of accent symbol, a pointer to a character to implement the accent in that font. Information about the geometry of the accent character must be stored with it in order that it may be aligned properly over the letter to be accented.

Another approach to accent marks is to treat each accented letter as a ligature and to devise manuscript sequences that ligate to the accented character.[1] A scheme like this has the advantage of properly handling those characters that must be ligated—a dieresis over a lower-case *i*, for example, requires that the dot over the *i* be eliminated: ï . The disadvantages of this scheme are that it greatly increases the size of the alphabet, it can produce only accented characters that are part of the font, and it introduces a large number of obscure ligature combinations into the manuscript language.

---

[1]The issue of how to *print* accented characters is entirely separate from the equally important issue of how to *specify* them. Language issues, such as the specification of accented characters, are discussed in Chapter 4.

3.2 Lineation and Word Placement

Once letters have been formed into words, according to the rules for word assembly set forth in the previous section, the words must be formed into lines and paragraphs.

3.2.1 Word Spacing and Justification

Words are assembled into lines of more or less even length; customarily the lines are then *justified* by adding extra spaces between words until the right margin is aligned. This practice was originally a mechanical necessity, as the type box full of lead slugs could not be used safely unless the text lines were securely clamped, and they could be clamped only if they were all the same length [47, p. 121]. Several studies have shown that unjustified text is often more readable than justified text, and never less readable [8, 45]. Typographic instructor and author J. R. Biggs points out, however, that a study of calligraphic manuscripts shows that scribes liked their lines to be about the same length, and frequently resorted to compressing or expanding their letters towards the end of the line in order to make the line lengths come out even [7, p. 32].

Word spacing is normally measured and specified in *spacing units*. A spacing unit is traditionally 1/18 of the width of the widest character in a font [47, p. 58]. Like many typographic traditions, this nomenclature arose from mechanical limitations of a particular technology, in this case the Monotype machines introduced in 1894 [30].

The preferred word spacing for text fonts is 4 to 6 spacing units. The narrowest word spacing that is generally considered by professional typographers to be reasonable in text is 3 spacing units or 1/6th of the width of the widest letter.[2] The widest word spacing that is generally considered acceptable is 9 units, or 1/2 the width of the widest letter [44, p. 59, 47, p. 121]. If the line cannot be justified without expanding or contracting the spaces outside these limits, it is customary to hyphenate (see Section 3.2.3). The new line breaking algorithm devised by Knuth, in which whole paragraphs are considered at one time and all the line breaks found simultaneously, greatly reduces the number of cases in which hyphenation must be attempted [24, p. 52].

---

[2] Prolonged exposure to any format will lead one to find it more readable; perhaps typographers have simply trained themself to be able to read such text.

In non-text situations, the word spacing often differs. Verse is normally set with word spacing of 6 to 8 units, somewhat wider than the preferred spacing for text. When setting tabular material and computer programs, one customarily uses a space that is the same width as the digit zero in the font in use.

For a compiler to get word spacing correct, it must have some way of computing the size of a spacing unit and a set of rules for how wide or narrow a space to place between words before attempting some other solution. Whether the compiler deals in actual Monotype spacing units or in relative character widths is not important, but the information must be available to it. The compiler must of course know whether it is setting text, verse, tabular material, or computer programs, in order that it choose the correct space width for the circumstance.

3.2.2 Paragraphing

When words are typeset into paragraphs, it is customary to take care that the last line of the paragraph not be too short. Sometimes this constraint is expressed as "the last line of the paragraph will not be a single word", at other times it takes the form "the last line of the paragraph will not be shorter than k% of the other lines." In either case, the intent is to prevent paragraphs with vestigial lines at the end of them; an example of this can be seen in the first paragraph of Section 3.2.2 on page 28.

It is considered bad form to typeset a paragraph so that there are any regular patterns in the word spaces from one line to the next [44, p. 59]. Figure 9 shows an example of a paragraph set with geometric patterns or "rivers" in the word breaks.

Any mechanism for arranging words into paragraphs must be able either to look ahead or to backtrack if it is to be able to do a satisfactory job of avoiding these conditions, as it is not possible to know at the beginning of a paragraph how the text will fall at the end, but the only way to control the placement of the text at the end is to adjust the placement at the beginning.

Some paragraphing styles call for a change in type size or type face after the first letter, word, few words, or line of text. Figure 10 shows several examples of this sort of style. Note particularly the third example, in which the precise location of the font change is determined by the location of the first line break, which cannot be determined by consideration of the text alone—the first line must actually be set in type before the location of the font change can be determined. These conditions amount to *events* in the text, and the compiler must have a pattern matcher able to recognize these events and trigger the appropriate action if it is to superimpose these formats on text. The compiler implemented for this research contains no such event

The guests included Sen. and Mrs. Edward F. Kennedy, Sen. and Mrs. John Anderson, Dr. and Mrs. Michael I. DeBakey, Dr. and Mrs. Edward N. Emery, Judge Bean Roy, Mr. James Marshall Hendrix, Mr. and Mrs. William O. Douglas, Fr. and Mrs. John Fetterman, and the Rev. Jonathan B. Appleyard.

Figure 9: Paragraph with "rivers" of white space.

It frequently happens in the history of thought that when a powerful new method emerges the study of those problems which can be dealt with by the new method advances rapidly and attracts the limelight, while the rest tends to be ignored or even forgotten, its study despised.

IT FREQUENTLY HAPPENS in the history of thought that when a powerful new method emerges the study of those problems which can be dealt with by the new method advances rapidly and attracts the limelight, while the rest tends to be ignored or even forgotten, its study despised.

IT FREQUENTLY HAPPENS IN THE history of thought that when a powerful new method emerges the study of those problems which can be dealt with by the new method advances rapidly and attracts the limelight, while the rest tends to be ignored or even forgotten, its study despised.

Figure 10: Unusual paragraphing styles.

detector and cannot generate such event-dependent formats, although Page-2 and Troff can.

3.2.3 Hyphenation

When the justification of a line requires the word spaces to be expanded or contracted so much that they are unsightly or ineffective, it is customary to hyphenate the last word on the line or the first word of the next line. The "correct" hyphenation of words has been an annoying problem throughout the whole history of document production. An examination of early manuscript documents, done with pen and ink, shows that new lines were started wherever the scribe found it convenient, even if it was in the middle of a word, and that no notion of a "hyphen" existed. A compositor setting type by hand spent as much as one third of his time on hyphenation, even when using linecasting machines designed to expedite the process [44, p. 58].

Standards for correct hyphenation vary among languages. In English it is customary to hyphenate between syllables, where syllable division corresponds roughly to pronunciation [36, p. xxv]. Unfortunately, English spelling does not correspond very well to pronunciation, and so there are no particularly good rules for finding hyphenation points in a word by examination of letter combinations. Many homographic pairs are hyphenated differently because of different etymology, e.g., ten-der (an offer) and tend-er (a ship), and sometimes the same word is hyphenated differently depending on its part of speech, e.g., prog-ress (noun) and pro-gress (verb) [17]. English hyphenation cannot be done correctly without an understanding of the text deep enough to recognize parts of speech.

Most other languages have rules for hyphenation that differ in detail but not in spirit from the English rules. Some are much more regular. In Finnish, words are divided between vowels except those that are part of a diphthong [16, p. 435]. There is a set of seven rules for hyphenating French; there are no exceptions to those rules [16, p. 442]. In German, a set of twelve rules for word division suffices [16, p. 448]. However, if a German spelling is an elision of a longer form, then if the word is hyphenated at the elided syllable, the long form must be restored: *glitschst* is hyphenated *glit-schest*, and *Luftschiffahrt* is hyphenated *Luftschiff-fahrt*. When the German double consonant *ck* is divided, it must be spelled *kk*: *Hacke* is divided *Hak-ke* [16, p. 450]. In Hungarian, when a word is divided at a "long" consonant such as *ssz* or *ggy*, the consonant is repeated completely in its short form: *hosszu* is hyphenated *hosz-szu* and *hattyu* is hyphenated *haty-tyu* [16, p. 471].

As noted earlier in this section, a compiler cannot hyphenate English perfectly

without understanding the context in the sentence of the word being hyphenated, but it can do an acceptable job with a hyphenation dictionary or with a set of rules and a dictionary of exceptions to them. A very elaborate commercial typesetting program might have 200 rules and 15,000 words in the exception dictionary [6]. The clever hyphenation algorithm used by D. E. Knuth in TEX has 5 rules and 350 words in the exception dictionary [24, p. 180]. TEX avoids having to hyphenate very often by considering the entire paragraph at once, to make the lines break more evenly. It therefore can get by with a hyphenator that in general does not find all of the legal hyphenation points in a word. The Scribe compiler uses a pure dictionary-based hyphenator; there are no rules to fall back on if the word to be hyphenated is not found in the dictionary. This scheme has the advantage of being very simple and being independent of the text language, but it is not very efficient in terms of the memory space consumed by the dictionary or by the I/O time expended in looking up words if it is not kept in primary memory. A technique for maintaining and using document-specific hyphenation dictionaries avoids this inefficiency; it is described in the chapter on the workings of the compiler, in section 8.6.4.

3.3 Tabular and Display Material

Any text not filled and justified in the usual way is called "display" text; complicated displays are called "tabular material". Simple displayed text can be centered, or flushed left or right to some fixed horizontal position. Complex displays include matrices, columnar material aligned on decimal points or with justified text set in tables, and so on. A centered display might have each line individually centered, like a wedding invitation, or be "block centered", wherein the lines in the display are set flush left to a margin chosen such that the longest line is centered.

Overlong lines in display material often cannot be automatically folded to the next line. Some means therefore must be found for making them fit on a page. Use of a smaller type face, or going outside the margins, or rotating the whole display to go sideways on a page, or some combination of these effects, is often used to make long lines fit.

When a large amount of tabular information must be fit into a small space, "dot leaders", a row of dots or dashes, are used to draw the eye from one part of the table to another. Dot leaders are often seen in telephone directories and tables of contents. The dots in separate rows must be vertically aligned. Frequently a dot leader is used in combination with a flush-right operation, so that the dots fill all of the space up to the text that is right flushed. At other times, dot leaders are used in

conjunction with filled but unjustified text, so that the dot leader begins at the end of the last word that was able to fit on the line, and follows from there to the end of the line.

In tables, the material in each row of a column must be harmoniously aligned with the other material in that column. Table columns might be flush left, flush right, centered, justified as text, or aligned on some punctuation character such as a decimal point. Similarly, the various columns of a table must sometimes be synchronized to a common vertical position before a new row can begin.

In very geometric or regular tables, it is customary to add blank space or a rule after every n lines. In poetry or prose that will be cited lineally, line numbers are often placed beside every n^{th} line.

3.4 Page Layout

"Page Layout", also called "makeup" or "dummying", is the assignment of lines of text to pages while coping with figures, footnotes, and various traditions and conventions. To a first order, it consists of putting as many properly-spaced lines on a page as will fit, while taking into account the page numbers, footnotes, and figures. Beyond that, the primary goals are legibility, consistency of design, and appearance. There are many traditional constraints and rules designed to assist a typesetter in producing legible and attractive pages. Not all of them can be satisfied simultaneously.

The last line of a paragraph should not be alone at the top of a page, and some standards call for the first line of a paragraph never being alone by itself at the bottom of a page. These lines are called *widow lines*, and the painstaking work that human typographers perform to get rid of them is referred to as *widow elimination*. The last word on a page should not be hyphenated.

When headings are used in text, the amount of text on the page below the heading should be roughly proportional to the significance of the heading. Major or chapter headings usually begin a new page. Second-order heads usually should be placed high enough on a page as to have several lines of text after them. Every heading should have at least two lines of text following it on a page.

When displayed material is interspersed in text, the line of text introducing the display should be on the same page as the display. Page breaks are not normally permitted inside displayed material, except when it is so long that one has no choice.

The first line of a text footnote must appear on the same page as the reference to it, and it is best if the entire footnote appears on that page. A footnote to a table

should appear with the table, at its foot, before the caption. When a page contains both full-width text and multiple-column text, footnotes to the full-width part should be set full width, below the column footnotes that are set column-width in the bottom of the column containing the footnote reference [16].

When figures are used with text, the figure should appear on the same two-page spread as the first reference to the figure. When possible, the bottom margins on the left and right pages of a two-page spread should be the same, though they need not be consistent from one page spread to another.

Page layout has of all aspects of typography yielded the least to reduction to rules, and remains the hardest unsolved problem in automated document production. Page layout is also the aspect of a document's appearance that is most heavily affected by considerations of the document design. The current Scribe compiler does a barely adequate job of page layout, using relatively inflexible algorithms. A compiler for the Scribe document specification language that is able to do a high-quality job of page layout for arbitrarily complex document designs will likely require an order of magnitude more knowledge about the layout task than the current compiler uses.

Part II

Design and Implementation

The goals for the Scribe system, as itemized in Chapter 2, were the design of a language for the specification of documents, the design and implementation of a compiler to process that language into finished documents, and the production of user documentation. The document specification language explicitly forbids the user from providing low-level device-specific information. For the compiler to be able to compile the document specification language properly into a finished document it must have considerable typographic expertise. The compiler must be able to recognize problem situations in the text (possibly aided by the writer), and to apply the correct typographic rule to produce appropriate output.

This organization makes the compiler design be a problem more in knowledge engineering than in formatting. The actual formatting is relatively trivial once the compiler has determined the rule or rules to apply. This determination often involves conflict resolution among multiple rules that apply. The major component of the compiler design was therefore a codification of the formatting task in terms that would make the knowledge representation simple, and the design of a knowledge representation suitable for storage in a database system external to the program. This codification resulted in the parameterization of the document production task in terms of about one hundred parameters; the behavior of the compiler is controlled by changing the values of these parameters. This parameterization and its impact on the solution are discussed in Chapter 5.

In order to be able to evaluate the effectiveness of the solution, especially the parameterization, the compiler was documented as production software and released to the university community at Carnegie-Mellon, and later to numerous other laboratories. As information came in from this field experience, the parameterization evolved somewhat, primarily by the addition of some new variables, but the basic approach has proved sound. A discussion of this field experience and its effect on the compiler is in Part III of the thesis.

The various pieces of knowledge needed by the compiler were divided into two groups: those that were likely to remain more or less fixed over all formatting tasks within the intended domain, and those that were likely to vary widely over those formatting tasks. The fixed knowledge was "hardwired" into the code of the compiler, and the variable knowledge was codified, organized into appropriate external form, and stored in database files. The compiler must retrieve the externally-stored knowledge and process it into an appropriate internal form before it can actually be applied.

The crucial factor in the compiler's ability to locate, control, and modify its formatting knowledge is the representation used for it. The requirements placed on the knowledge representation were:

1. It must be legibly representable in text files, not just in complex data structures in memory, to facilitate database management. An external representation can be designed for any data structure, but we also demand that:

2. It must be easily read and easily modified, both automatically by the compiler and manually by users.

3. It must be efficiently usable by the compiler, which is to say that once the compiler has retrieved the necessary knowledge from its database, it must run at a speed roughly comparable to one in which the knowledge is fixed in the compiler code.

Requirement 2 essentially eliminates any procedural knowledge representation: procedural knowledge sources are by definition coded in some programming language, to which automated modifications (such as those needed by the definition-by-analogy mechanism) are difficult or impractical. Furthermore, a procedural knowledge representation requires the user to learn the procedural language that is used before he can make substantive modifications. While there certainly exist procedural representations of knowledge and editing systems that operate on them to automatically perform the changes needed to redirect the behavior of the

procedure, they are not well understood. I deemed it risky to use such incompletely-understood techniques in such a crucial part of the compiler, since the primary research goal was not the investigation of knowledge representation techniques but the application of them.

The knowledge representation chosen to meet these various requirements, as discussed further in Chapter 5, is an association list. An association list is similar to the *property lists* used in LISP and the *description lists* used in IPL [28, 32]. The LISP property list is a list of attributes and their values that is attached to an object to show what properties it has. In IPL, the description lists are normally used to implement *associations*, which are single-valued functions that return a value for an object [32, p. 58]. Both organizations are used in Scribe, though the property-list form is dominant.

The document specification language, described in Chapter 4, has as its dominant characteristic the description of formats in terms of *formatting environments*. Each formatting environment causes the text contained within it to be shaped or styled in a certain way, as controlled by the value of the environment parameters. The overall collection of environments available to the compiler during the processing of a document is determined by its *document type*. The database of document and device types is discussed in Chapter 6.

Chapter 4
The Document Specification Language

Further explanation of the compiler mechanisms and implementation requires an understanding of the document specification language. This chapter outlines that language. The document specification language abstracted here is described in full in the *Scribe User's Manual* [37]. In this chapter, enough of the specification language is explained to give its flavor and to provide background for the chapters on mechanism.

The specification language is a scheme for marking (labeling) regions of the text and locations in it.[3] There is also a simple facility for passing information to the compiler via declarations at the beginning of the manuscript.

The strategy behind the language design is to have the writer identify segments of the text in abstract terms, and to have the compiler automatically retrieve the concrete details from the document design database. The language design process consisted of identifying the proper set of abstractions and giving them names, then devising a simple syntax that would allow those abstractions to be represented in a file of text characters.

4.1 Rationale

Although it is specifically intended that the specification language be representable as a linear stream of character text, a sequence of pictures can be used to explain it best. Figure 11a shows a paragraph of text that has been graphically labeled to show its component parts. One might envision a simple graphical notation like this being used informally at a blackboard when two people are discussing a format. Notice that there are several labeled regions, some nested inside others.

---

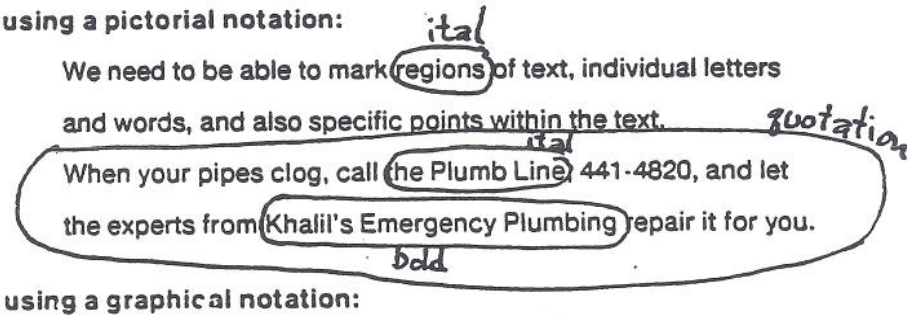
[3]The words *region* and *location* have precise technical meanings in this thesis; they are defined in section 4.2.

The desired document text:

We need to be able to mark *regions* of text, individual letters and words, and also specific points within the text.

> When your pipes clog, call *the Plumb Line*, 441-4820, and let the experts from **Khalil's Emergency Plumbing** repair it for you.

Markup using a pictorial notation:



Markup using a graphical notation:

We need to be able to mark *italic* regions *end italic* of text, individual letters and words, and also specific points within the text. *quotation*
When your pipes clog, call *italic* the Plumb Line *end italic*, 441-4820, and let the experts from *bold* Khalil's Emergency Plumbing *end bold* repair it for you. *end quotation*

Markup using an escape-character notation:

We need to be able to mark @i[regions] of text, individual letters and words, and also specific points within the text.@begin(Quotation)
When your pipes clog, call the Plumb Line, 441-4820, and let the experts from @b<Khalil's Emergency Plumbing> repair it for you. @End(Quotation)

Figure 11: Various schemes for marking text.

Figure 11b shows the same labeling, but this time the labels are differentiated from the text graphically: the labels are in script, and the text is in ordinary print. The printing industry uses proofreaders' marks in colored pencil to handle the text marking problem; both color and being handwritten serve to separate a proofreader's mark from the text being marked.

To represent this same labeling without resorting to graphics, special script, or color, one need only designate some character as the "color shift" character or *escape character*. We would like to choose a shift character that does not occur often in text and that is visually obvious to a person looking at a manuscript file. The Ascii character "@" satisfies these requirements; selecting "@" as the blue shift escape character yields Figure 11c, which is a syntactically correct Scribe manuscript.

4.2 Syntax

Three classes of notation are needed in the document specification language:

- Region labels: a notation for attaching a label, or attribute, to indicate the author's intention regarding a region of text. I will call these labels *environments*.

- Markers: a notation for marking specific points in the text, often with respect to the boundaries of some containing environment. Although it is a slight misnomer, I will call these *commands*.

- Declarations: a notation for passing values to the compiler to control certain details of its behavior. Most simple documents will need no declarations.

To describe all three of these notations, I shall borrow a word from printers and use the term *mark*, with collective plural *markup*.

A manuscript will consist of a mixture of text and markup, and the compiler must have some way of telling them apart. Although various schemes are possible, the fixed single shift-character scheme outlined in the previous section was selected because it places the least complicated restriction on the writer: *anything* following an "@" character is a mark. The shift character cannot be changed or redeclared; therefore no context dependencies are possible: a word or sentence from the manuscript can be moved or copied anywhere with confidence that it will still be syntactically correct in the new context.

All marks begin with an "@" character. If the character following the "@" is not alphanumeric, then the mark consists of exactly two characters, such as:

```
@#
@%
@=
```

If the character following the "@" is alphanumeric, then the mark consists of an identifier and a single delimited operand:

```
@Heading(The Document Specification Language)
@Label<L19>
@Style(Doublesided,Footnotes="*")
@Newpage()
```

Sometimes the delimited operand contains text that will be examined by the compiler (*e.g.* @Label and @Style, above), while other times it contains text that will be included in the finished document instead of being examined by the compiler (@Heading in the example above). Sometimes the operand is null (@Newpage). The mark is ended and text resumed by the closing delimiter that matches the opening delimiter that was used. Any of these paired Ascii characters can be used as delimiters: `[...]` `<...>` `(...)` `{...}` `"..."` `'...'` `'...'`. Any mark that takes a text argument can also be represented in "long form", with properly nested @Begin and @End:

```
@Begin(Heading)The Document Specification Language@End(Heading)
@Begin(Center)
Text to be Centered
@end(Center)
```

The syntax is not recursive; it is defined only at these two levels. @Begin(Begin)Heading@End(Begin) is not recognized.

Capitalization in alphanumeric marks is not important; any mixture of upper and lower case is equivalent to any other. End-of-line characters inside markup are equivalent to spaces, though in some environments end-of-line characters are significant.

4.3 Language Abstract

4.3.1 Environments

An *environment* is the mark attached to a piece of text identifying it to the compiler, and specifying certain goals that the author has for its appearance. If the text is a theorem, it would be marked as a *Theorem* environment; if the text is a footnote, it would be marked as a *Footnote* environment. Some environments

represent very simple concepts, like "italic" or "centered", while others represent relatively advanced concepts, like "bulleted list" or "footnote". Environments can be nested; for example, text can be marked as *italic* inside text that is marked as *footnote.*

Environments in the basic subset taught to the novice fall into two categories:

- Environments that define character shape, size, font, or appearance. These tend to have one-letter names: the *I* environment marks text as *italic*, the *C* environment marks text in SMALL CAPITALS.

- Environments that define paragraph shape (and sometimes paragraph font). These have multi-letter names: the *Itemize* environment marks paragraphs as elements of a bulleted list (like this one); the *Quotation* environment marks paragraphs as text quotations.

Figure 12 lists the "font-change" environments defined in the basic system, and Figure 13 lists the "paragraph shape" environments.

The "basic system" is not a separate or different part of the language; it is not implemented in any way differently than the more intricate parts. The concept of a basic system is rather just a documentation trick: the language features in the basic system are all simple, regular, stylistically similar, and guaranteed to be present in all document types.

4.3.2 Document Types

Whenever the compiler produces a document from a manuscript, it does so under control of the format set forth in a *document type definition* from the editorial data base. This document type definition completely determines the appearance of the document. The manuscript file is expected to contain a declaration of document type; if it does not, the compiler selects a default document type named *Text.*

All document types provide definitions for the basic environments; some provide additional definitions for environments that are peculiar to that document type. For example, the *Business Letter* document type provides environments for return address, greeting, and signature; the *Ph.D. Thesis* document type provides environments for chapter headings, a title page, and a bibliography.

`@i[phrase]`	*Italics*
`@b[phrase]`	Boldface
`@r[phrase]`	Roman (the normal typeface)
`@p[phrase]`	***Bold Italics***
`@c[phrase]`	SMALL CAPITALS
`@u[phrase]`	Underline non-blank characters
`@t[phrase]`	`Typewriter font`
`@+[phrase]`	print $^{\text{super}}$script
`@-[phrase]`	print $_{\text{sub}}$script
`@g[phrase]`	Greek (Ελλεν)

Figure 12: Font environments in the basic language.

`Center`	Unfilled environment. Each manuscript line centered.
`Description`	Filled environment. Outwards-indented paragraphs; single spacing with wider margins. This list of environments is in a Description environment.
`Display`	Unfilled environment. Widens both margins.
`Enumerate`	Filled environment. Numbers each paragraph. Widens both margins.
`Example`	Unfilled environment. Uses fixed-width typeface for examples of computer type-in or type-out. Widens both margins.
`FlushLeft`	Unfilled environment. Prints the manuscript lines, in the normal body font, flush against the left margin.
`FlushRight`	Unfilled environment. Prints the manuscript lines, in the normal body font, flush against the right margin.
`Format`	Unfilled environment. Normal body typeface. No changes to margins. Any horizontal alignment that is needed should be done with tabbing commands.
`Itemize`	Filled environment. Marks each paragraph with a tick-mark or bullet. Widens both margins.
`Quotation`	Filled environment. Single-spaced; widens both margins; indents each paragraph.
`Verbatim`	Unfilled environment. Fixed-width typeface. No changes to margins.
`Verse`	Semi-filled environment; fills lines, but starts a new line for each line break in the manuscript. Widens both margins.

Figure 13: Paragraph environments in the basic language.

4.3.3 Commands

While environments label whole regions of text, commands mark specific points in it. Some commands take arguments, others do not. Some sample commands:

`@|`	Permit a word break to occur here.
`@↑`	Set a tab stop at the current horizontal position.
`@Label(XYZ)`	Attach the cross-reference name "XYZ" to the current page and section number.
`@Ref(XYZ)`	Insert as text into the document at this point the section number that was attached to the cross-reference name "XYZ" elsewhere in the document.
`@PageRef(XYZ)`	Insert as text into the document at this point the page number that was attached to the cross-reference name "XYZ" elsewhere in the document.

Others include commands to do bibliography database retrieval, forcing of new pages, horizontal tabbing, and various other effects.

4.3.4 Declarations

Declarations in the specification language serve to control the compiler by passing it various parameters and values. Most declarations are restricted to the beginning of a manuscript, but some are permitted to occur anywhere.

Simple declarations include `@Device(name)`, which instructs the compiler to format the document for the named device, and `@Make(what)`, which instructs the compiler to produce a document of the requested type. More sophisticated declarations include `@Modify`, which alters the definition of an existing environment, and `@PageHeading`, which tells the compiler what text to put in the running page heads.

One declaration, `@Style`, serves as a catchall for passing miscellaneous scalar values to the compiler. There are several dozen "style keywords" whose values can be set by the `@Style` command. These include, for example, values to control the way dates are printed, to select a font family for the document, to select nonstandard paper sizes, and to select single-sided or double-sided formatting. The style parameters select small variations in document design.

Certain declarations, such as `@Define` and `@Form` (which define environments and macros, respectively) are intended primarily for use in document format definition

entries in Scribe's database. They can nevertheless be used in manuscript files, where their use permits expert users to develop new document types by gradual mutation of existing ones.

4.4 Character Sets and Font Variations

Western languages use alphabets, which consist of characters. The set of characters in each Western language has stayed essentially constant since the Renaissance, though not all Western languages use the same set of characters. When a character is typeset, the precise style and geometry of its appearance is determined by the *font* in which that character is typeset.

In addition to the alphabetic characters that are the basis of the written language, writers use many special characters. Some are punctuation marks, like "." or ";". Others are symbols borrowed from foreign alphabets, like Π or β. Others are purely fabricated, like "↑" or "$\leq$". When two printed letters of different appearance are visually compared, the difference sometimes arises because they are genuinely different letters and sometimes arises because they are the same letter printed in different fonts.

Pictorial representations of text, such as photographic copies or electronic facsimile transmission, do not need to concern themselves with identifying or encoding the letters—they merely store a picture of the letter and pass on to the reader the job of identifying the letter so pictured. When text is represented by character identity independent of the font in which the character is printed, there is a necessity to determine that identity and represent it with some sort of an unambiguous code.

Various codes for information interchange have been devised. Each defines a fixed set of characters to be represented, then assigns a numeric code to each. In the United States, for example, the BCD code defines 48 characters, the military Fieldata code 63 characters, the Ascii code 96, and the EBCDIC code 192. Whenever a character outside the defined set needs to be represented, one must go outside the interchange standard and use some private encoding. The specification of Ascii includes an explicit mechanism for extending the code, but does not assign character identities to any of the extended codes. As a result, no two users ever seem to produce the same set of extensions.

Some special characters are just ligatures of ordinary characters (ligatures are discussed in Section 3.1.2). For these cases, the compiler automatically substitutes the ligature graphic for the group of characters that were in the manuscript: *ffi* for "ffi", *ff* for "ff", and so on. Some special characters can be represented as

pseudo-ligatures: "—" for "--", for example. To represent special characters for which no common pseudo-ligature convention exists, the Scribe manuscript language uses a special-character convention that is not very satisfactory, and is one of the weakest parts of the design. It has been very difficult to maintain device portability of special characters as a result of this convention. A special character is represented by specifying an ordinary character in a special-character font: while @i[A] prints as "*A*" and @b[A] prints as "A", @f1[A] prints as "∀" and @f2[A] prints as "∮".[4]

It is worth noting briefly the several alternative specification schemes for special characters that were considered. TEX and EQN both use a "naming" scheme. To get an alpha character produced in TEX, one types \alpha (for lowercase "α") and \Alpha (for uppercase "A"). EQN recognizes the identifiers "alpha" and "ALPHA", although in the basic Troff system underneath EQN, an alpha is denoted instead by "\(*a".

These are implemented as fixed macros, encoded in whose definition is the information about how to print the special character on the printing device at hand. These naming schemes presuppose that the language designer knows all of the special characters that will be available on the printing device, and gives them all names in advance. Since the Scribe language is intended to be independent of printing devices, its naming convention would have needed to include all of the special characters expected ever to be available on any printing device. Fixed macro names for characters were therefore not adopted (although they are superior to the scheme actually used in the current Scribe language).

The TEX and EQN special-character schemes both require that the compiler (or the macro definitions) know the mapping of characters to slot numbers in fonts, for example, Troff must know that to generate a mu ("μ") character while using a certain font, it must switch to film 3 and generate a capital W; that font is arranged so that the character in the capital-W position is a lowercase mu.

A superior scheme for Scribe would have been to encode the font data such that there was no hard notion of a character slot, as exemplified by the "capital W slot" example above. Each font would have a name embodying its style and size, for example, "Helvetica 14-point lightface expanded italic" and would contain a set of definitions of characters. Some of these definitions would be *standard*, which is to say that they are valid graphics for the character set (Ascii, EBCDIC, etc.) being used, while others would be *non-standard*, meaning that they are not valid graphics for

---

[4]See Section 8.6.3 on page 97 for a discussion of the formatting issues for lines containing oversize characters like this one.

any characters in the base character set. The standard characters would be addressed by their slot in the font, while the non-standard characters would be addressed by name. The manuscript form of a document would be permitted to refer to any character by name; a symbol table associated with the base character set would identify those addressable in a particular slot. When a reference is encountered to a character not part of the base character set, it will first be looked up in the "current" font. If not found there, then fonts that are similar to the current font in shape and size must be searched until some definition for the character is finally found. This scheme requires standardization in naming, but not in allocation of non-standard characters to font slots.

4.5 Language Examples

Figure 14 (page 49) shows a simple manuscript prepared in the Scribe document specification language, and Figure 15 (page 50) shows the resulting document. Figure 16 (page 51) shows a reasonably elaborate one-page manuscript, and Figure 17 (page 52) shows the resulting document.

```
@Heading(What can be copyrighted)
Copyright protection exists for ''original works of authorship'' when they
become fixed in a tangible form of expression.  Copyrightable works include the
following categories:
@begin(enumerate)
literary works;

musical works, including any accompanying words;

dramatic works, including any accompanying music;

pantomimes and choreographic works;

pictorial graphic, and sculptural works;

motion pictures and other audiovisual works; and

sound recordings.

@End(enumerate)
This list is illustrative and is not meant to exhaust the categories of
copyrightable works.  These categories should be viewed quite broadly so that,
for example, computer programs and most ''compilations'' are registrable as
''literary works''; maps and architectural blueprints are registrable as
''pictorial, graphic, and sculptural works.''

@Heading(What cannot be copyrighted)
Several categories of material are generally not eligible for
statutory copyright protection.  These include among others:
@Itemize[
Works that have @i[not] been fixed in a tangible form of expression.  For
example: choreographic works which have not been notated or recorded, or
improvisational speeches or performances that have not been written or recorded.

Titles, names, short phrases, and slogans; familiar
symbols or designs; mere variations of typographic
ornamentation, lettering, or coloring; mere listings of
ingredients or contents.

Ideas, procedures, methods, systems, processes, concepts, principles,
discoveries, or devices, as distinguished from a description, explanation, or
illustration.

Works consisting @i[entirely] of information that is common property and
containing no original authorship.  For example: standard calendars, height and
weight charts, tape measures and rules, schedules of sporting events, and lists
or tables taken from public documents or other common sources.@Foot<
From @i[The Nuts and Bolts of Copyright (Circular R1)], U. S. Copyright Office.>
]
```

Figure 14: Simple Scribe manuscript.

What can be copyrighted

Copyright protection exists for "original works of authorship" when they become fixed in a tangible form of expression. Copyrightable works include the following categories:

1. literary works;

2. musical works, including any accompanying words;

3. dramatic works, including any accompanying music;

4. pantomimes and choreographic works;

5. pictorial graphic, and sculptural works;

6. motion pictures and other audiovisual works; and

7. sound recordings.

This list is illustrative and is not meant to exhaust the categories of copyrightable works. These categories should be viewed quite broadly so that, for example, computer programs and most "compilations" are registrable as "literary works"; maps and architectural blueprints are registrable as "pictorial, graphic, and sculptural works."

What cannot be copyrighted

Several categories of material are generally not eligible for statutory copyright protection. These include among others:

- Works that have *not* been fixed in a tangible form of expression. For example: choreographic works which have not been notated or recorded, or improvisational speeches or performances that have not been written or recorded.

- Titles, names, short phrases, and slogans; familiar symbols or designs; mere variations of typographic ornamentation, lettering, or coloring; mere listings of ingredients or contents.

- Ideas, procedures, methods, systems, processes, concepts, principles, discoveries, or devices, as distinguished from a description, explanation, or illustration.

- Works consisting *entirely* of information that is common property and containing no original authorship. For example: standard calendars, height and weight charts, tape measures and rules, schedules of sporting events, and lists or tables taken from public documents or other common sources.[5]

Figure 15: Document produced from manuscript in Figure 14.

---

[5]From *The Nuts and Bolts of Copyright (Circular R1)*, U. S. Copyright Office.

```
@Make(Wedding Program)
@Style(Font "Times Roman 10")
@begin(Introductory)
The Marriage of Loretta Rose Guarino and Brian Keith Reid
Saturday, May 12, 1979
The Church of St. Michael and All Angels,    Tucson, Arizona
@Separator()
@end(Introductory)
@Heading(Voluntary)
@Begin(Verse)
@i[Siciliano], from @i[Sonata #2 for Flute and Keyboard], J. S. Bach
@i[Prelude in Classic Style], Gordon Young
@i[Andante], from @i[Organ Concerto in F. Major], G. F. Handel
@end(Verse)
@Heading(Processional)
@begin(Verse)
@i[Adagio in A Minor], from the @i[Toccata, Adagio, and Fugue in C Major], J. S. B
@i[Rigaudon], Andre Campra
@end(Verse)
The text for the Marriage Ceremony may
be found in the @i[Book of Common Prayer] beginning on page 423.
@Heading(The Invocation@PageNum[p. 423])
@Heading(The Ministry of the Word@PageNum{p. 425})
@SubHeading(The Old Testament@>Tobit 8:5-8@\)
@SubHeading(The New Testament@>I Corinthians 13:1-13@\)
@SubHeading(Hymn 363)
@SubHeading(The Gospel@>John 15:9-12@\)
@SubHeading(Homily@>Fr. John Fowler)
@Heading(The Marriage@PageNum[p. 427])
@SubHeading(The Exchange of Vows)
@SubHeading(The Prayers)
@Heading(The Blessing of the Marriage@PageNum[p. 430])
@SubHeading(The Blessing)
@SubHeading(The Peace)
@Heading(The Holy Communion@PageNum[p. 361])
@SubHeading(The Great Thanksgiving)
@SubHeading(The Breaking of the Bread)
@SubHeading(The Prayer of Thanksgiving@>p. 432)
@SubHeading(Benediction and Dismissal)
@Heading(Processional)
@begin(verse)
@i[Toccata], from @i[Symphony #5 for Organ], C. M. Widor
@end(verse)
```

Figure 16: An elaborate scribe manuscript.

The Marriage of Loretta Rose Guarino and Brian Keith Reid
Saturday, May 12, 1979
The Church of St. Michael and All Angels, Tucson, Arizona

*** *** ***

Voluntary

Siciliano, from *Sonata #2 for Flute and Keyboard*, J. S. Bach
Prelude in Classic Style, Gordon Young
Andante, from *Organ Concerto in F. Major*, G. F. Handel

Processional

Adagio in A Minor, from the *Toccata, Adagio, and Fugue in C Major*, J. S. Bach
Rigaudon, Andre Campra

The text for the Marriage Ceremony may be found in the *Book of Common Prayer* beginning on page 423.

The Invocation p. 423

The Ministry of the Word p. 425

The Old Testament	Tobit 8:5-8	
The New Testament	I Corinthians 13:1-13	
Hymn 363		
The Gospel	John 15:9-12	
Homily		Fr. John Fowler

The Marriage p. 427
The Exchange of Vows
The Prayers

The Blessing of the Marriage p. 430
The Blessing
The Peace

The Holy Communion p. 361
The Great Thanksgiving
The Breaking of the Bread
The Prayer of Thanksgiving p. 432
Benediction and Dismissal

Processional

Toccata, from *Symphony #5 for Organ*, C. M. Widor

Figure 17: Document produced from manuscript shown in Figure 16.

Chapter 5
The Environment Mechanism

To facilitate knowledge representation and manipulation, the problem of text formatting was reduced to a set of almost-orthogonal parameters. The behavior of the formatting compiler is controlled by setting and manipulating the value of these parameters. The formatter interrogates the most recent value of appropriate parameters whenever it must make a decision.

The parameterization of the task for document formatting was crucial to the success of the compiler. It is therefore worthwhile to document the parameterization in detail, explaining the purpose and behavior of the parameters and the mechanisms that operate on them.

5.1 Environment Entry and Exit

Each environment (*environments* are defined in Section 4.3.1) specifies a value for some parameters, but not necessarily all of them. As environments are nested, a binding stack protocol is used; the current value of a parameter is the one found topmost on the binding stack, and therefore belonging to the innermost environment that specified a value for it. Because the parameters are static (no new parameters can be created without reconfiguring the compiler) the compiler is able to implement the binding stack much more efficiently than the classic LISP implementation.

All changes to the behavior of the output assembler—new margins, new fonts, new paragraphs, etc.—are effected by changing a state parameter. These parameter changes are made whenever an environment is entered, and they are unmade when the environment is exited. The initial values of the state parameters are determined by the initialization from the document type definition retrieved from the document design database.

An environment is a prescription for change to one or more state parameters. An environment could be represented as a program that operates on one set of state

parameter values to produce another, but for a variety of reasons it is implemented as a simple list of state parameter names and the change that is supposed to be made to them. An environment normally specifies a change to only a few of the parameters, leaving the rest to be inherited from outer environments.

When the compiler needs to know the value of a parameter during the formatting process, it uses the topmost value found in the binding stack for that parameter. When an environment is entered, its parameters and their values are pushed onto the binding stack; when the environment is exited, the values that it pushed onto the binding stack are removed. On both entry and exit, a change analyzer is called to examine the changes that have just been made in the state parameters to see if any support processing must be performed. Typical support processing functions signaled by the change analyzer include storage allocation, font structure initialization (the first time a font is used), and footnote placement.

5.2 Types

Every parameter has a type, and every value in an environment has a type. When the environment is entered and a new parameter value is computed, the value that the environment specifies for a parameter is coerced into the type of the parameter. Some of these coercions are context-sensitive, so that the same environment value can produce differing parameter values depending on context. For example, there is a state parameter named *WidestBlank* that specifies the largest size to which a blank can be stretched before the compiler will try to hyphenate the next word. The type of the *WidestBlank* parameter is *horizontal distance*—it specifies a genuine maximum size. However, a document format designer can specify a value in type *font width relative distance*—and it will be converted to a different absolute distance depending on the font currently in use. This permits most of the bookkeeping computations to be handled automatically. These types and their implementations are discussed in more detail in Section 8.4.1.

- Type *character* is a single Ascii character.
- Type *string* is a string of characters.
- Type *integer* is an ordinary machine integer, subject to the usual limitations of finite word size.
- Type *rational number* is a rational number represented as the quotient of two machine integers. They are used in distance calculations in which rounding errors must be avoided at any cost.

- Type *Boolean* is *true* or *false*.

- Type *vertical distance* is an absolute distance measured as an integral number of basic vertical spacing units of the destination printing device. Since it is always an integer, it is not subject to rounding error.

- Type *horizontal distance* is an absolute distance measured as an integral number of basic horizontal spacing units of the destination printing device.

- Type *font-width-relative distance* is a distance that is proportional to the width of the digit "0" in the current font. When an environment's value for a parameter is in type *font-width-relative-distance* and the parameter's type is an absolute distance, the environment's value is multiplied by the appropriate width at environment entry time, thereby yielding different absolute distances in different contexts.

- Type *font-height-relative distance* is a distance that is proportional to the height of the current font. Its coercion to absolute distance is context-sensitive; see above.

- Type *symbol* is a pointer to an entry in the compiler's symbol table. State parameters can take on symbolic values when they need to link the state to some external entity, such as a numbering counter.

There are also various enumerated types that are specific to the parameter whose value ranges over that type. These types are described along with the parameter for which they are the domain.

5.3 Dynamic State Parameters

Dynamic parameters are those that may change during a run of the compiler. They are classified into two groups, *inheriting* parameters and *non-inheriting* parameters. The inheriting parameters obey the binding stack protocol discussed in Section 5.1. The non-inheriting parameters do not: if an environment entry does not specify a value for a non-inheriting parameter, then a default value is used rather than an inherited value.

A sample dynamic parameter is the one that selects the font, which is an inheriting parameter: an environment whose definition makes no mention of font is produced in the same font as the containing environment. Another is the flag that

specifies whether or not a new paragraph is to be started on entry to the environment. It is a non-inheriting parameter. The complete set of dynamic parameters is listed in Appendix A, beginning on page 133.

5.4 Static State Parameters

Static state parameters are fixed during compiler initialization, and they do not change during a compilation. Their values are read in from various database files, or occasionally specified directly in the manuscript.

The various static state parameters that affect the formatting process are listed in Appendix A, beginning on page 139. These parameters are static not because of a conceptual or implementation need that they be static, but because there is no need for them to be dynamic, and static parameters are accessed much more efficiently. Examples of static state parameters are the width of the paper loaded into the printing machine and the flag that specifies whether or not the document type is to be set up for double-sided reproduction.

5.5 Pattern Templates

In designing or modifying a document a document format, one frequently needs to specify a style for numbering or marking or labeling. Are chapters numbered 1, 2, 3, 4 or I, II, III, IV? Or are they numbered one, two, three, four or One, Two, Three, Four?

In keeping with the general Scribe philosophy of nonprocedural specification, the Scribe compiler has a general mechanism for providing a schema for the generation of systematically-created names or numbers. This *pattern template* mechanism is similar to the Fortran FORMAT mechanism: the user provides a series of codes that show how the numbers are to be converted and where they are to be placed once they have been converted.

The Scribe pattern template mechanism supports about 15 different kinds of numeric conversion, including cardinal and ordinal Arabic (1, 2; 1st, 2nd), cardinal and ordinal English (one, two; first, second), upper- and lower-case Roman (I, II; i, ii), replicated tallies (*, **, ***, etc.) and selection from enumerated sets (dagger, double-dagger, etc.). Besides these format conversions, the pattern template can specify literal text (like the H format in Fortran conversion) and perform simple conditional tests on the numbers being converted.

There is sometimes a need to control the printing style of automatically-generated

text other than numbers. The Scribe compiler, for example, will insert the current date wherever it finds the construct @value(date). The default format is "13 December 1980", but many document styles require different date formats. A user can request the compiler to generate dates in a different format by providing it with a date template. A date template is a representation of the date 8 March 1952, using nearly any format. By parsing that template, the compiler can recognize fields as standing for the month, the day, the day of the week, the year, and so forth. When a date is inserted by the compiler into the text, it converts the components of that date into a string according to the fields found from parsing the template. Various examples:

> The template "8 March 1952" prints today's date as "13 December 1980".
> The template "08 Mar 52" prints today's date as "13 Dec 80".
> The template "8/3/52" prints today's date as "13/12/80".
> The template "03/08/52 (Saturday)" prints today's date as "12/13/80 (Saturday)".
> The template "The First of March, One Thousand Nine Hundred and Fifty-two"prints today's date as "The First of December, One Thousand Nine Hundred and Eighty".
> The template "Samedi, le 8 Mars, 1952" prints today's date as "Samedi, le 13 Decembre, 1980".
> The template "el 8 de Marzo de 1952" prints today's date as "el 13 de Diciembre de 1980".

The standard date was chosen so that a purely syntactic analysis could be used. The month number (3) must not duplicate the day number (8), the day number of that date within the week (6), or any of the digits of the year (1, 9, 5, or 2). Both the month number and the day number must be single digits, so that leading zeros can be detected (3/8/52 vs. 03/08/52). The month cannot be January, so that day-within-month and day-within-year can be disambiguated. The date must fall within the first 99 days of the year, so that leading zeros can be detected in a day-within-year value. Whatever month is used must have different spellings in all of the languages that we hope to recognize (English, Spanish, French, German, and Swedish); this eliminates April, which is spelled the same in English and German. Finally, I wanted the date to be relatively recent, so that it could be represented as a positive number in an offset Julian-day scheme whose values would fit into 16-bit machine words. February 2-8 and March 2-8 all provisionally satisfy these restrictions, though not all of them will work every year because of conflicts with the year digits. March 8 is my wife's birthday, so that settled it.

5.6 Definition by Analogy

The representation of environments as attribute lists permits a very simple *definition by analogy* mechanism. As introduced in Section 2.2.3, a definition by analogy is the definition of a new environment to be essentially like another, but with a specified set of differences.

Each environment is a set of pairs of attributes and their values. If environment x specifies values $v_1,\ldots,v_n$ for attributes $a_1,\ldots,a_n$, and environment y is defined to be "like x, but having value w_k for attribute a_k", then y will have values $v_1,\ldots,w_k,\ldots,v_n$ for attributes $a_1,\ldots,a_k,\ldots,a_n$. Environment x might or might not have had a value specified for attribute a_k.

This definition by analogy can also be used to make incremental changes to the definition of an existing environment, by the simple tactic of substituting x for y in the above transformation. x will then be redefined to be different in some set of attributes from its previous definition.

5.7 An Illustrated Example

For this example, please refer to Figures 18 and 19. The initial state, at sequence number 1 in Figure 18, is $v_1,v_2,\ldots,v_n$ for attributes $a_1,a_2,\ldots,a_n$. When the `@Begin(Quotation)` environment entry command is seen at sequence number 2, the definition of *Quotation* is retrieved and found to be

QUOTATION: $(a_3 = w_3, a_7 = w_7, a_9 = w_9)$

After the *Quotation* environment is entered, the formatter state is now

$v_1,v_2,w_3,v_4,v_5,v_6,w_7,v_8,w_9,v_{10},\ldots,v_n$

The next sentence, sequence 3 in Figure 18, is formatted according to that state vector. When the `@i` environment entry command is seen at sequence number 4, the definition of *I* is retrieved and found to be

I: $(a_8 = z_8)$

After the *I* environment is entered, the formatter state is now

$v_1,v_2,w_3,v_4,v_5,v_6,w_7,z_8,w_9,v_{10},\ldots,v_n$

```
[1]The quick brown fox said,
[2]@Begin(Quotation)
[3]Brian, you've just [4]@i[got][5] to think of a better
example for the environment/state vector figure.
[6]@End(quotation)
      --[7]@i[Anonymous]
```

Figure 18: Manuscript used for the example in Figure 19.

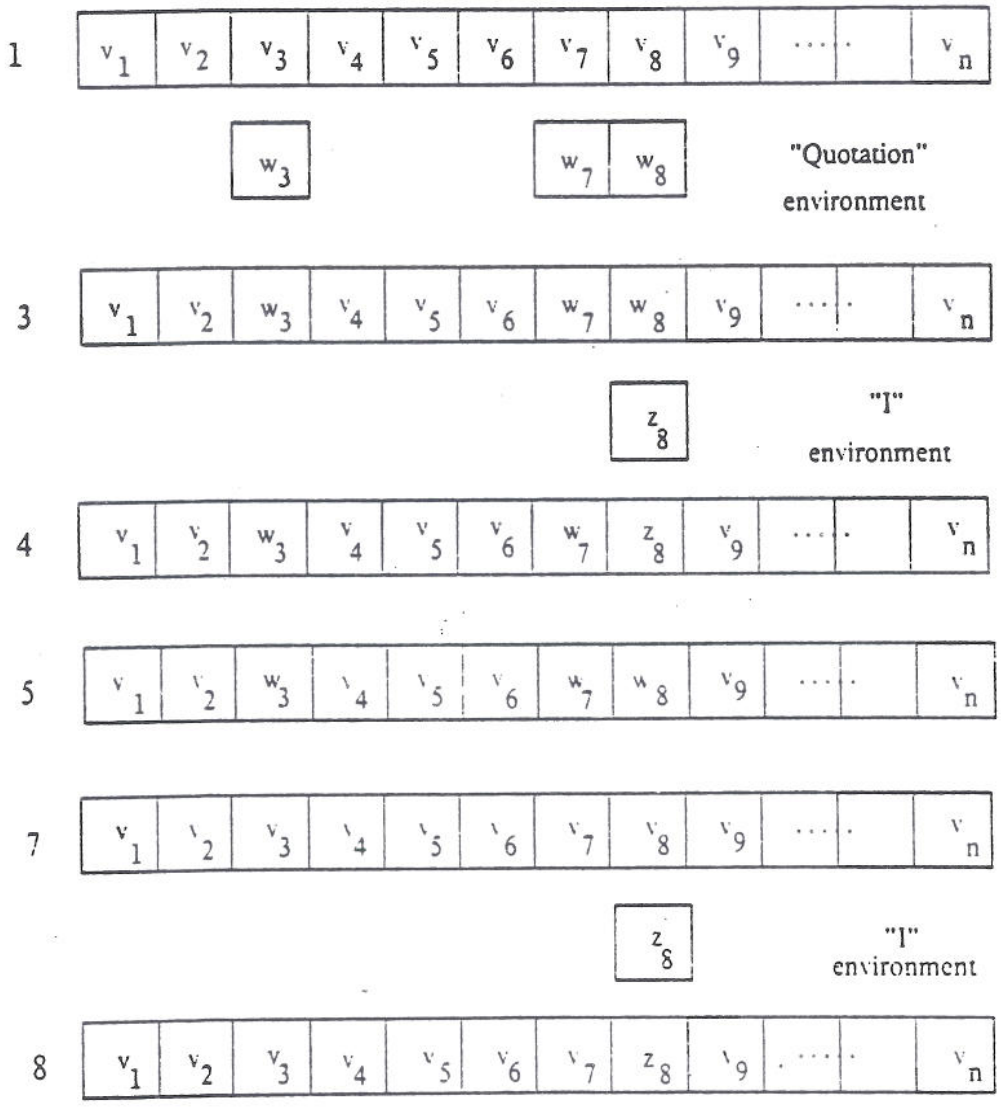


Figure 19: State vector changes during environment processing.

When the *I* environment is exited, sequence 5, the state is restored to be the same as it was at sequence 3, by popping back the old state vector from a stack on which it was saved when the new one was generated. When the *Quotation* environment is exited, the state is restored to its initial state. At sequence 7 another instance of the *I* environment is encountered, but this one is not nested inside *Quotation*. The same definition for *I* is retrieved, and after environment entry the state vector is now

$$v_1, v_2, \ldots, v_7, z_8, v_9, \ldots, v_n$$

Chapter 6
The Database

The Scribe database contains all of the information needed by the compiler to produce documents. There are two fundamentally different kinds of information in the database: *device* information and *format* information. There is a certain amount of interplay between the two, since the formats are device-dependent. Each time the compiler is run, it produces a document in a particular format for a particular device; it retrieves the necessary information from the database during compiler initialization.

6.1 Device Data

The first step taken by the compiler during initialization is to determine the printing device and retrieve the *device definition* from the database. The device definition contains specifications of physical properties of the device, specifications for retrieving font and format data pertinent to that device, and default implementations of various environments for that device. The data representation is a command language syntactically identical to the document specification language, in order that the same parser can be used for both.

Figure 20 shows the device definition for an optical photocomposer equipped with photographic fonts and a lens turret to change letter size. The first five statements assign values to static parameters. The "generic device" is the device retrieval key for future database retrievals; it permits device definitions for similar devices to share most database entries. The "driver" is the identifier for the particular output driver in the compiler; the GSI driver contains code that knows about optical photocomposers, and has the device commands for those devices hardwired into compiler tables.

The remainder of this database file is devoted to the default environment definitions for this device. A format definition can override any or all of these definitions, but most format definitions use the standard, default environment definitions.

```
@Marker(Device,GSI)
@Declare(GenericDevice="GSI",DeviceTitle="GSI CAT-8 Photocomposer",
        FinalName="#.GSI")
@Declare(Driver GSI,Hunits inch,Hraster 432,Vunits inch,Vraster 144)
@Declare(PaperWidth 7.75inches,PaperLength 11inches,ScriptPush=no)
@Declare(Underline available,Backspace available,
       Overstrike available,Fonts,Lenses,FontCount 8)

@Define(I,FaceCode I,TabExport)
@Define(B,FaceCode B,TabExport)
@Define(P,FaceCode P,TabExport)
@Define(C,Capitalized,Size -2,TabExport)
@Define(V,Capitalized,FaceCode R,Size -1,TabExport)
@Define(R,Underline off,Capitalized off,TabExport,FaceCode R)
@Define(U,Underline NonBlank,TabExport)
@Define(UN,Underline Alphanumerics,TabExport)
@Define(UX,Underline All,TabExport)
@Define(T=R,TabExport)
@Define(Plus,Script +0.41ines,Size -2,TabExport)
@Define(Minus,Script -0.41ines,Size -2,TabExport)
@Define(G,FaceCode G,TabExport)
@Define(Z,FaceCode Z,TabExport)
@Define(Y,FaceCode Y,TabExport)
@Define(F0,TabExport)@Define(F1,TabExport)@Define(F2,TabExport)
@Define(F3,TabExport)@Define(F4,TabExport)@Define(F5,TabExport)
@Define(F6,TabExport)@Define(F7,TabExport)@Define(F8,TabExport)
@Define(F9,TabExport)
@Define(W,Spaces NoBreak,TabExport)

@Counter(Page,Numbered <@1>,Referenced <@1>,Init 1)
@Define(Hdg,Fixed 0.5inch,Nofill,LeftMargin 0,RightMargin 0,Spread 0,
       Capitalized off, Spacing 1,Size 10,Font HeadingFont,FaceCode R,
       Columns 1, ColumnMargin 0,
       UnNumbered,Underline off,Indent 0,
       Initialize "@tabclear()",TabExport False)
@Define(Ftg=Hdg,Fixed -0.5inches)
@Define(Text,Fill,Justification on,Spaces compact,Break,WidestBlank 5pts,
       Blanklines Break)
@Define(Multiple,Indent 0,SpecialCase OpenBefore)
@Define(Transparent)
@Define(Group,Group,Break)
@Define(Float,Float,Break)
@Define(Bspace,Break,Above 0,Below 0,Group,Nofill,LeftMargin 0,RightMargin 0)
@Define(Bpage,FloatPage,Break,Continue)
@Define(Pspace,Break,Above 0,Below 0,Group,Nofill,LeftMargin 0,RightMargin 0)
```

Figure 20: Device definition for a photocomposer (part one).

```
@Define(Verbatim,Break,Continue,Nofill,Spaces Kept,FaceCode F,
        Above 1,Below 1,BlankLines kept,Spacing 1)
@Define(Format,Break,Continue,Nofill,Spaces Kept,Above 0.8 lines,
        Below 0.8lines,BlankLines kept,Spacing 1,Justification off)
@Define(Insert,Break,Continue,Above 0.7lines,Below 0.7lines,LeftMargin +4,
        RightMargin +4,spacing 1,BlankLines kept)
@Define(Center,Break,Continue,Above .8,Below .8,Spacing 1,
        LeftMargin 0,RightMargin 0,
        Centered,BlankLines kept,Initialize "@TabClear()",TabExport False)
@Define(Flushright=Center,Flushright)
@Define(Flushleft=Format,LeftMargin 0)
@Define(Heading,Use Center,Continue off,Above 2,Below 1.3,
        Font HeadingFont,FaceCode B,
        Need 1inch,Size +3,TabExport False,Spacing 1.2)
@Define(SubHeading,Use Insert,LeftMargin 0,Indent 0,Continue off,
        Font HeadingFont,FaceCode B,
        Above 0.8,Below 0.5,Need 4,Size +2,Spacing 1.2)
@Define(MajorHeading,Centered,Spacing 1.2,Continue off,Need 1inch,
        Font HeadingFont,FaceCode B,
        Above 2,Below 1,Size +5,Break,TabExport False)
@Define(Display,Use Insert,Nofill,Use R,Group,Blanklines Kept,
        Spaces Kept,TabExport False)
@Define(Example,Use Insert,Nofill,Spaces Kept,Group,Blanklines Kept)
@Define(OutputExample=Verbatim,LeftMargin 2)
@Equate(InputExample=OutputExample)
@Define(ProgramExample=Example)

@Define(Itemize,Break,Continue,Fill,LeftMargin +5,Indent -5,RightMargin 5,
        numbered <@@y[B] @,@@y[b] >,NumberLocation lfr,BlankLines break,
        Spacing 1,Above 0.5lines,below 0.5lines,Spread 0.5lines,
        Spaces compact)
@Define(Enumerate,Use Itemize,LeftMargin +6,Indent -6,
        Numbered <@1. @,@a. @,@i. >,
        Referenced <@1@,@a@,@i>)
@Define(Description,Break,Above 1line,Fill,LeftMargin +16,Spaces Compact,
        Indent -16,Spacing 1,Spread 0.3lines)
@Define(Quotation,Use Insert,Fill,Use R,BlankLines break,Size -1,
        Spaces Compact,TabExport False,Font BodyFont)
@Define(Verse,Use Insert,Fill,Spaces Kept,Justification off,Crbreak,Use R,
        LeadingSpaces Kept,
        indent -3,Spread 0,LeftMargin +8,TabExport False)
@TextForm(Bar="@begin(format)@tabclear()@&@+[]@\@end(format)")
@Define(Fnenv,Use Text,Above 1,Foot,Use R,LeftMargin 0,RightMargin 0,
        Size -2,CrSpace,UnNumbered,Indent 2,Spread 1,spacing 1,Break off,
        TabExport False)
@Define(FootSepEnv,Break,SaveBox <FootSep>,Nofill,LeftMargin 0,Above 1,
        Below 1)
```

Figure 20: Device definition for a photocomposer, continued.

The fonts and face codes referred to are defined in a font definition entry from the database. Font definition entries are described in Section 6.2. The syntax of the @Define statement is

@Define(*name*, *list of attributes*)

or the form for definition by analogy,

@Define(*name*=*existing name*, *list of differences*)

These environment definitions are processed into an internal representation of association lists (Section 8.4.1.4), forming a list of pairs of attributes and their typed values.

6.2 Font Data

Two kinds of font data are needed by the compiler. The first, font organization or family data, is a mapping between Scribe font names and face codes to the device-specific fonts. Figure 21 shows the font family definition for a font family named *Times Roman 10A*. This font family is commonly used for textbooks. It defines five named fonts, each of which contains several face codes. An actual device font is selected by a (Font, facecode) pair.

The second kind of font data found in the database is a *device font description.* A *device font description* is a map from Ascii characters onto code sequences sent to the final printing device with width information attached. It may also include specification of ligatures and special characters available in that device font. Figure 22 shows a portion of the device font description for the Times Italic Bold font available on this class of photocomposer. For ordinary Ascii characters, it specifies the width (in machine-specific distance units) and the particular device codes needed to get the photocomposer to print that character in the selected font. For ligature characters, it shows the ligature key (ff in this example), then the width and device codes.

6.3 Document Format Definition Data

After the compiler has processed the device data, it retrieves and processes the appropriate *document format definition* from the database. Selected by a @Make command in the manuscript (see Section 4.3.4), the document format definition selects fonts, defines or redefines environments as needed, and initializes dynamic state parameters. Document format definitions are sufficiently varied that one example will not suffice; we will discuss the *Letterhead* and *Thesis* document type definitions.

```
@Comment{       Times Roman
In this configuration, the following font segments must be mounted:

Quadrant 1:      Part #629-001A          #1 Pi font
Quadrant 2:      Part #603-008A          Helvetica Bold/Light
Quadrant 3:      Part #608-001A          Times Roman Italic/Regular
Quadrant 4:      Part #608-002A          Times Roman Bold/Bold-Italic
}
@DefineFont(BodyFont,
        I=<ascii "C02I">,
        B=<ascii "C02B">,
        R=<ascii "C02R">,
        G=<ascii "C02G">,
        F=<ascii "C02F">,
        Z=<ascii "C02Z">,
        Y=<ascii "C02Y">,
        P=<ascii "C02P">,
        T=<ascii "C02TL">)

@DefineFont(HeadingFont,
        R=<ascii "C02TL">,
        B=<ascii "C02TB">)
```

Figure 21: A font family definition (Times Roman 10).

```
(Id "l",Wid   9,Film 5,Code 5,Case L)
(Id "m",Wid  28,Film 5,Code 4,Case L)
(Id "n",Wid  19,Film 5,Code 3,Case L)
(Id "o",Wid  18,Film 5,Code 27,Case L)
(Id "p",Wid  17,Film 5,Code 17,Case L)
(Id "q",Wid  18,Film 5,Code 34,Case L)
(Id "r",Wid  13,Film 5,Code 29,Case L)
(Id "s",Wid  14,Film 5,Code 8,Case L)
(Id "t",Wid  10,Film 5,Code 2,Case L)
(Id "u",Wid  19,Film 5,Code 14,Case L)
(Id "v",Wid  16,Film 5,Code 31,Case L)
(Id "w",Wid  24,Film 5,Code 33,Case L)
(Id "x",Wid  18,Film 5,Code 11,Case L)
(Id "y",Wid  16,Film 5,Code 41,Case L)
(Id "z",Wid  14,Film 5,Code 7,Case L)
(Id "{",Wid  14,Film 1,Code 26,Case U)
(Id "|",Wid   2,Film 5,Code 41,Case U)
(Id "}",Wid  14,Film 1,Code 27,Case U)
(Id "~",Wid  36,Film 1,Code 58,Case L)
(Id "--",Wid  36,Film 5,Code 18,Case L)
(Id "fi",Wid  21,Film 5,Code 20,Case U)
(Id "fl",Wid  21,Film 5,Code 21,Case U)
(Id "ff",Wid  21,Film 5,Code 22,Case U)
```

Figure 22: Sample device font (Times Italic Bold).

Figure 23 shows the document format definition for a letterhead format. It consists of some environment definitions, some @Style commands to set static format parameters, a @Begin command to initialize dynamic state, and then some manuscript text that will place the return address and the current date. The manuscript file for the letter will contain a @Begin(Address)/@End(Address) and @Begin(Body)/@End(Body) delimiting the address and body.

Figure 24 on pages 68 and 69 shows the document format definition for the CMU Thesis format on a Xerox Dover printer. It consists of some static format parameter declarations, a number of counter declarations, initialization for generated portions, loading of several libraries (title page format library, math numbering library, etc.), but no canned text. The @Define(BodyStyle) and @Define(NoteStyle) at the beginning set up "subroutine" environments that are never explicitly referenced by the writer, but are referred to in various environments later in the file. The @Font command declares the default font for this document type to be Helvetica 10. The @Style command declares a few static state parameters. The @Enable command defines two generated portions, one for the document outline and the other for the table of contents. (The generated portions for the list of figures and list of tables are declared in the library file loaded by the @LibraryFile(Figures) command on the second page of the figure.) The @Send commands initialize the table of contents portion, setting it up with the correct page number and numbering style, and then giving it the heading "Table of Contents". The mechanism by which the table of contents is generated is described in more detail in Section 7.1.

The @Define commands that follow define environments HD0 through HD4, and TC0 through TC4. They are used for level-0 through level-4 body heads and table of contents heads, respectively. The @Counter declarations that actually define @Chapter, @Section, and the like use these environments explicitly in their numbering templates. There are two templates associated with each counter, one called its "Numbered" template and used for printing the actual number, the other called its "Referenced" template and used for printing cross references to the generated number (the cross-reference mechanism is defined in Section 7.2). The three @LibraryFile commands load the standard definitions for figures, equation numbering, and title pages.

The @Modify commands that follow change the numbering on equations and theorems from that found in the standard library file into a style wherein equations and theorems are numbered within the current chapter. The @Equate command defines a few abbreviations, and finally the @Begin(text) marks the entry to the outermost environment in which the basal text for the document will be formatted.

```
@string(Phone="(412) 578-2565",Department="Computer Science Department")
@String(Psychology="Department of Psychology",
        Math="Department of Mathematics",
        EE="Department of Electrical Engineering")
@String(CSD="@q(@@)",PSI="@ @ @ @ @ @ @J(Y)@ ",
        None="@q[@ @ @ @ @ @ @ @ @ @ ]")
@String(Logo=CSD)
@DefineFont(LetterHeadFont,Q=<ascii "CMUlogo18">,R=<ascii "Helvetica10B">,
            H=<ascii "Helvetica14B">,J=<ascii "Hippo18MRR">)
@Font(Helvetica 10)
@Define(Q,FaceCode Q)
@Define(H,FaceCode H)
@Define(J,FaceCode J)
@Style(TopMargin 0.3in,WidowAction Force)
@Define(Address,Nofill,LeftMargin 0,Break,Use R,Spacing 1,Spaces Kept,
        Sink 2.2in, above 0, below 0)
@Define(Body,Fill,Justification,Use R,LeftMargin 0,EofOK,
        Spacing 1,Spread 1,Spaces Compact,BlankLines Break,
        TopMargin 1in,
        Sink 3.2in, Above 1 line,Below 0.5in,Break)
@Define(Ends,Nofill,LeftMargin 3.3in,Spread 0,Break,Use R,
        Above 1.6in,
        RightMargin 0,BlankLines Kept,Spacing 1)
@Equate(ReturnAddress = Comment)
@Define(Signature, use Ends, above 0)
@Define(Notations,Nofill,LeftMargin 0,Spread 0,Break,BlankLines Kept,
        RightMargin 0,Spaces Kept,Sink 9.3in)
@Define(LogoFormat=Format,Font LetterHeadFont,FaceCode R,Break,Above 0,
        Below 0,NoFill,Spacing 0.2in,LeftMargin -0.23in,RightMargin -0.23in,
        Initialize "@TabClear()")
@Define(PS=Body,Sink 0,Above 0,Below 0)
@Define(Greeting=Flushleft)
@Equate(PostScript=PS,PostScripts=PS,Closings=Notations,Initials=Notations)
@LibraryFile(Math)
@Begin(Text,Justification,Font BodyFont,FaceCode R,LeftMargin 1.0in,
        Indent 0,LineWidth 6.3in,BottomMargin 1in,TopMargin 1.3in,
        Spacing 1)
@TextForm(NewLetter={@Set(Page=0)@NewPage()
@begin(LogoFormat,Fixed 0.625in)
@value(Logo)@|@!@h[@value(Department)]@>@h[Carnegie-Mellon University]@\
@end(LogoFormat)
@begin(LogoFormat,Fixed 1.2in)
@/@value(phone)@>Pittsburgh, Pennsylvania  15213@\
@/@>@value(Date)@\
@end(LogoFormat)
})
@NewLetter()
@Begin(Ends,Eofok)
@PageHeading(left "@value(Date)",right "Page @value(Page)")
```

Figure 23: Document format definition for a business letterhead.

```
@Comment{
This file defines the format for Ph.D. theses in the Computer Science
Department at Carnegie-Mellon University.
}

@Define(BodyStyle,Font BodyFont,Spacing 1.5,Spread 0.8)
@Define(NoteStyle,Font SmallBodyFont,FaceCode R,Spacing 1)
@font(Helvetica 10)
@Style(DoubleSided,BindingMargin=0.5inch,LeftMargin=1.25 in,
       WidowAction Force,References=STDalphabetic)

@Enable(Outline,Contents)
@Send(Contents "@NewPage(0)@Set(Page=LastPreContentsPage)",
      Contents "@Set(Page=+1)",
      Contents "@Style(PageNumber <@1>)")
@Send(Contents "@PrefaceSection(Table of Contents)")
@String(LastPreContentsPage=0)

@Define(HDX,LeftMargin 0,Indent 0,Fill,Spaces compact,Above 1,Below 0,
          break,Need 5,Justification Off)
@Define(Hd0,Use HdX,Font TitleFont5,FaceCode R,Above 2,Below 0.5inch,
        Centered,AfterEntry "@TabClear()",Sink 4in,PageBreak UntilOdd,
        Spacing 1.8)
@Define(Hd1,Use HdX,Font TitleFont5,FaceCode R,Sink 2in,
            PageBreak UntilOdd,Below 0.5inch)
@Define(HD1A=HD1,Centered,AfterEntry "@TabClear()")
@Define(Hd2,Use HdX,Font TitleFont3,FaceCode R,Above 0.4inch,Below 0.3in,
        Need  1.5 in)
@Define(Hd3,Use HdX,Font TitleFont1,FaceCode R,Above 0.4inch,Below 0.3in)
@Define(Hd4,Use HdX,Font TitleFont1,FaceCode R,Above 0.3inch,Below 0.25in)
@Define(TcX,LeftMargin 5,Indent -5,RightMargin 5,Fill,Spaces compact,
        Above 0,Spacing 1,Below 0,Break,Spread 0)
@Define(Tc0=TcX,Font TitleFont3,FaceCode R,Above .3in,Need 1 inch)
@Define(Tc1=TcX,Font TitleFont1,FaceCode R,Above 0.1inch,Need 0.6 inches,
        Below 0.1inch)
@Define(Tc2=TcX,LeftMargin 8,Font TitleFont0,FaceCode R)
@Define(Tc3=TcX,LeftMargin 12,Font TitleFont0,FaceCode R)
@Define(Tc4=TcX,LeftMargin 16,Font TitleFont0,FaceCode R)
@Counter(MajorPart,
        Numbered [@I],Referenced [@I],TitleForm
{@begin(Hd0)Part @parm(Numbered)@*@Skip(6 pts)@*@parm(Title)@end(HD0)},
        ContentsForm
{@Begin(Tc0)PART @parm(Referenced):@*@rfstr(@parm(page))@parm(Title)@end(Tc0)},
          IncrementedBy Use,Announced)
@Counter(Chapter,TitleForm
{@begin(Hd1A)Chapter @parm(Numbered)@*@Skip(6 pts)@*@parm(Title)@end(Hd1A)},
        ContentsForm
{@Begin(Tc1)@rfstr(@parm(page))@parm(Referenced)@.@ @parm(Title)@end(Tc1)},
          Numbered [@1],IncrementedBy Use,Referenced [@1],Announced)
```

Figure 24: Document format definition for CMU thesis.

```
@Counter(Appendix,TitleEnv HD1,ContentsEnv tc1,Numbered [@A.],
         ContentsForm "@Tc1(Appendix @parm(referenced)@.@
                       @rfstr(@parm(page))@parm(Title))",
         TitleForm "@Hd1(@=Appendix @parm(referenced)@*
                    @=@Parm(Title))",
          IncrementedBy,Referenced [@A],Announced,Alias Chapter)
@Counter(UnNumbered,TitleEnv HD1,ContentsEnv tc1,Announced,Alias Chapter)
@Counter(Section,Within Chapter,TitleEnv HD2,ContentsEnv tc2,
          Numbered [@#@:.@1],Referenced [@#@:.@1],IncrementedBy Use,Announced)
@Counter(AppendixSection,Within Appendix,TitleEnv HD2,ContentsEnv tc2,
          Numbered [@#@:.@1],Referenced [@#@:.@1],IncrementedBy Use,Announced)
@Counter(SubSection,Within Section,TitleEnv HD3,ContentsEnv tc3,
          Numbered [@#@:.@1],IncrementedBy Use,Referenced [@#@:.@1])
@Counter(Paragraph,Within SubSection,TitleEnv HD4,ContentsEnv tc4,
          Numbered [@#@:.@1],Referenced [@#@:.@1],IncrementedBy Use)

@Counter(PrefaceSection,TitleEnv HD1A,Alias Chapter)

@LibraryFile(Figures)
@LibraryFile(Math)
@LibraryFile(Titlepage)

@Modify(EquationCounter,Within Chapter,Numbered <(@#@:.@1)>,
        Referenced "(@#@:.@1)")
@Modify(TheoremCounter,Within Chapter)

@Equate(Sec=Section,Subsec=SubSection,Chap=Chapter,Para=Paragraph,
        SubSubSec=Paragraph,AppendixSec=AppendixSection)
@Begin(Text,Indent 1Quad,TopMargin 0.9inch,BottomMargin 1.2inch,
        LineWidth 6 in,Spread 0.075inch,
        Use BodyStyle,Justification,FaceCode R,Spaces Compact)
@Set(Page=0)
@PageHeading(Center "@value(page)")
```

Figure 24: Document format definition for CMU thesis, continued.

6.4 Libraries

Certain database files amount to "subroutine packages"; they define environments or facilities that are used in several document formats. One library defines title page environments, another defines figures and tables, and so forth. These libraries reduce the redundant storage of duplicate information in the database, making the database maintenance task simpler, but increasing the amount of I/O overhead involved in initializing the compiler.

Chapter 7
A Writer's Workbench

The Scribe system *in toto* provides a rich environment for the development of documents. Borrowing the name from Evan Ivie's system called "Programmer's Workbench" [21], I call the full set of support facilities the "Writer's Workbench".

The goal of the writer's workbench is to provide an integrated set of tools that automate as much as possible the clerical work involved in the preparation of a document. Some of these tools are implemented within the Scribe compiler, others are separable programs that operate on the manuscript file. In this chapter, only those facilities that are implemented as part of the compiler are described in any detail.

Many of the facilities in the Scribe Writer's Workbench were styled after those in various other document compilers, notably Troff and Pub. One difference is that these facilities are built directly into the Scribe compiler, while they were implemented as extensions to the other systems, usually by persons other than the original implementor. The Scribe Writer's Workbench facilities are more smoothly integrated with the overall system, but less flexible and redefinable.

7.1 Derived Text

Many of the Scribe Writer's Workbench tools have to do with collecting together in one part of the document various text that also appears elsewhere. Tables of contents, indexes, and glossaries are examples of derived text.

To facilitate the collection of various kinds of derived text, the Scribe compiler provides a mechanism whereby text can be saved during the processing of the manuscript file and then processed as manuscript text in its own right at specific "collection points" or when the end of the primary manuscript file is reached. Each such *generated portion*, as it is called, is built sequentially in a format determined by the document type, and then closed and processed automatically.

Tables of contents, tables of figures, lists of maps, and other "directory"

information are summaries of all objects of a certain type that are in a document. The table of contents is a directory of all of the numbered chapter and section headings that appear in a document. The table of figures is a directory of all of the numbered figures in a document. These directories are in the same order as the objects appear in the document.

The database language (see Chapter 6) allows a document format designer to specify that all objects of a certain type will be recorded in a specified portion. The standard document types normally produce a table of contents, a list of figures, and a list of tables. It is possible to attach to any numbered object an attribute that will cause instances of that object to be recorded in one or more tables.

7.2 Bookkeeping and Numbering

Another theme common to several Scribe Writer's Workbench tools is is the use of symbolic names to stand for cross reference numbers that will later be assigned by the compiler. Used directly, this facility allows the writer to mark objects for numbering without worrying about what the numbers will be in the finished document, but to be able to make cross references to the numbers so assigned by referring to a symbolic cross-reference label. This facility is styled after the counter and referencing mechanism used by L. Tesler in PUB [43].

7.2.1 Cross Referencing

The fundamental building blocks of the numbering and cross referencing facility are counters, labels, and symbolic references. A counter is a register that gets incremented by various events, and which contains a printing template that controls the conversion of the number in the register to a character string suitable for use in text. A label command binds a cross-reference identifier to the current value of a counter, recording the information in a symbol table and in the auxiliary file (Section 8.3). A symbolic reference command refers to a cross-reference identifier, and causes its recorded counter value to be included as text in the document in place of the command.

There are two distinct kinds of cross referencing, and several variations on one of them. The original design of the Scribe manuscript language did not take these distinctions into account properly, and as a result the manuscript language is unnecessarily baroque with respect to the subtle differences among them. A cross reference is a request to the compiler to fill in the actual number for something for which you know only the symbolic name. Unfortunately, there can be many

different "actual numbers" associated with a given symbolic name. One can refer to the identity of an object—"Figure 4"—or to the location of the object—"the figure on page 23" or "the figure in Section 2.4". Unfortunately, one can refer to section markers as either objects—"Section 2.4 on page 23"—or as locations—"the figure in Section 2.4". There is no way to precisely determine the object that a cross reference marker is marking simply by considering its location, as there are no syntactic ties to link a marker to an object. The language therefore defines two different commands, @Tag and @Label, to indicate the marking of sections as objects or as locations. This is a defect in the manuscript language—there is no way to express non-spatial links between two objects—but it actually manifests itself as an irritating property of the cross reference mechanism.

7.2.2 Indexing

Although the production of an index is tedious and begs automation, a satisfactory index cannot in general be produced entirely by the compiler. In the introduction to his work on indexing, G. V. Carey states "The true aim of an indexer is to be methodical rather than mechanical" [9]. An index is much more than just a list of the words that appear in the manuscript. An index lists ideas, not words, and a mechanically-produced index based only on the words will have superfluous entries and be missing entries.

Although indexing is conceptually trivial—find out what is covered in the manuscript, arrange it in alphabetical order, and add page or section numbers—the creation of a usable index is a higher art. Robert Collison, author of one of the definitive works on manual indexing, asserts that most authors are temperamentally incapable of making their own index, for they are too close to the material to see it the way the reader will [11].

Figure 25 lists Collison's twenty basic rules for indexers. Very few of these rules are specified rigorously enough that a computer program could follow them to produce the index mechanically. A program would have special trouble with items 8, 9, 10, 11, and 17, since they require a deep understanding of the text.

The unfeasibility of a perfect solution to automated indexing should not discourage an approximate or partial solution. There is general agreement among the authors of the classic treatises on indexing that the easiest books to index are those that deal with facts, and that the indexing of these is largely a mechanical operation [9, 11, 46]. Books of facts, such as reference manuals, comprise a good fraction of the expected problem domain for a computer document preparation system, and it is certainly feasible to include an indexing mechanism that will be of use only for them.

1. Index everything useful in the book—text, illustrations, appendices, foreword, notes, bibliographies, etc.
2. Include all index entries in one alphabetical sequence.
3. Choose popular headings, with references from their scientific equivalents, except where a specialist audience is addressed.
4. Be consistent in choosing one form of spelling—seismography or seismology; ae or e, etc. Use a standard dictionary as your authority.
5. Choose the most specific headings which describe the items indexed: Steam-Boilers, not Boilers; Finance—Haiti, not Finance or Haiti alone, etc. Use phrases as headings if generally accepted: Training within industry; Social life and customs; but not Disposal of surplus stores; Rights of the human person, etc.
6. Be consistent in the use of singular or plural terms.
7. Combine the word and the action which describes it, where it is useful and possible: Banks and Banking; but not Fish and Fishing, etc.
8. Invert headings, where necessary, to bring significant word to the fore: Agriculture, Cooperative; Sociology, Christian, etc.
9. Check for synonyms and make suitable references from forms not used: Clothes, with references from Dress, and from Costume, etc.
10. Check for antonyms and combine where suitable: Employment and Unemployment, etc.
11. Where words of the same spelling represent different meanings, include identifying phrase in brackets: Race (sport); Race (ethnology), etc.
12. Where possible, give full names of persons quoted: Darwin, Erasmus; not Darwin, etc.
13. Omit the name of the country in which the book is published in favor of direct entry under the subject: Trade, Board of; not Great Britain—Board of Trade, etc.
14. Use capitals for all proper names, and where the usage of foreign languages demands them: Aristotle; Menelaus; but silage; quantum theory. Ruhe; Zweifel; but paix, guerre, etc.
15. Make references from main subjects to subdivisions of these subjects, and to related subjects: Costume, see also Gloves; Shoes; Hats, etc. But avoid a 'vicious circle' of references leading the reader back to the first heading.
16. Subdivide alphabetically by aspects wherever possible, to avoid long lists of page numbers.
17. In the case of historical and biographical works, substitute chronological for alphabetical subdivision, where this will definitely assist the reader.
18. Spell out symbols and abbreviations, except where the meaning of the abbreviations is generally known. United Nations, not U.N.; but UNESCO, not United Nations Educational Scientific and Cultural Organization, etc.
19. Avoid the use of bold type wherever possible: use instead italics, capitals, parentheses, and any other legitimate typographic devices for distinguishing items.
20. If references are made to paragraph numbers and not to page numbers, include a note to this effect at the foot of every page of the index.

Figure 25: Twenty basic rules for indexers, from Collison [11].

The compiler should therefore be able to provide as much support as possible for the indexer, filling the role normally played by boxes of index cards. Perhaps the best compromise is for the human user to select the set of topics to be indexed, by consulting a list of the words found in the document, and then to place index marks in the manuscript noting the topic that is to be indexed there. The compiler will fill in the correct page number, and generate and sort the index, coalesce identical entries, and so forth.

The current Scribe compiler provides two different indexing facilities. The first is an "@Index" command that makes an index entry, complete with page number, from the text argument to the command. At the end of the manuscript, all entries made via @Index are sorted into alphabetical order and formatted into a one-level index, whose format is controlled in part by the document type. This facility is adequate for short or simple material. The second facility is a low-level primitive command that can be used to build higher-level indexing schemes. This alternative command allows a programmer to write macros that will independently control the text, the sort key, and the page or section reference number. This @Indexentry command is called from within a macro, which is the only interface that a writer would ever see. Various macro packages defining multi-level and cross-references index formats exist, and a document format designer can include one of those packages in his format definition file.

Even more intricate indexes can be constructed by the simple escape of redefining the @Index command so that instead of placing its argument into the index, it writes its argument into a generated portion file. That file can then be sorted, edited, or processed into an index by various means external to the Scribe compiler. When this method is used the compiler serves only as the data-gathering piece of the indexing facility.

7.3 Document Management

Several of the facilities in the Scribe compiler have the common purpose of helping the writer (or writers) manage large documents. This help consists of various tools for helping him better see the structure of large documents, a means of breaking large documents up into manageable pieces, and a means of synchronizing the work of multiple authors on the same document.

7.3.1 Division into Parts

During the development of a large document, it is rare for the entire document to be under active development by the same author at the same time. In the interest of saving time and paper, an author usually prefers to edit, process, and print only the section of the document that he is actively working on. The Scribe compiler permits a manuscript to be divided into numerous small files, which are structured into a tree to build the actual document.

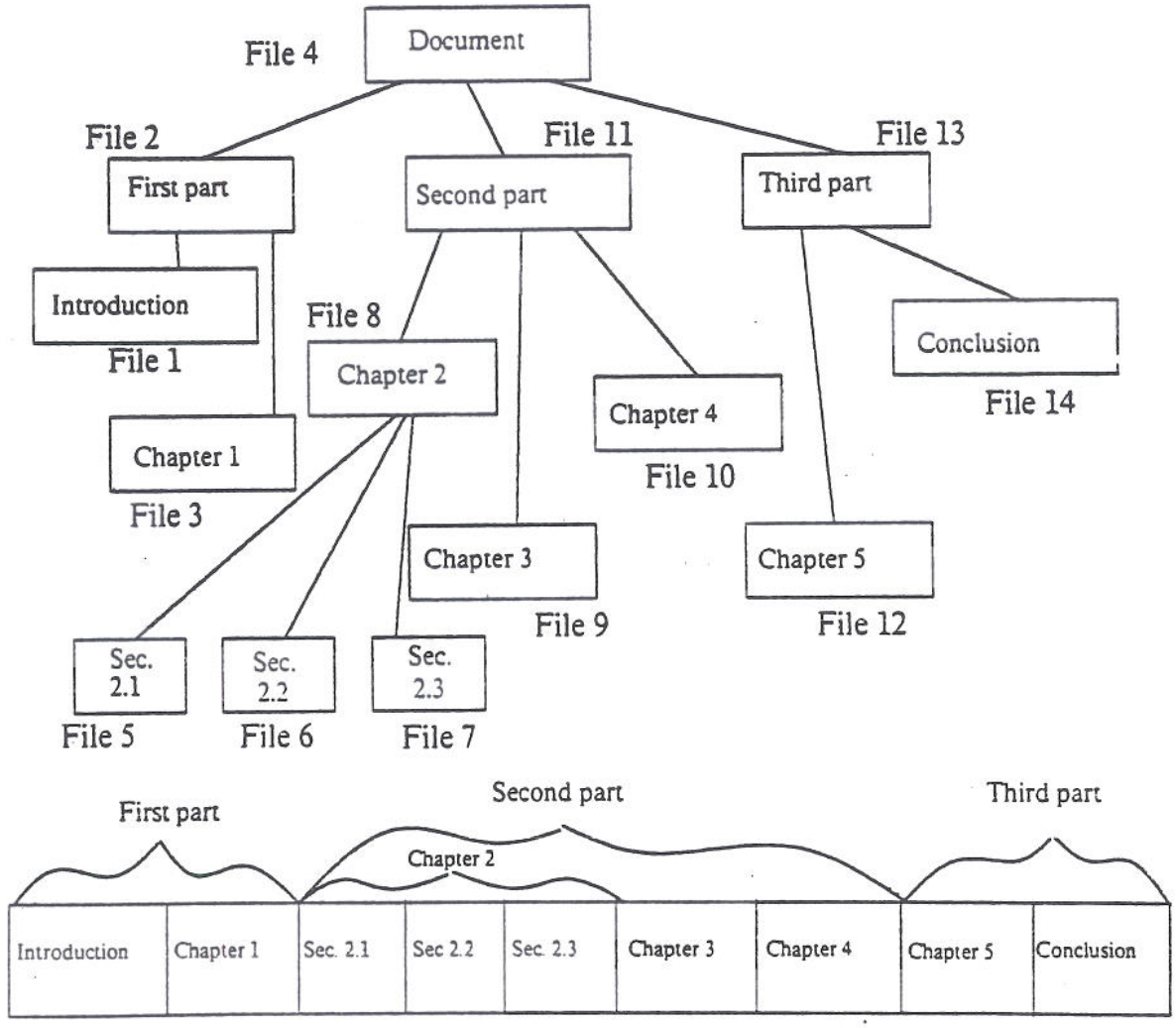


Figure 26: Decomposition of a document into a file tree.

Figure 26 shows a document partitioned into numerous subfiles.. The partitioning is hierarchical, and corresponds to the logical structure of the document. The sequence in which the files will be combined to produce the finished document is 1, 3, 5, 6, 7, 9, 10, 12, 14. If there is text in non-leaf nodes 2, 8, 11, or 13, it will be included mixed among the leaf text. This sequence amounts to a depth-first traversal of the file tree, and is equivalent to the sequence of text that would be used if each pointer to a sub-file were replaced by the entire text of that subfile,

recursively until no pointers remain. All text would then be in file 1, in the sequence shown above.

This mechanism is very ordinary, and has been used in document processors and other compilers for many years. The Scribe multifile partitioning mechanism is essentially identical to that used by most production compilers. Because it is so ordinary, this facility was only mentioned in passing in the first edition of the manual. I was quite surprised to notice that very few users made proper use of it to partition their big documents, so later editions of the manual described it as one of the major features of the Scribe compiler. While by itself this partitioning mechanism is convenient but not interesting, its presence in the compiler permits the implementation of a very interesting "partial compilation" mechanism, which is described in the next section.

7.3.2 Separate Compilation

When the compiler is invoked to process the root (file #4) of the file tree shown in Figure 26, it produces the entire document, as discussed above. However, when it is invoked to process some non-root piece of the manuscript file tree, it produces only that portion of the document that corresponds to the branch that was compiled. Page numbers, chapter numbers, footnote numbers, cross references, and all of the other compiler-generated text will be correct, *even if the cross references refer to labels in parts of the manuscript not included in this compilation.* For example, if the compiler is invoked to process file #7, then only the text of Section 2.3 will be produced, and the first page produced will be numbered according to the position of Section 2.3 in the document the last time the whole tree was compiled. If the number of pages in Section 2.3 changes, the global record of page numbers will be adjusted accordingly, so that if file #11 is now compiled, its first page number will be one greater than the last page in Section 2.3. If the document has an index, or other derived portion that is not in the same sequence as the document, then the final copy must be produced by a full-tree traversal. The mechanism by which this is made to work is discussed in section 8.3.

A variation on this "partial compilation" facility allows a manuscript to be simultaneously a complete document in its own right and a part of a larger document. Whether the compiler treats a file that looks like a root as a complete document or as a part of a larger one is determined by the presence or absence of a "@Part" declaration that contains a back pointer to a containing root: if the root pointer is present, the manuscript file is compiled as part of a larger document; if the root pointer is absent, then it is compiled as a full document in its own right.

Referring once again to Figure 26, if file #8 (Chapter 2) does not contain a @Part command linking it to file #4, then when the Scribe compiler is invoked to process file #8, it will produce a complete document beginning with page 1, in a format determined by the declarations in file #8. When the compiler is invoked to process file #4, and in the course of its processing encounters the declarations in file #8, they will be completely ignored because they are not in the root.

7.3.3 Document Analysis Aids

One way to help a writer manage a large document is to provide him with reports of its current status and structure. To this end, the Scribe compiler can produce a directory and cross-reference summary of any document that it compiles. The directory is similar to the table of contents, in that it is in the same sequence as the document. It shows the structure—all of the headings and their relationship to one another—without showing any of the text. It also shows all of the labels and tags defined for cross referencing. The cross-reference summary is a chart in alphabetical order by cross-reference label name, showing for each name the manuscript file location where it is defined, the value assigned to it, and the number of references to it.

Figures 27 and 28 show part of a sample cross-reference directory and listing produced by the Scribe compiler. Figure 27 is part of the directory, and Figure 28 is part of the cross-reference directory.

7.3.4 Draft Editions

Many documents are produced in draft dozens of times before a finished version is ready. When a compiler-based document production system is used, the compiler can just as easily produce a draft version of the document as a final version, and the format of the draft document can be completely different from the format of the finished document.

"Draft mode" is a state variable whose value can be interrogated in the database language so that a document type will yield a different format when draft mode is enabled. Document type designers typically make their formats have a slightly different appearance in draft mode, and usually add diagnostic information to the running heads or index. Together they deliver a useful draft capability that simplifies the management of the document during its development. Rather than having a draft mode and conditional code in the document format definitions, one could also define a separate document type for draft mode. Experience with both

Sec # and Title		Page #	MSS file location
2.1.3 Domain		4	PROBLE.THS, 00100/4
2.2 Language Goals		4	PROBLE.THS, 00100/5
2.3 Compiler Goals		5	PROBLE.THS, 00100/6
2.4 Documentation Goals		6	PROBLE.THS, 00100/7
3 Typography and Formatting		7	ISSUES.THS, 00200/1
	IssuesChapter	7	ISSUES.THS, 00300/1
3.1 Letter Placement and Spacing in Text		8	ISSUES.THS, 00100/2
3.1.1 Letter spacing and kerning		8	ISSUES.THS, 00600/2
	MatrixCompression 3-1	9	ISSUES.THS, 05500/2
3.1.2 Ligatures		9	ISSUES.THS, 00100/3
	RiversOfWhite 3-2	9	ISSUES.THS, 01600/3
	NEB 3-3	9	ISSUES.THS, 01800/3
	LigaturePicture 3-4	9	ISSUES.THS, 02100/3
	AccentCases 3-5	9	ISSUES.THS, 02300/3
3.1.3 Diacritical Marks		9	ISSUES.THS, 04500/3
3.2 Lineation and Word Placement		13	ISSUES.THS, 00100/4
	KernFigure 3-6	13	ISSUES.THS, 00500/4

Figure 27: Sample document directory.

Alphabetic Listing of Cross-Reference Tags and Labels

Tag or Label Name	Times Ref.	Page	Label Value	Source file Location
ACCENTCASES	1	12	3-5	ISSUES.THS, 02300/3
AUXFILE	0	37	7.3	SYSTEM.THS, 00200/4
BASICENVS	2	24	5-3	LANGUA.THS, 11800/5
BASICFONTENVS	2	24	5-2	LANGUA.THS, 06800/5
CHARACTERSETISSUES	1	25	5.5	LANGUA.THS, 00200/9
COMPILERSTRUCTURE	0	38	8	COMPIL.THS, 00300/1
DATAFLOW	0	41	8.3	COMPIL.THS, 00200/4
DATAFLOWFIG	1	43	8-6	COMPIL.THS, 01500/4
ENTERLEAVEFIGURE	1	42	8-5	COMPIL.THS, 06000/3
ENVIRONMENTREP	2	46	8.3.2.3	COMPIL.THS, 05700/8
ENVIRONMENTSEC	1	38	8.2	COMPIL.THS, 00200/3
FILETREE	1	33	6-1	WORKBE.THS, 03700/3
FLOWFIGURE	0	36	7-1	SYSTEM.THS, 01800/2
FORMATTER	2	50	8.5	COMPIL.THS, 00200/10
GLOBORG	1	39	8-1	COMPIL.THS, 00400/2
GOALCHAPTER	1	2	2	PROBLE.THS, 00300/1
GRAMMARFIGURE	1	49	8-7	COMPIL.THS, 02300/9
ISSUESCHAPTER	1	7	3	ISSUES.THS, 00300/1
KERNFIGURE	2	14	3-6	ISSUES.THS, 00500/4

Figure 28: Sample cross-reference summary.

styles has indicated that database maintenance is a serious problem when the number of formats in the database gets too large, and as soon as there are two document formats that are supposed to be nearly identical, the database maintainer must be extremely careful to maintain them in parallel. It is bad engineering practice to design a system that requires its users to exercise extreme care during a routine operation.

Document types having a draft mode typically adopt wider margins and wider line spacing, and disable double-sided format effects such as alternating page headings or margins and odd-page chapter openings. The compiler in draft mode provides extra diagnostic information about cross referencing and indexing, and depending on the document type, places it directly into the finished document. Cross reference label definitions in the manuscript appear explicitly in the document, and the generated index includes not only page numbers, but back pointers to the particular spot in the manuscript file that contains the index text. All index entries in the text appear as special footnotes in addition to being included in the index.

7.4 Database Retrieval

Many documents are valuable not because of the originality or uniqueness of the information that they contain, but because they have assembled in the right order various disparate pieces of information whose agglomeration is worthwhile. Examples of this sort of document are bibliographies, buyer's directories, and technical specification/repair manuals. The production of a document of this kind consists mostly of retrieving the necessary information from the appropriate place. A primitive version of this kind of document assembly can be had using the sub-file constructs explained in Section 7.3.2.

Support for more sophisticated automation of document assembly must include the interfacing of the document compiler with a database manager; the compiler will determine which records are to be retrieved from the database, then perform the retrieval itself, and include the retrieved text as part of the finished document.

The Scribe compiler contains a simple special-purpose database retrieval mechanism built to be a test bed for the more general task of generalized database retrieval from within a formatting compiler. Briefly, the author in preparing a manuscript makes citations to various bibliographic entries that he knows are stored in a bibliographic data base. The compiler collects the text of the bibliographic references, sorts them into an appropriate order, formats them into an appropriate format, and includes the resulting table in an appropriate place in the document,

then matches up the generated citation numbers with the citation markers in the text. M. Lesk of Bell Laboratories has implemented a very similar bibliography system that functions as a preprocessor to Troff [26, 34].

For example, the introduction to Chapter 3 of this thesis contains references to a pamphlet by Stanley Morison entitled "First Principles of Typography", and to a textbook by Arthur Turnbull and Russell Baird called *The Graphics of Communication*. The actual text of the manuscript file corresponding to the text on page 19 of this thesis contains the sequences

```
... and that all other factors are secondary@cite(Turnbull).
... people recognize its novelty''@cite(Morison ", p. 7").
```

The names "`Turnbull`" and "`Morison`" used in the @Cite commands are the retrieval identification keys from the bibliographic data base for those two references. The compiler has retrieved the appropriate entry, sorted the collective entries into alphabetical order by author's last name, and assigned reference numbers 44 and 31 to them. The citation style used in this document type specifies that references be put in square brackets, so the Scribe compiler produced entries in the finished document on that page that read:

... and that all other factors are secondary [44].
... people recognize its novelty" [31, p. 7].

If the citation style used in the formatting of this thesis had been other than the standard numeric citation style, then the examples above could have come out as:

... and that all other factors are secondary (Turnbull, 1975).
... people recognize its novelty"(Morison, 1967).

or perhaps as

... and that all other factors are secondary [TUR75].
... people recognize its novelty" [MOR67a].

or perhaps as

... and that all other factors are secondary[31].
... people recognize its novelty"[44].

The database mechanism built into Scribe to handle this bibliography task has all of the properties of an ordinary database system except generality: it selects, sorts, reformats, and configures, but only on bibliography data. Within this scope,

however, it is completely general—the Scribe format database includes definitions for several dozen different bibliography formats, including those required by major journals. A future writer's workbench system should certainly include a more general database retrieval mechanism, to allow retrieval tasks of this complexity to be used for other purposes than bibliography.

7.5 Summary and Prospectus

The Scribe Writer's Workbench was not so much designed as evolved. With the exception of the cross reference and bibliography mechanism and the index facility, all of the tools on the Writer's Workbench evolved to meet specific needs for specific projects, during critical periods in the activity of those projects. The flavor of some of the facilities comes from the relatively primitive support facilities that the host operating system and file system can deliver. Nevertheless, there are several common themes in the Writer's Workbench facilities: facilities for collecting and processing derived text, facilities for numbering and crossv referencing objects in the text, facilities for assisting with the management of large documents, and facilities for helping to automatically assemble documents from external databases.

Fruitful topics for further work include the development of a set of tools that is integrated with the host operating system and file system as well as the text editor, and integration of the compiler and text editor with a database system. With cooperation from the text editor, which would need to be able to read the Scribe databases and understand Scribe manuscript language syntax, facilities like the cross-reference summary chart would no longer be needed (the editor could serve as a query system for answering questions about the structure of the document) and various new facilities, such as automatic generation of "revision bars" would be possible. Close cooperation with a database system would permit more facile generation of documents such as form letters or statistical summaries.

Chapter 8
The Compiler

The Scribe compiler is a one-pass processor, written in Bliss [49], which processes manuscript files into finished documents. This chapter is a survey of its design, organization, and construction.

8.1 Overall Organization

Figure 29 shows the global structure of the compiler; it indicates the various interface layers between the compiler and the host computer. As the diagram indicates, the compiler proper is implemented to run on a mythical high-level machine running a mythical high-level operating system; an appropriate implementation exists for each real operating system that supports Scribe. The percentage figures on the various blocks of Figure 29 indicate the percent of the total object code size of the compiler represented by that block. The compiler proper occupies 45% of the code space, and the database manager and environment control mechanisms occupy another 18% of the code space. The remaining 37% of the code space is occupied by support routines that one might envision belonging in a subroutine library: they are not directly specifically oriented towards the Scribe compiler.

The various support modules are built on top of the data-type support, which in turn interfaces with the virtual operating system for memory management. Each of these pieces is discussed in more detail later in this chapter.

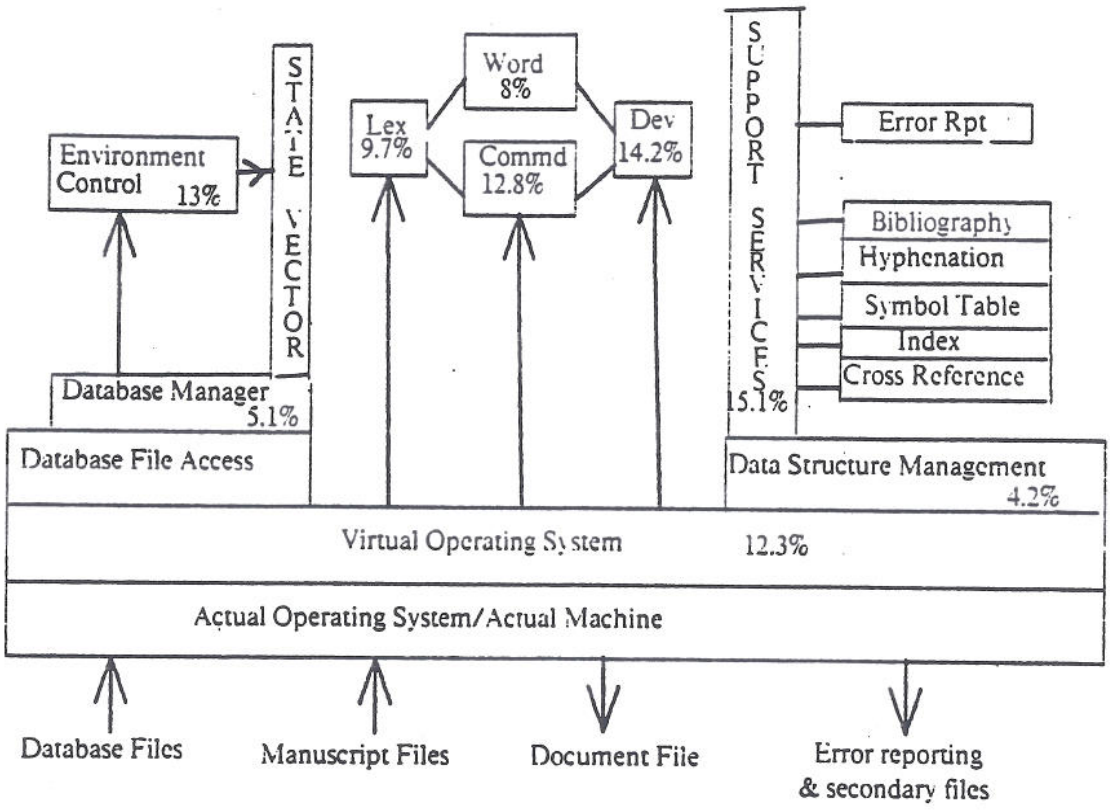


Figure 29: Conceptual structure of the compiler.

Module group	*Size*	*Percent*
Low-level support	16397	31.7%
Command/Environment processing	13676	26.4%
Lexical Analysis, Error Reporting	7559	14.6%
Device drivers	7361	14.2%
Justification and formatting	4165	8.0%
Database management	2638	5.1%
Total	51796	100.0%

Figure 30: Code space distribution.

8.2 Information Flow

To a first order, the data flow through Scribe is trivial: the user prepares a manuscript file in the document specification language and processes it with the compiler to produce a device-specific document file. That file is then printed on the appropriate printing device. To the unsophisticated user, this model is adequate.

Looking at a finer resolution, more data paths emerge. Figure 31 shows the minor data paths. The manuscript file may be supplemented by an auxiliary file, which was generated by a previous run of the compiler, a hyphenation dictionary that provides hyphenation information specific to this document, and a root file, which provides definitions global to the entire document of which the current manuscript is a part.

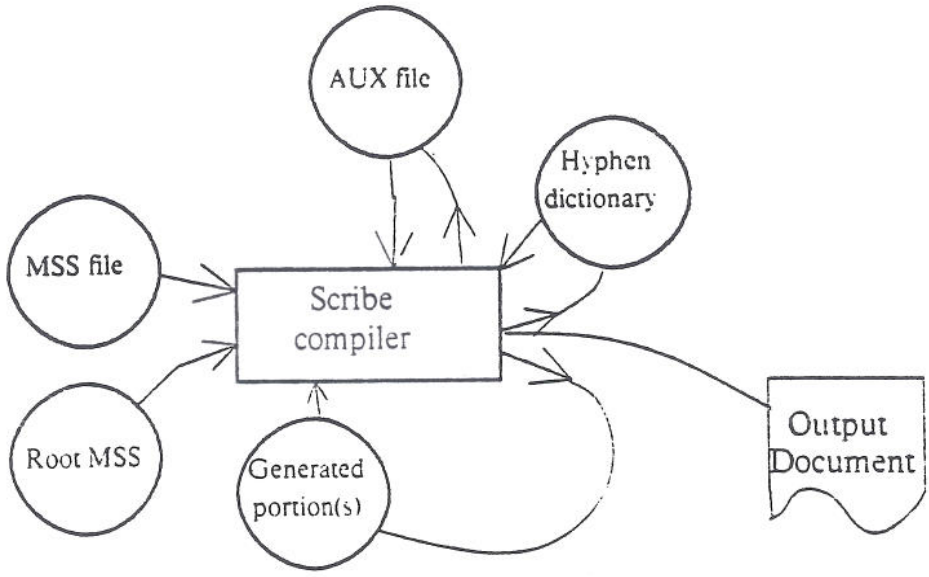


Figure 31: Scribe data flow paths.

During the compilation, the compiler retrieves device and font information from the "device" portion of its database, and retrieves format, style, and layout information from the "format design" part of its database. In addition to the intended document file, the compiler can generate an updated auxiliary file, an outline file, a vocabulary digest file, and a hyphenation dictionary update. These few derivative files are not crucial to the workings of the compiler, but are

bookkeeping aids to help either the author or the compiler with management of the document.

All files except the final document file that are read or written by the compiler are Ascii text files. In particular, all of the database files are text files. At the price of compiler speedup—they must be reparsed every time the compiler is run—this scheme offers flexibility and self-documentation. The fixed overhead time needed to locate and parse all of the database files needed by the compiler in a given run is about one second of processor time on a 1-MIP KL10 processor. Troff, by comparison, takes 10 to 20 seconds on a 0.6-MIP PDP-1/70 processor to read in its macro definition files. Implementation of a cache to permit the compiler to retain preparsed copies of database files would be relatively straightforward, though the current compiler does no such thing. Cache invalidation would need to be by creation time of the database file, since cooperation from the text editor used to edit the file cannot be guaranteed.

8.3 The Auxiliary File Mechanism

One of the most important characteristics of the system organization is its use of auxiliary files to simplify what would otherwise be iterative or multipass processing into a single-pass scheme. The essence of the scheme is straightforward: at the beginning of each compilation, the compiler reads in an auxiliary file that was produced by the previous compilation. The auxiliary file contains an edited dump of the compiler's symbol table, including information about forward references and the file structure of the document.

The auxiliary file contains:

- Information about the definition of every cross-reference label that has been defined, even if there is no reference to it.
- Information about the correspondence between manuscript files and compiler-generated data like chapter numbers or page numbers.
- A list of the fonts used in the document.
- A record of the tree structure of the document, if any, showing how the various part files are combined.

When a document is broken up into multiple files, each perhaps holding a chapter or a section, the auxiliary file contains enough information to allow a subfile to be

separately compiled, yet have all of the page numbers, section numbers, footnote numbers, and so forth, be correct.

As the document evolves under development by the writer, the auxiliary file used for each compilation will be slightly in error. Page number references will be wrong by the number of pages that have been added or removed, section number references will be wrong by the number of sections that have been added or removed, and so forth. If there is ever a need for a draft with all of the cross references correct, the author can compile the manuscript twice in succession without intervening changes. Otherwise the cross references will be syntactically correct—a page number wherever a page reference is expected, a section number wherever a section reference is expected—even if the correct values do not appear. The compiler always notifies the user if the document contains incorrect cross references. Since most drafts of a document do not have to be perfect, and since the rate of substantial change slows down as a document nears perfection, one very rarely finds the need to recompile a document for the sole purpose of getting the cross references to come out right.

8.4 Data Structures and Data Flow

Figure 32 shows the compiler data flow in block form. Each block has an indication of its relative code size in the compiler. There are numerous minor data paths within the compiler that are not shown in Figure 32; for example, the word assembler informs the hyphenator if a word being processed contains an optional-hyphen mark, and the command evaluator can send specific instructions to almost any module in the entire compiler.

8.4.1 Low-level data Types

The Bliss language is typeless. It provides typeless scalars, vectors, and records. The programmer must build the types and structures that he needs out of those primitive parts.

8.4.1.1 Simple Types

Scribe low-level support recognizes and supports 7 simple scalar types. Some are quite ordinary—integers and characters—while others are very specific to the problem domain of text formatting. The support for these types consists of routines to do input and output translation on them, to coerce them into other types, and

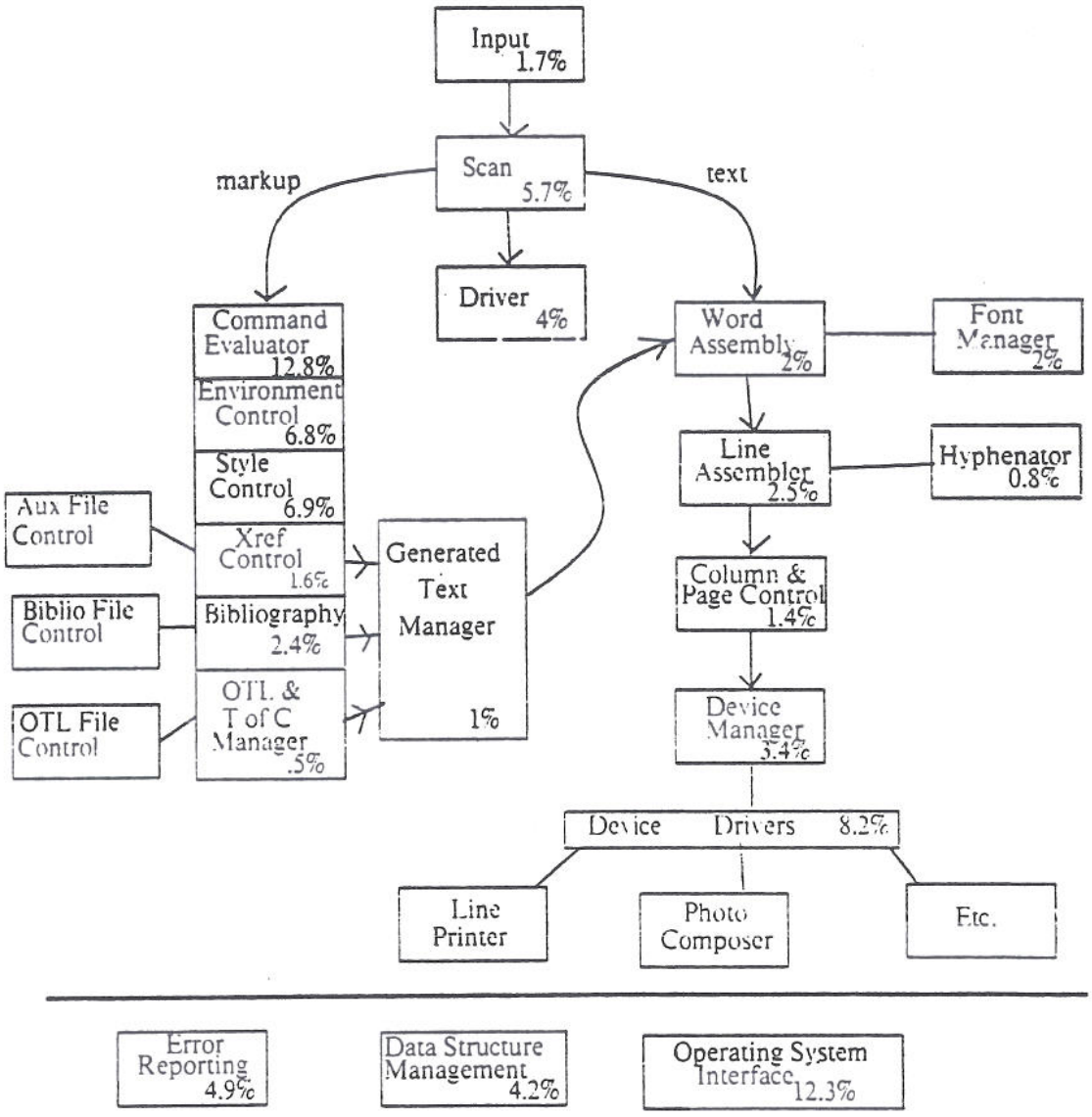


Figure 32: Major data flow paths within the compiler.

sometimes to create and destroy objects of that type if they occupy more than one machine word and therefore cannot be simple Bliss variables. The simple types supported are *integer, character, file-character, vertical distance, horizontal distance, font-relative vertical distance, font-relative horizontal distance,* and *type* (the token for type *integer* is of type *type*). Type *character* is a subrange type of *file-character.* File-character includes a special end-of-file character whose index value is 1 larger than the largest character.

8.4.1.2 Records and Storage Management

The Bliss language does not support records, but it does support typeless pointers that can be used to implement primitive record structures. The record system in Scribe is the means of dynamic storage allocation: memory is allocated by creating a record, and deallocated by destroying a record. No garbage collection of any kind is available in Bliss and no reference counts are kept. The intimate support from the compiler that is necessary in order to implement even a simple garbage collector was not available, and modification to the compiler was not a reasonable option. The absence of a scan/mark garbage collector was a serious impediment to the speedy development of the Scribe compiler.

8.4.1.3 Strings

The strings used in the Scribe compiler are variable-length objects built on top of the record system. A string consists of two parts: a token record and a buffer record:

```
type String_Token = record
 Buffer: pointer to String_Buffer;
 String_size: integer;
 Left_pointer: Character_index;
 Right_pointer: Character_index
end

type String_Buffer = array [1:N] of character;
type Character_index = 1:N;
```

String_Buffer has a varying length, and Character_index is an index into String_Buffer. The basic operations defined on strings are:

- `Create:String_Token`: Create a new empty string.
- `Destroy(S:String_Token)` Destroy string S and release its space.
- `Right_Append(S:String_Token;C:Character)` Append C to the right end of S.

- `Left_Remove(S:String_Token):Character` Remove the leftmost character of S and return its value, or the null character if S is empty.
- `Length(S:String_Token):Integer` Return as a value the number of characters in the string S.
- `Erase(S:String)` Make string S be empty.

Note that a string is not randomly addressed or edited, and that all changes to a string other than right-append or left-remove must be accomplished by copying. Note also that a string is defined out of characters and not file-characters, so it cannot contain the end-of-file character.

8.4.1.4 Association Lists

An association list, or pair list, is a list of pairs of typed values. Each cell of the list carries two values, with an explicit record of the type of each. These pair lists are sometimes used as property lists—one list for each object with its contents being attribute/value pairs, and sometimes as associations—one list for each attribute with its contents being object/value pairs. A list of N cells consists of N+1 list_cell records.

```
type list_pointer = pointer to list_cell;
type list_cell = record
 next_cell: list_pointer;
 value_1: any;
 type_1: type;
 value_2: any;
 type_2: type
end
```

The basic operations defined on lists are:

- `Create:List_Pointer`: Create an empty list and return a pointer to it.
- `Destroy(P:List_Pointer)`: Destroy a list and deallocate all storage assigned to it.
- `Insert_Before(P:List_Pointer;C:List_Cell)`: Inserts the cell C in front of the list denoted by P. The pointer P will have a pointer to the newly-inserted cell after the Insert_Before operation.
- `Delete_Cell(P:List_Pointer)`: Deletes the cell at the head of the list pointed to by P. After the Delete_Cell operation, P will point to the new head of the list.
- `Next_Cell(P:List_Pointer):List_Pointer`: Returns the cell following P in the list pointed to by P.
- `Find_Cell(P:List_Pointer,V:Any,T:Type):List_Pointer`: Returns

a pointer to the first cell in the list that has an attribute (first field) of V with type T.

All other list support functions are built from these.

8.4.2 High-Level Data Structures

The Scribe compiler proper deals with manuscript files, fonts, environments, word buffers, line buffers, text buffers, and various dictionaries and tables. Each of these structures is built from one or more of the low-level data types described in the previous section. The high-level data structures are described here in some considerable detail to impart a better sense of the operation of the compiler than could otherwise be had without reference to the code.

8.4.2.1 Manuscript Files

As a manuscript file is processed, it is represented as a sequence of records, each corresponding to one line of the manuscript file:

```
type Manuscript_line = record
 Line_name: string;
 Text_of_line: string;
 Processing_cursor: integer
end;
```

When the manuscript line is processed, it is read nondestructively by advancing the processing cursor so that the error message reporter will be able to display the entire text of the manuscript line as part of an error message. Most strings are processed destructively because it is more convenient.

8.4.2.2 Fonts

The Scribe compiler keeps font information for several purposes, which are discussed in more detail in Section 3.1.1:

- To know the sizes (widths and heights) of letters and symbols for the purpose of deciding how many words to place on a line.

- To know ligature combinations that are available in a font (since ligatures are font-specific).

- To know the codes to send to the printing device in order for it to be able to print or draw the desired letter.

Font information for various printing devices is kept in Scribe's database. When ready to use, a font has the following structure:

```
type font = record
 Font_name: string;
 Font_size: vertical_distance;
 Character_widths: array [1:127] of horizontal_distance;
 Character_displacements: array [1:127] of horizontal_distance;
 Character_contsructions: array [1:127] of string;
 Ascii_translation: array [1:127] of integer;
 Draw_codes: array [1:127] of string;
 Ligatures:  pairlist of (name: string, code: integer);
 Special_symbols: pairlist of (name: string, code: integer)
end;
```

8.4.2.3 Environments

Environments are implemented as pair lists used as property lists. An environment is an unordered set of pairs of dynamic parameter names and the changes to be made to those parameters. The mechanism of environments is described in Chapter 5. The structure is:

```
type Environment = pointer to Environment_Pair;

type Environment_Pair = record
 Next_Cell: pointer to Environment_Pair;
 Parameter_Name: integer;
 Change_value: any;
 Change_type: type
end;
```

The value in Change_value is used to update the dynamic parameter value of the parameter identified by Parameter_Name, according to the particular coercion rules for values of type Change_type.

8.4.2.4 Text Buffers

The manuscript text is assembled by the formatter (Section 8.6) into words, lines, boxes, and pages. Each of these is kept in an appropriate record. Word records are assembled into line records; line records are assembled into box or page records. When the assembly is complete, a device driver is called to write the assembled page image to the device file.

A word buffer holds one word:

```
type word_buffer = record
 Text: string;
 Bounding_width: Horizontal_distance;
 Bounding_height: Vertical_distance;
 Left_spacing: Horizontal_distance;
 Right_spacing: Horizontal_distance;
 Top_spacing: Vertical_distance;
 Bottom_spacing: Vertical_distance;
 Footnote_box: Text_box;
end;
```

A line buffer holds one line:

```
type line_buffer = record
 Text: string;
 Next_line: pointer to (Line_buffer or Box_Buffer);
 Parent_box: pointer to Box_buffer;
 X_origin: Horizontal_distance;
 Y_origin: Vertical_distance;
 Bounding_width: Horizontal_distance;
 Bounding_height: Vertical_distance;
 Left_spacing: Horizontal_distance;
 Right_spacing: Horizontal_distance;
 Top_spacing: Vertical_distance;
 Bottom_spacing: Vertical_distance;
 Footnote_box: Text_box; end;
```

The text of the line buffer is the concatenation of the text strings from all of the words in the line, and the various box widths and heights are the maxima (not sum) of the widths and heights of the corresponding fields of all of the words in the line. The X_origin and Y_origin fields are the X and Y distance of the lower left corner of the bounding box of this record from the lower left corner of the bounding box of the containing box record, or else are absolute coordinates if there is no containing box record. The use and contents of the other fields is described in Section 8.6.

A box buffer is similar to a line buffer, but instead of a text field, it has a Child_box field, which points to a list of line records that contain the actual text.

```
type box_buffer = record
Child_box: pointer to (Line_buffer or Box_Buffer);
Next_line: pointer to (Line_buffer or Box_Buffer);
Parent_box: pointer to Box_buffer;
X_origin: Horizontal_distance;
Y_origin: Vertical_distance;
Bounding_width: Horizontal_distance;
Bounding_height: Vertical_distance;
Left_spacing: Horizontal_distance;
Right_spacing: Horizontal_distance;
Top_spacing: Vertical_distance;
Bottom_spacing: Vertical_distance;
Footnote_box: Text_box;
end;
```

8.4.2.5 Symbol Table

The compiler uses a very standard block-structured symbol table. All commands, environments, user-defined names (except cross references), and file names are kept in the symbol table.

8.4.2.6 Dictionaries

The compiler maintains a number of dictionaries, in addition to the symbol table. A *dictionary* is a table of words, with some sort of value information stored for each. For example, the hyphenation dictionary is a table of words to be hyphenated, and the value is the list of the legal hyphenation points. The cross-reference dictionary is a list of cross-reference names, and the value is the page number and section number on which each is defined.

These dictionaries are maintained in association list data structures with the text word as the attribute field and the associated value as the value field.

8.5 Parsing and Error Reporting

The Scribe manuscript specification language is not a programming language, and one of the ways in which this difference is most manifest is the structure of the parser. There are no syntactic restrictions on the location or context of text or well-formed copymarks, though several semantic restrictions are enforced.

The specification language is syntactically trivial; its entire grammar is shown in Figure 33. The parser is therefore conceptually trivial. The considerable size and complexity of the parser implementation (there is more code in the parser than in the formatter) is due entirely to its error detection and recovery algorithms and its error reporting.

8.6 Formatting and Justification

The formatting section of the Scribe compiler receives as input from the parser a stream of words and word fragments, and produces as output a 2-dimensional image of the finished page. When the page buffer is full, or when some external agent requests a new page, the device driver is called to output the entire page to the output device. During the page assembly process, the formatter sometimes interrogates the device driver about certain properties or requests it to perform certain device-specific formatting functions; it is otherwise device-independent.

8.6.1 Word Assembly

The innermost loop of the formatter is the word assembler. Its job is to look up character widths, find and substitute ligatures, and perform any translation or capitalization requested in the current state. Ultimately it produces a *word token*, which is passed to the line assembler. This word token contains

- The text of the word
- Dimensions of a bounding box that bounds the word (i.e., the height and width of the word)
- Dimensions of a spacing box that surrounds the bounding box, to be used for positioning that word relative to other words
- All necessary font and magnification information about text within the word.

<manuscript> ::=	(<text> | <copymark>)*
<text> ::=	((<word> <word break>)* <sentence break>)*
<word> ::=	(Any printing character but '@')* | <word><copymark><word> | <null>
<word break> ::=	(Any nonprinting character)*
<sentence break> ::=	('.' | '?' | '!') <closure sequence> <space> <space> (<space>)*
<closure sequence> ::=	(')' | ']' | '"' | " ' ")* | <null>
<copymark> ::=	'@' (<punctuation character> | <named command>)
<punctuation character> ::=	'!' | '@' | '#' | '$' | '%' | '↑' | '&' | '*' | ')' | '_' | '=' | '~' | '[' | '|' | '\' | ':' | ';' | '>' | '.' | '/'
<named command> ::=	(<letter> | <digit>)* <delimited argument>
<delimited argument> ::=	'(' <argument> ')' | '<' <argument> '>' | '[' <argument> ']' | '{' <argument> '}' | '"' <argument> '"' | '"''" <argument> '"''" | '"''" <argument> '"''"
<argument> ::=	<text argument> | <keyword argument>
<text argument> ::=	<text> | <text> <copymark> <text> | <null>
<keyword argument> ::=	<keyword> ('=' <space> '/') <value> | <keyword argument> ',' <keyword argument>
<keyword> ::=	(<letter>)*
<value> ::=	<delimited string> | <typed value>
<delimited string> ::=	'(' (<any character but ')'>)* ')' | '<' (<any character but '>'>)* '>' | '[' (<any character but ']'>)* ']' | '{' (<any character but '}'>)* '}' | '"' (<any character but '"'>)* '"' | '"''" (<any character but '"''">)* '"''"
<typed value> ::=	(<integer> | <real>) | <value name> | (<integer> | <real>) <unit name>
<unit name> ::=	inches | points | cm | mm | ...
<value name> ::=	true | false | <some keyword>

Figure 33: Document specification language grammar.

Some sample word tokens, with their bounding and spacing boxes indicated, are pictured in Figure 34 next to the fragment of manuscript text that generated them.

8.6.2 Line Assembly

As the word assembler tentatively finishes each word, its token is passed to the line assembler for inclusion in a line record. The word completion is tentative because the line assembler might signal "that word does not fit, so try to hyphenate it". In this case, the word assembler must begin the assembly process anew, replacing ligatures by their unligated text and regenerating pairs of word tokens if the word can be hyphenated.

One way or another, the line assembler builds an output line by concatenating word tokens. The spacing boxes are not abutted, but overlap: the assembler places two words next to one another as shown in Figure 35. The bounding box of one word and the spacing box of the next are not permitted to overlap.

When a word occurs at the left end of a line, the left end of its spacing box is ignored, and the left edge of the bounding box is aligned with the left margin of the line. Similarly, the right end of the spacing box of a word at the right end of a line is also ignored.

The "stretchable glue" concept used by Knuth in TEX is a more general realization of this bounding box/spacing box concept. TEX needs to keep on hand more information about each character than does Scribe, but it is able to do a better job of line assembly because of the more general "glue" mechanism [24].

8.6.3 Box and Page Assembly

When the line assembler finishes a line, its record token is passed to the box assembler for inclusion in a box. Box tokens and line tokens are structurally identical, so that a box containing several lines can be recursively passed to the box assembler for inclusion in a larger containing box. The page output buffer is just a box with a restricted size.

The algorithm for vertical assembly of lines into boxes is essentially the same as that for horizontal assembly of words into lines. The spacing boxes of two vertically adjacent lines are overlapped, but the spacing box of one line is never allowed to overlap the bounding box of another.

When lines contain characters of radically mixed sizes, as for example in the line with the integral sign on page 47, there are two different strategies that the page

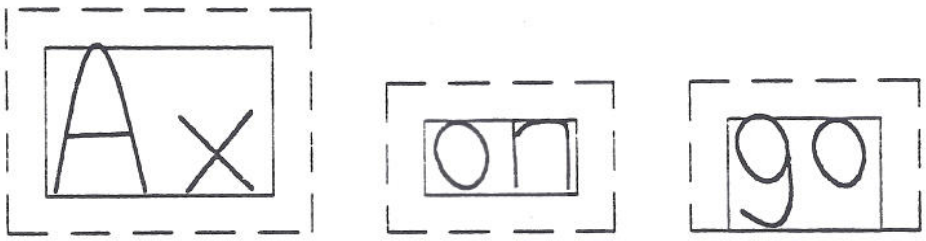


Figure 34: Word tokens, showing bounding and spacing boxes.

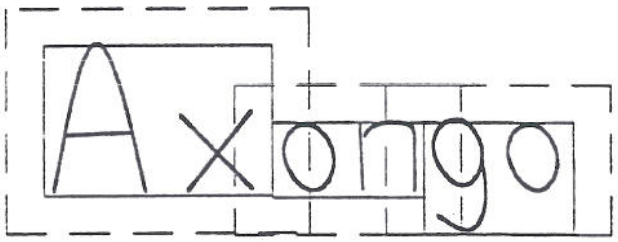


Figure 35: Use of bounding and spacing boxes in line assembly.

assembler can use. The first is to let the line spacing remain constant, regardless of the characters that are in the line; this is in fact how the text on page 47 was spaced. The second is to let the line spacing be increased until there is no actual overlap of characters, or possibly even until there is some minimum amount of space between characters on adjacent lines. One of the dynamic state parameters, *Line push* (item 32 on page 137) selects between these two modes. In general it is best to set *Line push* false in ordinary text environments and true in environments containing formulas or built-up characters.

8.6.4 Hyphenation

The Scribe compiler uses a dictionary-based hyphenator, with no rules of any kind used by the compiler when checking words in the dictionary. The hyphenation dictionary is handled as a sort of auxiliary file. During compiler initialization, the hyphenation dictionary or dictionaries associated with the document are read in. As the formatting progresses, any word that needs to be hyphenated but cannot be found in the hyphenation dictionary is recorded in the error log. At the end of the compilation, the error log contains an alphabetized list of the words that could not be hyphenated. A simple merging program is used to update the document-specific hyphenation dictionary by looking up the unhyphenatable words in a master dictionary and copying the results into the document's own pocket hyphenation dictionary. The user can manually enter into the hyphenation dictionary any specialized words that he needs to use as well as any words whose standard hyphenation he disapproves of.

Even though no purely syntactic hyphenator can guarantee perfect results (this problem is discussed in Section 3.2.3), dictionary hyphenation is perfectly adequate in practice. Scribe's pure-dictionary scheme works equally well for most Western languages, although it is not particularly convenient for any of them. The use of a master hyphenation dictionary to which the compiler would always refer when it could not hyphenate a word, and the automatic updating of the document-specific hyphenation dictionary whenever such a word is looked up (thereby making the document-specific hyphenation dictionary a cache of words from the master dictionary) would improve the convenience of the hyphenator at the expense of a little extra code in the compiler.

8.6.5 Footnotes

As noted in Section 8.4.2.4, every text assembly record has the potential for an attached footnote box. When a footnote is found in the manuscript, the formatter is called recursively to produce a box record that contains the body of the footnote. This footnote box is then attached to the footnoted word. When the footnoted word is placed on its line, its attached footnote box is placed in the line's footnote box. When a line containing footnotes is placed in a page, its footnote box is inserted (as a line) into the page's footnote box. When a page box with attached footnotes is passed to a device driver for output, its footnote box is "anchored" by being inserted into the page box as a line record.

Footnotes that occupy a substantial fraction of a page present an especially sticky problem in page assembly. The usual style conventions for footnote placement specify that the footnote appear on the same page as its callout, but good page assembly also mandates that the bottom margins of pages be reasonably consistent. When a footnote is so large that moving it and its associated text line to the next page causes a large amount of white space to be left behind, the correct solution is to put the text line and the first few lines of the footnote on one page, then continue the footnote on the next page. The Scribe compiler makes no attempt to break long footnotes across page boundaries.

8.6.6 Floating, Grouping, and Page Break Control

Good page layout requires that certain sets of lines be kept together on a page, possibly by "floating" them to a convenient nearby page top or bottom. Objects requiring this kind of treatment include equations, tables or figures, and some kinds of displayed text.

One of the dynamic parameters is a "clustering" value that, if set, denotes that all text generated while it is set must be made to satisfy certain requirements of pagination. The details of the pagination requirements—grouping or floating—are indicated by the value of the clustering attribute.

An environment whose text must be clustered must specify as part of its definition a value for the cluster attribute. When the change analyzer (Section 5.1) is called during processing of the environment entry, it will notice that the value of the cluster attribute has changed from one that does not specify clustering to one that does. The change analyzer will therefore allocate a new assembly box so that all lines passed from the line assembler will be placed not in the page box but in this new box. When the environment is exited, the change analyzer will discover the

reverse situation, and place that box into the page box or delay it for a later page, as needed.

Widow and orphan elimination is accomplished by a crude set of special-purpose tests in the box assembler, which monitor incoming lines and force various lies to be told by the routines that determine whether or not a line will fit on a page.

Part III

Results, Conclusions, and Future Directions

An operable Scribe compiler, with moderately complete databases, was released in February 1978 for use within Carnegie-Mellon University and several other laboratories. On the basis of that experience numerous small changes were made to the compiler and language, and the need for various larger changes was noted. The compiler was almost completely rewritten during the summer of 1978 to take those needs into account. This last part of the thesis details the experience gained from the user community, evaluates the finished product, and reflects on the original goals and principles in the light of this experience.

Chapter 9 outlines the chronology of the project, including occasional setbacks and redesigns, and comments on the effectiveness of the finished product from the point of view of a user. Chapter 10 is a critical retrospective on the project's goals and principles, with an eye towards more ambitious future work.

Chapter 9
An Evaluation of the System

9.1 Chronology

Work on the language design began in Spring 1976 after extensive discussions with W. Wulf, M. Shaw, and D. Lamb about the nature of the problem. Lamb and Shaw surveyed the habits of users of existing document preparation systems. The study revealed that there was a relatively small number of fundamentally different formatting effects that users were trying to achieve, but that there were a large number of minor variations of each of them. We therefore concluded that there was sufficient uniformity of usage style to make an environment-based language practical with a small number of basic environments *provided* that there was a simple mechanism for making small changes to their behavior—the number of fundamentally different styles of formatting seen in Lamb and Shaw's survey was small, but almost no two documents used quite the same set of details.

Lamb, Shaw, and I then designed a prototype language that allowed users to express directly the formatting effects that they were trying to achieve rather than the procedures necessary to achieve them. I then implemented this language as a set of Pub macros. Following a suggestion by G. Yuval, the language was named Cafe (a civilized pub). These macros came into reasonably general use in the Carnegie-Mellon computer science department community, but they were error-prone, extremely slow, and had unpleasant semantics that masked the simplicity of the language that they were trying to implement. I then launched upon the ambitious project of implementing an entirely new compiler that would process the Cafe language directly. During the course of planning for that implementation, I devised a new language, sufficiently different from Cafe to warrant a new name: Scribe.

9.2 Evolution of the Compiler

The original goal for the compiler was to produce a body of code that was portable among all computers in the CMU environment, namely PDP-10's, PDP-11's, and the C.mmp multiprocessor. For this reason, the implementation was begun in Bliss [49], since it was the only language system available on all of these machines. Bliss proved to be an extremely difficult language in which to get started on such a project, since it has utterly no low-level support for any data types besides scalar words and stack-allocated vectors.

I began an implementation on the PDP-10 in September 1976, spending the first six months building a programming environment in which the rest of the development could take place. This programming environment included runtime and diagnostic support for strings, lists, and heap-allocated vectors, as well as an operating-system interface intended to be portable to other machines. I began work on the actual compiler in May 1977, and had a system producing output for line printers by November 1977. XGP support was completed by January 1978, and the first release of the compiler was in February 1978. Development effort was suspended while I attended to passing my qualifying exams.

The first Scribe compiler had device drivers for line-printer-class and XGP-class devices, a database with five document types, and a number of serious restrictions on the manuscript.

The first Scribe compiler implemented an option for "idempotent update" of the manuscript file so that its line and page breaks corresponded to the line and page breaks in the final document. This manuscript update also placed generated text—chapter and footnote numbers, values of string expansions—into the manuscript file as comments. While convenient, this idempotent manuscript update facility had two serious drawbacks that led to its being removed from the second compiler: it led to manuscript files that could not be edited easily on ordinary monofont terminals, and it made all compiler bugs become major, since the manuscript file itself would be damaged by any mistake in the formatting. Manuscript file update is a valuable facility, but for it to be practical, experience with Scribe leads to the conclusions that:

1. The interactive editing device must be no less powerful than the printing device with respect to line widths and fonts available.

2. The file system must support multiple versions of a file, so that the original manuscript is never lost in case of a compiler failure.

3. The file format for the file representation of the manuscript must be rich enough to be able to represent text that will not appear in the output—comments, false conditional branches, etc.—in such a way that it does not interfere with line lengths and page size computations.

A second compiler was begun in June 1978 and released in January 1979. It shared low-level support routines with the first compiler, but most of the substantive code was completely redesigned and rewritten. Various defects in the first compiler motivated this rewrite.

The first compiler placed severe restrictions on the relative placement of declarations in the manuscript file; the placement sequence corresponded to phase sequences within the compiler. This proved to be too restrictive, and was a constant source of difficulty for users. A sort was added to the second version of the compiler so that mansucript declarations could be in any order; the compiler sorted them before actually processing them. This technique has been entirely satisfactory, except that semantic errors in declarations are not printed in the same order as the declarations appear in the manuscript file.

The first compiler did not have the bibliography facility (see Section 7.4), and also did not have a macro facility. Macros were omitted from the initial compiler to encourage more creative use of the environment mechanism; the presence of familiar procedural macros would be too much of a temptation for a programmer learning to use the system. No way of implementing the bibliography formatting templates without the use of parameterized macros was ever devised, and so the second compiler had the macro facility to support the bibliography mechanism. Macros remain a painfully clumsy and error-prone mechanism for most text processing applications, but nothing that is clearly superior has evolved.

9.3 Evolution of the Manuscript Language

While the compiler has evolved in the direction of increasing complexity and sophistication, as do all maintained software systems, the document specification language has evolved in the opposite direction. The final published language has fewer commands, fewer restrictions, and a smaller number of predefined names than did the earlier editions. On the other hand, the number of environment attributes in the mechanisms used to implement language constructs in the database has doubled.

All of the language simplification has happened as a direct result of field experience: when users were unable to comprehend the difference between two similar language constructs, they were merged into one. For example, the earliest

versions of the manuscript language had a @Font declaration to specify font family, in addition to the @Style declaration to specify other style parameters. Users found the distinction incomprehensible, so the @Font declaration was eliminated in favor of a @Style keyword named *Font*.

The first version of the manuscript language had separate commands for retrieving the values of user-defined strings and system-defined strings. The motivation for this distinction was that it permitted users to define string names for their own use without needing to worry about whether or not their names collided with system names. In practice all users who were sophisticated enough to use the string definition facility were able to remember the list of predefined string names, and occasionally wanted to redefine system strings. The two commands were merged into one.

9.3.1 Evolution of the Databases

The databases have undergone the most substantial evolution, both in terms of their content and the database language used to express that content. The database language is still relatively weak, but it has been enhanced slowly in order to learn what sort of expressiveness is actually needed. Initially the database language permitted only environment and counter definitions and static parameter values. The ability to place manuscript text in a database file was added to permit the description of document formats with constant text, such as letterheads. It was next found necessary to add to environment definitions the ability to specify initialization and wrapup text with them.

The database language currently lacks a means of event recognition, i.e. triggering specific action upon the first or every instance of certain events, such as counter incrementation or page completion. The *trap* mechanism used in Troff is a good example of an event trigger for spatial events such as line or page completion [34].

Many sophisticated users, especially those who define their own document types, have requested that the database language be expanded to include a Turing-equivalent programming language, allowing arbitrary computations to be performed. I have avoided the installation of such a facility for two reasons. The first reason is that a procedural language will reduce the amount of feedback that I get from users about the kinds of formatting that they find themselves unable to do in Scribe. Although their goals are to design document formats and to produce documents, my own goals are to learn about the requirements of document formats. If users in the field were given an algorithmic database language, they could program around deficiencies in the design of the system, thereby preventing me

from ever finding out about those deficiencies. The presence of this algorithmic language capability would improve the usability of the compiler to demanding users, but decrease its usefulness as a data-gathering tool. The second reason for the continued absence of a procedural basis for the database is that in the absence of enforcement mechanisms or user training, programmability invariably leads to a diversity of style, which makes documents harder to read, harder to merge or combine, and harder to transport. Furthermore, a programmed system implemented by a diverse variety of people without central control, namely the union of the procedural extensions with th basic system, will invariably be more obtuse and difficult to understand and use than a more unified one.

Chapter 10
Critical Retrospective

The Scribe system *in toto* is a resounding success, though no software system pleases everyone. My estimate of the size of the user community in September 1980, based on sales of the user's manual, distribution of the code, and rates of complaints received, is five to six thousand active users. The majority of them appear to be reasonably satisfied with it. Nevertheless, it is our responsibility as systems designers to understand the weaknesses as well as the strengths of a system, so that we can learn from it as much as possible for similar future systems. This chapter is a narrative discussion of the various successes and failures of the Scribe project with respect to the goals stated in Chapter 2 and the expectations of its users.

10.1 Language Goals

The goals for the document specification language were that it be nonprocedural, syntactically trivial, and easily parsable. During the development of the compiler and especially during the initial release period, there was strong temptation to warp the language somewhat in order to make it easier for the compiler to handle some difficult construct properly. Once the language has decayed by the addition of some new construct whose purpose was the solution of a specific problem, it is politically difficult to remove that construct from the language once the compiler has been enhanced to no longer need it: there is always a community of users who have come to depend on the full set of features, good and bad, of the language.

10.1.1 General Language Issues

Many of the worst problems remaining in Scribe are actually language design problems, even though most users see them as bugs in the compiler. Once Scribe was in heavy use in the field, problem reports trickling in showed that various changes in the language, some small and some radical, were needed. On the other

hand, the very community of users whose feedback helped locate those problems makes it difficult to repair them, since they have built up a large investment in source files in the old language, and will be seriously disrupted by incompatible changes in it. As a result, many problems that would best have been fixed by incompatible changes to the document specification language were instead fixed, or at least ameliorated, by changes to the compiler.

Sometimes language problems could be solved by the addition of new declarations, which would allow users to patch specific problems themselves. As a result, the document specification language is slightly impure. Although largely nonprocedural, there are various procedural commands in it that allow users to overcome other shortcomings. The @Newpage command is an example of this: the original document specification language did not include any mechanism whereby users could modify environments to fall automatically on a new page. The @Newpage command was therefore added so that users could explicitly request a fresh page. Later a dynamic state parameter ("Page break", item 27 on page 136) was introduced, allowing environments to be set up so that they automatically started on a new page. By this time the @Newpage command was in use by most of the user community, so that it could not be removed without inconveniencing them.

The document specification language does not include any clean way for passing multiple text arguments to a declaration, or even for passing a single text argument at the same time as one or more identifiers or scalar parameters. While this property has made the language much more robust—users are never confused about the correct delimiters to use for a text argument—it has the side effect of making certain declarations clumsy or incorrect. The most glaring example of this is the @Tag vs. @Label confusion in the cross reference facility, which is discussed in Section 7.2. To unambiguously attach a cross reference label to a section number, in order that the compiler can know that it is referring to that section number as an object and not as the location reference for a nearby object, the cross reference label needs to be attached to the section marker.

The Scribe user currently defines a section and labels it for cross referencing it with two consecutive commands:

```
@Section(Numbering and Cross Referencing)
@Label(XrefSection)
```

He should instead be able to define the section and give it a cross reference label all in a single operation. By contrast, the GML system [13] performs this same operation with the ":h2" command, which defines a second-level heading:

```
:h2 id='XrefSection'.Numbering and Cross Referencing
```

In this case the beginning of the command is the colon character in the first column and the end of the command is the end of the manuscript line. A Scribe syntax for the same sort of combination command might be something like:

```
@Section(Numbering and Cross Referencing@,XrefSection)
```

This syntax. however, implicitly assumes that the field after the "@," separator is an identifier. It would be more general, but less convenient, to allow a syntax such as this:

```
@Section(Numbering and Cross Referencing@,Label=XrefSection)
```

The document specification language could be redesigned around the notion that declarations can have one or more keyword arguments in addition to a single text argument. This would substantially increase the complexity of the language, and is therefore probably not worth doing.

One of the goals for the document specification language was simplicity—the user should specify as little information as possible, and the compiler should be able to figure out what to do. One constant source of ambiguities in the document specification language is the disposition of spaces and carriage returns (end-of-line characters) in the manuscript. The ambiguity is in whether or not the spaces or carriage returns that follow a command in the manuscript are actually part of that command—and therefore should be ignored—or are actually text, and should be processed as such. In the following example, the carriage returns after the @Index commands are part of the commands themself, and should not be processed as text. The user cannot be expected to understand this obscure distinction, and most are reduced to trial and error in attempts to get it right.

```
Fill the crankcase with 30-weight motor oil.
@Index(crankcase, filling)
@Index(oil, crankcase)
@Index(motor oil)
Now start the engine.
```

In the following example, however, the carriage returns after the @B commands are intended to be text, and should not be discarded:

```
Begin the assembly with the following parts
@display[
@b[4 sections of pipe]
@b[1 can of pipe joint compound]
@b[1 hacksaw]
]
Begin by opening the can of pipe joint compound.
```

No syntactic clue can be used to distinguish these two cases. Only the semantic difference between the two commands tells us how to handle them. Such tasks as handling of carriage return characters should be handled by the lexical analyzer, and this need for semantic information in the lexical analyzer substantially increases the complexity of the compiler. Many other systems get around this problem by favoring consistency over convenience. TEX, for example, always discards all blank spaces and carriage returns after a command, and if the user wants them to be part

of the text, he must place an explicit marker specifying that. The equivalent feature applied to Scribe syntax would require that the @; noop code be used after every command for which the following carriage return is text:

```
Begin the assembly with the following parts
@display[
@b[4 sections of pipe]@;
@b[1 can of pipe joint compound]@;
@b[1 hacksaw]@;
]
Begin by opening the can of pipe joint compound.
```

Instead of requiring this more-complicated syntax in the document specification language, the Scribe compiler goes to great and complicated lengths to handle carriage returns properly. It sometimes gets them wrong.

10.1.2 Portability

Another goal for the document specification language was portability, of all kinds: device portability, site portability, and computer-type portability. By and large this goal was successfully met, though there were a few problems. An unstated aspect of the portability goal was that the specification language was supposed to *force* users to produce portable manuscript files, whereas in reality it only encourages them to do so. A clever user can always find ways, usually by misuse of the @Modify command, to make a manuscript be committed to a particular device.

One of the most difficult aspects of device portability was the treatment of overlong lines. No two printing devices seem to have quite the same set of fonts or maximum paper widths, and in frequently occurs that a line that fits within the margins on one device must somehow be truncated, wrapped, or shrunk on another device. Tabular or unfilled lines that just barely fit within the margins on one printing device will go far beyond it on another, so that the lines need either to be broken, compressed or printed in a smaller font. In order to do an acceptable job of any of these, the compiler must know not only that they are nofill lines, but why they are nofill lines. If they are computer program text, then there are certain rules that can be applied, a kind of prettyprinting, for breaking up the overlong lines into several shorter lines. If they are tabular material, then perhaps the inter-column spacing can be reduced or the table split into two parts or turned on its side. If they are mathematical formulae, then there are standard (though complex) ways of breaking formulae across lines.

In general there is always a "reasonable" way to break long lines, but the compiler does not necessarily know what it is. The solution to this problem would be to create a large number of specialized environments, each corresponding to a

different kind of material with a different breaking rule, and then to add a lot more knowledge to the compiler about how to break overlong lines in various kinds of environments. It is worth nothing that this problem of overlong lines is not peculiar to computerized text formatting. The correct disposition of lines too long to set in a textbook format is a constant source of dispute between editors and authors of manually-produced books.

A more serious problem with device portability of a manuscript is its character set. As discussed in Section 4.4, there is no absolute identification for characters outside the standard set (whether that standard set is Ascii or BCD or Chinese), and therefore no standard manuscript conventions can exist for specifying those characters. The Scribe document specification language completely ignores the whole issue of character sets, leaving the user to fabricate his own special-character schemes from a set of primitive low-level tools. A proper treatment of special characters would associate with each document a directory of non-standard characters used in it, giving each a symbolic name for use within that document. The directory would provide for each symbolic name a definition or description of the character, possibly in the form of a digitized picture of the character or an output-device-specific command sequence that will construct that character. In the worst case, the directory would provide an alphabetic name or description of the character that the compiler could use to construct some sort of surrogate for it.

Site portability—the ability to transmit a manuscript from one computer site to another and have the receiving site be able to print it reasonably—requires device portability and more. Certainly if the sending and receiving site do not have the same printing device, then the manuscript must be device-portable before it can be site-portable. Site portability requires that the versions of the compiler used at the two sites be compatible, that the versions of the database used at the two sites be compatible, and that the document be device-portable.

Since the compiler has been centrally maintained, by me, and distributed in an immutable object form to essentially all sites that have received distribution, the problems of divergent versions of the compiler have been minor. This central maintenance has undoubtedly led to poor responsiveness to user complaints at all sites except Carnegie-Mellon, but it has made site portability possible. The database compatibility problem, on the other hand, is not solved. I produced about 15 different document types for the initial release of the compiler, and all of the sites that received copies of the initial release of the compiler immediately set out to develop their own document types. While most of the document types developed by people other than me are more attractive and better documented than my own designs, they do not in general tend to be environment-name-compatible with the "standard" set of document types or with each other. This has led to a situation in

which each of the major installations using Scribe has its own set of document types, none of which are entirely compatible. It is relatively simple to convert a document from dependence on one format definition to dependence on another, but it is not automatic, and therefore complete site portability is not achieved.

A more subtle problem in site portability of documents is the use of the partitioning facility described in Section 7.3.1. If one part of a document is transmitted to another site, but not all of the parts that it refers to are transmitted to that site, then the piece is useless out of that context. The bibliography database files used in the automatic bibliography facility are different from one site to another, which means that the retrieval keys used in @Cite commands will not be the same at any two sites. It is probably not practical to maintain a centralized bibliographic database with standardization of retrieval keys, but unless that is done or unless bibliography database files are transmitted along with the manuscript source (not a difficult task), then site portability is lost.

10.1.3 Domain

Scribe has been successfully applied to a very wide range of documents. I am aware of four hardbound books for which camera-ready copy was produced entirely with Scribe; one a biography [10], one a computer science textbook [50], one a monograph on an operating system [51], and one a monograph on multiprocessors [39]. Dozens of theses (including this one, of course), hundreds of manuals, and thousands of shorter documents have been produced at CMU, and I am certain that useful documents of other varieties have been produced at other sites.

As predicted, it is extremely difficult to bludgeon Scribe into producing a format that it was not designed to produce. C. Leiserson has succeeded in getting it to produce respectable mathematical formatting, but to do so he has had to abandon all pretenses of portability. A system designed with mathematics in mind, such as EQN or TEX, can be completely portable for mathematical expressions, and there is no reason why such a facility could not be added to Scribe, or why a system could not be built that combines the document portability achievable in Scribe with the equation portability achievable in EQN or TEX.

A programmable manuscript language, or even a programmable database language, would greatly increase the domain flexibility of the Scribe compiler. I discuss in Section 9.3.1, on page 108, why such programmability was carefully left out of the Scribe language or database language.

10.2 Compiler Goals

The goals for the compiler were that it work well enough for people to use it voluntarily, and that it be sufficiently mutable that the majority of its users would be able to achieve the format variations that they wanted. An unstated goal, perhaps not realized soon enough during the implementation, was that I had to have something running and released to the department community within about 6 months of when I started the implementation.

It actually took about 15 months to get the first compiler running; about 6 of these 15 months were spent implementing the low-level data type support and operating system interface that should have been a part of the programming language support system. The decision to use BLISS was made in large part because we wanted to be able to carry the compiler to our experimental PDP11-based multiprocessors, and the only language common to both machines was BLISS. The complete lack of runtime support or data typing probably made the debugging task a whole order of magnitude more difficult, and as I look back on the implementation and reflect on the nature of the debugging task, I am completely amazed that the compiler works at all. The largest single source of implementation problems, by at least a 2:1 margin, was management of pointers to heap-allocated objects. All dynamically-created objects must be explicitly erased, and it is far too easy to accidentally hide away a pointer to an object that is subsequently erased and then reallocated, with the result that the pointer now points to some entirely different object. Strong typing, preferably with complete garbage collection, would have made this debugging more tractable. On the other hand, the implementation takes considerable advantage of the typelessness of BLISS structures to build an almost LISP-like symbol manipulation environment that was crucial to the environment mechanism and the definition-by-analogy mechanism.

Since I was the only programmer involved in the implementation effort, and since its implementation was not my primary goal in the project, my interest in proper software maintenance often left something to be desired. After the initial release, which hardly worked at all, the compiler went through two periods of intense instability, each following a switchover to a new release of the compiler. Although the user community was greatly inconvenienced by these periods of instability, and may never forgive me for it, a significant piece of data emerged from watching three complete cycles of the redesign-rebuild-restabilize loop. That result is that it is much more important for a compiler that is trying to be smart to actually be so than it is for a compiler that is trying to be obedient to actually be so. The whole Scribe approach is based on putting all of the intelligence in the compiler, and keeping the manuscript language relatively simple. A byproduct of this approach is that when

the compiler is not properly debugged—which it almost never was—the users had essentially no mechanism for circumventing compiler bugs. More traditional algorithmic compilers, whether for programming languages or document production, make it much easier for a user to circumvent bugs by programming around them.

Besides reliability, the other major goal for the compiler was that it support a definition-by-analogy mechanism that would make it easily mutable by casual users. This goal was met almost perfectly. The @Modify and @Define commands work well in practice, and users have tended not to go overboard in defining new environments just because the definition mechanism exists.

There are two difficulties with the mutation scheme, both minor, that nevertheless bear mention. The first is that there is no simple way to *remove* an attribute from an environment definition—the @Modify command permits only the addition of new attributes or the alteration of existing attributes. This was not a serious problem, because one could always look up the existing definition and copy it exactly, minus the attribute to be removed. The second difficulty with the mutation scheme turned out to be that many users did not understand the difference between static and dynamic state, and could never form a working mental model for when to use the @Style command, which changes static state parameters, or the @Modify/@Define commands, which change the dynamic state parameters. Probably no more than 15% of the user community understood the distinction well enough to be able to use the commands without consulting the manual every time; this indicates that the distinction between the two kinds of modification mechanism is either too obtuse or too arbitrary and should be eliminated.

10.3 Documentation Goals

Nobody ever reads manuals, or so it would seem to the people who write them. Nevertheless, a good manual is an integral part of any software system, and as mentioned in Chapter 2, the manual is actually an informal specification of the intended behavior of the system, and is therefore available as a tool for finding problems in the design.

The documentation goals were to produce a tutorial, an advanced manual, and a pocket reference. A very abbreviated 40-page tutorial was released with the first version of the compiler in February 1978. By August 1978 I had finished the first genuine edition of the tutorial, and began work on the advanced manual.

As I began to produce drafts of the advanced manual, which I had tentatively titled the *Scribe Expert's Manual*, I noticed that often people would steal them from

the printer before I had a chance to go downstairs and pick them up. Everybody wanted to be an expert, and owning a copy of the *Expert's Manual* put you halfway there, even if it was a bootleg copy. As various drafts of the advanced manual got into circulation one way or another, I found that people immediately started using the features described in that manual, even if they didn't need to, just because the features were there. When I subsequently made changes to those "expert" features, such as the database language, people objected violently that their documents suddenly didn't work any more.

I then embarked on a campaign to destroy all extant copies of the *Expert's Manual*, in order that I could do further work on the design of the database language without disrupting people's work. Most of them were actually purged, but enough people had memorized its contents that there were still a number of people busily making their own document format definitions and filling their documents with just the sort of low-level commands that I didn't want people using directly in their documents.

The second compiler called for a second edition of the tutorial, and together with J. Walker of Bolt Beranek and Newman, I produced a second edition of the manual only six months after the second compiler was released. Walker and I also collaborated on the third edition of the tutorial, which did not contain much new material but which was very much reorganized according to what we had learned about how to present the material from a year of experience with the second edition of the manual.

We have found that people who have no particular background in computer science or programming can in an hour or two of reading the tutorial manual learn enough about Scribe to be able to produce simple but useful documents. On the other hand, those people who have a programming background, especially those who have extensively used procedural document preparation languages, have much more trouble getting started because they seem not to believe the explanations in the manual and keep reading until they learn how to program it, and in the process building a completely incorrect mental model of how the system works.

There is still no adequate documentation on the database language or the macro facility, primarily because I wanted the freedom to continue to make changes to them. A pocket reference guide was printed in August 1979, but it is very slightly too wide to fit in some pockets.

References

[1] Addison-Wesley Publishing Company.
Principles to Observe in Paging.
Addison-Wesley internal memorandum.

[2] Wm. Atkins (editor).
The Art and Practice of Printing.
New Era Publishing Co. Ltd., Holborn, London, W.C.2, 1915.

[3] N. A. Badre and C. H. Thompson.
Yorktown Mathematical Formula Processor User's Guide
IBM T. J. Watson Research Center, Yorktown Heights NY, 1977.

[4] M. P. Barnett, D. J. Moss, D. A. Luce, and K. L. Kelley.
Computer Controlled Printing.
In *Proceedings of the Spring Joint Computer Conference, Vol. 23.* AFIPS, 1963.

[5] M. P. Barnett.
Computer Typesetting: Experiments and Prospects.
MIT Press, 1965.

[6] N. Edward Berg.
Electronic Composition.
Graphic Arts Technical Foundation, Pittsburgh, 1978.

[7] John R. Biggs.
Basic Typography.
Faber and Faber, London, 1968.

[8] Sir Cyril Burt.
A Psychological Study of Typography.
Cambridge University Press, 1959.

[9] Gordon V. Carey.
Cambridge Authors' and Printers' Guides. Volume 3: *Making an Index (Third Edition).*
Cambridge University Press, 1963.

[10] B. Cohn.
The End is Just the Beginning: the life of U. A. Whitaker.
Carnegie-Mellon University Press, 1980.

[11] Robert L. Collison.
Indexes and Indexing: Guide to the Indexing of Books, and Collections of Books Periodicals, Music, Gramophone Records, Films and other Material, with a Reference Section and Suggestions for Further Reading.
John de Graff, Inc., New York, 1959.

[12] G. F. Coulouris, I. Durham, J. R. Hutchinson, M. H. Patel, T. Reeves, and D. G. Winderbank.
The Design and Implementation of an Interactive Document Editor.
Software—Practice and Experience 6:271-279, June, 1976.

[13] Charles Goldfarb.
Document Composition Facility: Generalized Markup Language (GML) User's Guide.
Technical Report SH20-9160-0, IBM General Products Division, 1978.

[14] M. Gorlick, V. Manis, T. Rushworth, P. van den Bosch, and T. Venema.
Texture User's Manual
Department of Computer Science, University of British Columbia, Vancouver, B.C. V6T1W5, 1975.

[15] Edward M. Gottschall.
Communications Typographics.
IEEE Transactions on Professional Communication PC21(1):18-23, March, 1978.

[16] *U.S. Government Printing Office Style Manual*
Revised edition, Washington, D.C., 1973.

[17] *Word Division Supplement to the Government Printing Office Style Manual*
Seventh edition, Government Printing Office, Washington, D.C., 1976.

[18] John Guttag and J. J. Horning.
Formal Specification as a Design Tool.
In *Conference Record.* Seventh Annual ACM Symposium on Principles of Programming Languages, ACM/SIGPLAN-SIGACT, January, 1980.

[19] Allen V. Hershey.
A computer system for scientific typography.
Computer Graphics and Image Processing 1:373-385, 1972.

[20] *IBM SCRIPT/370 Version 3 User's Guide, manual SH20-1857-0*
IBM Data Processing Division, White Plains, NY, 1976.

[21] Evan L. Ivie.
The Programmer's Workbench - A Machine for Software Development.
Communications of the ACM 20(10), October, 1977.

[22] Paul E. Justus.
There is more to typesetting than setting type.
IEEEPC PC-15(3):13-16, March, 1972.

[23] Brian W. Kernighan and Lorinda L. Cherry.
A System for Typesetting Mathematics.
Communications of the ACM 18(3):182-193, March, 1975.

[24] Donald E. Knuth.
TEX: A System for Technical Text.
Technical Report AIM-217, Stanford University, November, 1978.
Republished by Digital Press as Chapter 2 of *TEX and METAFONT, new directions in typesetting*, 1979.

[25] Butler Lampson.
Bravo Manual
Xerox Corporation, Palo Alto, CA, 1978.

[26] M. E. Lesk.
UNIX Programmer's Manual, p. REFER(1)
Bell Laboratories, Murray Hill, NJ, 1979.

[27] M. V. Mathews and Joan. E. Miller.
Computer Editing, Typesetting, and Image Generation.
In *Proceedings of the Fall Joint Computer Conference, Vol. 27*, pages 389-398. AFIPS, 1965.

[28] John McCarthy.
LISP 1.5 Programmer's Manual.
Technical Report, MIT Computation Center and Research Laboratory of Electronics, Cambridge, MA, 1962.

[29] Douglas C. McMurtrie.
The Book: the Story of Printing and Bookmaking (Seventh Edition).
Oxford University Press, London, 1943.

[30] *The Monotype Machine Book of Information*
Monotype Corporation, Leeds, England, 1946.

[31] Stanley Morison.
Cambridge Authors' and Printers' Guides. Volume 1: *First Principles of Typography (Second Edition).*
Cambridge University Press, 1967.

[32] Allen Newell, Fred M. Tonge, Edward A. Feigenbaum, Bert F. Green Jr, and George H. Mealy.
Information Processing Language-V Manual.
Prentice-Hall Inc., Englewood Cliffs, N. J., 1964.

[33] William M. Newman.
Page Makeup and Editing.
In James Foley (editor), *Introduction to Raster Graphics.* Sixth Annual Conference on Computer Graphics and Interactive Techniques, ACM/SIGGRAPH, May, 1979.

[34] J. F. Ossanna.
TROFF User's Manual.
Computing Science Technical Report 54, Bell Laboratories, 1977.

[35] John Pierson.
Computer Composition Using PAGE-1.
Wiley-Interscience, 1972.

[36] Jess Stein (editor).
The Random House Dictionary of the English Language.
Random House, New York, 1970.

[37] Brian K. Reid and Janet H. Walker.
Scribe User's Manual, Third Edition
CMU Computer Science Department, 1980.

[38] Robert P. Rich.
Information Handling.
Methodik der Information in der Medizin IV(4):159-163, December, 1965.

[39] M. Satyanarayana.
Multiprocessors: A Comparative Study.
Prentice-Hall, 1980.

[40] Kern E. Sibbald.
DPS User's Guide.
Technical Report CN-16.0, University of Maryland, April, 1976.

[41] S. H. Steinberg.
Five Hundred Years of Printing.
Penguin Books, Harmondsworth, England, 1974.

[42] Will Strunk and E. B. White.
The Elements of Style, Second Edition.
Macmillian, 1972.

[43] Lawrence Tesler.
PUB: The Document Compiler.
Technical Report ON-70, Stanford University Artifical Intelligence Project, September, 1972.

[44] Arthur T. Turnbull and Russell N. Baird.
The Graphics of Communication (Third Edition).
Holt, Rinehart, and Winston, New York, 1975.

[45] Daniel Berkeley Updike.
Printing Types: their History, Forms, and Use (A Study in Survivals).
Harvard University Press, 1937.

[46] Henry B. Wheatley, F.S.A.
How to Make an Index.
Elliot Stock, London, 1902.

[47] Hugh Williamson.
Methods of Book Design (Second Edition).
Oxford University Press, London, 1966.

[48] N. E. Wiseman, C. I. O. Campbell, and J. Harradine.
On making graphic arts quality output by computer.
The Computer Journal 21(1):2-6, February, 1978.

[49] Wm. A. Wulf, D. B. Russell, and A. N. Habermann.
BLISS: a Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

[50] Wm. A. Wulf, M. Shaw, P. Hilfinger, and L. Flon.
Fundamental Structures of Computer Science.
Addison-Wesley, 1980.

[51] Wm. A. Wulf, Roy Levin, and Sam Harbison.
Hydra/C.mmp: An Experimental Computer System.
McGraw-Hill, 1980.

Acknowledgments

I would like to thank my advisor, Bob Sproull, and my reading committee, Brian Kernighan, Mary Shaw, and Bill Wulf, for all of the help they gave me in making this thesis be reasonably coherent. David Lamb, Chris van Wyk, and Don Knuth also provided valuable critical feedback on intermediate drafts.

Scribe was a big project, and its design, development, debugging, and documentation have involved a lot of people. It's impossible to thank them all, but I'd nevertheless like to try. I am certain that I have forgotten to include people who have made contributions as significant as these.

Thanks to Bill Wulf, David Lamb, Mary Shaw, and Paul Hilfinger for solid design principles and getting me started in the right direction. They deserve full credit for the original ideas behind Scribe and the language design principles that guided it. Thanks to Larry Tesler and Les Earnest for inventing PUB, without which I never would have had the design tools for Scribe. To Doug Clark and Roy Levin, whose unflinching insistence on quality raised my consciousness about automated document formatting. To Gideon Yuval, for offering mad suggestions often enough that I stopped thinking they were mad.

In the implementation of Scribe, two years of programming in Bliss, my feeble programming skills were supplemented by the awesome wizardry of Craig Everhart more times than I can count. The job of exporting Scribe to other laboratories was made easier because of the assistance of Dwight Cass, Benjamin Hyde, Janet Walker, Chris Ryland, Chuck Weinstock, Wayne Gramlich, and Gary Baczkowski.

Jan Walker consistently played the role of The User. She helped me build more realistic models of how users perceived the Scribe system, she helped me understand how the system could be made more conceptually uniform in order to assist those users. She wrote half of the Second Edition of the manual and most of the Third, and also makes incredibly good Chinese food.

Dan Lynch and the SRI Computer Resources group sponsored the original development of the GSI photocomposer interface, and Tim Basinski generously made a photocomposer available to me at CMU for followup development. James Adams and Lawrence Butcher helped immeasurably in coping with the cantankerous photocomposer, and Jim Gasbarro designed and built the clever interface that connects it to our computer.

Many kind people helped with the debugging. Ivor Durham was Scribe's first user, and without his legendary patience the debugging effort might not have succeeded at all. The entire Computer Science department at CMU has suffered through two years of having Scribe constantly changing out from under them, with new bugs replacing the old. Special thanks to those people who were unusually helpful in pinpointing problems for me: Bruce Leverett, Philip Lehman, Paul Hilfinger, James Gosling, David Lamb, Chuck Weinstock, Les Lamport, Joe Newcomer, Lee Cooprider, Bill Brantley, Bob Schwanke, Walter Tichy, and John Nestor. Bruce Leverett and Kevin Brown put many hours into the creation and standardization of bibliography formats.

Finally, very special thanks to my wife, Loretta Guarino Reid, whose skills as a systems designer, debugger, proofreader, cook, and counsellor have helped me in every aspect of the Scribe project.

Glossary

Collected definitions of terms that are peculiar to the typography or printing field, and that are used in the text.

Ascender	The parts of lowercase letters that protrude above the basic body height in the letters *b, d, f, h, k, l,* and *t.*
Descender	The parts of lowercase letters that protrude below the baseline in the letters *g, p, q,*, and *y*, and in certain fonts for some capital letters as well.
Diacritical	An accent mark.
Face	Meaning varies. As used in this thesis, *face* is the attribute that determines the style of letter to be used within a particular font. Typical values of *face* include *italic, bold, small capital. Cf. font.*
Filling	Placing as many words on one line as will fit, in an attempt to make line lengths approximately even; *cf.* "justification".
Font	Meaning varies. As used in this thesis, a *font* is a family of alphabets whose letters are stylistically similar. Within a font, various faces can be selected and various sizes of letters can be made. This thesis is set in the Times Roman font, in the 12-point size.
Justification	The expansion or contraction of spaces within a filled line so that the line is exactly the prescribed length, in order that the right margin will be even.
Kerns	Parts of a type slug for italic or slanted letters that protrude past the edges of the type slug. See the diagram on page 21.
Kerning	Fine adjusting of the horizontal spacing between letters in a word so as to take into account the nuances of their geometry.

Leader — A row of some punctuation character, usually periods or dashes, to fill white space in a table.

Ligature — A single letter that takes the place of a group of two or more, such as *fl* for fl instead of the *fl* that appears when the individual letters are simply abutted.

Markup — Instructions to a typesetter written on a typescript by a copy editor. In discussing Scribe, the *markup* is used to describe all of the uses of the "@" character to pass special information to the compiler.

Mechanical spacing — Horizontal spacing of letters within a word that is identical to the spacing that would be had when metal type slugs are used, even if there are no physical restrictions on letter spacing.

Optical spacing — Horizontal spacing of letters within a word such that the space between two letters depends on their shape. Optical spacing is achieved by kerning from mechanical spacing.

Orphan — The first line of a paragraph placed by itself at the bottom of a page. *Cf. Widow.*

Pagination — Division of running textual material into pages, taking simultaneously into account the placement of footnotes, figures, headings, and other non-textual material.

Point — A unit of distance approximately equal to $1/72^{nd}$ of an inch.

Running heads — The portion of a page that contains the page numbers and other information. The running heads in this thesis include the name of the chapter.

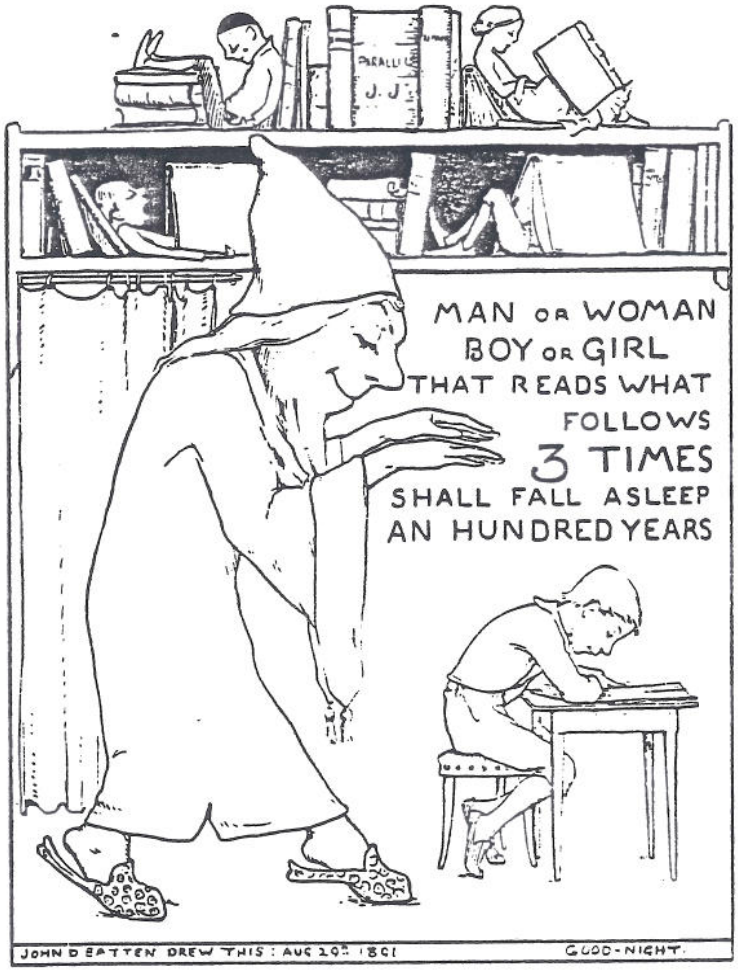
Serif — Serifs are the difference between f and f. They are the small horizontal and vertical lines that characterize Roman type faces.

Slug — See *type slug.*

Type slug — A rectangular piece of metal used in classical hand typesetting. A drawing of a type slug appears on page 21.

Widow — The last or first line of a paragraph left by itself at the top or bottom of a page. Also called *widow line* or *widowed line.* Sometimes the word *orphan* is used to describe a last-line-of-page

widow.


MAN OR WOMAN
BOY OR GIRL
THAT READS WHAT
FOLLOWS
3 TIMES
SHALL FALL ASLEEP
AN HUNDRED YEARS
JOHN D BATTEN DREW THIS : AUG 20TH 1801
GOOD-NIGHT.

Appendix A
The State Parameters

Chapter 5 discussed the environment mechanism, and explained the difference between static and dynamic state parameters in the context of the environment mechanism. This appendix lists those parameters, with a brief explanation of their semantics. The type names used in this chapter are explained in Section 5.2 on page 54.

A.1 Dynamic State Parameters

Dynamic parameters are those that may change during a run of the compiler. Static parameters are fixed during compiler initialization, or remain constant for the entire compilation.

Dynamic parameters are classified into two groups, *inheriting* parameters and *non-inheriting* parameters. The inheriting parameters obey the binding stack protocol discussed in the previous section. The non-inheriting parameters do not: if an environment entry does not specify a value for a non-inheriting parameter, then a default value is used rather than an inherited value.

For purposes of this explanation, the parameters are also classified as either *format control parameters*, *manuscript language interpretation parameters*, and *bookkeeping parameters*. These classes do not have significance in the actual implementation of the compiler.

Remember that the type of the value specified for a state parameter need not be the same as the type of the parameter; wherever meaningful, the necessary type coercion will be made when the environment specifying that value is actually entered. This means, for example, that a *font-relative distance* can be specified as the value for a parameter whose type is *horizontal distance*. The environment-entry processing will perform the necessary multiplication.

Format Control Parameters

1. **Font family**: the identity of the current font family. Inheriting. Type: *symbol.* Typical values: "Heading Font", "Body Font", etc. Font family names are declared in font definition files in the database.

2. **Face code**: the face code within the current font family. Inheriting. Type: *character.* Typical values: "R", "I", "B". Face codes select a particular font from a font family.

3. **Font size**: Inheriting. Type: *Absolute distance.* The size of the letters to be generated in the font selected by the previous two parameters. The precise meaning of the font size with respect to the geometry of letters in the font depends on the font designer's measurements. It is usually the height of a box that is guaranteed to be tall enough to contain any letter in the font while its baseline is at a fixed point in the box. The box height is therefore the sum of the maximum above-baseline height of characters in the font and the maximum below-baseline height of characters in the font.

4. **Spacing**: the vertical spacing between ordinary text lines, measured from the baseline of one line to the baseline of the next. Inheriting. Type: *vertical distance.*

5. **Paragraph spread**: the additional spacing, over and above **spacing**, that is placed between text paragraphs. Inheriting. Type: *vertical distance.*

6. **Left margin**: the distance between the left edge of the paper and the left edge of the text lines. In justified text, the **left margin** applies to all lines but the first in a paragraph. Inheriting. Type: *horizontal distance.*

7. **Indention**: the horizontal distance between the left margin and the left edge of the first line of a paragraph. An ordinary indented paragraph has a positive value for **indention**; a block paragraph has a zero value. Outdented paragraphs have negative indentions. Inheriting. Type: *horizontal distance.*

8. **Right margin**: the distance between the right edge of the paper and the right edge of justified text lines. Inheriting. Type: *horizontal distance.*

9. **Top margin**: the distance between the top edge of the paper and the top edge of the first line of actual text on a normal page. Inheriting. Type: *vertical distance.*

10. **Bottom margin**: the distance between the bottom edge of the paper and the bottom edge of the last line of actual text on a normal page. Inheriting. Type: *vertical distance.*

11. **Fill mode**: Inheriting. Type: *Boolean.* True if the compiler is to "fill" text lines, i.e. to put as many words on each as will fit. False otherwise.

12. **Line disposition**: Inheriting. Type: enumerated from {*flushleft, flushright, centered, justified*}. After the line has been closed, and possibly filled, what full-line processing is done with it as it is placed onto the page.

13. **Transformation**: Inheriting. Type: enumerated from {*none, capitalized, initial capitalized*}. Dictates capitalization transformation performed on text before width computation.

14. **Sink margin**: Inheriting. Type: *vertical distance.* Specifies a distance from the top edge of the paper such that the first line of this environment is permitted to be no closer than **sink margin** to the top edge of the paper. If the position on the page is already farther from the top edge of the paper than **sink margin**, then it has no effect.

15. **Fixed location**: Type *vertical distance.* Specifies a distance from the top edge of the paper to which the first line of this environment is forced, regardless of context. Used for page headings and other running material.

16. **Script displacement**: Type *vertical distance.* The amount by which the text in this environment is displaced from the current baseline, for superscripting or subscripting. Positive values generate superscripts, and negative values generate subscripts.

17. **Underlining**: controls underlining in the text. Inheriting. Type: enumerated from {*none, all, nonblank, alphanumeric*}.

18. **Overbar**: controls generation of overbars on the text. Type same as **underlining.**

19. **Widest blank**: the largest amount to which a blank can be expanded by the justifier before the formatter will try to hyphenate. Inheriting. Type: *horizontal distance.*

20. **Narrowest blank**: the smallest amount to which a blank can be compressed by the justifier before the formatter will try to hyphenate. Inheriting. Type: *horizontal distance.*

21. **Hyphenation**: Inheriting. Type: *Boolean.* Set *true* if hyphenation is permitted in this environment, else false.

22. **Columns**: the number of columns into which the text is to be set. Inheriting. Type: *integer.*

23. **Column margin**: the horizontal spacing between columns. Inheriting. Type: *horizontal distance.*

24. **Running heads**: permit running headers. Inheriting. Type: *Boolean.* Set *true* if any new pages opened during this environment are to have running headers.

25. **Resume paragraph on exit**: Type: enumerated from {*No, Required, Permitted*}. Non-inheriting; default *Permitted.* When an environment is exited back to the outer containing environment, this parameter controls whether or not the outer environment is resumed in the same paragraph or whether a new paragraph is begin.

26. **Line break**: Controls line break upon entrance and exit to and from the environment. Type: enumerated from {*break on entry, do not break on entry*} cross {*break on exit, do not break on exit*}. Non-inheriting; default: *do not break.*

27. **Page break**: Controls page break upon entrance to and exit from the environment. Type: enumerated from {*page break before entry, break until even page before entry, break until odd page before entry, do not break on entry*} cross {*break on exit, do not break on exit*}. Non-inheriting; default: [*do not break on entry, do not break on exit*].

28. **Block disposition**: Controls the disposition of the entire environment's text. Type: enumerated from {*none, group, float, footnote*}. Non-inheriting; default: *none.* If the value of this parameter is other than *none*, then its text will be clustered and handled as a unit. The various values allow for figure floating, equation clustering, and note placement.

29. **Float disposition**: Type: enumerated from {*none, float down, float up, float defer, float whole page, float to line end*}. If the value of **block**

disposition is *float*, then the value of **float disposition** controls the floating process. Non-inheriting; default: *none*.

30. **Minimum above spacing**: Type: vertical distance. Non-inheriting; default: 0. Specifies that the first line of text of this environment can be placed no closer to the bottom of the last line of the previous environment than the indicated value.

31. **Minimum below spacing**: Type: vertical distance. Non-inheriting; default: 0. Specifies that the last line of text of this environment can be placed no closer to the top of the first line of the following environment than the indicated value.

32. **Line push**: Type: boolean. Inheriting. If true, then line spacing is increased to accomodate oversize characters. If false, then line spacing is left constant regardless of the characters on that line. See section 8.6.3 for a discussion of this effect.

33. **Page need**: Type: vertical distance. Non-inheriting; default: 0. Specifies that the first line of this environment can be placed no closer to the bottom of the paper than the sum of the indicated value and the page bottom margin.

Manuscript File Interpretation Parameters

34. **Carriage-return action**: action to be performed on a carriage return/line feed pair in the manuscript file. Inheriting. Type: enumerated from {*paragraph-break, spaces, ignored*}. The carriage return is treated as a paragraph break, a word break, or ignored completely depending on the value of this parameter.

35. **Blank line action**: action to be performed on a blank line in the manuscript file (two or more consecutive carriage returns). Inheriting. Type: enumerated from {*paragraph break, word break, keep line, keep and hinge*}. The value *word break* means to treat a blank line in the same way that multiple blank spaces are handled, which is controlled by the **Space action** parameter, below. The *paragraph break* value means to cause a paragraph break at a blank line (this is the usual case). The *keep line* value means to retain an actual blank line in the produced document, after performing a paragraph break. The *keep and hinge* value means to permit a grouped environment to hinge at this blank line. Grouping is one of the **block disposition** options; see item 28.

36. **Space action**: how to treat blank spaces in the manuscript file. Inheriting. Type: enumerated from {*retained, compressed, normalized, discarded, retained significant*}. A retained significant space is treated as a letter, and can never cause a word break.

37. **Leading space action**: like space action, but applies to leading spaces on manuscript lines.

38. **Overlong line action**: action to be performed when a line in the manuscript file is too long for the margins, and the formatting parameters do not specify line filling. Inheriting. Type: enumerated from {*chop, wrap, keep*}. The line is either truncated at the right margin, allowed to extend past the right margin, or wrapped to a following output line.

39. **Newpage disposition**: disposition of "new page" characters in the manuscript file. Inheriting. Type: enumerated from {*ignored, text, start-new-page*}.

Bookkeeping parameters

40. **Attached counter**: Inheriting. Type: *symbol*. If non-null, the symbol table name of a counter defined for numbering objects in this environment.

41. **Number location**: Selection of where to put a generated number for generated objects in this environment. Inheriting. Type: enumerated from {*beginning, end*} cross {*flush left, flush right*}.

42. **Process text after entry**: A string of manuscript text to be processed immediately on entry to the environment. Type: *string*. Non-inheriting; default: *null*.

43. **Process text before exit**: A string of manuscript text to be processed immediately before exit from the environment. Type: *string*. Non-inheriting; default: *null*.

44. **Process text after exit**: A string of manuscript text to be processed immediately after exit from the environment. Type: *string*. Non-inheriting; default: *null*.

45. **Tab export**: Type: *Boolean*. Non-inheriting; default: *False*. If *true*, then

tab stops set within this environment are to be exported to the outer containing environment.

A.2 Static State Parameters

Static state parameters are fixed during compiler initialization, and do not change during a compilation. Their values are read in from various database files.

Device Parameters

1. **Paper width**: Type: *horizontal distance.* The physical width of the paper in the printing device.

2. **Paper height**: Type: *vertical distance.* The physical height of each page of paper in the printing device.

3. **Horizontal width increment**: Type: *rational number.* The number of horizontal width units in a centimeter, expressed as a quotient of two integers.

4. **Vertical width increment**: Type: *rational number.* The number of vertical width units in a centimeter, expressed as a quotient of two integers.

5. **Can backspace**: Type: *Boolean.* True if the printing device is capable of executing a backspace command; false otherwise.

6. **Bare carriage return**: Type: *Boolean.* True if the printing device is capable of executing a carriage return without a corresponding line feed, to move to the leftmost printing position on the page.

7. **Bare line feed**: Type *Boolean.* True if the printing device is capable of executing a bare line feed, without corresponding carriage return, to move to the same printing position on the next line.

8. **Has fonts**: Type: *Boolean.* If the printing device is capable of changing font, then true, else false.

9. **Has lens**: Type: *Boolean.* If the printing device can change font size or scale without changing font, then true, else false.

10. **Overstrike**: Type: *Boolean.* If the printing device is able to overstrike characters then true, else false.

11. **Paged**: Type: *Boolean.* If the printing device operates on discrete pages, then true, else false.

12. **Underline**: Type: *Boolean.* If the printing device is capable of underlining, then true, else false.

Static Format Parameters

13. **Double sided printing**: Type: *Boolean.* If the document is being prepared for double sided reproduction, then true, else false.

14. **Binding margin**: Type: *horizontal distance.* When a document is printed doublesided and bound, a certain amount of the inside margin is used up by the binding. The value of the **binding margin** parameter should be equal to the amount that is covered by the binding. It will be added to the left margin on odd pages and to the right margin on even pages.

15. **Font family**: Type: *Symbol.* The name of the font family to be used for typesetting this document. A font family is a selection of fonts chosen by a designer to look harmonious when used together. It provides the bindings for the *heading font* and *title font* names used in the dynamic **font** parameter.

16. **Note disposition**: Type: enumerated from {*inline, end of chapter, end of document, bottom of page*}. Disposition of footnotes in the text.

17. **Widow disposition**: Type: enumerated from {*ignored, forced, give warning*}. How the compiler is to treat widow lines.

Static Bookkeeping Parameters

18. **Generic Device**: Type: *String.* Used as a common retrieval key for database entries shared by several device types.

19. **Page numbering**: Type: *symbol.* A pointer to a counter to be used for numbering pages.

20. **Note numbering**: Type: *symbol.* A pointer to a counter to be used for numbering notes.

21. **Bibliography type**: Type: *symbol.* The name of the database file to be used as the format definition for the bibliography and citations in this document.

Appendix B
Compiler Implementation Details

B.1 The Generic Operating System Interface

The Scribe compiler was coded to deal with a generic operating system; various specific operating systems are used by means of an operating system interface. This Scribe Generic Operating System is a minimalist system; it is the simplest possible OS that was reasonably able to support the compiler without it seeming alien to users experienced in the behavior of the host operating system. It offers no surprising or innovative services, and is worth recording because of its simplicity.

The Scribe GOS supports files, terminals, address space management, and environment inquiry. It has no notion of processes, synchronization, interrupts, or communication. All I/O is synchronous.

A *text file* is a stream of bytes. It is read sequentially, one byte at a time. Every file has a name and a creation date/time. Its text can optionally be divided into named zones, pages, lines, or records. These names are used in error messages generated by the compiler, for the purpose of tying errors to particular locations in the file. A *binary file* is a vector of bytes, which are read or written by position within the file. A binary file can be opened for input or output, but not both simultaneously. A *terminal* is a text file that can be opened for input and output simultaneously. When non-printing characters are written to a terminal, the operating system either honors them as control characters or else translates them into some appropriate sequence of printing characters.

The GOS manages the computer's address space. The compiler is not permitted to reference a memory address that has not been allocated to it by the GOS. The compiler can request blocks of memory from the operating system and also can return them if it so desires. The overhead of requesting space from the operating system is large enough that the compiler is expected to retrieve large chunks of address space and subdivide them itself.

The client program can request environment information of various limited kinds from the operating system, including the date and time of various events, the name of the current user, and so forth.

B.1.1 The File System

An open file is represented by an Open File Descriptor Record, or OFDR:

```
type OFDR = record
 client's_name: string;
 true_name: string;
 short_name: string;
 open_type: {in,out};
 location_name: string
end;
```

When a file is opened, the GOS is passed a string that contains the client's name for the file. The GOS locates the file, creates an OFDR, and returns a pointer to it. One of the fields of the OFDR is the "true name" of the file. The true name may differ arbitrarily from the client's name—it might just be a sequence number—but the expected property of the true name is that it be a copy of the client's name, expanded by the addition of supplementary text.

The remainder of this section details the file-related services provided by the Generic Operating System.

B.1.1.1 Open for Text Input

The function Open_For_Text_Input(Client's_name: string) returns a pointer to an OFDR. The GOS locates the requested file, opens it for sequential text input, creates an OFDR record, and returns a pointer to that record. The GOS goes to considerable effort to ensure that some file will be found. If the file named by the client cannot be found, the GOS engages in a dialog with the user at the terminal to find a replacement file name, and uses that name instead. If the user refuses to provide a substitute file name, or if there is no terminal available, then an OFDR to a zero-length nameless file will be returned.

B.1.1.2 Open For Text Output

The function Open_For_Text_Output(Client's_name: string) returns a pointer to an OFDR. The GOS creates a new file with the requested name, opens it for sequential output, creates an OFDR record, and returns a pointer to that record. If the GOS is unable to create or open such a file, then it engages with a dialog with the user at the terminal, as above. If there is already a file with the requested name, the GOS is permitted to delete it at this time, but not required to.

B.1.1.3 Check For Text Input

The function Check_For_Text_Input(Client's_name: string) returns a Boolean value. It checks to see whether or not an "open for input" request would succeed if issued. If a call to Open_For_Text_Input of Client's_name would succeed without needing to interrogate the user, then Check_For_Text_Input returns True. In any other circumstance, it returns False.

B.1.1.4 Check For Text Output

The function Check_For_Text_Output(Client's_name: string) returns a Boolean value. The function checks to see whether or not an "open for output" request would succeed if issued. If a call to Open_For_Text_Output of Client's_name would succeed without needing to interrogate the user, then Check_For_Text_Output returns True. In any other circumstances, it returns False.

B.1.1.5 Open Unique Text Output

The function Open_Unique_Text_Output takes no arguments, and returns a pointer to an OFDR. It is identical to Open_For_Text_Output, save that it invents a file name, and returns the name so invented in the Client's_name field of the returned OFDR. The invented file name is guaranteed not to duplicate or interfere with any existing file.

B.1.1.6 Close File

The function Close_File(File_record: **pointer to** OFDR) closes the file whose OFDR is pointed to by File_Record, and then destroys that record. If the file was open for input, there are no side effects. If the file was open for output, the Close_File operation must perform any housekeeping operations related to deleting or inactivating old versions of the file.

B.1.1.7 Close and Delete

The function Close_And_Delete(File_Record: **pointer to** OFDR) closes an open file and deletes or suppresses it. No value is returned. If the indicated file is open for input, then it is closed as by Close_File, above, then deleted. If the indicated file is open for output, then it is closed as by Close_File, except that (a) the indicated file is not created, (b) no housekeeping or deleting of old versions of the file is performed, and (c) any deleting or modification of files that was performed by Open_For_Text_Output is undone.

B.1.1.8 Rewind

The function Rewind(File_Record: pointer to OFDR) returns a pointer to an OFDR. It accepts as input a file that is open for input or output, and returns an OFDR to the same file open for input, ready to read the first character of the file.

B.1.1.9 Read Text Character

The function Read_Text_Character(File_Record: pointer to OFDR) returns a value of type File_Character. It reads one character from a file and returns it as the value of the function.

B.1.1.10 Write Text Character

The function Write_Text_Character(File_Record: OFDR, Char:Character) writes the designated character to the designated file, which must be open for output.

B.1.2 Address Space Management

The GOS provides a heap protocol for allocating and deallocating blocks of memory. A simple quickfit algorithm is used to manage space. When the GOS runs out of space, it negotiates with the actual host operating system for more memory. The released memory is periodically compacted into larger blocks during the release process. The algorithms used for free-list management and allocation strategy are not specified, and the GOS is free to manage them however it chooses.

B.1.3 Environment Inquiry

The Scribe compiler needs very little information about its environment. The GOS provides these service routines.

B.1.3.1 Determine Date

The function Determine_Date:integer returns the number of whole days that have elapsed since Sunday, March 0, 1948, in local time, as of the start of execution of the program. All calls to Determine_Date made during the same compiler run will return the same value.