6-1987

The Evolution of Text Formatting Languages

Dirk Herr-Hoyman

Western Michigan University

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation
https://scholarworks.wmich.edu/masters_theses/1211

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master’s Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
THE EVOLUTION OF TEXT FORMATTING LANGUAGES

by

Dirk Herr-Hoyman

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
June 1987
THE EVOLUTION OF TEXT FORMATTING LANGUAGES

Dirk Herr-Hoyman, M.S.
Western Michigan University, 1987

Text, as seen in books and magazines, can take on three forms: string, graphic (two-dimensional), and image (digitized pictures). Text formatting processes text into a representation suitable for printing. Since a printer is really a computer, this representation is machine code for the printer. ASCII is one such code.

Six historically significant text formatting languages are surveyed: Runoff, Troff, TeX, Bravo, Scribe, and Postscript. The emphasis is on the text types available and the code generated. The main evolutionary forces are the changes in printers. Comparisons are made with programming languages.

Each of the six languages has ASCII as its basis, even though ASCII was originally meant for string text processing. With the electronic text fast approaching, the reliance upon ASCII will need to change in the future.
ACKNOWLEDGEMENTS

I would like to thank my thesis committee: Dr. Iyad Natour, Dr. Robert Trenary, and Dr. Ben Pinkowski for their efforts in both advising and making helpful comments in the preparation of this thesis.

Dirk Herr-Hoyman
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University
Microfilms
International
300 N. Zeeb Road
Ann Arbor, MI 48106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The evolution of text formatting languages

Herr-Hoymon, Dirk Alan, M.S.
Western Michigan University, 1987
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS ........................................................................................................ ii</td>
</tr>
<tr>
<td>LIST OF TABLES ........................................................................................................... v</td>
</tr>
<tr>
<td>LIST OF FIGURES ......................................................................................................... vi</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1. INTRODUCTION ........................................................................................................ 1</td>
</tr>
<tr>
<td>2. CONCEPTS ............................................................................................................... 3</td>
</tr>
<tr>
<td>2.1 Text .................................................................................................................. 3</td>
</tr>
<tr>
<td>2.2 Formatting ......................................................................................................... 7</td>
</tr>
<tr>
<td>2.2.1 Abstract to Concrete Object ................................................................... 7</td>
</tr>
<tr>
<td>2.2.2 Printer as a Computer ............................................................................ 9</td>
</tr>
<tr>
<td>2.3 Operations ......................................................................................................... 10</td>
</tr>
<tr>
<td>2.3.1 String ......................................................................................................... 10</td>
</tr>
<tr>
<td>2.3.2 Graphics ..................................................................................................... 12</td>
</tr>
<tr>
<td>2.3.3 Image .......................................................................................................... 13</td>
</tr>
<tr>
<td>2.4 Markup ............................................................................................................ 14</td>
</tr>
<tr>
<td>3. THEMES ............................................................................................................... 15</td>
</tr>
<tr>
<td>3.1 Co-Evolution of Text Formatting and Printers ........................................ 15</td>
</tr>
<tr>
<td>3.2 Parallel Development of Programming and Text Formatting Languages .......... 16</td>
</tr>
<tr>
<td>3.3 ASCII as the Basis of Text Formatting Languages .................................... 16</td>
</tr>
<tr>
<td>4. SURVEY ................................................................................................................. 17</td>
</tr>
<tr>
<td>4.1 Runoff ............................................................................................................. 18</td>
</tr>
<tr>
<td>4.2 Troff .............................................................................................................. 20</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table of Contents—Continued

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 TeX</td>
<td>24</td>
</tr>
<tr>
<td>4.4 Scribe</td>
<td>28</td>
</tr>
<tr>
<td>4.5 Bravo</td>
<td>30</td>
</tr>
<tr>
<td>4.6 Postscript</td>
<td>32</td>
</tr>
<tr>
<td>4.7 Summary</td>
<td>36</td>
</tr>
<tr>
<td>5. ANALYSIS</td>
<td>38</td>
</tr>
<tr>
<td>5.1 User Interface</td>
<td>38</td>
</tr>
<tr>
<td>5.2 Programming Language</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Printer Architecture</td>
<td>43</td>
</tr>
<tr>
<td>6. FUTURE</td>
<td>46</td>
</tr>
<tr>
<td>6.1 Electronic Text</td>
<td>46</td>
</tr>
<tr>
<td>6.2 Some Current Research</td>
<td>47</td>
</tr>
<tr>
<td>6.2.1 Standard Generic Markup</td>
<td>47</td>
</tr>
<tr>
<td>6.2.2 Music Printing</td>
<td>48</td>
</tr>
<tr>
<td>6.2.3 Abstract Object Editor</td>
<td>48</td>
</tr>
<tr>
<td>6.3 Is ASCII Good Enough?</td>
<td>50</td>
</tr>
</tbody>
</table>

APPENDIX

Seven Bit ASCII | 52

BIBLIOGRAPHY | 54
LIST OF TABLES

1. Example of a Table .................................. 4
2. Comparison of Text Types .............................. 6
3. Example Troff Commands ............................... 21
4. Example Troff Registers ............................... 22
5. Summary of TFL Capabilities ........................... 37
6. Generations of TFLs ................................. 42
7. Printer Architecture of TFLs .......................... 45
LIST OF FIGURES

1. Hierarchical Text Structure .................................. 8
2. Example of Runoff ............................................. 20
3. Troff at the Heart of UNIX Text Formatting .................. 24
4. TeX Line Breaking ............................................... 27
5. Bezier Cubic Representation of a Curved Line ............... 35
6. Classification of Surveyed TFLs .............................. 40
CHAPTER 1

INTRODUCTION

Language is our mode of communication. Certainly, language is and has been an important part of our human civilization. In particular, written language was a vital force in the growth of our western civilization. Consider how the introduction of movable type in the early 1500's fueled the Renaissance. The ability to produce and reproduce written language easily by mechanical means is undoubtedly a necessary condition for the world today.

Since the introduction of movable type, the printing profession has evolved as both a trade and a craft. Many technical developments since that time have increased the printer's productivity. In the 1960's, electronic digital computers entered the printing arena in the form of computerized typesetters. But, a good printer is also an artisan. Knowing what style and format to use for a particular document remains something of an art. Consisting of a large body of knowledge built up by centuries of expertise, the art of formatting text is not easily encapsulated mechanically.

At the same time that the printing industry was integrating computers into its environment, developments were made to utilize the computer as a document generator. Judging by the piles of computer generated reports and the existence of a word processor at every desk, one could reasonably surmise that producing written material is
quite important for the typical computer user.

More recently, the gap between document production systems and printers has been bridged, allowing the end user to produce typeset quality documents. In the popular press such systems have been called desk top or computer aided publishing.

The purpose of this thesis is to explore the heart of computer document production systems, text formatting, and the evolution of a particular type of computer language, text formatting languages.

The remainder of this thesis is organized as follows:

1. Concepts: basic ideas about text formatting
2. Themes: underlying ideas in this thesis
3. Survey: a look at some text formatting languages
4. Analysis: viewpoints on the surveyed languages
5. Future: some research into text formatting
CHAPTER 2

CONCEPTS

This section develops a number of concepts which are important for text formatting.

The first two concepts are text and formatting, which are central concepts for this paper.

2.1 Text

Generally, text is thought of as written language, the kind of thing one finds in a book. For example, this sentence is text and so is this paragraph. With this simple form of text, the meaning is carried entirely by the sequence of the characters. Thus,

The rain in Spain falls mainly on the plain.

has the same meaning as

The rain in Spain

falls mainly on the plain.

Even though the layout on the page of the first sentence is different than the second, both have the same meaning. The meaning is given by the sequence of characters, assuming that it is read left to right and top to bottom. With this first type of text, which is known as a string, a written language such as English can be represented.
However, the contents of a book are typically more than just sentences of written language. Tables and diagrams also have a place in text. Such text is displayed in such a manner that the layout on the page has a meaning. For example, consider a table (see Table 1).

Table 1
Example of a Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Dishwashing Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Sunday</td>
</tr>
<tr>
<td>Tom</td>
<td>Monday</td>
</tr>
<tr>
<td>Julie</td>
<td>Tuesday</td>
</tr>
</tbody>
</table>

Meaning is associated not only to the sequence of characters "Bob," but also to their relative position, in the same row as "Sunday" and in the column "Name." Thus, the meaning for "Bob" is not just the character sequence, but also the two-dimensional position of "Bob" relative to the other parts of the table.

Other examples that show meaning given via special attributes of text are: a title that is underlined or italicized is understood to be a book or magazine, chapter headings are enlarged, subscripts and superscripts have special meaning in mathematical formulae. The point is that meaning can be ascribed to text that is outside the range of a "pure" written language.
To encompass this range of written material adequately, this paper will broaden the common meaning of text. The term text will refer to the type of material found in a book. While some (e.g., Kimura 1984) have used four categories of text: string, tables, math, graphics, others (Adobe Systems Inc. 1986, Barrett & Reistroffer 1987) find that only three are necessary:

1. String—a sequence of characters, strings have only a one dimensional constraint, their order

2. Graphics—any text whose elements have a two dimensional relationship to each other

3. Image—an image is a dot matrix (digitized) representation of a picture

The elements of graphics can be either strings or some special symbols, such as line segments. Thus, one can form graphics from other graphics. Graphics can also be formed entirely from text, making tables a graphic. Graphics represent a large category of text which have only recently been integrated into text processing.

Since both graphics and images are two-dimensional, what differentiates the two? It is the dot matrix representation that distinguishes images from graphics. This dot matrix format forces a lower level data representation which is more costly in terms of storage requirements than graphics (see Table 2). In essence, the reason for graphics' existence is to prevent this expensive representation and utilize some aspect of the semantics of the text entity to store it more compactly. While this may not be a problem for two-dimensional text such as a table, a more compact
representation for a line drawing can make an order of magnitude difference in storage costs (Adobe Systems Inc. 1986).

The view of text composed of string, graphic, and image types is supported by the Postscript language (Section 4.6) (Adobe Systems Inc. 1986), while the view of Kimura (Kimura 1986) is shared by TeX (Section 4.3) and Troff (Section 4.2).

It can be seen that these three text types form a hierarchy, with images on top, graphics next, and finally strings. This means that any text could be formed as an image, e.g., the dots of ink on a page produced by a dot matrix printer. Of course, computationally this is a nightmare. A 300 dpi (dots per inch) laser printer on an 8.5 x 11 piece of paper would have: 300*300*8.5*11 or about 3,000,000 dots. With a 9600 baud communications line to such a printer, it would take about 300 seconds to transfer a single page from a host.

Table 2 is a comparison of these three text types. In general, the type which is the easiest to use and has the least storage requirement is used.

Table 2
Comparison of Text Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Ease of Use</th>
<th>Storage Required</th>
<th>Information Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Highest</td>
<td>Lowest</td>
<td>Lowest</td>
</tr>
<tr>
<td>Graphic</td>
<td>Intermediate</td>
<td>Intermediate</td>
<td>Intermediate</td>
</tr>
<tr>
<td>Image</td>
<td>Lowest</td>
<td>Highest</td>
<td>Highest</td>
</tr>
</tbody>
</table>
2.2 Formatting

Two definitions of text formatting are considered here. Both have their place, although this paper will emphasize the latter one.

2.2.1 Abstract to Concrete Object

Commonly, text formatting is thought of as preparing an input file, and processing it by the text formatter to produce the document. More formally it is a mapping or transformation of the input string, which is a one-dimensional object, to a two-dimensional space, the page of text.

A more general treatment of these mappings is to look at a document as an abstract object (Futura, Scofield & Shaw 1982). Formatting is a mapping of these abstract objects to concrete objects, while editing is a mapping to other abstract objects.

Underlying the concept of an abstract object is a high level description of a document. Of course at the lowest level is characters; a document is composed of a sequence of characters. To see a higher level description, consider a document such as this paper. It can be described as: title page, table of contents, sections, subsections, and bibliography. Using this description, a tree structure is formed for the document (see Figure 1).
2.2.1.1 Views

As Section 2.1 indicated, there are many ways of formatting a document without changing its meaning; this is to say the abstract to concrete object mapping is one-to-many. Any particular mapping is a view of the document.

If structure of the document is maintained, but some physical attribute of the concrete object is changed, for example the margins, this would be called a physical view. The notion of a physical view encompasses more than just the layout, however. Stylistic elements can also be changed. Consider how citations are formatted. One could use citations in the style of (Knuth 1984a) if the citations are alphabetical by author, or (2) if the citations are numbered. The structure and meaning is maintained, only with a different reference style.
The other type of view is a logical view. Here a part of the structure of the document is suppressed. A logical view of this paper could be the section and subsection titles. Thus, a table of contents is a logical view of this document.

2.2.2 Printer as a Computer

An alternate view of formatting sees a printer as an automaton or a remote computer. A printer has a current state which is defined by the position of the print head. Actions of the printer are to move or print some character.

Any file that is printed is composed of code words to control the action of the printer. One such code is ASCII (American Standard Code for Information Interchange). When the printer receives the code for an "A," it prints this character and then moves to the next position. Alternately, the printer might receive a move command such as "carriage return" "line feed" pair, which means move to the beginning of the next line. This shows how a print file can be thought of as a program, and printing this file is equivalent to running the program on the remote computer which is the printer.

Given that a file sent to a printer can be thought of as a program, then there could be some source code which is compiled to generate this program. This is exactly what a text formatter does. In this case the source program is a text file. The data here are the text to be printed, while the programming language is the sequence of formatting commands, or markup (see Section 2.4), embedded in the text. This shows the linkage between text formatting
and printing.

Obviously, the latter definition is more substantial from a computer science point of view. There are some interesting questions here, such as: What type of architecture would be useful for a printer? Can the compiler used for text formatting be optimized? And in general, how does the print/text formatter compare to the general purpose computer/programming language?

2.3 Operations

Various operations have been developed for text formatting. It should be noted that, in general, a text formatter will not be able to perform all of these operations. Each text type will have its own operations. The operations on string text are well known and well defined. However, the operations on graphics and images are not as clear and can vary greatly from one text formatter to another. This should not be a surprise, though, since graphics and images are two-dimensional objects, while strings are one-dimensional.

2.3.1 String

2.3.1.1 Alignment

The primary goal in text formatting is the placement of the text on a page. There are a number of operators to position text, which fall into two categories: vertical and horizontal.
Examples of alignment operators are:

1. Margins—left, right, top and bottom, margins beyond which text cannot be placed
2. Center—centering text either horizontally or vertically
3. Indent—temporarily change the left margin

More than anything, these operators resemble the features found on a typical typewriter.

2.3.1.2 Filling and Justification

Filling and justification work together to produce the aligned margins which are normally associated with good looking string text. Certainly the alignment operators, that is setting margins, have an effect, but the process goes beyond alignment.

Filling is similar to the lexical phase of a compiler (Aho, Sethi, & Ullman 1986). The string text is broken up into words (tokens), then as many words as possible are fit onto the current line. After the line has been filled, sufficient space is added, between the words, to align the right margin. This is justification.

A more sophisticated version involves hyphenation, determining where words at the end of lines can be broken.

2.3.1.3 Fonts

A font is a typeface. Different typefaces can be used for emphasis, style and so on. The operator would simply be a request to change typeface.
2.3.1.4 Indicators

Indicators create string text within the running text of a document in order to indicate or draw attention to the text.

Examples of indicators are:

1. Headers and footers—a line of text at the top or bottom, respectively, of each page
2. Lists—lists can be numbers, as is this list, or have some other special symbol mark each item
3. Page numbers
4. Footnotes—the footnote number or special symbol is the indicator; the actual footnote text would be a footer

2.3.1.5 External Text

This category, like indicators, also creates text, but not within the running text.

Some examples of external text would be: (1) table of contents, (2) index, (3) floats—text which does not have a definite place, but will "float" to the first place that it fits; figures and tables are often floated.

2.3.2 Graphics

Graphics are formed in one of two ways, by alignment or by relationship.
Alignment graphics have a definite position on the page. The operators would specify a horizontal and vertical position in the same manner as string alignment.

A relationship graphic is specified by showing the relationship among the elements. For example, a table can be specified by giving each item membership in two sets corresponding to a particular row and column.

Another form of relationship is constraint. The graphic is formed by constraining it with some mathematical formula (Van Wyk 1981). A circle is given by a center point and a radius. More sophisticated shapes, ones with curves given by 3 degree polynomials, are defined using Bezier cubics (Adobe Systems Inc. 1986; Foley & Van Dam 1982; Knuth 1986).

Since the defining feature of graphic text is the representation in a compressed (non-matrix) format, other graphic operators could be possible. However, this paper will concentrate on Bezier cubics, since they play an important role in Postscript graphics (see Section 4.6).

2.3.3 Image

The image operators come from computer graphics (Adobe Systems Inc. 1986; Foley & Van Dam 1982). Generally these involve matrix multiplication and are computationally expensive. In other words, their use delays the time it takes to produce the desired text.
Examples of image operators are: (1) translation, moving horizontally or vertically, (2) scaling, stretch or shrink, and (3) rotation.

2.4 Markup

Any text formatting language uses markup to control the formatting process. In order to differentiate markup from the rest of the text, an escape character is used. For example, if a Runoff source file had the line

```
.center This text
```

the "." would serve as the escape character for the command "center." In this case the command would center the text "This text" on a line.

The markup is thought of as the commands of the text formatting language, in much the same way as there are reserved words in a programming language. The use of markup makes the text formatting languages appear similar to programming languages. Section 5.2 explores this similarity in more detail.
CHAPTER 3

THEMES

While numerous ideas are brought forth in this paper, there are several basic themes which are emphasized. These themes are explained in this section.

3.1 Co-Evolution of Text Formatting and Printers

Hardware and software cannot evolve separately because each depends intimately on the other. Typically, the theoretical basis for software is developed before its effective implementation is possible, as is seen with a language like LISP. However, developments in hardware often spur interest in some particular types of software; large scale integration (LSI) and micro-programming are an example of this effect.

In the same manner I contend that printers and text formatters have evolved together, or co-evolved—to borrow a term from the biologist Paul Erlich (Brand 1975). The primary hardware development was from a character oriented printer from the earliest computer printers in the early 1960s to a raster printer in the late 1970s (Adobe Systems Inc. 1986; Barrett & Reistroffer 1987), while the software was changing from string text formatting to graphic and image text formatting.
These developments occurred concurrently and affected one another's progress. Each text formatting language discussed in Chapter 4 is related to the introduction of some new printing technology. On the other hand, printing technology, especially raster printers, are built around the previously existing text formatters, as described in Section 5.3.

3.2 Parallel Development of Programming and Text Formatting Languages

The development of a class of languages other than programming languages gives us the opportunity to compare and contrast. In many ways text formatting languages grew up just like programming languages. However, since the goals of these two types of languages are not the same, differences exist in their respective developments. This is discussed in Section 5.2.

3.3 ASCII as the Basis of Text Formatting Languages

It will be seen that all of the text formatting languages explored here have the American Standard Code for Information Interchange (ASCII) as their base because, at so many lower levels, ASCII is used to store the text. Even the newest languages whose emphasis is on graphics utilize ASCII.

While ASCII has been a useful standard, one must question if ASCII is the correct vehicle from which to build a text formatting language. In Section 6.3, this issue is explored in some detail.
CHAPTER 4

SURVEY

Since a text formatter is a compiler, there must be some text formatting language (TFL) which is analogous to a programming language. This being the case, one would expect the development of text formatters to follow the development of programming languages. This development is traced here.

In order to examine TFLs, there need to be criteria by which to judge them. Three criteria have been chosen for this purpose.

The first criterion is what text types, string, graphic, or images, does the TFL handle? This will show the relative strength of each TFL.

The next criterion is motivated by the comparison of TFLs to programming languages: What services does the TFL provide? Traditionally, programming languages have been broken into generations, each succeeding generation providing more services than the previous. An analogous, though differing, breakdown of TFLs is also possible.

The last criterion is does the TFL support a high level view of the document?
Certainly, there are many TFLs that could have been chosen in this survey. Six have been selected on the basis of impact on the development of TFLs. These six are:

1. Runoff
2. Troff
3. TeX
4. Bravo
5. Scribe
6. Postscript

This survey differs from the definitive survey (Futura, Scofield, & Shaw 1982) in its concentration on language constructs, the ability to support text types and views, and what type of printer the language was developed for. Many changes have occurred in the area of text formatting since 1982, especially with regards to laser printers and compact disk-read only memory (CD-ROM). Both are becoming readily available for a realistic price ($2,000 - $10,000). Because these technologies have the possibility to make the storage of text on electronic media feasible, text formatting will become even more important. Thus, the development of these technologies make this treatment timely.

4.1 Runoff

Runoff was originally developed at Massachusetts Institute of Technology (MIT) by J. E. Saltzer in the early 1960s (Saltzer 1965; Kernighan & Pike 1984). This makes Runoff one of the oldest text formatting languages. Many text formatters have copied the style of
Runoff, including some more powerful ones such as that used by Bell Laboratories UNIX system (Kernighan & Pike 1984), Troff (Futura, Scofield, & Shaw 1982; Ossana 1976). Just like the programming language FORTRAN, Runoff has proven to be very durable, continuing in use even today. The particular version of Runoff that will be examined is Digital Equipment Corporation's Digital Standard Runoff (Digital Equipment Corporation 1984).

Runoff uses the "." as an escape character to markup the text. This was a vast improvement over merely typing in the text, especially if one were using cards (remember, Runoff was developed in the early 1960s).

Runoff has the ability to easily change the margins. The command

```
.LEFT MARGIN 10
```

sets the left margin at space 10.

```
.PAGE SIZE 54,60
```

sets the page length to 54 lines and the width to 60 spaces.

As should be apparent from these commands, Runoff looks at the printer as a glorified typewriter. As such, one can only use string text.

Runoff also performs filling and justification. These are thought of as modes. Thus, to turn off filling

```
.NOFILL
```

and back on again

```
.FILL
```

Using Runoff is much like compiling a program. A source Runoff file

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
is created with the Runoff commands interspersed with the text itself. Figure 2 shows the source Runoff text used to create the list on page 18.

.list 1
.le;Runoff
.le;Troff
.le;TeX
.le;Bravo
.le;Scribe
.le;Postscript
.eis

Figure 2. Example of Runoff

Once the source Runoff file is created, Runoff is invoked and the desired text is output to some file. This is quite analogous to compiling a program.

The markup in Runoff is static, which is to say one cannot create his or her own markup commands. This being the case, views can only be created by editing the Runoff source and reinvoking Runoff. In this way, physical views are possible, but not logical ones.

4.2 Troff

High quality document production has long been a part of the UNIX environment. Indeed, one of the original selling points for the development of UNIX was just this ability (Kernighan & Pike 1984).
At the heart of UNIX text formatting is Troff (Kernighan & Lesk 1982; Ossanna 1976). Troff stands for Typesetting Runoff, and as such is capable of utilizing raster type output devices such as laser printers or phototypesetters. In terms of text types, Troff will handle string and graphic text, but not images.

Typically there will also be Nroff on a UNIX system. Nroff accepts the same commands as Troff, but produces output for use on character oriented printers. Since the languages accepted by Troff and Nroff are the same, with only a few exceptions, it will suffice to examine only Troff.

Few people use Troff commands directly, just as few people use assembly language. Table 3 has some typical low level Troff commands.

Table 3
Example Troff Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.fl</td>
<td>Filling on</td>
</tr>
<tr>
<td>.vs N</td>
<td>Vertical space N</td>
</tr>
<tr>
<td>.de XX</td>
<td>Begin macro definition XX</td>
</tr>
<tr>
<td>.wh H XX</td>
<td>Set location trap for condition H</td>
</tr>
<tr>
<td>&amp;</td>
<td>Non-printing zero-width character</td>
</tr>
<tr>
<td>\c</td>
<td>Interrupt text processing</td>
</tr>
</tbody>
</table>
The analogy of Troff to assembly language is deeper than just usage. Many important processing values are stored in so-called registers (see Table 4). For example, there is a page number register which contains the number of the current page, a character type register for the current font selected, and many others.

Table 4

Example Troff Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Page number</td>
</tr>
<tr>
<td>ct</td>
<td>Character type</td>
</tr>
<tr>
<td>ln</td>
<td>Output line number</td>
</tr>
</tbody>
</table>

However, user interaction is generally not with these low-level commands, but rather with macros. Troff macros expand to form one or several commands and may have several parameters. This gives rise to libraries of macro commands which format a document according to a certain style. Typically, several of these macro libraries are standard under UNIX. For example, the ms library will format a manuscript (much in the manner of this document).
Another method of utilizing Troff is via so called filters (Kernighan & Lesk 1982). This would be an external program which filters out certain commands and replaces them with Troff commands. Examples of Troff filters are

1. EQN—mathematical expressions
2. TBL—tables
3. PIC—diagrams with symbols and connections

Troff filters capture the notion of a program toolbox, which is one of the central design principles of UNIX. Rather than adding more functionality to Troff itself, Troff is used as a tool, adding the functionality outside of it.

Perhaps the biggest contribution of Troff to text formatting is device independent output. As Troff was being developed, more sophisticated printers became available. These printers were ASCII compatible in only a general way. Each would have its own special codes to control it. So, inside of trying to produce the necessary code for each type of printer, Troff was designed to produce a sort of generic code, a device independent code. The task of actually implementing all of the device specific code was delegated to a post processor.

Thus, one sees Troff at the center of UNIX text formatting, acting as the engine or driving force. From Figure 3 it is apparent how all UNIX text formatting must at some point utilize Troff.
4.3 TeX

TeX is a text formatting language aimed at producing high quality, typeset-like documents, particularly those with mathematical formulas. TeX began as an interest by its author, Donald Knuth, in typesetting his series of textbooks *The Art of Computer Programming* (cited in Vose & Williams 1986). However, what started out as a side interest for Knuth soon grew into a project that would last for seven years and generate several books, theoretical discoveries, and several doctoral theses (Knuth 1984a, 1984b & 1986), as well as the TeX system itself.

In many ways, TeX resembles Troff. It has a command character (typically \) which precedes the formatting command. For example, the command \\

\b
In many ways, TeX resembles Troff. It has a command character (typically \) which precedes the formatting command. For example, the command
\b
will change the current typeface to bold.

A more complex example shows the box and glue model that TeX uses. This line
\hfil\hbox{The quick brown fox}\hfil
will center the phrase "The quick brown fox" on a line. \hbox is the command to form a horizontal box around some text, while \hfil is glue that will fill up the space on either side of a \hbox. Compare this command sequence to that of Runoff. While TeX is more complicated here, it is also more flexible. The hfil command shows how the user can control the spacing in a fairly powerful manner.

Like Troff, TeX uses macros to define "high level" commands out of primitive ones. The \b command is actually a macro which will change your typeface to bold for whatever typesize you are currently using. The output from TeX is device independent requiring a translation for the particular printer to be used, as with Troff.

A macro could also be defined for the centering operation given above as
\def\center#1{\hfil\hbox{#1}\hfil}
Now one could center the text from the previous example by
\center{The quick brown fox}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Thus, even though TeX has sophisticated low level operators, like hfil, the ability to create macros allows these operators to be hidden.

The TeX programmer sees a virtual machine with registers similar to Troff. This means that the programmer thinks about programming some abstract machine, as opposed to the actually printer itself. The programmer need not be bothered with the idiosyncracies of the particular printer. However, TeX goes beyond Troff in its ability to define new registers for entities such as boxes (see below for details on boxes). Such a register can be thought of as containing a copy of the text object in the box.

So, with respect to producing device independent output and utilizing registers, TeX resembles Troff. But the similarities stop there, as the underlying philosophies of operation and representation are quite different compared to Troff.

TeX is a unified program, as opposed to the software engine for a suite of programs, as is Troff. Rather than relying on preprocessors, all of the functionality has been written into TeX itself. It can handle string, table, math and pictorial text. Written in Pascal (Jensen & Wirth 1975; Knuth 1984b), TeX is not too terribly large, as it has been ported to an IBM PC (Varian 1986).

Internally, TeX builds a representation of the page to be printed from boxes. These boxes come in two types, horizontal and vertical, and are built hierarchically. Between boxes, TeX will apply "glue" that stretches and shrinks (within limits) as needs to allow for the proper placement of boxes. For example, each character
has its own horizontal box. A group of characters forming a word (perhaps delimited by a space) will be grouped in another box. This words on a line will have a 3rd level of horizontal box. The line is then put into a vertical box and these vertical boxes of lines together will form a page.

The Troff preprocessor EQN also uses boxes to represent mathematical text. TeX, on the other hand, uses boxes as its basic data structure for all text, not just mathematical text.

By using this box and glue model, Knuth applies a dynamic programming technique to obtain better line breaks (Knuth 1984b; Plass & Knuth 1982). To see how this happens, consider Figure 4.

<table>
<thead>
<tr>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>The quick brown fox</td>
</tr>
<tr>
<td>jumped over the lazy dog</td>
</tr>
<tr>
<td>while the gigantic ape died.</td>
</tr>
<tr>
<td>worst</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First fit</th>
</tr>
</thead>
<tbody>
<tr>
<td>The quick brown fox</td>
</tr>
<tr>
<td>fox jumped over the lazy dog</td>
</tr>
<tr>
<td>while the gigantic ape died.</td>
</tr>
<tr>
<td>worst</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optimal fit (TeX)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Initially there is an example of first fit line breaking; this is what filling does normally. As many words as possible are placed onto the current line. Compare the result with optimal fit. Optimal fit attempts to break the line for an entire paragraph so that</td>
</tr>
</tbody>
</table>
overall the space is added uniformly to each line. Usually first fit and optimal fit will give the same results; however, in Figure 4 one sees the difference. Essentially optimal fit has moved the word "fox" from line 1 to 2, with a marked improvement in line 3.

By utilizing a dynamic programming technique, TeX will break its lines in the manner of optimal fit. However, there is a great deal of computing expense associated with dynamic programming, so TeX first attempts a first fit of the line, before using optimal fit.

TeX should be called the macro language. Each macro definition can have up to 9 parameters. Knuth (Knuth 1984b) allows for multiple macro definition types: local, global, and without parameter expansion. One wonders whether all of this is needed, or if Knuth is simply showing off his programming skills.

One of the most intriguing aspects of TeX is that Knuth has donated it to the public domain. Furthermore, he has released the source code. This may be the largest and best documented program in the public domain. Computer Science educators should consider using this as a classic example of programming by one of the field's acknowledged masters.

4.4 Scribe

The Scribe document compiler (Reid 1982) was developed by Brian Reid as his doctoral dissertation at Carnegie-Mellon University in Pittsburg in 1979. Scribe has since been marketed as a commercial product.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Scribe assumes that the author of a document is not concerned with the actual formatting of his/her document, but is really only concerned with the concepts being effectively transmitted. The model here is someone handing a paper which is marked up much in the manner of an editor (a publishing editor, not a text editor) to a person who will format the document appropriately. Exactly how the document is formatted will be up to this person. So, an author using Scribe will add this markup to his/her document as it is being written.

The markup used in Scribe is called generic markup, because it has no specific meaning until the document is formatted. For example, there could be a markup

@title{The Evolution of Text Formatting}

Exactly what effect this will have will only be known when the document is formatted. The title could be centered on a single page, or be 13 lines down from the top of the first page and so on. In terms of programming, this would be called declarative, since there is no concern with exactly how the operations are performed, only their meaning (Chamberlin, King, Slutz, Todd, & Wade 1981).

Scribe looks at a document as a tree (see Figure 1). Each node of the tree is called an environment. The basic use of environments is to inherit attributes to lower levels. For example, a subsection environment would inherit the attributes of the section environment.

Another way environments are useful is producing numbering. One might wish to number figures or theorems consecutively throughout the document. Thus, an environment can maintain the numbers automatically. This numbering feature is also combined with a cross
reference feature, so that references to a numbered object, like a figure, can automatically be updated when the numbering changes (due to additions or deletions of preceding figures).

Scribe obviously supports physical views, since it can easily change the physical format without changing the meaning of the markup. Since Scribe represents a document internally by a tree, it also has logical views. Scribe implements these views with what are called document styles. This would correspond to a style sheet given out by a publisher to an author. So, if the Scribe document needs to be formatted in a different style, it is only necessary to apply a different document style.

Although Reid implemented Scribe as a separate TFL, the major features in Scribe can be incorporated into other TFL. The macro package LaTeX (Lamport 1986) has done just this, adding support for environments and document styles to TeX.

4.5 Bravo

Any consideration of text formatting must take into account word processing. Word processors are quickly replacing typewriters as the tool of choice for document preparation. Certainly, any modern office will have one. And one could argue that the word processor, more than any other type of software, has most helped to popularize micro based computers.
At first glance it might seem odd to consider a word processor as a computer language. However, the software techniques used to build compilers are also used for text editors (Aho, Sethi & Ullman 1986), and a word processor is really a combination of a text editor with a text formatter (Foley & Van Dam 1982; Reid 1982). In terms of computer languages, a word processor corresponds to an interpreted language, since the commands affect the format of the document (as seen on the screen) immediately. This is in contrast to a language such as Runoff, where the results can be seen only after a batch run.

Because you can look at the format of your document as you edit it with a word processor, this type of system is often called "what you see, is what you get" or WYSIWYG. The debate as to whether WYSIWYG is really the best method for text formatting is an ongoing one. For more details on this debate see Section 5.1.

To represent the WYSIWYG systems, the Xerox Bravo (Lampson 1978; Reid 1982) was chosen. Bravo was an editor/formatter developed at the Xerox Palo Alto Research Consortium (PARC) during the mid 1970s as a part of the Alto computer and has greatly influenced small system design with its desktop metaphor and mouse driven input. While Bravo was a software package, its laboratory success spurred the creation of the Xerox Star and Apple MacIntosh computers (Futura, Scofield, & Shaw 1982).

What makes a Bravo-type system interesting for text formatting purposes is its high resolution bit mapped screen. Indeed, the prime motivation for the development of Bravo was as a text formatting machine. The high resolution graphics make the representation of
highlighting and multiple fonts very much a possibility.

At the moment, the Apple Macintosh computer stands at the vertex of micro based computer aided publishing, otherwise known as desk-top publishing. The release of desk-top publishing software in 1985 has marked a new era for printing. The current front running software is Pagemaker (Aldus Corp. 1986). While it does give the user the power of a print shop for pamphlets and newsletters, it is limited (currently) to only 16 pages, hardly a book or even a technical paper. Additionally, the user must compose the text entirely outside of Pagemaker and then lay out the document in a rather manual manner. Each text type is unrelated and must be positioned by the user. This means that ANY change in the text calls for reformatting the entire document with this manual method. More work needs to be done here before these types of products will be useful for generalize text formatting.

4.6 Postscript

Almost ten years in the making, Postscript is the newest member of the six TFLs being considered here; it was released in 1985 by Adobe Systems Inc. (Adobe Systems Inc. 1986). Postscript may also turn out to have the greatest impact. The driving force behind Postscript is John Warnock, who spent time at PARC developing a predecessor to it.
What makes Postscript so important is that it has been implemented directly on a printer, acting in effect as the machine language for that printer. By implementing the same language on many printers, the same file can be printed on any of those printers without modification. For example, the same Postscript program that will print on an Apple LaserWriter, will also print on a book quality (1200 dpi) phototypesetter (Adobe Systems Inc. 1986).

While others manufacturers having written their own graphic languages which reside either on the printer itself or a printer controller (Telaris Systems Inc. 1984), Adobe is attempting to establish Postscript as a standard language for printers. An interesting sideline is that a competitor language, Interpress, was also being developed at PARC while Warnock was there, although the two are somewhat different (Adobe Systems Inc. 1986).

Postscript is known as a page description language (PDL). Rather than printing a character at a time, as a character printer might, a PDL forms a virtual image of the entire page before printing it. This is appropriate for a raster output device such a laser printer.

The difference between a PDL and ASCII is that of dynamic versus static codes (Adobe Systems Inc. 1986). ASCII is a fixed length code (either 7 or 8 bits) and all commands must fit into this framework. While ASCII does provide for non-ASCII commands, using the escape character this usage is cumbersome. A PDL, on the other hand, is more like a programming language in that there are primitive operations that can be combined to form higher level operations. A
PDL does not need fixed length codes.

Of course the purpose of a PDL is to generate a page of text, and, as such, Postscript provides a rich assortment of operators to manipulate all the text types. But, the power of Postscript is enhanced by its use of stacks. Not only is there a stack to process commands (remember, Postscript is interpreted), there are also various ancillary stacks for other functions such as saving some text object. As a language, Postscript resembles Forth (Adobe Systems Inc. 1986).

Postscript assumes it is using a virtual machine, much as TeX and Troff do. As opposed to them, Postscript's machine is stack based. And since it is stack based, Postscript uses postfix commands.

Here is the Postscript command to generate the phrase "The rain in Spain."

(The rain in Spain) show

This example demonstrates simple string text. To treat this phrase as a graphic and specify a particular position on the page, say the upper left corner, then the commands are

newpath 0 0 moveto
(The rain in Spain) show

Postscript provides a wide variety of operations which can be used by any of the text types, as the above example shows.
Postscript graphics are represented as Bezier cubics (Foley & Van Dam 1982; Adobe Systems Inc. 1986), as are its character fonts. With a Bezier cubic one can draw a curve represented by a 3rd degree polynomial (see Figure 5).

![Figure 5. Bezier Cubic Representation of a Curved Line](image)

Four points are chosen, two as the end points and two to control the curve. Call these four points $z_0, \ldots, z_3$ and let $z_0$ and $z_3$ be the end points. The curve, $z(t)$, is the solution to this parametric equation

$$z(t) = (1-t)^3 z_0 + 3(1-t)^2 t z_1 + 3(1-t) t^2 z_2 + t^3 z_3$$

letting $t$ vary from 0 to 1.

This curve can be approximated to whatever degree of accuracy (within the bounds of the numeric representation) by building up what might be described as scaffolding. Line segments are drawn to approximate the curve in the following manner:
1. Connect the four points in order

2. Connect the midpoint of each line to the midpoint of its neighbor

3. Continue step 2, using the new lines generated by the last step

From Figure 5 it can be seen that after only three iterations the approximation is good.

Character fonts in both Postscript and TeX are represented with Bezier cubics. But, Postscript has Bezier cubics available for general graphic text usage, while TeX does not. The separate program METAFONT (Knuth 1986) is used to create TeX fonts.

Typically, Postscript is generated by a program, as opposed to being entered by the user. Since it has such a rich and powerful mixture of both graphic and image operators, Postscript is tedious for a human to enter. On the other hand, the use of postfix commands and a stack based machine make it easy for a program to generate Postscript code.

It is easy to see how the device independent output of Troff or TeX might be translated into a Postscript program and then sent to a Postscript driven printer. So, rather than making these languages obsolete, Postscript enhances their potential.

4.7 Summary

Table 5 summarizes the capabilities of the TFLs surveyed. No one language has all of the capabilities. It is possible to add more capabilities by combining TFLs. For example, TeX could produce
output that is later processed by Postscript.

Table 5
Summary of TFL Capabilities

<table>
<thead>
<tr>
<th>TFL</th>
<th>Text String</th>
<th>Type Graphic</th>
<th>Image</th>
<th>View Physical</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runoff</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Troff</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>TeX</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bravo</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Scribe</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Postscript</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Of course just looking at the capabilities of a TFL does not answer the question of which one is better. As is often the case, the answer depends upon your application. To produce a short document, a Bravo type system is preferable. However, for longer documents, whose style may need to change, Scribe would be better.

The next section, analysis, will address these larger issues, especially the subsection on user interface.
CHAPTER 5

ANALYSIS

This section will look at how text formatting languages have evolved from three points of view: (1) user interface, (2) programming languages, and (3) printer architecture.

Each of the surveyed languages has its own plus or minuses from each point of view, and no one language will be the best from all angles.

5.1 User Interface

Essentially, the interface to a text formatter can be described as either interactive or batch. An interactive formatter is most often called a word processor. Sometimes batch formatters are described as pure formatters. Each type of interface has its supporters and detractors (Futura, Scofield & Shaw 1982, Seybold 1982).

An interactive formatter is sometimes called "what you is see is what you get" or WYSIWYG. The appeal of such a system is obvious, you merely make your document look correct right on the screen.
This is contrasted with batch formatting, sometimes called "guess formatting" (Futura, Scofield & Shaw 1982). It can often be very difficult to visualize exactly how your document will turn out. You must exit your editor, run the formatter, then view the finished product. This edit/format/view cycle can be rather frustrating, as well as time consuming. In effect, the user has become a programmer.

But then, an interactive formatter does not make our life so easy either. As noted, Kernighan has called WYSIWYG systems "what you see is all you get" (Lamport 1985). Once you type in your document in a certain style, that's it, you are stuck with it. If you later change your mind, and want to make some global change in style, it can be very difficult with a WYSIWYG system. By contrast, with pure formatters, one can change the style easily.

Another advantage of pure formatters comes up when you attempt to use a higher resolution printer, such as a laser printer. What you see may not really be what you get, unless your terminal screen has resolution on par with the laser printer (typically 300 dots/inch). Getting such a terminal is currently a high priced affair, on the order of $5,000. Most users will opt for buying one expensive printer, and many inexpensive (low resolution) terminals.

But perhaps the biggest reason to use a pure formatter is the increased functionality that comes with doing a complete pass of the document. Such items as table of contents and index entries can be generated automatically only with a complete pass.
There are factors in the user interface other than simply whether the TFL is interactive or batch, however. The basic aspect of these factors is how much control the user has over exactly how the document will look.

A language like Scribe assumes one does not wish to bother with the lower level details of exactly how your document looks. On the other extreme, Postscript has extremely powerful operators, so powerful that Postscript is usually generated by some other programmer, rather than entered directly by the user.

Figure 6 summarizes what could be termed a taxonomy of TFLs. This shows a way of classifying the languages according to how a user interacts with them.

---

Figure 6. Classification of Surveyed TFLs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
5.2 Programming Language

The correspondence between text formatting and compiling was noted. It would not be unreasonable, then, to expect a correspondence between text formatting languages and programming languages. Also, one might expect to see similarities in development.

Typically programming languages are categorized by generations, to date there are four generations in use (Martin & Leben 1986). The earlier generations are closer to machine code, while the later are higher level, closer to English and the way humans think.

However, the programs generated by text formatting do not contain loops, since the concept of a loop does not enter into the concept of text. After all, text is a static entity. Moreover, the purpose of text formatting is different than that of programming languages. One does not typically think of programming when creating text. So, these differences may cause dissimilarities in the development of TFLs compared to programming languages.

Looking at TFLs from the perspective of what lower level operations are controlled by the user, one might come up with a correspondence between TFLs and programming languages as shown in Table 6. It is interesting to see that the developments in TFLs do not parallel those of programming languages.
Table 6
Generations of TFLs

<table>
<thead>
<tr>
<th>PL Generation</th>
<th>TFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Runoff</td>
</tr>
<tr>
<td>2,3</td>
<td>Troff</td>
</tr>
<tr>
<td>2,3</td>
<td>TeX</td>
</tr>
<tr>
<td>1</td>
<td>Bravo</td>
</tr>
<tr>
<td>4</td>
<td>Scribe</td>
</tr>
<tr>
<td>2</td>
<td>Postscript</td>
</tr>
</tbody>
</table>

Indeed, the latest TFL in the survey, Postscript, actually corresponds to an early generation programming language. One might even argue that since Postscript was designed to reside on a printer that it is a machine language or generation 1.

Bravo is called a generation 1 language because in a sense you are working with exactly the machine language, that is how the text will look. Perhaps this correspondence is not exact, since some powerful graphic and image operators are at the user's disposal. Still, if the ASCII as machine code analogy is accepted, a WYSIWYG formatter essentially works on the level of ASCII code. If you wish to start a new line, you enter the ASCII codes for line feed/carriage return, and so on.
It is difficult to see a precisely parallel development between text formatting languages and programming languages, even though the underlying software, compilers, is the same.

5.3 Printer Architecture

If text formatting languages did not evolve along the same lines as programming languages, what forces did drive their development? One good candidate would be changes in printer technology (Barrett & Reistroffer 1987).

Certainly Runoff has to be considered as a first generation TFL. When Runoff was created, the only types of printers available were character oriented printers. Essentially, a character printer is just a typewriter that is connected to a computer.

ASCII is a typical representation for the text printed on a character printer. A look at the scheme for ASCII (see the Appendix Seven Bit ASCII) shows that the majority of codes (the last 96 except for the delete character (DEL), to be precise) will print a single character. Other codes, for example linefeed (LF) and carriage return (CR), give movement commands. The CR code, carriage return, is especially indicative of the typewriter paradigm that is implicit in ASCII.

But the designers of ASCII did not wish to restrict its capabilities to only those printers which followed the typewriter paradigm. There is special character, ESC, in ASCII through which one can define a sequence of characters as a new command. ESC means, in essence, that what follows is not actually an ASCII code, but
rather has some special meaning that falls outside the scope of ASCII. One could call ESC a metacommand, a command for a command.

So, when new technologies where incorporated into printers, for example the dot matrix, and the printing possibilities fell outside the usual scope of ASCII, the ESC command was used. A typical example of this would be bold characters.

A dot matrix printer is still, to a large extent, a character printer, since there is a print head. The next big jump in printer technologies is to a full raster printer (Adobe Systems Inc. 1986, Barrett & Reistroffer 1987); a laser printer is an example of a raster printer. ASCII is truly inadequate when it comes to a raster printer. There is no print head associated with a raster printer. Each element can be accessed at random in a relatively brief time. These elements are often called pixels. What a raster printer prints has been termed image text in this thesis.

There is an enormous overhead associated with using ESC commands with a raster printer, since the concept of pixels is nowhere to be found in ASCII. However, there is a temptation to somehow stick within the ASCII framework, even when using a raster device. All of the surveyed TFLs utilized ASCII, especially for their character fonts. Even Postscript, a language designed with a raster printer in mind uses ASCII as the basis for its fonts (Adobe Systems Inc. 1986).

The move away from strict character printers and towards raster printers appears to be the driving force, then, behind the evolution of text formatting languages. Certainly Troff, TeX and Postscript were designed with higher resolution raster printers in mind. Bravo
was designed for a high resolution terminal, as well as printer. Scribe actually assumes no printer whatsoever, but this still supports the move towards better printers, since the details of these printers are complex and Scribe attempts to insulate the user from these details. Table 7 contains a summary of printer architecture.

Table 7

<table>
<thead>
<tr>
<th>Printer</th>
<th>TFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>Runoff</td>
</tr>
<tr>
<td>Raster</td>
<td>Troff</td>
</tr>
<tr>
<td>Raster</td>
<td>TeX</td>
</tr>
<tr>
<td>Raster</td>
<td>Bravo</td>
</tr>
<tr>
<td>None</td>
<td>Scribe</td>
</tr>
<tr>
<td>Raster</td>
<td>Postscript</td>
</tr>
</tbody>
</table>
CHAPTER 6

FUTURE

Obviously, this is not the end of the evolution of text formatting languages. Indeed, there are some very exciting development, some of which may change the whole way in which text is viewed.

6.1 Electronic Text

The concept of text stored in a dynamic, electronic form has tantalized us (Nelson 1976) even before computers were built (Bush 1945). The development of write once, read many times (WORM) and CD-ROM technologies certainly has changed these possibilities in the last few years (Peels, Janssen & Nawijn 1986; Yankelovich, Meyrowitz, & Van Dam 1985; Weyer & Boring 1985). Most text once published is never revised.

Consider the possibilities for periodicals. Already the American Mathematical Society is accepting articles written with TeX (Knuth 1984b) and a data base of references is being maintained on-line. It is not a very large step to being able to access the entire article on-line. Then, if one were reading a list of citations on-line and wished to see a particular article, it would be possible to bring that article up on the screen quickly.

46
Additionally, one could then look at the citations from this article (which may be different than the original list) and quickly bring these articles onto the screen. This could greatly aid researchers in all fields.

6.2 Some Current Research

Some of the more interesting research being done in the area of text formatting is surveyed in this section.

6.2.1 Standard Generic Markup

Standard Generic Markup Language (SGML) (Goldfarb 1981) was developed by International Business Machines (IBM) to standardize text formatting, as its name implies. In many ways, it resembles Scribe, since SGML is declarative. Currently there is a International Standards Organization (ISO) draft standard for SGML (Smith 1986) and electronic documents. This standard allows the document to be electronically transferred, and later be formatted.

However, even before the draft standard is accepted, there are extensions being formulated. Digital Equipment has created Digital Document Interface Format (DDIF) (Hannula 1987) for electronic documents. The difference between SGML and DDIF is that DDIF can handle all three text types, while SGML can only deal with string text. This limitation on SGML would appear to make it an outdated standard before it is even adopted.
6.2.2 Music Printing

Ohio State University has embarked on a music printing project (Gourlay 1986). However, the aim is more than merely printing; it is to create a format for the electronic representation of a music score. With this format, one can create, edit, play, and transmit electronically a musical piece. The key element is a high level view of the music.

6.2.3 Abstract Object Editor

At the University of Washington in Seattle, workers have attempted to formalize the concept of text formatting (Futura, Scofield & Shaw 1982; Kimura & Shaw 1984; Kimura 1986) using abstract objects. This work influenced the definition of formatting given in Section 2.2.1. Essentially, they see text editing and text formatting as different parts of the same overall process. In this view, every document to be formatted is composed of abstract objects. Editing transforms abstract objects to other abstract objects, while formatting transforms them to concrete objects. Concrete objects would then be our hardcopy from a printer. A view is one particular abstract to concrete transformation.

The key issue behind this is what comprises an abstract object. A simple minded version merely uses strings as the objects; Runoff works in this manner. However, a more powerful implementation would use many other types of objects, for example, a tree to represent a document with chapters, sections and so on.
Kimura has implemented a prototype abstract object editor (Kimura 1986). However, he views text as being of four types: string, tables, mathematical, and diagrams. Based on current text formatting products, Postscript, DDIF, it seems there are really only three types of text, as discussed in Section 2.3.1.

The types of abstract objects that are being proposed here are trees and sets. What is not clear is how someone might effectively use these or any other abstract object. The basic underlying problem is that there could be many abstract objects possible for a particular document. Ideally, these objects would conform to the way humans think about text and formatting. For string text, there does appear to be a clear methodology; however, graphics and images are another matter.

Consider graphic text, text for which there is a two dimensional relationship among the components. This is another way of saying that there is semantic information in how text is formatted. But there is a catch here in trying to formulate abstract objects: for example, a table is thought of in terms of its two-dimensional representation, that is, how it is ultimately formatted. Perhaps one may have some special way of looking at a table in his or her mind, but in general it is thought of in terms of how it actually looks.

Thus, one really needs to look for reverse mappings from concrete objects to abstract objects. Since this is a many-to-many mapping, the problem is essentially intractable. Put another way, the best way to deal with graphics is with a WYSIWYG editor. But once it is decided how a graphic looks, how can it be transformed
into an abstract object?

There needs to be more work on what abstract objects are appropriate for text documents.

6.3 Is ASCII Good Enough?

In each of the TFLs surveyed, as well as most commercial document formatters, one finds the American Standard Code for Information Interchange (ASCII) lurking about. Naturally one expects to see ASCII in an earlier TFL like Runoff; however, one still sees ASCII at the heart of the latest TFL surveyed, Postscript.

What role does ASCII play in Postscript? Text may require many fonts, which poses a problem for text formatters: how to represent characters from different fonts? In Postscript, it is possible to define font directories, each containing 256 characters (Adobe Systems Inc. 1986). During string text processing, Postscript will have a current font directory from which characters are taken. To change the font, one must bring into the memory of the printer a font directory. Several of these directories can be stored in the memory, the number depending on the sizes of the directories and the memory. It is no coincidence that the font directory contains 256 characters. This is precisely the number of characters possible with an 8 bit ASCII code.

TeX also has ASCII at its heart. With so much of TeX alterable, which character is the space, which is escape and so on, this one aspect of TeX is hard coded. Knuth specifically declares that one should create new fonts for TeX with ASCII in mind (Knuth 1984b).
ASCII has certainly been one of the more useful standards developed. Without such a standard, it is hard to conceive of the state of personal computing. However, as was noted in Section 6.3, ASCII was created for character oriented printers. Both Postscript and TeX were made for the new generation of raster oriented printers. Is it really necessary to have ASCII at the heart of the latest generation of TFLs?

Of course, one might say that it's not really ASCII that is at issue here, but rather an 8 bit fixed length code. This is true, so the question could be whether 256 characters are enough for a single font?

In languages such as English or German, certainly 8 bit ASCII is sufficient. But consider Chinese or Japanese; 256 characters will hardly make a dent on these languages. Implementations using an ASCII base might have to switch often between fonts for these languages. Perhaps unintentionally (or even intentionally), computer standards are forcing people to create text in a Western language.

Another problem with ASCII is that it is a fixed length code. Certainly, mainly characters, especially space, will be more frequent than others. A Huffman encoding (Tanenbaum 1982) of the characters might make more sense, particularly for electronically transmitted text. This encoding could be stored at the beginning of the document. Or perhaps a standard Huffman encoding could be adopted for all documents in specific languages. There are many possibilities.
APPENDIX

Seven Bit ASCII
<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 NUL</td>
<td>0000 NUL DLE 0 0 0 ` p</td>
</tr>
<tr>
<td>0001 SOH</td>
<td>0001 SOH DC1 1 1 A Q a q</td>
</tr>
<tr>
<td>0010 STX</td>
<td>0010 STX DC2 &quot; 2 B R b r</td>
</tr>
<tr>
<td>0011 ETX</td>
<td>0011 ETX DC3 # 3 C S c s</td>
</tr>
<tr>
<td>0100 EOT</td>
<td>0100 EOT DC4 $ 4 D T t</td>
</tr>
<tr>
<td>0101 ENQ</td>
<td>0101 ENQ NAK @ 5 E U u</td>
</tr>
<tr>
<td>0110 ACK</td>
<td>0110 ACK SYN &amp; 6 F V v</td>
</tr>
<tr>
<td>0111 BEL</td>
<td>0111 BEL ETB , 7 G W w</td>
</tr>
<tr>
<td>1000 BS</td>
<td>1000 BS CAN ( 8 H X x</td>
</tr>
<tr>
<td>1001 HT</td>
<td>1001 HT EM ) 9 I Y y</td>
</tr>
<tr>
<td>1010 LF</td>
<td>1010 LF SUB * : 9 J Z z</td>
</tr>
<tr>
<td>1011 VT</td>
<td>1011 VT ESC + ; 9 K k</td>
</tr>
<tr>
<td>1100 FF</td>
<td>1100 FF FS , &lt; L \ l</td>
</tr>
<tr>
<td>1101 CR</td>
<td>1101 CR GS - = M ] m</td>
</tr>
<tr>
<td>1110 SO</td>
<td>1110 SO RS / &gt; N ^ n</td>
</tr>
<tr>
<td>1111 SI</td>
<td>1111 SI US : ? O _ o DEL</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


