AN EXPERIMENTAL CURRICULUM FOR TEXT PROCESSING

A project submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science

by

Glen Williams

June, 1979
The Project of Glen Williams is approved:

Professor Motil

Professor Walsh

Professor Abbott

California State University, Northridge
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iv</td>
</tr>
<tr>
<td>The Objectives</td>
<td>1</td>
</tr>
<tr>
<td>The Realities</td>
<td>13</td>
</tr>
<tr>
<td>TEO</td>
<td>14</td>
</tr>
<tr>
<td>The Editor Assignment</td>
<td>21</td>
</tr>
<tr>
<td>An Aside On PASCAL</td>
<td>24</td>
</tr>
<tr>
<td>Pattern Matching</td>
<td>32</td>
</tr>
<tr>
<td>Formatting</td>
<td>37</td>
</tr>
<tr>
<td>Typesetting</td>
<td>42</td>
</tr>
<tr>
<td>First Generation Typesetters</td>
<td>47</td>
</tr>
<tr>
<td>TTS</td>
<td>52</td>
</tr>
<tr>
<td>The Advent of Photocomposition</td>
<td>54</td>
</tr>
<tr>
<td>Second Generation Principles</td>
<td>57</td>
</tr>
<tr>
<td>Third Generation Typesetters</td>
<td>61</td>
</tr>
<tr>
<td>Composition</td>
<td>66</td>
</tr>
<tr>
<td>Hyphenation</td>
<td>70</td>
</tr>
<tr>
<td>Input To A Typesetter</td>
<td>74</td>
</tr>
<tr>
<td>Pagination</td>
<td>76</td>
</tr>
<tr>
<td>Word Processing</td>
<td>80</td>
</tr>
<tr>
<td>Office Automation</td>
<td>85</td>
</tr>
<tr>
<td>Conclusion</td>
<td>91</td>
</tr>
<tr>
<td>Bibliography</td>
<td>96</td>
</tr>
<tr>
<td>Appendix A (III 1070 Editor User Guide)</td>
<td>97</td>
</tr>
<tr>
<td>Appendix B (A Sample &quot;Read&quot; Program)</td>
<td>100</td>
</tr>
<tr>
<td>Appendix C (CDC Character Set)</td>
<td>103</td>
</tr>
<tr>
<td>Appendix D (A Set Manipulation Program)</td>
<td>105</td>
</tr>
<tr>
<td>Appendix E (Editor Specification)</td>
<td>107</td>
</tr>
<tr>
<td>Appendix F (&quot;FINDL&quot; Pattern-Matching Program)</td>
<td>117</td>
</tr>
<tr>
<td>Appendix G (&quot;FINDA&quot; Pattern-Matching Program)</td>
<td>121</td>
</tr>
<tr>
<td>Appendix H (Flow of RUNOFF Justification)</td>
<td>125</td>
</tr>
<tr>
<td>Appendix I (Text Processing Midterm and Final)</td>
<td>127</td>
</tr>
</tbody>
</table>

iii
This project details the definitions and implementation of an experimental curriculum for an upper division class in Text Processing within the Computer Science Department at CSUN in order 1) to determine what information should be presented about this emerging field and 2) to experiment with a method of presenting that information.

The class was offered in the Spring of 1979 under the direction of the objectives defined in the first section of this paper. In teaching the class, certain realizations arose concerning the implementation details specified to meet these objectives. As often as not, these details were not predicted.

Therefore, this paper consists of two conceptual divisions: a statement of the a priori objectives; and a description of the material actually covered -- interspersed with remarks on the class's reaction to the
material.

Topics covered include the use of TECO as a text processing language, the implementation considerations for a text editor program, pattern matching, implementation considerations for a formatting program, the history and development of typesetting and phototypesetting devices and composition software, word processing, electronic message systems and the automated office. The implementation of a text editor was assigned as a project, which necessitated some discussion of the PASCAL programming language.
The Objectives

In considering the nature of "text processing" many aspects may come to mind. Text processing is as yet somewhat loosely defined in relation to the other disciplines that together comprise "Computer Science". It has been with us in one form or another almost since the first computers (in the form of holerith cards carrying information). Yet it existed almost 'by the leave' of the more urgent tasks at hand -- the loading of data (programs included) into the computer. Even now text processing involves many of its sister disciplines. For instance, pattern recognition is involved in the process of capturing text during OCR while data compaction can be invoked in storing characters in a DBMS (as with ADABAS). And, of course, text processing may be said to 'take over' such heretofore non-computer disciplines as typewriting (e.g., word processors and editors) and typesetting (phototypesetters).

The text processing field is currently expanding faster than most other areas of computer science in the form of the automated office (wherein text processing plays a central role), electronic message systems, and document/data processing interfaces.

The automated office and the Electronic Message System (EMS) are closely related, as it is the EMS that provides the capability for sharing and storing data from heretofore isolated word processors. The EMS also enables the
travelling executive to maintain intimate ties with his organization at any time of day. His time is saved in that recipients of his communications need not be "present" to be reached.

Another new linking of disciplines is introduced with the relation of document processing to data processing. Document processors are adding capabilities to directly link to data processing applications in an interesting form: the document "hides" binary information in text files for data processing programs to utilize in providing up-to-date reports. For instance, the latest figures in a company's financial report may exist in a text file that is accessed by data-processing programs to make financial projections.

Therefore the objectives of a text processing class seek to identify the reach of text processing. In introducing the field, a great deal of emphasis is placed on knowing the problems inherent in a well engineered text editor. This knowledge equips the student with an appreciation for the nature of 'text' as well as methods for its manipulation. The rest of this section discusses the objectives in more detail.

TECO

This course therefore introduces the students to TECO, a powerful character-oriented text editor and programming language sold by Digital Equipment Corporation. TECO may
smallest addressable unit of text. They are mentioned as an introduction to a discussion of the various ways to specify and implement a text-processing program: in particular, an editor. Augmenting this is a discussion of SOS. SOS is a program written at Stanford University, where SOS stands for "SON OF STOPGAP". It is so named because it is meant to bridge the gap between line-oriented editors (like LINED at DEC) and TECO. Stopgap is primarily line-oriented in that one may address lines as the smallest addressable unit of text. However, it differs from simple line-oriented editors in its 'Alter' command. This command allows the user to enter an intra-line editing mode in which he may move through the current line with character-cursor moving commands, and of course, he may modify individual character strings within the line without retyping the whole line.

The Editor Project

The purpose of placing the editor discussion at the beginning of the course is to allow the assignment of a simple editor as a project. The class is required to hand in a functional specification of an editor: each student specifying the editor he will implement. To ease their trauma, discussions for the following few sessions will center on what a reasonable editor should incorporate. The subject of how to implement each feature and how the internal routines may be defined complimentarily are also addressed.
To be specific, some considerations are:

1. The language the editor is to be written in
2. What type of device will it run on (TTY or smart terminal?)
3. Line editor or character editor? The choice affects the cursor-positioning commands at the disposal of the user
4. How will user get a display of his text?
5. How is the cursor represented?
6. Will the cursor position follow typeout (as with XEDIT, a line-oriented editor for CDC machines)?
7. How is the cursor moved? In what increments?
8. What form will commands take? One-letter commands?
9. How many commands will the user be able to enter at one time if commands are entered in line-mode? -- i.e., the program gets characters from the terminal only after a 'break' character is entered.
10. What form should insert take and how should it work? That is, should text be inserted before or after the cursor? What bearing does this have with entering text before the first line or after the last line?
11. Should the delete command delete characters or lines or both?
12. How do we divide a file into pages--or should we?
13. What form should the editing buffer and/or the Q-registers take?
14. How is text entered into a Q-register (a Q-register is an unbounded temporary storage area for saving text)
15. How is text retrieved from a Q-register and how is this specified?
16. Searching -- should it search around end-of-lines? should it search backwards? what happens to the cursor in the event of either failure or success? Should there be class-matches for pattern elements?
17. Help command. How will provision be made for allowing a user to get a refresher of which commands are available, etc.?

18. Meta characters: How does the user specify characters that are to be in the output file if the character is not in the keyboard's character set? Meta characters are one possibility. For instance, SOS uses the question-mark as a meta character.

To insure the completion of this editor, milestone-style assignments are made throughout the semester.

Pattern Matching

A subject of great importance within text processing is pattern matching. Concerns within pattern matching theory include minimizing the number of operations required for an average search and matching classes of patterns that can be defined as regular expressions. In introducing simple pattern matching, the procedure FIND that is described by Horowitz and Sahni\textsuperscript{1} is examined. This procedure runs in time $O(mn)$ where $m$ is the length of the pattern and $n$ the length of the string. Such an algorithm is obviously quadratic. From this procedure we can easily move to an improved FIND, also in Horowitz and Sahni\textsuperscript{2},


\textsuperscript{2}Ibid., p. 192.
that introduces the short-cut of checking the first and last characters of the pattern with their counterparts in the string and also of terminating the search when the pattern length is greater than the number of remaining characters in string. This algorithm, while potentially also running in time O(mn), actually exhibits better performance in actual usage.

From concerns of speed it is possible to move on to a discussion of "A Fast Searching Algorithm". This algorithm introduces more preprocessing of the pattern as a means of reducing the number of characters examined in the course of a search. It employs a technique for ensuring that we never move backwards in the string while progressing through the string. The actual loop for cycling through the string is also shortened by employing two arrays that provide the information of how far we can "slide" in the string as a function of either the character at the cursor or as a function of reoccurrences of subpatterns within the pattern. This technique has been found to be sublinear in the number of comparisons required to find a successful match.

---

attern-class matching techniques to speed such types of search as found in SNOBOL. In discussing so-called "wild-card" matches as employed in TECO, the subject of very general pattern searches may be broached. TECO allows a pattern of up to 36 characters. In any character position of the pattern we may specify any character or class of characters as a match. For instance a control-X character entered as a character for a given position will cause TECO to accept any character as a match. One may also specify to accept any element of an enumerated set of characters. By prefacing a specification with a control-N character, we search for any match except that specified by the following specification. The beauty of TECO's wildcard patterns is that a character in the string can be accepted or rejected as a match in one machine instruction. The search loop itself is less than 7 machine instructions. The reason, of course, is due to the first pass TECO makes to encode the pattern. Thus it should become obvious to the student that a small amount of preprocessing can save one or two orders of magnitude on operations, depending upon the length and complexity of the pattern. The obvious drawback is that we cannot make searches that contain an elipsis.

A very general pattern search on the order of SNOBOL is coded and described by Kernighan and Plauger⁴.

---

Their pattern matching algorithm covers all regular expressions except of patterns of the form $x(ab|bc)y$, the pattern which matches either $xay$ or $xbc$. It also preprocesses the pattern, but not for any speed considerations. It is mainly to make the classes readable by the actual matching logic. Their treatment is valuable to the student in that it clearly delineates all the steps involved in implementing a sophisticated (in the sense of handling closure) pattern matcher. It gives an extensive treatment to both the building the pattern representation as well as to the utility of recursion in following possible leads for a closure match. Following the treatment of matching, the idea of searching and replacing in the same operation is detailed. The final beauty of their work is perhaps its accessibility to the avid student: all their routines are written in a pseudo block-structured language: RATFOR.

Finally, a discussion on blending the speed characteristics of the Boyer and Moore sublinear searching algorithm with the generality of either TECO's or Kernighan and Plauger's probes most of the important aspects of pattern matching. Such a blend is indeed possible but to my knowledge, not yet realized in a commercial product.

Text Formatting

Kernighan and Plauger also give a fine, if not too detailed, discussion and example of a text formatting program.5
Incorporating material from *Software Tools* is useful in that it allows discussion of fine details by examining the example program developed therein.

Two prime functions of a text formatter are covered: to do "page layout" and horizontal justification. Within these discussions appropriate commands to allow the user control over these functions arise, as well as the criteria for their implementation.

However, their logic for justification seems wanting in that it does not seem to distribute extra spaces as evenly as does RUNOFF. Therefore, the exact algorithm RUNOFF employs is examined in detail. Examples of RUNOFF input files are also given in order to make the sense of certain commands more concrete.

The reason page layout was in quotes above is that neither RUNOFF nor the Kernighan and Plauger program deal with page layout as thought of by printers. Page layout to RUNOFF is managing to get headers and footnotes as well as the text on the page in a manner pleasing to the eye. Page layout as envisioned by printers is much more complicated and is only recently being attempted by commercial phototypsetting concerns. One illustrative problem is that of specifying a position for a photograph on a page and asking a typesetting program to "flow" the text around it, say in a double-column format. But while RUNOFF does not attempt such a task, a discussion of its

---

concerns prepares the student to deal with many of the concerns of real typesetting.

Computer Driven Phototypesetting
The major text for the typesetting is Fundamentals of Modern Composition by John Seybold (of the "Seybold Reports" fame). Concepts in this text are augmented with material from Electronic and Electro-optical Publishing Equipment Market. In addition, programming manuals published by Information International, Inc. are referenced when dealing with typesetting programs.

The sequence of the Seybold book is followed somewhat closely. The first six chapters deal with the history of printing as well as the various characteristics of "type". These chapters are important in various ways: in studying this facet of text processing the student must be made aware of the complications inherent in dealing with printing, whether it is created electro-optically or mechanically. The main complication is non-standardness within the field. Almost all measures are relative (and the relations are not necessarily transitive.) Aside from

6 John Seybold, Fundamentals of Modern Composition (Media, Penn.: Seybold Publications, Inc., 1977.)

exposure to inherent complications, the student profits from a survey of the methods employed throughout the history of typesetting. For instance, the idea of a "meta" character is introduced in the 1930's for shifting 'rails' on a paper-tape-driven linecaster. Even earlier (the 1880's), the inventers of Monotype developed the system used even today in measuring characters as values relative to the widest character in the font.

Therefore, the first nine chapters of the book are assigned and discussed. This leads the discussion up to current typesetters. The other chapters covered include the logic of hyphenation and the specification and actual programming of composition languages. As mentioned above, some material on the latter topics are derived from actual composition language software at III.

Office Automation

The relationship of composition to word processing and office automation is explored, followed by a proper study of office automation and electronic mail. Material for these topics is drawn not only from Seybold, but also from reports from various researchers and companies as Xerox, Stanford Research Institute and Datapro. These reports detail the beginning of electronic mail as a logical progression from the network technology developed in the form of the ALOHANET and the ARPANET.
The Realities

In this second part of the paper the main notes to the class are detailed along with a discussion of the actual progress of the class. It is here that the discovery is made that concepts clear to one with prior understanding or experience are not made familiar to the novice by intimation alone. I hope that statement becomes clear as we progress...
TECO

The introduction to TECO was accomplished mainly by example. Such conventions as how to use the escape key (as a delimiter and command invoker) and its visual representation (a dollar-sign) are given. Four main genres of command type are then given: Insertion, Type-out and Pointer manipulation, Deletion and Search. An example page of FORTRAN code used for demonstrating these commands was xerographed and distributed to the class as well as a copy of the TECO pocket guide (a mini reference manual).

The lecture content up to this point did not seem to challenge the class, as it appeared to be an editor with simple commands. It was after the introduction of TECO as a "programming" language that some interest was demonstrated in the form of probing questions (as opposed to "Is it true that this effects...?" questions). TECO as a text processing language is the most powerful I have seen. Given enough insight and/or patience, almost any task can be accomplished. (Note that I didn't say quickly --TECO is an interpreter.) For instance, TECO will return a flag as to whether a file exists on the disc and then can be persuaded to use that file as a data file or as a TECO program to be executed. However, it would have taken much longer to thoroughly teach all the capabilities of the language than was time for. Therefore, with a simple but challenging assignment in mind, I covered many of the more powerful constructs (including macro generation and execution). The pith of this follows.
TECO is a character-oriented editor and the character "." signifies the current cursor address. A "Q-register" is a place of storage and can store two data types: integers and text. It may act as holding either, but cannot be thought of as holding both simultaneously. When used for storing an integer value, it is loaded with that value by preceding the letter "U" with an expression that evaluates to an integer. For instance, .U1 loads Q-register "1" with the address of the cursor. Q-register names are any digit or letter of the alphabet. In the same manner 0U1 zeros Q-register 1. Other numeric operations of use with Q-registers are % and =. The percent operator increments the following expression while = types the value of an expression out to the terminal. Thus %A would increment the contents of Q-register "A" and QA= would type out the value of Q-register "A". A simple "QA" returns the value within Q-register "A". Q-registers can also hold text. One easy way to load one with text is to position the cursor "in front of" the text to be stored away and type nX*, where n is the number of lines to be stored (beginning with the current cursor position) X is the operator, and "*" is a Q-register name. In order to retrieve this text, position the cursor at the point you want the text and type G* where * is a Q-register name. The contents of the Q-register will be inserted between the two characters addressed by the cursor (the cursor's address may be thought of as "how many characters that precede the cursor in the buffer", and the cursor is always "in between" characters). If a G command is issued and the
buffer has an integer stored in it, an error message is generated. Another item of interest is the "nA" expression. This returns the equivalent ASCII code (in the prevailing radix, default is decimal) of the .+n+1th character in the buffer. For example, a "0A" return the character code of the character immediately following the cursor, while a "+1A" will return the value of the character immediately preceding the cursor. Now, there are two ways to create "macros" in TECO. A "macro", by the way, is somewhat equivalent to a procedure in an algorithmic language. The first appears in the form of a "command loop". This, in turn, is usually employed in two forms. The first is easier: n{command string}. This syntax (including the braces) causes the string of commands within the braces to be executed "n" times. Unfortunately, braces won't print on the printer used for this paper, so parentheses are used in their stead. A more general form of this is: <string1 n; string2>string3. The three strings are commands with the following interpretation: string1 is executed and hopefully generates a numeric argument to the semi-colon. The semicolon causes control to pass to string2 if n negative. After string2 has been executed, control is passed to string1 again. In the event the numeric argument evalutes to a non-negative integer, control is passed to string3. This loop is then referred to as a "conditional command loop" (by virtue of the semi-colon evaluation.) Now, if a search (for a string, of course) fails while in a command loop, it generates a zero for use by a semicolon that may follow it; or a -1 if the
search succeeds. So \(<SFOO$;0tt> will search for an occurrence of the string "FOO". If it finds one, it will type out the line on which it is found, and begin the command loop again; if not, it exits the loop. Command loops may be nested.

The other form of macro looks more like a program:

\begin{verbatim}
!TAG! anything between exclams is a label or comment
Otag$ unconditional go-to; transfers control to label !tag!
exp"#string1'string2 Expressions exp is evaluated.
# is an L, E, N, G. So if
# is L, string1 is executed,
then string2 if exp is less-
\hline
than zero.
E = equal to zero, N = non zero
G = greater than zero.
If condition not met, control is passed immediately to \\
\end{verbatim}

Example

```
Macro Body Running Comments
!START!
-10u1 Load Q-reg "1" with a -10.
!LOOP!c "c" moves the cursor up one
%1 character and the %1 incre-
\hline
ments the contents of Q-reg "1".
Q1"L OLOOP$' The contents of Q-reg "1" are
!DONE! used in the " evaluation.
```

Spaces and carriage-returns are ignored in the macro body.

If the content of Q-reg "1" is less-than zero, control returns to label !LOOP! via the "O" unconditional jump command, otherwise (when contents are .GE. zero) the "Oloop$ command is skipped and control passes to label !DONE! (a comment, but can be the next command). This simple macro has the effect of moving the cursor ten characters to the "right". The same macro can be coded
using a command loop: \(-10U1\langle%1Q1;\rangle\).

Another useful variable is "Z". "Z" always contains the address of the last character in the buffer. In other words, it is the number of characters in the buffer. Some useful constants are the ascii values of carriage-return and linefeed, decimal 13 and 10, respectively. The "Z" variable can be useful in terminating macros. For instance, \(<C.-Z;\rangle\) will move the cursor "to the right" until it is moved past the last character in the buffer. At that point, the loop is exited. The same may be coded as:

\[!\text{LOOP!} \cdot -Z"E\text{OFINISH}$'\text{COLOOP}$!\text{FINISH!}.\]

The class also was told (in the first lecture) than the "L" command would move the cursor to the beginning of the next line. If the user is on the last line, the "L" command will position the cursor past this line.

A macro may be executed by typing its body followed by two alt-modes. Since it is usually easier (and more prudent) to develop programs with an editor, then "execute" it, another method for executing macros is given. As was mentioned above, it is possible to "load" a Q-register with text. A Q-register may then be "executed" by typing Mq where q is a Q-register name. Therefore, we may develop a macro in TECO's editing buffer then load a Q-register with the macro body for later execution. Macros then become, effectively, subroutines. So, to load a Q-register with the above macro, we just type

\[!\text{LOOP!} \cdot -Z\text{EOFINISH}$'\text{COLOOP}$ \text{!FINISH!}$$. Not quite. We cannot insert alt-modes using the insert command since
alt-mode is the delimiter. One way to insert an altmode is to type 27I. This is a TECO command that inserts the ASCII value preceding the "I" into the buffer, and "27" is the decimal ASCII value of an altmode.

Given this knowledge, the class was assigned 1) to create a text file of ten lines or so using TECO, then 2) to write a TECO macro to count the number of lines in the editing buffer. They could write any kind of macro they wished, so long as it worked. I instructed them to get the macro into the editing buffer in text form, and to have the file of text (to be counted) available on disk. To save further nausea on their part, I gave them an incantation that would load their macro into a Q-register, then load the text file into the editing buffer. They were to turn on the hard copy to the terminal then type the following commands:

```
HT$$   !Type out macro body!
HXA$   !Load Q-reg A with macro!
HK$    !Clear buffer!
ERFOO$ !Find file FOO on dsk!
Y$     !Load file into buffer!
HT$    !Type contents of file!
MA$    !Execute macro in Q-reg "A"!
```

Their macro was to type out value of the number of lines in the buffer. One acceptable solution would be

```
0u1     !zero out Q-reg 1 for counter !
<s    $;\%1> !search for a literal carriage-return!
Q1=$$   !by typing a carriage-return in the !
        !search string. Increment Q-reg 1 if !
        !found, exit loop if search fails !
```
Other solutions could involve 1) moving through the buffer examining each character and incrementing the Q-register on seeing a carriage-return or on seeing a line-feed (but not incrementing on both), 2) moving through the buffer with the "L" command. Both of the latter versions would need to detect the bottom of buffer by testing the cursor against "Z".
The Editor Assignment

The TECO assignment acquainted the class with some of the options that must be considered when specifying an editor. In explaining the characteristics of this editor/language alternative schemes of specifying TECO were considered. The fact that it is character-addressable locks it into a certain cast already. Characteristics unique to other three other editors were then discussed. One, which I wrote, is a display-oriented character editor (a list of its commands appear in Appendix A). The other two are essentially line-editors. One of the latter was the SOS mentioned in the Objectives, the other an editor written for use under the UNIX system and is described in detail in Software Tools.¹

The class was made aware the first day that they were to write an editor as part of the requirements for the class. They were to hand in an external specification of the editor they would implement. The editor was to provide certain minimum editing capabilities, and written, preferably, in PASCAL. The requirement to write it in PASCAL had its rational in staying clear of FORTRAN and SNOBOL. The reason is the writing of the editor was to teach the class how to build and organize tools for manipulating text. This included searching. If the editor were written in SNOBOL, the task of implementing a searching routine and therefore understanding how searching

¹Kernighan and Plauger, op. cit., pp. 163-217.
is effected, would not be accomplished as the routines are part of the kernel. FORTRAN was foregone for reasons I hope are obvious to the reader. PASCAL has more affinity for data structuring. SIMULA was foregone because I wanted to have a common basis for discussing coding problems. Thus bugs in one compiler are worth discussing in class; allowing two languages would, in my opinion, turn the class into a programming languages class.

In covering the other editors, I emphasized that each convention or feature the implementor allows in the editor molds the other features as yet unspecified. For instance, if a line editor is contemplated, the implementor must anticipate the need to modify characters on a given line. He is faced with the decision of forcing the user to completely retype the line or of allowing the modification only of those characters requiring modification. If he chooses to offer the latter capability, specification and implementation of that command must be consistent with commands already specified.

During these discussions the class exhibited concern in that they had no idea as how to specify an editor: the first milestone assignment in the project. So questions exemplified in the Objectives (concerning editor design) were posed academically. This discourse was interrupted by a deep anxiety on the class's part about writing the editor in PASCAL; only two people in the class admitted knowledge of the language. I decided that I really wanted a common basis of PASCAL in the project, so I agreed to deliver a
few lectures by way of introducing it.
An Aside On PASCAL

In order to spare the class the expense of buying too many books, a copy of Software Tools was put on reserve in the library. I then strongly urged them to buy a copy of Grogono's Programming in Pascal\textsuperscript{1}. This book was augmented with examples of my own design, but for the most part, it gave a clear treatment of the language and examples of valuable routines. For instance, the Dispose routine in our (CDC) particular implementation was unsatisfactory. Grogono provides a routine to test the Dispose in a given implementation, then offers replacement Create and Return routines that manage unique data types (usually record structures). In addition to the Grogono text, I informed the class that there is a documentation file on NOS that outlines the peculiarities of the NOS PASCAL.

PASCAL was introduced to the class by quickly naming the "gross" or categorical features of the language. I.e., that the complete syntax of the language can be represented on two pages using Wirth's diagrams as well as the following:

1. All variables must be declared
2. Keywords (reserved words) are capitalized in the diagrams
3. The semicolon is a statement separator

\textsuperscript{1}Peter Grogono, Programming in PASCAL (Reading, Mass.: Addison-Wesley Publishing Co., Inc., 1978).
4. There are four built-in data types
   a) real
   b) integer
   c) boolean
   d) char(acter)

5. There are also four built-in data structures
   a) array
   b) record
   c) set
   d) file (sequential in standard PASCAL)

6. There is facility for user-defined data types.
   Example: Type iodevice = (diskdevice, printdevice, readdevice, drumdevice)
   The four scalars are the only values the type iodevice can assume.

7. There is dynamic storage allocation (but not for arrays unless defined as a component of a record).

8. Associated with records are typed pointer variables.

9. Labels are unsigned integers and must be declared.

10. There is a CASE statement (but has no "others" label).

11. The FOR statement can take one of two forms:
    FOR id := exp1 TO exp2 DO statement
    FOR id := exp1 DOWNTO exp2 DO statement
    The implication is the "step" is always "1".

12. Procedures and Functions are automatically recursive.

13. Can call by reference or by value.

14. All blocks are named.

15. The boolean operators are AND, OR, and NOT.

The remainder of the PASCAL lecturing was divided into two genres: providing in-depth examples of the above features and pointing up caveats involved in using the NOS PASCAL. Most of the time in covering the first genre involved familiarizing the class with the record type definition facility and set definition and manipulation. I
began the record definition material with an example of setting up doubly-linked lists: one record type defined a node. The data part of the node was to be a character. This was the first opportunity I had to gauge the programming level of the class. They had performed the TECO assignment quite well and I am not sure as to whether it was due to cooperation among members of the class or my hints in delivering the TECO material and assignment. But in dealing with the list example, a number of students asked me to explain what a linked-list was. This, of course, led to a short treatment of linked-lists. The way PASCAL handles file I/O was then covered in detail. I thought the class needed some sort of spring-board into PASCAL. So, in concert with the lecture, I wrote an example program using a PASCAL implemented in Hamburg, West Germany that runs on a DECSYSTEM-10. The program appears in Appendix B.

The program was examined in class. The "program declaration" heading differs from CDC's, and was pointed up in class. Before any assignment was made, we covered characteristics unique to CDC's implementation. First, text files were discussed. It seems NOS interferes a bit with I/O in the following way: any line is stored an an integral number of words at ten characters per word; and an end-of-line condition is represented as 12 bits of zero (two six-bit quantities). Now for the catch: character positions between the last non-blank character on the line and the 12 bits of zero that represent the end-of-line are
padded out with blanks. Under certain conditions this means the user can expect up to eight blank characters before detecting the end-of-line! Second, the string delimiter characters in the CDC implementation are #'s, not apostrophes. Third, the character for defining a pointer variable is not an up-arrow, but an apostrophe. These changes are not out of capriciousness, but result from CDC's limited character set. There are only 64 characters available under the NOS timesharing subsystem -- and there is no up-arrow in it. The character set is given in Appendix C. Fourth, as mentioned above, the CASE statement has no "others" clause, and the compiler is not forgiving in that the program will abort if one of the CASE labels does not match the selector. While this seems to be "standard" PASCAL, it is prudent to note that most, if not all, CASE statements should be enclosed in an "IF" statement for safety.

The thought was raised in class that there is another (simpler?) way to read a string of characters into a linked list. One appends each character to the beginning of the list (thus not requiring a pointer to the last node of the list). When the string is completely read in, one simply reverses the list.

The class was offered its choice of methods in completing an assignment to read the entire contents of a file into memory in the form of linked lists. Each list represents a line and the lines are to be connected with another list. Thus we have a linked list of text lines
where each line is represented by a linked list of characters. Then the "buffer" is to be written to an output file.

The motivation for making the above assignment was twofold: to familiarize the class with the NOS PASCAL and to give them a kernel from which to develop the rest of the editor. In completing the assignment, the class had its input and exit routines.

By this time it was becoming clear that the editor might be implemented using linked-lists for buffer representation. The possibility of using an array was discussed but not encouraged much on the following basis. I consider a Q-register-type facility (somewhere to temporarily store text) a basic component of an editor. The task of memory management seemed less imposing to the class when conceived as lists than as one long array that is not dynamic. Thus a Q-register is loaded by copying lists rather than managing free "core". With this in mind, I tested the NOS PASCAL resident Dispose routine. The test consisted of getting an instance of a PASCAL record then "Disposing" it. The program aborted due to exhaustion of available memory. This is clearly not the action one would desire. Therefore, I mentioned to the class that we should define a separate New and Dispose (under different identifiers) for each record type that is used dynamically by the program; because, among other reasons, the pointer variables receiving pointers to these records are strongly typed. The assignment was trivial as the procedures are
given, as mentioned above, in the Grogono text.

Perhaps the last major topic covered under PASCAL was definition and use of sets. We decided it would be useful to employ the set type in identifying characters retrieved from the terminal. Thus a set "digit" could be defined, as well as "alpha". Unfortunately the NOS PASCAL does not allow the declaration "TYPE CH = SET OF CHAR". This results from the convention of implementing sets as bit strings on 60-bit words. Only one word is used and, sadly, there are 64 characters in the timesharing character set. Therefore, as "illumination" to a lecture on sets and their manipulation, I wrote and handed-out a program that defines and employs sets; it appears in Appendix D.

In discussing features of PASCAL with the class, I was able to fit examples to the discussion at hand, that is, text editors. So when we returned our discussion to specifying an editor the class had gotten over some of its fears of writing one -- but not of specifying one. I offered them the choice of specifying the editor each of them would eventually implement to specifying one en masse and they unanamously chose the latter. In retrospect, this appears to have been the better choice. I was able to turn the class into more of a seminar than a lecture (at least for awhile) by continually asking what may be termed "the naive user" questions. For instance, the user would like to see, from time to time, the current state of all or part of his editing buffer. The class considered from the designer's vantage the implications of allowing the user to
specify "type-out" in absolute and/or relative commands. The same question appeared in terms of the deletion of lines. When we tried for a plurality in specifying the action of the insert command (and the form it should take) the class became embroiled in what can only be termed a "controversy". As the specification progressed, I continually (I think) reminded them that each syntactic construct implied the form of the as-yet-to-be-specified commands; that the user must be spared the complication of non-symmetric syntax; and that definitions of some commands begged the definition of others. For example, it was decided that the user could insert or delete whole lines only. So how was the user to "merge" two adjacent lines? The "merge" command was thus born: it essentially "deleted" the conceptual end-of-line of the first line and made the following line part of the first. The class responded immediately that we needed to specify an opposing command; one that could split one line into two at a user-specified point -- a command that I had not anticipated.

The subject of allowing more than one command per user-input line was rejected as a requirement due to time restrictions. This allowed the class to use a CASE statement for a control structure to the editor (by having a set of legal commands, then dispatching within the case statement). Also, a method for saving and retrieving text in Q-registers was discussed. One simply defined a copy routine that replicated the line and character nodes of
desired lines (for saving) and the same for retrieving (new nodes are obtained, not pointers to data in the buffer).

Later in the semester I handed out a document that reiterated the consensus on the editor specification. It appears in Appendix E along with the RUNOFF source file used to create it. One student requested that I give them longer than a few weeks to complete programming assignments. As the only assignment pending was the editor, I assigned all the capabilities except "Search", "Change", "Q" and "G" to be completed 4 weeks before the final, with the balance due a week before the final.
Pattern Matching

The editor discussion was ended by a discussion of searching techniques. In particular, I described the simple searching algorithm presented by Horrowitz and Sahni. However, I had meant to allow this algorithm as sufficient for our editor and therefore decided code it and test it in PASCAL. I thought the rest of the editor would be challenging enough (for one semester's work). I coded it exactly the way I found it in the book and it terminated with an "invalid pointer reference". This usually means a NIL pointer value. Upon investigation, I discovered I could not use the following construct:

\[
\text{WHILE (NOT(P=NIL) AND NOT(Q=NIL) AND (P'.DATA=Q'.DATA)) DO } \\
\text{BEGIN P:=P'.NEXT; Q:=Q'.NEXT END;}
\]

It seems that PASCAL evaluates the whole predicate, even if P=NIL or Q=NILL. Thus the (P'.DATA=Q'.DATA) will eventually always attempt to utilize a NIL pointer. I left that particular code within a comment in my program FIND in Appendix F for illustration (and in case the above context is vague). I found that I had to write some "ineligant" code in order to get around this shortcoming. As may be seen upon inspection, I attempted to remain as faithful as possible to the original algorithm, but this necessitated my introducing another variable OLD P. This is used to save the value of P (which points into the target string) in anticipation of P being set to nil in the event the pattern and target strings are of equal length. Otherwise, the pointer would be lost. Other "ineligance" is introduced in order to ensure termination of the first loop.
As mentioned above in passing, this procedure is given in its entirety in Appendix F. It assumes two lists: the pattern and the target string. I decided to code the algorithm (for illustrative purposes) so as to assume the pattern to be in an array rather than a list. This appears in Appendix G and is called FINDA. This algorithm differs in one important way from FIND (because the pattern is in an array). We find that yet another piece of information must be supplied — the length of the pattern. The reason becomes obvious when we consider that with such a limited character set, there is no convenient character to use as a delimiter. We can't use a zero or any other non-character data-type as the array is defined as consisting of elements of type "CHAR", and PASCAL enforces this. Thus, any character in the array may be considered part of the pattern. Therefore, actual length of the string is passed as a parameter to FINDA. In other respects FINDA is similar to FIND (and in fact, simpler).

In Fundamentals of Data Structures, algorithm FIND is followed by another algorithm NFIND which exhibits better characteristics (in terms of number of characters examined to achieve a match). As mentioned in the Objectives the algorithm compares the corresponding last character in the strings before starting the actual loop. Of course, this algorithm can exhibit behavior as bad as FIND, and this was mentioned to the class.
The TECO algorithm was discussed next, with an emphasis on the preprocessing of the data in order to achieve generality and speed. The user is able to build a pattern string that allows specification of special match characters. In the following table a control character is represented as two characters, a ' followed by the character to be struck while holding down the control key. For example, a control "X" is represented as 'X. Some of the match characters consist of a control character followed by another character, a few follow:

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>T X</td>
<td>accept any character in this position</td>
</tr>
<tr>
<td>T S</td>
<td>accept any separator. (punctuation, tabs and spaces)</td>
</tr>
<tr>
<td>T EA</td>
<td>accept any alpha (of the 52 upper+lower)</td>
</tr>
<tr>
<td>T ED</td>
<td>accept any digit</td>
</tr>
<tr>
<td>T ES</td>
<td>accept any non-null string of spaces</td>
</tr>
<tr>
<td>T E ch1,...,chn</td>
<td>accept any of the characters specified</td>
</tr>
<tr>
<td>T N</td>
<td>accept any char except the following</td>
</tr>
</tbody>
</table>

The control-N needs explanation. It is used as a modifier to a match specification. For instance, if the search string were 'N'EA (control-N, control-E, A) the meaning is "accept any character in this position if it is not an alpha character. The algorithm is conceptually two staged. In the first stage the pattern array is "compiled"; in the second, the pattern array is used for determining matches. The array may be thought of as two-dimensional. The columns correspond to the character position in the pattern, while the rows determine matching status. The depth of a column is set to 128 (corresponding to the cardinality of the character set). In this stage, the array is set to zero then the first character position is "set". A string-specifying character is read and the
appropriate cells of that column are exclusive-or'd with a 'true'. In non-case-sensitive mode, say an 'a' is read. The ASCII value of the character is used as an index into the column for that character position. The program notes that the character was alphabetic and so subtracts an octal 40 from its value, thus making it upper-case. This value is also used as an index into column one. Thus two elements of the column have been set to true. The program then increments the column index and the second position in the pattern string is set. For an example of a special match-control character (from the table above), let's say a control-X is read. The control-X, as may be remembered, specifies that any character found in this position of the search string is accepted as a match. Thus, all the cells in column two are set to true using the exclusive-OR operator. Obviously, if the character had been a control-D, we would have set only the cells whose row values corresponded to the ASCII values of zero through nine. Using this mechanism, the control-N operator becomes trivial. Upon reading a control-N in setting up the pattern string, all the cells in the column are set with the inclusive-OR operator. After setting these cells, the column number is not advanced. Therefore, when any other arbitrarily complex specifier is encountered, the action of that specifier is to turn off those bits it is concerned with with an exclusive-OR. Our first example would serve well. If the 'a' had been read following a control-N, the effect would have been to set to false the two cells corresponding to upper and lower case 'a'.
The second stage of the algorithm resembles "FIND" above. The algorithm attempts to match the pattern "string" with characters in the buffer. If no match is found, the buffer pointer is moved only one character and the match is tried again. However, this is carried out in a very few instructions as the algorithm is written in a powerful assembly language. A character in the buffer is found to match by loading it into an accumulator then the determination is made in one instruction. The instruction uses the accumulator as an index register into the row in the pattern table corresponding to the ASCII value of the character in the accumulator. It skips if the cell is zero and returns to pick up the next character from the buffer if the cell is non-zero (i.e., the characters matched). If the characters did not match, the process is repeated with the character following the first character in the target string as the new beginning of the target string.

Unfortunately, after the TECO search was covered, there was insufficient time to cover any of the other algorithms mentioned in the Objectives. The time for Text formatting had come.
Formatting

Two format programs are discussed: one from Software Tools and the other, RUNOFF. Actually, the former was modeled on RUNOFF, but is written in RATFOR for a UNIX system. Most of our discussions concerned the former, but RUNOFF was appealed to for justification. As mentioned by Kernighan and Plauger, the purpose of a formatter is to neatly format a document that is pleasing to the eye\(^1\). The "pleasing to the eye" is mine and implies a minimal degree of page layout. The commands implemented in this formatter are characters in length that always follow a period. Some representative commands follow:

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ti</td>
<td>toggles to &quot;fill&quot; mode</td>
</tr>
<tr>
<td>.nf</td>
<td>turns off &quot;fill&quot; mode</td>
</tr>
<tr>
<td>.br</td>
<td>force out current output line and start new line</td>
</tr>
<tr>
<td>.sp</td>
<td>produces a blank line</td>
</tr>
<tr>
<td>.ls n</td>
<td>sets spacing on the page</td>
</tr>
<tr>
<td>.bp n</td>
<td>&quot;begin page&quot; -- skips to top of new page, numbers it &quot;n&quot;</td>
</tr>
<tr>
<td>.ce n</td>
<td>center the next &quot;n&quot; lines</td>
</tr>
<tr>
<td>.ul n</td>
<td>underline the next &quot;n&quot; lines</td>
</tr>
<tr>
<td>.in n</td>
<td>sets left margin</td>
</tr>
<tr>
<td>.rm n</td>
<td>sets right margin</td>
</tr>
<tr>
<td>.ti n</td>
<td>do temporary indent of &quot;n&quot; spaces</td>
</tr>
</tbody>
</table>

Most of the commands are for "status setting" (the setting of margins, etc.). There is also a provision for specifying header and footer lines (meaning a "running head" (foot) that is put at the top (bottom) of each page. The treatment of this genre is quite good, in that the text is exemplifying programming by top-down methodology. It therefore shows the development of some of the more

\(^1\)Kernighan and Plauger, op. cit., pp. 219.
important routines through up to four generations. The page layout code is encapsulated in the output routine in that it detects whether we are at the top of the page or past the bottom (in which case the header is printed) and puts out the current line; then a test is made to see whether the footer should be output. One level down, we find that the routine that puts one more word into the output "buffer" is responsible for detecting whether the word will overflow the measure. If so, the justification routine is invoked. This structure is similar to typesetting programs, as will be later discussed. The justifier is similar to typesetting programs in that all the "spacing" is done before any of the line is output. That is to say, the output buffer is an array within which the text is shuffled until it is justified "in core", and afterwards the contents of the array are output (up to the "linewidth"). Another useful concept pointed up is that of distinguishing between the number of printing characters in the output buffer as opposed to the number of characters. The non-printing characters may be backspaces (for underlining) or later, in typesetting programs, "meta" characters for controlling the typesetter. Thus the user should be sure the total number of characters in his output buffer does not overflow the buffer length.

The concept of the "break" is central to text formatting programs. A break may be issued explicitly or called by another of the routines that comprise the formatter. Its action is to force the output of the line
in the output buffer without justification. Thus, in discovering a "begin page" or "space n lines" command, a call to the break routine is made to finish the previous text and insure the text following appears in the correct place on the document.

It was mentioned above that the underline command accepts a numeric argument (corresponding to the number of lines to be underlined). Also, if we wanted to underline just a few words of a line, we would put these words on a separate line. Since the "underline" command does not cause a break, the words on that separate input line appear on the same line as the text encompassing it. Its implementation is straightforward: immediately after the underline command is parsed, the next line is read into the input buffer. Each character of this buffer is copied into a temporary buffer — but with a backspace and underline character following each character of the input buffer word. Then the whole temporary buffer is copied back into the input buffer and the words are sent to the output buffer by the normal operation of the program.

The centering routine is even simpler: it simply sets a variable "tival" that is sensed by the output routine. Tival is mnemonic for "temporary indent", and the output routine simply outputs "tival" number of spaces before outputting the line.
Probably the main drawback I found with the Software Tools algorithm for justification is that it is tricky and elaborate (the word "tricky" is their own description). Nonetheless, the algorithm was discussed. Immediately afterward, however, I presented the algorithm that RUNOFF uses, which seems, to me at least, more accessible. That algorithm differs from the former in that it adds the "justifying" spaces to the output line as the line is being output (i.e., sent to the output buffer). On discovering the line to be full, RUNOFF calculates two values "EXSPI" and "EXSP2". The first is the number of spaces to be added unconditionally to each interword space in the line, while the latter's interpretation is contingent on whether we are weighting the "white space" to the left or right of the line. That is, the number of extra spaces is alternately added to the right or left of the output lines in order to avoid "rivers" -- gaps of white space on one side of the page or other. So, if we are biasing the spaces to the left, EXSP2 is the number of times we add a space to a "real" output space, otherwise it is a count of the number of interword gaps to pass up before adding an extra space between words. Thus if we have a line with ten words on it, there are 9 interword gaps or "real" spaces on the line. If we need to add three spaces to the line to make it justify, exsp2 will be equal to six if we are biasing the space to the right. Therefore an extra space will be added between words 7 and 8, 8 and 9 and 9 and 10. A flow diagram of the algorithm that was presented to the class appears in Appendix H.
At this point the class was given a midterm. The text of both the midterm and the final appear in Appendix I. I considered it a somewhat easy test, but then again, I wrote it. The results were a bit surprising to me in that I thought people were absorbing the material better than was demonstrated. There were excellent scores: 5 in the very high 80 and 90 per cent range. The highest and lowest scores were 97% and 28% respectively.
This section concerns the business of teaching typesetting -- by both "early machine" and phototypesetter methods. The corresponding chapters covered in the book were 1-9, 12-15 and 21. The content of other chapters (that deal with word processing) are dealt with using other sources.

The area is introduced with a discussion of the difference between typewriters and typesetters. The IBM Executive is an example of "cold type" in that no lead is cast. If any plates are made, it is a result of photographing a typewritten page produced by the Executive, then making offset plates from that. The Executive also introduces the student to the idea of proportional spacing. This is the case in which the machine "escapes" proportionately different for a capital "M", say, than for an "l". The concern here is for the appearance of the text -- type is traditionally designed so that some characters are naturally wider than others and the Executive is a compromise between a typewriter and a typesetter.

Since this was an introduction phase, the chase and its use is also demonstrated (verbally). The chase is a device for locking rows of type close together for the purpose of either making plates or printing directly onto a page.
page. In such a device, all the lines are required to be of the same length. Therefore, if a line is not wide enough to fit in the chase tightly, pieces of lead (called quads) are added. From this came the concept of justifying lines, where the last word of the line appears at the very right edge of the chase. This is accomplished by adding pieces of metal between the words (and sometimes between the letters), and is called "inter word" and "inter letter" spacing. The introduction includes a short comparison between the linecasting machines and the Monotype, but I will postpone such a discussion until they are treated at length.

Probably the most obvious difference between typesetters and typewriters is that of character repertoire. For example, a typewriter has only a hyphen character and it must fulfill the role of the n-dash (used for separating say, dates: 1888-1960) and the m-dash—used for emphasis. Another difference is that of achieving justification more smoothly than a mono-spaced font can. On a mono-spaced device like a typewriter, justification cannot be achieved by distributing small increments of space among each of the inter-word spaces because the smallest and largest increment are the same—1 unit. Thus traditional printing is concerned with the aesthetics and two other attributes: readability and legibility. Readability is the ease of reading a printed page (the type arrangement) and legibility refers to the speed with which each letter or word can be recognized (the type design). And of
course, a printer also has bold and italic fonts to draw upon in order to draw attention to specific phrases.

For purposes of discussion, it should be noted that there are 72 "points" to the inch and one "pica" is equal to 12 points. In detailing some of the characteristics of type, two terms come to mind: the ability to kern and mortise. Kerning is the process of causing two characters to "overhang" one another. For instance, if we wanted the cross of a capital "Q" to underline the next character, we could put both of the characters on the same piece of lead (and thus creating a ligature) or we could cast the "Q" so that the cross overhangs the body of lead it is cast on. Thus any character that follows will fit under the cross. Mortising is another aesthetic nicety wherein two characters that normally would be spaced too far apart if they were to "sit" on their respective bodies are "pulled" together by physically sawing off parts of each of their bodies. This could be desired in the case of a capital V followed by a capital A.

---

*** ←-- ASCENDER
***

---

*** ←-- X-HEIGHT
***

---

*** ←-- DESCENDER
***

---
Referring to the above figure, the point size of type is generally given as the height of the body, where the body is the distance from the top of the ascenders of the type to the bottom of the descender of the type. Note that a lower case "l" may be larger in one font than a lower case "l" of another font of the same "point size". This may occur because the lower case l "sits" on the base line -- and the base line in one font may be in a different relative position than the base line of another. The position of the base line is determined by the x-height of the font, where the x-height is the height of a lower-case x (it has no ascenders or descenders). The larger the x-height of a font, the shorter its descender can be. Unfortunately, capital letters are dependent on the base-line also; so some, in turn, are larger in one font than another of the same point-size. This in turn creates the problem of getting characters of various fonts to "base align", meaning they all sit on the same imaginary line.

This brings up the topic of leading. Leading is the distance between the base lines of running text, in a "solid setting". A solid setting implies there is no extra space added between consecutive lines of text. The term leading derives from the days in which consecutive lines were spaces out by inserting strips of lead between the rows of type. It is used to make a page more legible; the usage is "11 on 13" where "11" refers to the point-size of the type and "13" refers to the actual distance between base-lines. This implies there is "two points of leading"
Another handy term of measurement is the idea of the "em" space. It is the amount of horizontal space that is equal to the point size one is setting in. It derives from the fact that the capital M was traditionally square, and that it is the widest character of the font (normally). An "en" space is either 1/2 an "em" space or 1/3 an "em" space. The "em" concept is very handy in that it always specifies a relative width. When using typesetters that use a relative-width system of escapement (to be defined later) the em-space is the widest character in the font, and all the other characters are defined in terms of the em-space. Thus if there are 18 relative units, an em-space is 18 units wide, while an en-space would be 9 units (if it is defined as 1/2 em).

If that were not confusing enough, the concept of "set" should help us along. It has two definitions. In terms of the design of the original type, it defines the case when the width of the em-space has been narrowed or widened (the height remains the same). Thus if a character is 6pt/8set it means it is 6 points high and 8 points wide. 8pt/6set implies the opposite. 2) In the context of a typesetter, it is the amount of "side bearing" to add or subtract from the character's normal width. Thus we achieve a 6pt/8set appearance by adding two points to the width of each character that is set. With a photocomposition device, the amount can be negative.
First Generation Typesetters

Seybold concerns himself with two genres of typesetter in this category: linecasters and Monotype-style typesetters. The Linotype is covered first.

The Linotype is still used in the United States, and has was the most popular style of typesetter throughout the first half of the Twentieth Century. Its advantages over hand-composition are significant. First, a printer can set over 6000 characters per hour on the Linotype as opposed to 2000 characters per hour by hand. Second, the matricies (type bodies) are automatically recirculated back into the machine's matrix magazine (this implies sorting them into the proper positions). The Linotype is so named because it composes a line at a time, the lead "line" is called a slug. The slugs are "recirculatable" also in that they can be re-melted for reuse. Third, justification is simple: when the operator strikes the space bar, a "spacing" wedge is dropped into the setting channel. When the line becomes within "justification range" (the point at which the line may be justified aesthetically), the operator may command the machine to "set" the line. At this point the machine pushes the spacing wedges causing them to become wider (the wedge consists of two "wedges" that, when placed against each other, form two parallel planes). The spacing wedges thus force the words apart with an equal amount of space between adjacent words, until the line is spaced apart the width of the measure. Immediately afterward, the machine injects lead on top of the matricies to create a lead block.
of type. When a hand-compositor tries to achieve the same effect, he must do it by experimenting with various interword spaces until the correct width is achieved. The last advantage of the linecaster over hand-composition is one of order: blocks of type are easier to keep in order (and are all the same width) than lines comprised of individual pieces of type. The individual pieces are also "lost" for the period of time they are used for the actual printing.

The Linotype has four divisions. The first is the magazine that contain the matrices (a magazine corresponds to a font). Each of the matrices contains a standard character and an Italic of that character (usually). The magazines are interchangeable and thus the operator may have two magazines -- one for roman and one for bold type faces. Then that is the keyboard which releases into the "channel" one matrix per corresponding keystroke. As mentioned above, there is a casting mechanism which injects the lead, and finally the distribution mechanism. This recirculates each used matrix into the magazine by utilizing a special sort key on the top of each matrix. Some linecasters may be fitted with two magazines, and is where the terminology "upper case" and "lower case" originates: the upper case contained the capital letters!

One of the disadvantages of the Linotype is if a correction must be made on a line, the whole line must be recast and possibly the surrounding lines. Also, if a widow occurs, the whole page is sometimes reset to a
tightly set size. Obviously, pages are "pulled" throughout the operation to analyze for errors. With composition software for second- and third-generation typesetters, the detection of a widow is detected before any copy is pulled. Other more obvious drawbacks are that matrices may be mixed-up (they are sometimes dropped into the assembly area of the machine out of order -- a "feature" of the machine.)

The Monotype was introduced a year after the Linotype (1887). This machine differs substantially from the Linotype. It is divided into two main constituents: the keyboard and the caster. The two communicate by paper tape, the caster senses the presence or absence of a hole by blowing air against the tape. In the event air gets through a hole, it blows against a small piston, which in turn actuates the setting mechanism. Because the keyboard is physically separate from the caster, there needed to be a mechanism for determining whether the line was within justification range. This was provided by a mechanical counter which counted the sum of the widths of the characters as they were keyed. There was also a counter which tallied the number of interword spaces that appeared on the line. These counters were used to calculate the number of units required for justification, which number was punched at the end of the line. The caster would read the tape backwards to access the amount of space that needed to be added to each interword space as the line was cast. Justification is achieved by adding discrete amounts of space between words, as opposed to a Linotype.
In order to tabulate all these amounts, there needed to be a counting system to relate all the character widths. A relative system was developed wherein there were so many "units" to the "em" (which is, as may be recalled, the widest character in a font). The inventors of Monotype chose to implement a system in which an "em" was 18 units. Thus, an "en" space would be 9 units. Note that this system will work for any point size, as characters are measured only in relation to their width relative to an "em". This system caused type for the monotype to be designed in specific quanta (in relation to each other, rather than randomly as for the Linotype).

The casting component contains a removeable matrix (in the mathematical sense this time) in that it contains a number of "combs" locked side by side, where a comb is a row of characters. The combs contain 15 characters each, and there are 15 combs, thus 225 characters are available on the machine at one time. Each row contains characters of the same set size, and the range is usually from 5 relative units to 18, although this can be easily customized. Corresponding to a given configuration of matrix is a "normal" wedge. The normal wedge is essentially a "hardwired" table. Although there are no wires involved, given notches on the wedge signify a given set size. Thus the wedge describes the configuration of combs in the matrix. If the user decides to reconfigure his matrix by making one of the rows contain characters of 9 relative units rather than say, 12, another comb that
reflects this data must be inserted into the machine. Another correlation that must be dealt with is that of the code punched onto the tape by the keyboarder and the actual piece of type that appears in that position in the matrix case (a modern equivalent would be changing type "balls" on an IBM selective typewriter).

The actual casting process proceeds in the following manner. The caster reads the tape which causes the matrix case to be moved in a manner such that the proper matrix appears over a "casting" hole. At this point a square tube is fitted to the selected matrix and molten lead is forced into the tube and against the matrix face, causing an impression of the type to be formed. This process is repeated until the line is set.
TTS

Perhaps the most important development after the invention of these typesetters occurred in the 1930's. Walter Morey invented the Teletypesetting code. It was a system for sending typesetting information (to drive typesetters) over telephone lines. It didn't catch on as quickly as would be expected for two reasons. First the newspapers didn't want to be locked into a particular editorial content, and second, the local typesetting unions did not want to lose the local keyboarding work. However, by 1951 UPI began sending pre-justified anyway because some newspapers by that time were using TTS internally to drive their Linotypes. This implies, of course, that the matrices in Linotypes had to have been standardized to the 32-unit-to-the-em system (TTS instituted more accuracy). This was done not by redesigning the type, but by estimating with a micrometer the value of a matrix and rounding to the nearest 1/32 of an em. This caused rounding errors that usually balanced out by the end of the line, but not always. The tapes were punched on a machine that resembled a typewriter called a "multiface perforator". In the very early days, Linotype keyboards were fitted with punches connected to solenoids, so that as the paper tape was read, the keys were punched automatically. Later, of course, linecasters accepted tape directly.
TTS is a descendent of both the Baudot code and the Murray code. The Baudot scheme allows the user to strike a chord on a 5-key keyboard. The cord is fed out serially providing 32 distinct representations. The Murray code used a keyboard that resembled a typewriter. It also produced a 5-level code, but in parallel, much in the style of the modern paper tape. It used 26 codes as the alphabet, and two for line feed and carriage-return codes, respectively. There was also a shift code that acted as a toggle. When in "shifted" mode, the codes became figures. There were no lower-case characters. Morey adapted this system to a 6-level code, thus adding 32 codes, and used it to drive the typesetters described above. He used information theory (wittingly or not) and assigned the characters used most frequently to the representations with the least holes. This minimized tape wear; a space had a one-hole representation. The addition of 32 codes in conjunction with a shift-precedence code allowed upper and lower case characters. In addition, he assigned such typesetting commands as:

thin space
quad left
upper rail
lower rail
em space
rub out
quad center

Upper rail would select the upper portion of the matrix, which would have the roman font as opposed to the lower portion which would contain the italic.
The Advent of Photocomposition

The devices described herein are termed "first generation typesetters". The reason is that while these machines accomplish composition by exposing a photographic medium through a "master" character image, they do not substantially differ from their hot metal progenitors. For instance, they are capable of setting only one line at a time. Then the paper is moved and another line is "set". An example would be the Intertype Fotosetter. This machine has a system of machinery that uses matrices in much the same manner as a Linotype. Each character is on a separate matrix which is "called up" by a keyboard. The matrix contains a piece of photographic film with the image of that character on it. Thus the matrix is suspended in front of a light source, then the light is strobed. The result is a character is "written" onto the film at that spot. The mechanism that contains the light source then moves a distance determined by the "width" of the character (encoded in the matrix by virtue of the width of the matrix) and the process continues. At the end of the line, the camera mechanism measures the height of the accumulated matrices to determine the amount of space needed for justification. All justification on this machine is effected by "letterspacing" -- the padding out of the line by adding space between letters in the words. The Fotosetter provides the capability of various point sizes (from 6 to 36 pts.) by incorporating a system of eight pre-focused lenses. Otherwise, the Fotosetter very
strongly resembles a Linotype (i.e., matrix returning mechanism, etc.).

Another new entry at the time was the Monophoto. It is based on the Monotype. In this case, the matrix-mat (that contains the character images) contains photographic images that are positioned below a lamp using the same machinery a Monotype incorporates. The character image is passed through a few lenses, then imaged onto the photographic medium by travelling mirrors. Thus it is the mirror that determines the position on the film that the character is put.

The AFT Justowriter is typesetter that also evolved from an earlier technology, this time from a strike-on typesetter. Like the Monotype and Monophoto, it consists of a tape-punching part and a tape-reading unit. The tape-puncher punched TTS type codes, while the reader is much like a typewriter. The typewriter part evolved into a phototypesetting unit. It is simpler in workings as there is a "standard" set value for the type, and only one size of type. The imaging is accomplished by shining a light source through characters on a daisy wheel. Incidentally, the wheel stops before the light is strobed.

The first second generation typesetter was the Photon 200, developed in the late 1940's. The Photon 200 images characters through photographic film masters that are arranged in concentric circles around a disc. There are 16 fonts in the eight circles. Thus, a font is selected by
moving the axis of the disc, then the character is strobed onto the medium as it passes the light source. The disc remains in motion during this process. The images may be "sized" by the use of 12 lenses mounted on a rotating turret. One of the innovations developed with this machine is that of the "width card". This was essentially a lookup table for character widths, and, the width cards are changed to reflect the characteristics of the fonts in use.

A competitor to the Photon 200 is the Mergenthaler Linofilm. This device also imaged characters by strobing a light source through a photographic master, but in this case, there are 18 grids of one font each. The grids are mounted on a carousel and, when a font and character are requested, the carousel rotates to the proper grid then all the grid is mechanically masked out except the desired one. The image may then be sized through a resident zoom lens.
Second Generation Principles

There are many similarities in the second generation of typesetters. For instance, all fonts are stored as negative images on photographic film. The manner of storing the font is determined by the manner in which it will be selected. Two steps are involved in the latter process:

1) position the character in front of the light source
2) mask out, somehow, all characters except the one desired

Let's examine some of the attendant pitfalls. If characters are exposed using shutters to mask out unwanted characters, it is not uncommon for the shutters either to stick or to clip off portions of the desired character. In the case where characters are selected by means of moving the axis of a disc, vibrations are often induced, causing a blur of the image. This was avoided by sending "null" characters on the tape following a font change, but this is, of course, an undesireable feature. Then there are objections to characters that are exposed "on the fly" (as with the Photon 200). The claim here is that a faint "tail" can be seen following the character.

There are almost as many schemes for "sizing" characters as there are typesetters. Sizing is the task of reducing or enlarging a character designed for use as a certain point size. The utility is that one need not maintain up to seventy different sizes of a character for each font. Some machines do character sizing by using a
separate image size for each font used, and so use a fixed-size lens (as with some Compugraphic equipment). In this case, only certain sizes are desired, perhaps two or three. The resulting image is excellent, but this type is good mainly for "straight" text, i.e., galleys. One drawback of some of these machines is that one cannot change the font strips without exposing the photographic medium.

Some typesetters have two lenses that may be switched, but they must be changed manually by either unscrewing them or revolving a turret. Aside from these examples are the phototypesetters that can size automatically. These devices use a zoom lens. It is implied that one can set type in the range of 6-72 points, but this is rarely the case. Most machines with zooms can size at specific discrete sizes. To achieve better appearance, some devices maintain fonts in a few size ranges, the lower two ranging from 6-12 pts. and 12-24 pts., respectively, so that they can be sized within that range with good appearance. An important caveat (or another one, I should say) is that many machines can offer this sizing, but not as adjacent characters on the same line: as the characters are sized their apparent base lines do not match. Thus the characters appear to be on different lines! Of course there are also turrets of lenses in some devices that offer better resolution since they are fixed-length lenses. They sometimes include special lenses for distorting the images so as to simulate italic, and others that "squeeze" the
characters into a different set size.

The issue of character registration also received varied treatment. There is a number of alternatives. First, the photographic medium may be moved horizontally, implying that the image is always projected to the same spot. Or, the image assembly may move while the film advances vertically at the end-of-line. Then again, both the image and film may remain stationary. In this case a collimator lens picks up the image and channels it to a moving lens. A variation would be to have a prism or mirror that sweeps the page, laying the images on the medium as it moves. A different method is to use a fiber-optic bundle to convey the image to the spot of exposure.

Character escapement -- the providing of space between successive words or characters, is the last major concern of second generation typesetters. There are two concerns, calculating escapement before sizing the image or afterwards. If the sizing is done before the registration the escapement must be larger or smaller than the true width we were to move, since the character is distorted. In this case, the magnification factor must be taken into consideration in determining the distance to move. If the sizing is done after the escapement, the problem is not quite so bad since the escapement of the true width is magnified along with the character. The escapement values must all be translated from the relative measurement system of ems and ens to the absolute machine units (internal
units), whether they are in parts of a meter or parts of a point. Thus if we had a 9 point font and needed to lay down three units of space, and the machine moved in 100ths of a point, and the system is 18 units to the em, the following calculation would be performed:

\[
\frac{9 \text{ pts}}{1 \text{ em}} \times \frac{1 \text{ em}}{18 \text{ units}} \times \frac{3 \text{ units}}{100 \text{ internal units}} = 150 \text{ IU.}
\]
Third Generation Typesetters

The main distinction between the third and second generation of typesetter is that the third exposes the photographic medium not directly from a photographic character master but from the face of a cathode-ray tube. Some other distinctions are:

1. There are no mechanical parts involved in the selection of the character.
2. The device can manipulate the characters electronically (for sizing and skewing).
3. In most of the devices, there is no need for font masters, the fonts are stored digitally on disc files.
4. They are almost always driven by a computer.
5. Some can compose full pages without moving the paper.

The Linotron 505 may be thought of a "2.5 generation" typesetter in that it scans photographic masters within the machine. The masters are arranged in a grid of 16 plaques with 16 characters per plaque. The character selection is accomplished by producing scan lines in a relatively small patch of space on a crt. These lines are imaged through a grid of sixteen lenses (all in the same plane), through one character position out of each of the 16 plaques (thus selecting 16 characters out of 256; actually, 238) then through 16 more lenses, (again, all in the same plane) onto 16 photo-multiplier tubes. One character is selected from the 16 by accepting input from only one of the 16 PMT's. The impulses from the pmt are fed to a second CRT that sits on a travelling bed. The automation of the scanning is
effected by the carriage sensing calibration lines (1300 per inch); with the sensing of each calibration line, a new scan is begun, thus the characters are laid down with vertical strokes. There can be four grids of 238 characters each; the grids are mounted on a windmill-looking wheel. This device skews characters by skewing the scan lines on the output CRT. Also, sizing is accomplished by causing the output CRT to amplify the stroke lengths by the desired magnification size.

The technique of creating images by scanning photographic masters has some limitations. The images are not as "clean" as they could be because in the scanning of a straight line, the line is detected randomly (the signal oscillates). Also, the "pincushion effect" of the CRT is not adequately compensated for. This is handled, to some extent, by locating characters that distortion that has a minimal effect on (as some punctuation) at the edges of the grids. Another drawback is that the PMT's do not age uniformly, nor do they stay calibrated uniformly, causing some characters to register brighter than others.

The Videosetter is another machine that scans a character grid within the typesetter. In this case, there is one character grid of 106 characters. All the characters are projected onto the face of the "Image dissector tube" (like a television) which does the character selection. It too uses a second CRT for output, but this CRT has a fiber optics bundle fused to its face. Since the fiber optics bundle comes into contact with the
photographic medium, there is no sizing done with lenses. This machine reads the width values for the characters directly off the character grid. Next to each character is a bar code encoding 4 bits. A constant, "3", is added to each value obtained, thus giving the capability to encode a range of 3 to 18. In later versions of this machine, 8 grids were added in turrets to allow larger "on-line" font capability.

Digital phototypesetters differ from the preceding devices in that the character masters are scanned only once (and usually on a different device.) The character data is then stored in memory as "stroking information", or run lengths. Characters are written one at a time by turning the beam on and off for limited duration defined by "on" and "off" edges of the characters.

Characters are usually digitized in three or four size groups at 450 to 900 strokes/inch. The reason is that sizing is accomplished by lengthening strokes and also widening them by increasing the voltage to the CRT. Thus if a character that was digitized at 720 lines per inch is enlarged 50%, it reduces the number of strokes to 540 per inch. Even with thicker lines, character stroking cannot be pushed too far as the roughness can be detected at larger point sizes. We cannot start with an assumed large point size and scale down, as this will cause the character to appear too dense and smudged. Thus it is possible to enlarge an image by 40-60% and reduce it 20-40% effectively.
The early versions of the RCA Videocomp required all fonts to be used to be loaded at the front of the data tape, then into core. If there weren't enough fonts in core and a character of another font were required, a complete font was required to be overlayed from disc. This shortcoming was overcome in the 830.

Autologic developed machines that set the type as the film advanced past a setting "window" in an attempt to speed up the process. These machines had no electronic sizing, so every font was digitized separately. This is not as bad as it seems as the company also stored the character stroking data in a compressed format, thus packing more characters into memory.

A totally different method of producing characters was introduced in the Fototronic CRT. It stored fonts on a fixed-head drum (it held 1024 chars). The characters, however, weren't stored as stroking information, but as patches of dots. As the information was read in from the drum, it was used to direct the movements of the CRT beam in "spraying" the character out. Thus it didn't matter where the initiation of the read took place as long as the cycle lasted only one drum revolution. This machine could size electronically and produce forward or reverse leading in 1/10 point increments (the utility of this will become clear in the discussion of vertical justification).
Most phototype setters receive their character information as predefined strokes -- that is, the "stroking" takes place off line by reading what are called "perimeter" characters. Perimeter characters are simply perimeters formed by co-ordinate pairs. The Metroset typesetter reads perimeter characters from disc and generates stroking patterns as the data is read in. This implies that all the characters, no matter what the point size, have the same stroking density. This is done by assuming the largest point size and throwing away strokes to narrow it down. Another innovation it made was to implement a "look ahead" algorithm that anticipates characters of fonts that are not in core, so that these characters may be read-in in advance for use at the time they are needed. The other major manufacturers soon adopted this technique.

One would think that newspapers would have been quick to use phototypesetters, but they needed to await a technology for making plates from photographic film. Now, however, they are the largest users as they can have all the type fonts on line; they can include half-tones as part of the composition process, and they have better logo versatility. The speed of phototypesetters also allows them to take classified advertisements up to the last minute, thus creating more revenue.
Composition

The Fundamentals of Modern Composition was set using a Comp/Set 500. This particular typesetter is a "direct entry" style typesetter, meaning each line is composed as it is typed in. Before beginning a sequence, the operator must indicate various control parameters:

1. point size
2. font number
3. line length (in picas)
4. leading
5. manual or auto end-of-line decisions
6. minimum interword space
7. maximum interword space (expressed in relative units)

In manual mode operation, the user types until he decides to end the line. The machine aids in the decision by "beeping" when the operator has entered justification range. In automatic mode, the operator continues typing and the machine ends the lines automatically and pushes any word that doesn't fit to the beginning of the next line. The drawback of such a machine is that it cannot redistribute lines that have already been set (for vertical justification). The vertical justification may be desired to move a line from the bottom of the current page to the top of the next.
More sophisticated processing can be accomplished by using a "front end" computer. There are two contexts for this application -- batch and interactive. The interactive system is similar to the Comp/Set machine described above, except the system writes a file for driving a typesetter rather than setting text immediately. This system can also perform sophisticated hyphenation using an exception dictionary. In addition, the user has the option to reset the entire page using other parameters if he so desires or, the file may be sent for typesetting and reused for other contexts. The interactive system relies on the user to "override" hyphenation offered by the system if he so desires. The mechanism is called a discretionary hyphen that, when inserted by the operator, takes precedence over the system's hyphenation.

A batch system is a much larger software package that contains extensive logic for balancing lines on consecutive pages and other considerations. It usually reads files that have been marked up much as a RUNOFF file is, and the imbedded commands direct the software as to which actions to take for the given context. A language called PAGE-3 was discussed in class; its commands appear within square brackets. Thus `tf,1;ps,36` instructs the program to set the type face to "1" (which is an index into a table of font names) and a point size of 36. The commands given to the program are encoded into the binary language that drives a particular typesetter. Each font used by a phototypesetter driven by 8-bit codes reserves seven to
nine of the codes as meta characters for providing escape sequences from the normal text.

The Comp/Set typesetter mentioned above lacks the capability to set characters of widely differing point sizes on the same line. However, for machines as the Videocomp 500, this capability must be provided for in the composition software that outputs code for that machine. Therefore, the software scans the next line in its entirety looking for the character with the largest body leading. This value then becomes the amount the typesetter will advance the CRT beam when it begins writing the new line.

Getting back to justification, a composition package can take one of four main tacks in its end-of-line decision processing. In all cases, words are set into the output buffer until the counter of space left on the line goes negative. The first tack in dealing with the line is to throw away that last word and attempt to justify the line without it. The second tack must be considered if the line would look to spaced-out without the last word: space out the remaining words on the line using inter-character spacing also. A third alternative is to hyphenate. Otherwise the program should go back to previous lines and attempt to reset them tighter or looser in order to make the current line set correctly. The way Page-3 handles this is to keep at least three pages available for resetting at any one time.
The specification of text position on a page is also a capability of batch systems. All typeset material is set into "blocks". A block is an imaginary rectangle that provides the bounds for the text as it is set, and the block must be defined, using the commands within the language, before any type can be set. There may be any number of blocks per page, and the blocks may be positioned anywhere on a page, also with specifiers within the language. Blocks may be named and used multiple times on a page or many pages. Most batch composition systems also allow the user to define macros that, when called, substitute typesetting commands and/or text into the stream. Added to this is the ability to test various of the variables controlling the composition, and conditionally execute statements. Thus the composition language really is a language.
Hyphenation

Mention was made above of hyphenation routines. A hyphenation routine is actually an extension of the justification routine. Most composition of aesthetic quality incorporates hyphenation, but the degree to which one relies upon it is dependent on the application. For instance, newspapers hyphenate the most often, while high-quality books try to minimize it; however, a total lack of hyphenation can be as disasterous as too much of it -- the result is a page with "rivers" of white space in it.

We know that when we hyphenate we do so on the last word of the line. For this purpose, we must define the exact nature of a word. A first approximation would be to define it as any string of non-blank characters with spaces on either side of it. This may be extended to include a hyphen on either side of it also -- the word "micro-computer" may be divided at the hyphen. We can also add the em-dash and sometimes the slash.

Now that we have our word, we find we must "clean it up" even more by stripping the punctuation from either side of it (remember, this is for purposes of discovering a hyphenation point). Another class of characters that must be removed from the word would be those of the composition language that may be imbedded in the word. For example, the user may wish to hyphenate the sub-word "ever" in "forever". The mark-up commands may appear as for \[fn,2]\ever[fn,i] to specify that we switch to font 2 (italic, say) before setting the word "ever". There are
some exceptions, but they must be handled as special cases, for instance, with 6:45 P.M., the P.M. must appear with the 6:45.

There are two main approaches to hyphenation: logical and by dictionary. Often the two are blended. One of the methods of the logical approach is to take a four-character window into the character and checks to see whether a division can be made mid-way. It starts with the last two characters that will fit on the line with the hyphen and looks at the next two characters. This assumes the first break will achieve the tightest fit (a preference of many printers). Thus if the word that is to be divided is "photographer" and the characters "photogr" will fit with a hyphen on the line, we analyze the tetrad "grap". Three questions are asked: 1) can a syllable begin with ap? 2) can a syllable end with "gr"? and 3) may an English word be hyphenated between an "r" and an "a"? The answer to the second question would be no, so the algorithm moves to the left in the word, character by character, posing the same three questions until the answer to all three is "yes".

Some hyphenation routines start at the first character of the word and work to the right, searching for a possible breaking point. This insures that if we take the first breaking point, we find it will either set or it won't, but we needn't keep trying as any other attempts clearly will fail.
Another method analyzes character pairs. For instance, a syllable will never end with the characters bc, bd, bg, bh ... and a syllable will never begin with bb, bc, bf .... Thus for each letter in the alphabet, we keep all the illegal occurrences. This, however, is only useful as a checking mechanism for some general purpose hyphenation routine, as it tells us only what won't work. Hyphenation is more challenging when there is a vowel on either side of a consonant. For instance, basil and basilic both hyphenate in different fashions. One therefore has a 50% chance of being right.

A less storage-intensive method is to use word structure analysis. We merely store a list of prefixes and suffixes around which it is usually acceptable to hyphenate. Then words as reentry, restructure, etc are easily hyphenated. Dis-, in-, mis-, and un- are common prefixes, while -er, -tion, -sion, -ible, -ness are common suffixes. One still runs into trouble with such exceptions as inchworm, inkless, union, and unified. These may be resolved with more logic, but one must stop somewhere if the system is not to be overburdened with hyphenation logic. There is also the unresolvable problem of homographs, words that hyphenate differently for their different connotations. Examples would be minute and produce.

The logic above may be augmented with a dictionary of which there are two types, full dictionaries and exceptions. Thus a word may be hyphenated using the above
logic then that syllable or word may be checked against such a dictionary. If the word appears as the logic hyphenated it, it is wrong and should be tried over. It is not unreasonable, however to maintain a full dictionary on line. Such a dictionary may be searched in one disc access and have a very good chance of having the word. In that case, there would be no need for logic. As mentioned above, the user is able to override hyphenation logic or lookup by keying in a discretionary hyphen. If this is done, the user's hyphenation is accepted without further analysis.
Input To A Typesetter

Because the "turnaround time" on many typesetters includes the developing of the photographic medium, there is a recognized need to insure the perfection of the textual input to the composition program. One way to minimize typo's is to write the equivalent of a TECO macro to look for commonly misspelled words and correct them, an example being "teh" for "the". Such a program could read a driver file that contained all the search and replace options. This is even more cogent when the input has been produced with an OCR machine. The more sophisticated OCR machines can read pages of material consisting of type of various fonts, and some can even read handwritten data. These machines sometimes make "common" mistakes as recognizing an "1" as a "1". Thus one could search for words that have imbedded numbers (assuming the material is not supposed to have that sort of construct) and fix them. Other "format" errors can also be found, as when the machine has inserted a space between the last character of a word and the period following the word. When the input is believed to be correct, it may be submitted to a dictionary program that underlines words that look "suspicious" i.e., that are not recognized. The words may then be scrutinized by a human for final judgement. This added care saves time and chemicals.

The correction of errors in the actual typesetting commands can be more difficult. These errors serve to transform the typesetter or person entering the typeset
commands into a programmer of sorts, since he must debug the commands. Often an incorrect mark-up command causes a typesetter to abort execution in unfathomable ways, i.e., with no meaningful diagnostics. The error can be as simple as entering the column width as ten points instead of ten picas. In such a case, an indention of 2 ten-point ems would cause unresolvable problems.

Clearly, text processing is involved in two seemingly disparate stages, the data exists as "text" in a computer memory, but also as ink on a page. If an error occurs wherein the text is positioned on the wrong part of the page but is otherwise correct, it is more prudent to "paste up" the output or to change the input to the typesetter and run it again? Both methods are employed. In the instance in which we re-run, it may be because the file will be used again to produce the same typeset output (as when the file generates a telephone book). The other method would be used when the job is a one-time-only job or if it is done in a small shop that wants to spare the expense of running the job again.
Pagination

In computer typesetting, pagination is the process of outputting codes which cause a typesetting unit to generate fully composed pages. This stage of the industry is new; in fact, many of the problems are not fully resolved. Looking back, one of the first (and simplest) problems addressed by the industry was that of setting multi-column text. Assume we want to set a page of three columns that are 60 lines deep each. We should also assume that the leading and point-size of the lines don't change too. The procedure would then be to read the input file (randomly if possible) and write the 1st, the 61st and the 121st lines across the top line of the output page. Then the paper is advanced within the typesetter and the 2nd, the 62nd and the 122nd lines of the file are written as the second line of the page. The reason this must be performed in so arcane a fashion is that the typesetter could write only one horizontal line at a time, with no backing-up possible. This scheme worked ok as long as the leading was constant. Now suppose the first and third columns are set as 10 on 12 text (i.e., 10 point letters on 12 point bodies) and the second column is set as 8 on 9. The program must then write the first line, then drop down nine points to write line two of column two. Afterwards, it drops down another three points to write line two of columns one and three and so on.
Another problem that had to be dealt with was that of the serial nature of typesetting commands. For instance, if a column of text is to be set, the leading and point size and font number are all specified, then all the words in that column are set with those parameters. When column two is begun, the appropriate commands precede it and these commands remain in force for the duration of the setting of the column. Now, if the algorithm specified above is used to set three columns, the typesetting commands corresponding to the individual columns will be "lost". Therefore, the program must discover and save the state of each column for future use.

Of course there is a way around all this computing and that is to use a typesetter that can move the paper backwards or a typesetter that can write a whole page at a time. These exist now, but did not exist when the above problem was first addressed. As mentioned above in the discussion of "blocks" for composing, the blocks may be specified to sit anywhere on the page, and therefore the page may be "made up" in that fashion.

Perhaps the one driving force in page composition is the widow. A widow is a line that is quad-left that is immediately preceded by a justified line. Now the rule is that a widow cannot begin a page, and this is sometimes extended to include two or three lines not being able to begin the page. The reason for this is lost in the oral annals of printing. Since a widow cannot begin a page, programs must deal with it in one of three ways. One is to
go back a page or two and reset the text either tighter or looser so as to either pull the widow back onto the preceding page or to push an additional line or two onto the page with the widow, thus making it acceptable. Another method would be to go back to a previous page and perform vertical justification by "carding". The term comes from the printer's term for inserting cards of lead between the lines of print so as to increase the leading. This, of course, pushes more lines onto the widow's page. The other alternative resembles the previous one, and that is concerned with illustrations or photographs. A certain amount of "white" space surrounds an illustration on a page, and this space may be tampered with in order to satisfy a widow condition. The ability to do vertical justification implies the typesetter can advance the page in increments at least as small as one tenth of a point. A similar printer's condition, that facing pages should be of equal depth, is handled in much the same way.

There are therefore a few different ways to do pagination. One mentioned already is in the Page-3 style of specifying positions of blocks from within a batch program. Another would be to arrange blocks interactively with a "page makeup terminal" in which the operator has the ability to move the blocks around at will, then to see the whole page before it is set. Other special-purpose terminals are used in newspaper advertising offices. These allow the user to type in text then to enlarge or reduce it using special keys. The characters are "stick figure"
characters that are meant to give a general idea of the proportions of one type to another and not as the actual representation of the type. These terminals also have the capability to define irregular shapes around or into which text may be "poured".

This completes the treatment of printing and composition. The remainder of the semester was devoted to an introduction to the technology of word processing and office automation. The treatment of these topics from the point of view of their impact on the office personnel is beyond the scope of this course.
Word Processing

The trend of what we now call word processing began with the introduction of the typewriter. The "trend" is that of making scribes more efficient in terms of both accuracy and speed, where a "scribe" is defined in the normal manner. Word processing made an advance in the 1930's with the introduction of the Friden Justowriter. This device punched and read roller-paper (similar to a player-piano) and later, in the 1950's, the medium was changed to paper-tape. These devices provided the capability to merge text from various paper tapes, and could also perform a rudimentary text search.

The term "word processing" was finally coined by IBM in 1964 when it introduced the Mag Tape Selectric Typewriter (MT/ST). This device uses tape cartridges for storing textual information. While the technology was not a radical change from the Justowriter, it offered improvements in storage density (it could store 20 chars/inch vs 11 chars/inch with paper tape), the tapes were recyclable, and the transport machinery (for the tape) was faster. In 1969 IBM introduced the Mag Card Selectric Typewriter (MC/ST). Characters were stored on a magnetic card that could store a minimum of one page of text. Until the introduction of the MC/ST, IBM had little competition.

---

Most of the material on word processing derives from Data Pro Reports on Word Processing, (Delran, New Jersey: "The Data Pro Research Corp., 1979).
in the market, but since then, over 100 companies have entered the word processing arena.

In word processor technology there are two main schemes for providing the function of a word processor: standalone units and shared logic.

Within the standalone genre, there are four types. The first is the standalone mechanical system. It is a keyboard and platen wherein the keyboard is coupled with edit and control logic, internal memory and a magnetic medium recorder -- card, cassette, or diskette. The platen is used (with paper) to generate the text and to perform subsequent editing functions. The final document is output on the same platen. These devices are priced from $5,000 to over $10,000. Standalone display systems are another type within this genre. In this type of system, the editing and input are registered on a display screen while the output (i.e., the final product) goes to a daisy wheel printer or a selectric printer. These systems usually have more internal memory than that required for one page's worth of text, so the user may "scroll" throughout a very long page, and even divide this long page into smaller pages. Some of these systems are user-programmable to provide special functions, and of these, some are capable of being down-loaded from a magnetic medium. These devices sometimes offer a limited data-processing facility, say tallying a column that is flagged for this purpose, but this is more normally found on a shared-logic type of system. These systems usually price around $10,000 to over
$25,000. The third category is really a sub-category of the previous type of system. It is called a standalone "thin window" display system. This device offers a partial- or full-line display (either CRT or gas-plasma). The idea is that the user can see all or part of a particular line in memory. It differs from the first category in that it saves paper, and has a lower purchase price than a full display system. The fourth type covers the low end of word processing technology. It is called the electronic typewriter and is usually an upgrade from a normal typewriter. It has internal memory, but differs from most word processors in that there is no provision for secondary storage.

Many of the above systems are dual-media stations. This is the case in which it has provision for two tapes or diskettes to run concurrently. This facilitates both copying and the merging of "boilerplate" text with address lists. A "boilerplate" being a section of non-varying text, such as for a form letter. Some of the systems also offer a communications interface and the capability to drive a higher-speed printer.

Shared logic word processors are similar to time-sharing in that one minicomputer is used to drive a number of stations, but the CPU is dedicated to word processing and document processing. Document processing contrasts to word processing in that word processing is "word stream generation" in nature while document processing includes software resembling RUNOFF, automatic
abstracting software, and a table description language (for, perhaps, driving a phototypesetter). Some include a "data processing interface". This provides the capability to store data as both binary integers and ASCII. This enables "DP" programs to search text files for needed information, as figures from a current annual report, etc. Typically, up to 12 stations are supported by a central minicomputer (but there exist some that drive up to 35). The stations may share common peripherals such as high-speed line printers, OCR equipment, large discs, computer-type tape drives, network communications and photo-comp interfaces. This stems from the obvious point that there is a non-trivial computer controlling them all. Such systems share not only the logic but the cost as well. Thus offices with high volume may be cost effective: if costs are measured on a per/station basis, the price is $10,000 to $20,000 (from an overall cost of $25,000 to over $150,000). Other advantages are non-trivial: the power of the CPU may be used to all searching of very large files, maintaining page format and pagination over numerous updates; also, the printing of documents may proceed as other text is being input. Most standalone word processing systems have right margin justification, but on shared logic systems there are frequently fairly sophisticated hyphenation routines that use dictionaries. A standalone system normally asks the operator to perform hyphenation. Some advantages of word processors follow:

1. Higher-quality output due to better equipment.
2. Output is error-free.

3. Higher utilization of installed machines (this implies a "word processing" department).

4. Reduced typing time due to the "rub-out" key.

5. Faster output speeds (450 words per minute and higher).

6. Drastic reduction of "repeat" typing.

7. No retyping at all if for photocomp (a re-run case).

In terms of operational capabilities, the gap between word processing and text processing seems to be narrowing quickly. In the following list enumerates some of the features of many word processors.

1. Characters may be displayed as variable-width, thus showing where lines will break.

2. Underscored characters are underscored on the screen or the start and stop control codes for underscoring are visible.

3. Multi-column display format available.

4. Superscripted and subscripted characters appear so.

5. Ability to store often-used text for insertion.

6. Displaying of such format-control characters as line-spacing, pitch, etc.

7. Stored-forms processing where the operator moves from one field to the next by typing the carriage-return.

8. Ability to perform queued-printing, where central CPU manages the print queues.


10. Automatic alignment of numbers.

11. Automatic page numbering, headers and footers.

Office Automation and Electronic Mail Systems

Office automation may be described as the "glue" concept that seeks to incorporate discrete technologies into a tool for establishments (corporations, governmental departments) and groups of establishments (research institutions). Its advent is made possible by the growing capabilities of word processors and/or timesharing computer systems. However, word processors are only element of an automated office; the realization of which would be impossible without the experience gained with computer networks. Thus, technically at least, computer networks, with message software and editors are the basis on which the rest of the system relies. The linking of local intelligence in the form of either editor programs or word processors is realized in the form of electronic message systems (also called computer message systems). There was not enough time to dwell upon the implications of office staff reorganization attributable to office automation (with two lectures left to the semester). There is time enough, however, to describe the tool that has made the transformation to an, if not paperless, then less-paper-oriented office: the electronic message system (EMS). The harbiger of the EMS is, in a way, the ARPANET. This network is a packet-switched network capable of transmitting up to 1K bits in one packet. It is important to stress at this point that an EMS is not simply a network. It provides for temporary storage, if needed, and also performs long-term (up to weeks) monitoring of
messages. The characteristics of EMS's that I will be describing are not of any one particular EMS, but are a collage of MM¹ (a message system written at SRI), HERMES (by BBN), the DMS² message system (MIT), the ETHER-NET (Xerox), and one proposed at Information International.

An EMS is a somewhat misleading term in that the messages are really English text kept in the memory of a computer. In any case, such a system requires both hardware and software for coordination of messages (the hardware being the network and file system). An EMS is used to send messages of any length to other people on the "system". System in this case means a network or networks within which a person or concern may be addressed. In an EMS one can 1) send a file created by an editor; 2) type the message directly to the EMS by running the EMS program. There is an underlying assumption that one needs a terminal to access messages addressed to him: in an automated office, there is normally one terminal associated with each secretary or work station. A message always includes a header of the following form as the beginning of the


Messages are normally appended to a file in the user's file storage area. In some systems, the messages are not duplicated; a program maintains a file of "pointers" in which pointers to recipients are maintained. In this system, when all the addressees have "seen" the message, it is deleted from the system (unless otherwise specified). Messages may be sent to distribution lists and/or specific individuals (self included). Also, on some "in house" (meaning not over an external net) EMS's, messages may have an expiration date; after which the addressees who didn't respond within a set time are made known to the sender. In such a system, messages within the system longer than a week are automatically returned. A message may be "appended" by an addressee. If this occurs, the message is marked as "unread" by all others and recirculated.

Messages are accessed through a program. To make the action of such a program more concrete, I give here some sample commands from MM (SRI)³:

The user may ask for the status of messages in his "mailbox" in a number of ways.

³Cracraft, op. cit., pp. 5-14.
1. Since dd: type titles of all messages since date dd.
2. Subject string: give titles of all messages with "string" mentioned in the subject heading.
3. New: give titles of all messages sent here since program started.
4. Seen: give titles of all messages marked as "seen".
5. Text string: titles of messages with "string" in the text body.
6. Unanswered: titles of unanswered messages.
7. Deleted: messages deleted in this session (they really are deleted on program termination).
8. From string: titles of messages from "string".

Disposition commands follow:
1. Read n: Read message number n (all messages in your box are numbered by the message system).
2. Answer n: EMS responds with TO: The proper response is ALL or sender. EMS then accepts your message either as a file specification or as input.
3. Copy n,filename: Copies message n into file "filename".
4. Forward n,addressee: Forward message n to addressee.
5. Delete n: Delete message n at end of session.
7. Mark : Mark message n as seen.
8. Next : Type the next message.

Now to send a message one types:

MM)send
To: Soloman,McLure@SRI-KL (the network address appears after the "@".)
text, etc.

Why would anyone want electronic mail? Vezza and Broos\textsuperscript{4} say the average business letter costs

<table>
<thead>
<tr>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writer's time</td>
<td>$1.45</td>
</tr>
<tr>
<td>Secretary's time</td>
<td>$1.76</td>
</tr>
<tr>
<td>Postage and paper, etc.</td>
<td>$0.59</td>
</tr>
</tbody>
</table>

The postage and paper would be taken care of with EMS and the first two would be cut substantially. The same study projects that the Post Office delivered 9 billion pieces of mail that were writer-to-reader in nature. The advent of electronic mail to households would cut the costs of this organization substantially.

Then there is a question of privacy -- for the message originator. In business, the secretary of the originator sees the message, as does the receiver's secretary. With electronic mail, this is circumvented.

Ease of access is another point. An executive can access his mail from anywhere there is a terminal -- or carry his own on business trips. This allows him to swiftly answer any pressing business at the "home office".

Other obvious attractions are electronic speed of the messages, the receiver need not be "present" to receive messages, and the computer may be used as a tool for composing and storing the message.

\textsuperscript{4}Vezza, \textit{loc. cit.}, p. 90.
to generate intelligent composition software. Lectures on fonts and normal printing techniques should remain; they provide the reasoning for other composition concerns as kerning and relative character widths.

The time saved could be employed in lectures on such semantic concerns as general macro processing (i.e., in editors, composition languages and general computer languages) and data base publishing. The latter is currently under development by companies like Information International, Inc. for use in composing text shared by composing and general data processing programs. The text would be described by its schema entry in the DBMS in terms of its semantic meaning but also in terms of its part in a document, as a "running head", "paragraph head", or "paragraph text". As mentioned, the software is currently under development, but by the next offering of this class, the details may be more formalized.

It is difficult to restrain the content of this course so as not to attempt lengthy discussions of all branches of Computer Science. For instance, the Office Automation material may be expanded (for the class's edification) to a detailed discussion of packed-switching networks and queueing theory. Once again, some restraint in lecturing on typesetting hardware may allow more time for such discussions, but not enough. The Office Automation topic alone could fill a semester. A sample curriculum, based upon the preceding reflections, appears on the following page.
Professor Ed Fredkin at MIT\textsuperscript{5} suggests additions to a system that supports EMS in the form of a calendar program that reminds the user of pre-defined meetings, access to on-line databases and light computing. Obviously, these are elements that could comprise the automated office.

\textsuperscript{5}Edward Fredkin, The MIT Comprehensive System, (Cambridge: Artificial Intelligence Laboratory, MIT, unpublished).
Conclusion

When the class expressed shock at specifying the editor as part of the project, I was a bit discouraged. This stemmed from the anxiety on my part of using "valuable" class time to specify it by committee. In reflection, however, it was time well spent. The method employed was that of me suggesting a need to offer a certain class of capabilities, say print-out (or display) commands, and asking for commands to supply the capabilities. Members of the class sometimes became embroiled in heated debate in fighting for their preferences. In fighting for supremacy, each group, of necessity, was required to offer counter examples to the proposals of others. This process transformed the class proceedings from one of lecture to one of debate. From these debates came ideas that none of us individually would have offered. For instance, the idea of a Q-register abstract data type was born. This essentially made each of the Q-registers a buffer, and there would be no difference between editing the "buffer" and a Q-register. A simple context switch could provide the generality.

It is appropriate to mention here that some of the ideas developed in those discussions were not assigned due to the size of the overall project; but it was made clear that students would receive "extra credit" if they were to implement any extra features. Some students did so, the favorite feature being the ability to enter multiple commands on an input line.
While the decision to require the class to implement the editor project in PASCAL appears not to have hampered the implementation, it did hamper the flow of the class in that it brought about four or five unplanned lectures. On closer examination, however, it allowed a detailed discussion of one manufacturer's attitude towards text handling. The CDC Cyber 174 is computationally fast, but in the light of, say, Digital Equipment software, its implementation of text-handling tools is not adequate. The particulars are given in the main text. In discussing features of PASCAL we were able to discuss the ability of the language to perform text processing tasks. The "set" data type was found to be useful in creating classes of characters and thereby lessening the amount of code generated.

I think it is a good idea to require the project to be written in one particular language for the reason mentioned earlier: the discussion of implementation problems remains homogeneous. That there need be such discussion at all may be questioned until one realizes many of the students hadn't sufficient programming experience to develop an editor without some direction. That is, there was insufficient knowledge of such standard data structures as linked-lists.

In teaching this course again, I would recommend a condensation of much of the typesetting hardware lectures. However, some knowledge of the hardware, even the earliest versions, is necessary to appreciate the methods developed
A Sample Curriculum For Text Processing

Although the course lasts 15 weeks (45 meetings), one week is left unspecified for expansion.

1. (6 meetings) Introduction
   Introduction to "text processing" through use of TECO as an editor and programming language.

2. (6 meetings) Survey of editors
   The class is to specify an editor. This is followed by discussion of implementation details, including the source language.

3. (5 meetings) Pattern Matching
   Details of simple pattern matching as well as TECO's method. May also cover search-optimization techniques and/or generalized search techniques.

4. (6 meetings) Formatting Programs
   Internal details of text formatting programs. This lays groundwork for the "composition programs" discussion later.

5. (5 meetings) Typesetting
   Includes a survey of typesetting devices (and their capabilities) and a discussion of printing in general (i.e., characteristics of type).

6. (6 meetings) Composition Programs
   A discussion of composition programs comparing batch-oriented systems to interactive systems. Should include discussions on hyphenation techniques, page layout and database publishing.

7. (5 meetings) Word Processing and Office Automation
   A discussion of standalone vs. shared-logic WP systems. Office Automation as a hybrid of word-processing technology and networks. A discussion of such networks as ARPANET and CHAOSNET. Also Electronic Message Systems and how they differ from networks.

8. (3 meetings) Macro Processors
   A discussion of macro processors, their implementation, and their place in text editors and assembly and composition languages.
One point I emphasized and one I feel is important is that of the dual role text processing assumes. On one hand, it requires the logic of complex hardware and software in order to "compose" text onto a page. Printing, and by extension, text processing is therefore an art form that requires intuition and diligence. On the other hand, "text" is a vehicle of information, and so text-processing assumes a dimension of data processing. The database application mentioned above is a case in point. In its purest form, the text stores information that is not easily represented by integers. For instance, a database containing real-estate listings captures information such as owner's requirements for assumption of the title. These requirements are rarely amenable to standard codes. One requirement could be "the buyer must pay 20% down and not remove the rose bushes from the front yard." This is not a standard stipulation that can be represented by a coded number for "20% down" and so must be included in its original form.

Text processing is therefore information theory and calligraphy.
Bibliography


Appendix A

III 1070 EDITOR USER GUIDE COMMAND LIST BY GROUPS
III 1070 EDITOR USER GUIDE COMMAND LIST BY GROUPS

I. CURSOR POSITIONING COMMANDS

- **B** Bottom of page
- **nC** Move n Characters forward (-n reverses)
- **J** Jump to top of page
- **nL** Move n Lines down (-L moves up)
- **nR** Move n characters backward (same as -nC)
- **nU** Move n Lines Up (same as -nL)
- **nSText$** Search current page for 'text' (nth occurrence)
- **nNtext$** Search entire file for 'text' (nth occurrence)

II. SIMPLE TEXT MANIPULATION COMMANDS

- **Atext$** Append 'text' to right of cursor.
- **nD** Delete n characters in current line (n.GT.0 only)
- **Etext$** Exchange 'text' for character
- **Itext$** Insert 'text' to left of cursor
- **nK** Kill n lines including current line (n.GT.0)
- **Otext$** Overwrite character(s) with 'text'
- **M** Merge next line to end of current line

III. Q-REGISTER COMMANDS (% is Q-register A,B,C,D,E,F or G)

- **nX%** Load % with n lines beginning at character
- **QX%** Load % with text between copy-cursor pair "\" and ";
- **Q<** Set left copy-cursor "\" to current character
- **Q>** Set right copy-cursor "\" to current character
- **QZ** Clear copy-cursor marks "\" and ";
- **QK** Kill text between copy cursor marks "\" and ";
- **Q6** Get contents of % and insert to left of character
- **QI%text$** Insert 'text' into %
- **QT%** Type contents of % (follow by typing 'T')

IV. FILE MANIPULATING COMMANDS

- **PN** Turn page
- **PC** Write out entire file, continue running EDITOR
- **PX** Write entire file, EXIT to monitor
- **PA** Append the next page of the file to your buffer and insert a "NOT" L
- **PM** Append the next page of the file to your buffer, inserts nothing extra.
- **PK** Release files and restart editor. Q-registers are preserved.

V. OTHER USEFUL COMMANDS

- **F** Force case change of current character
- **VS** Toggle switch for case-sensitive search
- **VT** Toggle switch for case of TTY
- **T** "****Special--refresh display on screen****"
- **nT** Display n lines on screen, minimum=3

SPECIAL KEYS WHICH ACT AS COMMANDS IN COMMAND MODE
COMMAND KEYS
L L key, Carriage return, line feed, down arrow on keypad
C C key, Space bar, right arrow on keypad
R R key, Rubout, back arrow on keypad
U U key, Up arrow on keypad
I I key, Tab key (inserts tab too)
? ? key, Types this message

A NOTE ABOUT CHARACTER CONVERSION:

The character allows the creation of any character in the output file. During output (letter) is converted to corresponding control character. (number) is converted to the corresponding character in the range of 170-177: i.e. 75 becomes a).
Appendix B
A Sample "Read" Program
(* PROGRAM TO MAKE LIST IN CORE OF A LINE OF TEXT 
   AND THEN TYPE IT OUT TO TTY, UNTIL EOF SEEN *)

TYPE
  NODE = PACKED RECORD (* DEFINE NODE STRUCTURE *)
  NEXT: 'NODE (* POINTER TO NEXT NODE IN LIST *);
  PREV: 'NODE (* POINTER TO PREVIOUS NODE *);
  DATA: CHAR (* ONE CHARACTER OF LINE *)
END;

LINK = 'NODE (* DEFINE POINTER TO THIS TYPE *);

VAR
  LBEG, LEND : LINK (* POINTERS TO BEGINNING, END OF LIST *);
  STATUS : INTEGER (* FOR RETURNING STATUS FROM PROCEDURES *);
  CH : CHAR (* VARIABLE TO READ CHARACTER INTO *);

PROCEDURE READLINE (VAR BEG, EN : LINK; VAR STAT: INTEGER);
 (* READLINE CONSTRUCTS A LINKED LIST AND SETS POINTER 
   VARIABLES BEG AND EN TO POINT TO THE FIRST AND LAST 
   NODES OF A LIST IF EOF IS ENCOUNTERED BEFORE ANY CHAR-
   ACTER, STATUS IS SET TO 1 AND A NIL LIST IS RETURNED, 
   OTHERWISE STATUS=0. THIS ROUTINE READS PAST THE END-OF-
   LINE ON ENCOUNTERING IT AT THE BEGINNING OF THE 
   PROCEDURE. *)

VAR
  P : 'NODE (* A LOCAL POINTER VARIABLE FOR "NEW" *);

BEGIN
  BEG:=NIL; EN:=NIL; STAT:=0; P:=NIL;
  IF EOF(INPUT) THEN STAT:=1 (* RETURN EOF *)
  ELSE BEGIN
    IF EOLN(INPUT) THEN READLN(INPUT) (* GET OVER POSSIBLE EOL *);
    IF EOF(INPUT) THEN STAT:=1 (* IF WAS LAST LINE *)
    ELSE BEGIN
      WHILE NOT EOLN(INPUT) DO
        BEGIN (* CREATE A NEW NODE FOR EACH CHAR *)
          NEW(P); read(input, ch); P'.DATA:=ch;
          IF (BEG=NIL) (* N:B. THERE IS NO *)
            THEN (* CARRIAGE-RETURN NODE *)
              BEGIN
                BEG:=P; EN:=P (* INITIALIZE LIST *)
              END
            ELSE BEGIN
              EN'.NEXT:=P;
              P'.PREV:=EN;
              P'.NEXT:=NIL;
              EN:=P END
            END
        END
      END;
  END;
%$D-%$L-
(* NO DEBUGGING, NO LIST *)
(* PROGRAM TO MAKE LIST IN CORE OF A LINE OF TEXT
AND THEN TYPE IT OUT TO TTY, UNTIL EOF SEEN *)

TYPE
NODE = PACKED RECORD (* DEFINE NODE STRUCTURE *)
    NEXT: 'NODE (* POINTER TO NEXT NODE IN LIST *);
    PREV: 'NODE (* POINTER TO PREVIOUS NODE *);
    DATA: CHAR (* ONE CHARACTER OF LINE *)
END;
LINK = 'NODE (* DEFINE POINTER TO THIS TYPE *);

VAR
LBEG, LEND : LINK (* POINTERS TO BEGINNING, END OF LIST *);
STATUS : INTEGER (* FOR RETURNING STATUS FROM PROCEDURES *);
CH : CHAR (* VARIABLE TO READ CHARACTER INTO *);  
PROCEDURE READLINE (VAR BEG, EN : LINK; VAR STAT: INTEGER);

(* READLINE CONSTRUCTS A LINKED LIST AND SETS POINTER
VARIABLES BEG AND EN TO POINT TO THE FIRST AND LAST
NODES OF A LIST IF EOF IS ENCOUNTERED BEFORE ANY CHAR-
ACTER, STATUS IS SET TO 1 AND A NIL LIST IS RETURNED,
OTHERWISE STATUS=0. THIS ROUTINE READS PAST THE END-OF-
LINE ON ENCOUNTERING IT AT THE BEGINNING OF THE
PROCEDURE.*)

VAR
P : 'NODE (* A LOCAL POINTER VARIABLE FOR "NEW" *);
BEGIN
BEG:=NIL; EN:=NIL; STAT:=0; P:=NIL;
IF EOF(INPUT) THEN STAT:=1 (* RETURN EOF *)
ELSE BEGIN IF EOLN(INPUT) THEN READLN(INPUT) (*GET
OVER POSSIBLE EOL *);
IF EOF(INPUT) THEN STAT:=1 (* IF WAS LAST LINE *)
ELSE BEGIN WHILE NOT EOLN(INPUT) DO
BEGIN (* CREATE A NEW NODE FOR EACH CHAR *)
    NEW(P); read(input, ch); P'. DATA:=ch;
    IF (BEG=NIL) (* N.B. THERE IS NO *)
    THEN (* CARRIAGE-RETURN NODE *)
    BEGIN
        BEG:=P; EN:=P (* INITIALIZE LIST *)
    END ELSE
    BEGIN (* ADD TO AN EXISTING LIST *)
        EN'. NEXT:=P;
        P'. PREV:=EN;
        P'. NEXT:=NIL;
        EN:=P
    END
END;
END;
PROCEDURE WRITELINE (VAR BEG, EN: LINK; VAR STAT: INTEGER);
(* SIMILAR TO READLINE, BUT SIMPLER: RETURNS A "1" IF
PASSED A NULL LIST, OTHERWISE STAT=0. WRITES OUT
THE CHARACTERS OF THE LIST ONE AT A TIME, THEN
OUTPUTS A CARRIAGE-RETURN *)
VAR
POINTER: 'NODE;
BEGIN
STATUS := 0;
IF (BEG = NIL)
THEN STATUS := 1
ELSE
BEGIN
POINTER := BEG;
REPEAT
WRITE(TTY, POINTER'.DATA);
POINTER := POINTER'.NEXT
UNTIL (POINTER = NIL);
WRITELN(TTY);
END;
END;
BEGIN (* BEGIN THE MAIN ROUTINE *)
WHILE (STATUS = 0) DO
BEGIN
READLINE (LBEG, LEND, STATUS);
IF (STATUS = 0)
THEN WRITELN(TTY, 'EOF SEEN') (* HERE IF ERROR RETURN *)
ELSE
BEGIN
WRITELINE (LBEG, LEND, STATUS);
IF (STATUS = 1)
THEN WRITELN(TTY, 'Line is NIL');
END;
END;
END.
Appendix C

CDC Character set
<table>
<thead>
<tr>
<th>ASCII CODE TERMINAL</th>
<th>CORRESPONDENCE CODE TERMINAL</th>
<th>INTERNAL DISPLAY CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR (8-BIT OCTAL)</td>
<td>CHAR (8-BIT OCTAL)</td>
<td>CHAR (7-BIT OCTAL)</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>A</td>
<td>072</td>
<td>: 153</td>
</tr>
<tr>
<td>B</td>
<td>101</td>
<td>A 171</td>
</tr>
<tr>
<td>C</td>
<td>102</td>
<td>C 172</td>
</tr>
<tr>
<td>D</td>
<td>303</td>
<td>D 052</td>
</tr>
<tr>
<td>E</td>
<td>304</td>
<td>E 112</td>
</tr>
<tr>
<td>F</td>
<td>305</td>
<td>F 163</td>
</tr>
<tr>
<td>G</td>
<td>107</td>
<td>G 043</td>
</tr>
<tr>
<td>H</td>
<td>110</td>
<td>H 046</td>
</tr>
<tr>
<td>I</td>
<td>311</td>
<td>I 031</td>
</tr>
<tr>
<td>J</td>
<td>312</td>
<td>J 103</td>
</tr>
<tr>
<td>K</td>
<td>113</td>
<td>K 032</td>
</tr>
<tr>
<td>L</td>
<td>314</td>
<td>L 106</td>
</tr>
<tr>
<td>M</td>
<td>315</td>
<td>M 141</td>
</tr>
<tr>
<td>N</td>
<td>316</td>
<td>N 122</td>
</tr>
<tr>
<td>O</td>
<td>317</td>
<td>O 105</td>
</tr>
<tr>
<td>P</td>
<td>320</td>
<td>P 013</td>
</tr>
<tr>
<td>Q</td>
<td>321</td>
<td>Q 133</td>
</tr>
<tr>
<td>R</td>
<td>322</td>
<td>R 051</td>
</tr>
<tr>
<td>S</td>
<td>323</td>
<td>S 045</td>
</tr>
<tr>
<td>T</td>
<td>324</td>
<td>T 002</td>
</tr>
<tr>
<td>U</td>
<td>325</td>
<td>U 062</td>
</tr>
<tr>
<td>V</td>
<td>326</td>
<td>V 061</td>
</tr>
<tr>
<td>W</td>
<td>327</td>
<td>W 165</td>
</tr>
<tr>
<td>X</td>
<td>330</td>
<td>X 142</td>
</tr>
<tr>
<td>Y</td>
<td>331</td>
<td>Y 147</td>
</tr>
<tr>
<td>Z</td>
<td>332</td>
<td>Z 126</td>
</tr>
<tr>
<td>D</td>
<td>060</td>
<td>O 144</td>
</tr>
<tr>
<td>E</td>
<td>260</td>
<td>E 040</td>
</tr>
<tr>
<td>F</td>
<td>261</td>
<td>F 020</td>
</tr>
<tr>
<td>G</td>
<td>063</td>
<td>G 160</td>
</tr>
<tr>
<td>H</td>
<td>262</td>
<td>H 020</td>
</tr>
<tr>
<td>I</td>
<td>263</td>
<td>I 160</td>
</tr>
<tr>
<td>J</td>
<td>264</td>
<td>J 020</td>
</tr>
<tr>
<td>K</td>
<td>265</td>
<td>K 020</td>
</tr>
<tr>
<td>L</td>
<td>266</td>
<td>L 160</td>
</tr>
<tr>
<td>M</td>
<td>267</td>
<td>M 020</td>
</tr>
<tr>
<td>N</td>
<td>268</td>
<td>N 020</td>
</tr>
<tr>
<td>O</td>
<td>269</td>
<td>O 020</td>
</tr>
<tr>
<td>P</td>
<td>270</td>
<td>P 020</td>
</tr>
<tr>
<td>Q</td>
<td>271</td>
<td>Q 020</td>
</tr>
<tr>
<td>R</td>
<td>272</td>
<td>R 020</td>
</tr>
<tr>
<td>S</td>
<td>273</td>
<td>S 020</td>
</tr>
</tbody>
</table>

† THE OCTAL CODES LISTED FOR ASCII CODE TERMINALS ARE SHOWN WITH EVEN PARITY (NORMAL)
‡ THE OCTAL CODES LISTED FOR CORRESPONDENCE CODE TERMINALS ARE SHOWN WITH ODD PARITY (NORMAL)
§ USE OF THE COLON IN PROGRAM AND DATA FILES WILL CAUSE PROBLEMS. THIS IS PARTICULARLY TRUE WHEN IT IS USED IN PRINT AND FORMAT STATEMENTS.
Appendix D

Set Manipulation Program
(*$D-,L+*)
(* Program to demonstrate a method for defining
sets consisting of characters. Written on a
DEC-10 *)

```
type uletter = 'A'..'Z';
        digit = '0'..'9';

    alset = set of uletter (* upper case chrs *);

var    al,all : alset;
        ch : char;

begin
    al:= 'A'..'C';
    all:= 'X'..'Z';
    ch:= 'C';
    if (ch in al) OR (ch in all) then
        writeln(tty,'ch is indeed in al or all')
        else writeln(tty,'in neither set');

    if ch in 'A','B','C' then begin
        case ch of
        'A': writeln(tty,'was an A');
        'B': writeln(tty,'was a B');
        'C': writeln(tty,'was a C')
        end;
    end;
end.
```
Appendix E

Editor Specification Followed By Its RUNOFF Source File
The consensus concerning editor functions

This editor will be line-oriented; therefore, the smallest addressable amount of text is the line.

The editor is written in PASCAL. As a result, such common file operations as specifying an input file name and an output file name from within the program are not possible (that is, we don't know how to do it, and no one at DIS does either). So for the time being, filenames must be specified at compile time.

The commands to this editor consist of one-character each, followed by its parameters, if any. Any characters in parentheses immediately following the one letter command are given as mnemonic interest and should not be typed as part of the command.

A list of the agreed-upon commands and conventions follows:

1. The cursor may be thought of as an integer ranging from zero (in the event of an empty buffer only) to "L" where "L" is the number of lines in the buffer. Thus the character $, as mentioned below, has the current value of "L".

2. The period character may be used in the stead of a numeric argument to signify the cursor position (i.e., the current line).

3. $ The dollar-sign may be used in the stead of a numeric argument to signify the last line's address.

4. L(list) m Print out line 'm' of the buffer onto the terminal
5. L(ist) m,n
   Print out lines 'm' through 'n' of the buffer onto the terminal

6. P(rint) m,n
   Print onto the terminal the current line and the following m-1 lines if m is positive. If m is negative, print the previous m-1 lines in addition to the current line.

7. P(rint) m,n
   Print onto the terminal the current line and the previous m lines and the following n or n-1 lines, however the implementor decides.

8. T(op)
   This command positions the cursor at the first line of the buffer.

9. B(ottom)
   Position the cursor at the last line of the buffer.

10. D(own)
    Move the cursor to address the line following the current line.

11. U(p)
    Move the cursor to address the line previous to the current line.

12. K(ill) n
    Delete (or "kill") n lines starting at the current line. If n not present, delete one line.

13. K(ill) m,n
    Kill relative to the cursor. So kill the same lines that would be displayed with a P m,n command.

14. I(nsert)
    Insert mode accepts lines that have at least one character on each one. Each such line is inserted above the cursor as it is read. A line with only one space on it is considered a blank line, and the space is not inserted into the buffer: only the line node signifies the presence of a null line. If the line is of zero length on read-in, insert mode is exited.

15. A(ppend)string
    Append mode is similar to insert mode except that lines are added to the buffer following the cursor line.
16. **M(erge)**
Concatenate the line following the cursor line to the end of the cursor line.

17. **S(earch)text**
The search command attempts to find a match of pattern "text" in the buffer by checking each line, commencing with the line after the cursor line. No matches are possible around end-of-lines. No delimiter is needed.

18. **E(nd)string**
The end-of-line command searches the current line for string "string" and in the event of a successful search, it breaks the line following the string. Breaking the line is defined as making the remainder of the line a new line that follows the current line. In the event the string comprises the last characters on the line, the new line is blank. Thus if we were to type "Ethe " to the program for the line on which "Ethe " appears above, the new line would begin with "program".

19. **C(hange)/string1/string2/**
The change command is valid only for the current line. The character "/" used above signifies any delimiter character, being by convention the first non-blank character following the "C" command. The usage is to find the first occurrence of string1 and replace it by string2.

20. **Qxn**
This command puts the cursor line and the next n-1 lines following into Q-register "x" where "x" is one of the 26 letters of the alphabet. It's implementation should put a copy of these lines into the Q-register, not a pointer to these lines. The usage could be to save the content of the lines then kill the lines from the buffer. The lines can be re-inserted into the buffer at a different position with the "G" command. The "n" is optional.

21. **G(et)xn**
This command inserts a copy of the contents of Q-register "x". If "n" is present, this command is repeated so as to get n copies in all. The "n" is optional.

22. **X**
The eXit command causes the contents of the buffer to be written to the output
file and the program terminated.

23. Z(ed)
The Zed command causes the program to be terminated without writing the contents of the buffer to be written to the output file. This command is useful if a drastic mistake has been wrought, and the best course of action is to restart editing from scratch.

24. H(elp)
Last but not least is the help command. Typing an 'H' to the program causes a brief list of these commands to be output to the screen for the user's benefit. The description of each command should be worded in the spirit of refreshing the user's memory of available commands and not as a tutorial.
The consensus concerning editor functions

This editor will be line-oriented therefore, the smallest addressable amount of text is the line.

The editor is written in PASCAL. As a result, such common file operations as specifying an input file name and an output file name from within the program are not possible (that is, we don't know how to do it, and no one at DIS does either). So for the time being, filenames must be specified at compile time.

The commands to this editor consist of one-character each, followed by its parameters, if any. Any characters in parentheses immediately following the one letter command are given as mnemonic interest and should not be typed as part of the command.

A list of the agreed-upon commands and conventions follows:

- The cursor may be thought of as an integer ranging from zero (in the event of an empty buffer only) to "L" where "L" is the number of lines in the buffer. Thus the character $, as mentioned below, has the current value of "L".

- The period character may be used in the stead of a numeric argument to signify the cursor position (i.e., the current line).

- The dollar-sign may be used in the stead of a numeric argument to signify the last line's address.

Print out line 'm' of the buffer onto the terminal
Print out lines 'm' through 'n' of the buffer onto the terminal.

Print onto the terminal the current line and the following m-1 lines if m is positive. If m is negative, print the previous m-1 lines in addition to the current line.

Print onto the terminal the current line and the previous m lines and the following n or n-1 lines, however the implementor decides.

This command positions the cursor at the first line of the buffer.

Position the cursor at the last line of the buffer.

Move the cursor to address the line following the current line.

Move the cursor to address the line previous to the current line.
Delete (or "kill") n lines starting at the current line. If n not present, delete one line.

Kill relative to the cursor. So kill the same lines that would be displayed with a P m,n command.

Insert mode accepts lines that have at least one character on each one. Each such line is inserted above the cursor as it is read. A line with only one space on it is considered a blank line, and the space is not inserted into the buffer: only the line node signifies the presence of a null line. If the line is of zero length on read-in, insert mode is exited.

Append mode is similar to insert mode except that lines are added to the buffer following the cursor line.

Concatenate the line following the cursor line to the end of the cursor line.

The search command attempts to find a match of pattern "text" in the buffer by checking each line, commencing with the line after the cursor line. No matches are possible around end-of-lines. No delimiter is needed.
E(nd)string
.1m 10
.rm 52
The end-of-line command searches the current line for
string "string" and in the event of a successful search,
it breaks the line following the string. Breaking the line
is defined as making the remainder of the line a new line
that follows the current line. In the event the string
comprises the last characters on the line, the new line
is blank. Thus if we were to type "Ethe#" to the program
for the line on which "Ethe#" appears above, the new line
would begin with "program".
.1m 0; .rm 59
.1m 9
.1e
C(hange)/string1/string2/
.1m 10
.rm 52
The change command is valid only for the current line.
The character "/" used above signifies any delimiter
character, being by convention the first non-blank
character following the "C" command. The usage is to
find the first occurrence of string1 and replace it by
string2.
.1m 0; .rm 59
.1m 9
.1e
Qxn
.1m 10
.rm 52
This command puts the cursor line and the next n-1 lines
following into Q-register "x" where "x" is one of the 26
letters of the alphabet. It's implementation should put
a &copy of these lines into the Q-register, not a
pointer to these lines. The usage could be to save the
content of the lines then kill the lines from the buffer.
The lines can be re-inserted into the buffer at a
different position with the "G" command. The "n" is
optional.
.1m 0; .rm 59
.1m 9
.1e
G(et)xn
.1m 10
.rm 52
This command inserts a copy of the contents of Q-register
"x". If "n" is present, this command is repeated so as
to get n copies in all. The "n" is optional.
.1m 0; .rm 59
.1m 9
.1e
X
.1m 10
.rm 52
The eXit command causes the contents of the buffer to
be written to the output file and the program terminated.
.1m 0; .rm 59
.1m 9
The Zed command causes the program to be terminated without writing the contents of the buffer to be written to the output file. This command is useful if a drastic mistake has been wrought, and the best course of action is to restart editing from scratch.

Last but not least is the help command. Typing an 'H' to the program causes a brief list of these commands to be output to the screen for the user's benefit. The description of each command should be worded in the spirit of refreshing the user's memory of available commands and not as a tutorial.
Appendix F
"FINDL" Pattern-Matching Program
TYPE
  NODE = PACKED RECORD
    (* DEFINE NODE STRUCTURE *)
    NEXT: 'NODE (* POINTER TO NEXT NODE *);
    PREV: 'NODE (* POINTER TO PREVIOUS NODE *);
    DATA: CHAR (* ONE CHARACTER OF LINE *)
  END;
  LINK = 'NODE (* DEFINE POINTER TO THIS TYPE *);
VAR
  LBEG, LEND : LINK (* POINTERS TO BEGINNING AND END OF LIST *);
  POS, PATPNT, PATHEAD : LINK;
  STATUS : INTEGER (* STATUS FROM PROCEDURES *);
  CH : CHAR (* VARIABLE TO READ CHAR INTO *);
PROCEDURE READLINE (VAR BEG, EN : LINK; VAR STAT: INTEGER);
  (* READLINE CONSTRUCTS A LINKED LIST AND SETS POINTER VARIABLES BEG AND EN TO POINT TO THE FIRST AND LAST NODES OF A LIST IF EOF IS ENCOUNTERED BEFORE ANY CHARACTER, STATUS IS SET TO 1 AND A NIL LIST IS RETURNED, OTHERWISE STATUS=0. THIS ROUTINE READS PAST THE END-OF-LINE ON ENCOUNTERING IT AT THE BEGINNING OF THE PROCEDURE. *)
VAR
  P : 'NODE (* A LOCAL POINTER VARIABLE FOR "NEW" *);
BEGIN
  BEG:=NIL; EN:=NIL; STAT:=0; P:=NIL;
  IF EOF(INPUT) THEN STAT:=1 (* RETURN EOF *)
  ELSE BEGIN
    WHILE NOT EOLN(INPUT) DO BEGIN
      (* CREATE A NEW NODE FOR EACH CHAR *)
      NEW(P); read(input, ch); P'.DATA:=ch;
      IF (BEG=NIL) (* N.B. THERE IS NO *)
      THEN (* CARRIAGE-RETURN NODE *)
      BEGIN
        BEG:=P; EN:=P (* INITIALIZE LIST *)
      END
    END
  END
END
PROCEDURE FIND(S:LINK;PAT:LINK;VAR I:LINK);

(* IMPLEMENTATION OF ALGORITHM "FIND" FROM
HOROWITZ AND SAHNI. FIND IN STRING S THE FIRST
OCCURRENCE OF STRING PAT AND RETURN I AS A POIN-
TER TO THE NODE IN S WHERE PAT BEGINS; OTHERWISE
RETURN I AS NIL. *)

VAR SAVE,Q,P : LINK;

BEGIN
IF ((PAT=NIL)OR(S=NIL))
THEN I:=NIL
ELSE (* EXECUTE REST OF ROUTINE ONLY IF NOT NIL *)
END

BEGIN
I:=NIL (* MAKE SURE I IS NOT VALID ON ENTRY *);
P:=S;
REPEAT
SAVE:=P; Q:=PAT (*SAVE THE STARTING POSITION *);

(* COMMENT OUT DEFECTIVE EXAMPLE CODE
WHILE ((NOT(P=NIL))AND(NOT(Q=NIL))AND
(P'.DATA = Q'.DATA)) DO
BEGIN THIS CODE IS INCLUDED AS AN
EXAMPLE--IT DOESN'T WORK;

END;
*)

WHILE NOT ((P=NIL)OR(Q=NIL)) DO
BEGIN
IF (P'.DATA=Q'.DATA)
THEN
BEGIN
P:=P'.NEXT; Q:=Q'.NEXT;
END;
ELSE P:=NIL;
END;

IF (Q=NIL)
THEN

BEGIN
  P:=NIL; I:=SAVE (* SUCCESS! P IS SET TO NIL TO STOP LOOP *)
END
ELSE P:=SAVE'.NEXT (* OTHERWISE CHECK STRING BEGINNING AT NEXT CHAR *);
UNTIL (P=NIL);
END (* OF ELSE ASSOCIATED WITH "IF ((PAT=NIL)OR(S=NIL))" *);
END (* OF PROCEDURE 'FIND' *);
BEGIN (* BEGIN THE MAIN ROUTINE *)
NEW(PATPNT) (* CONSTRUCT A PATTERN IN A LIST *);
PATPNT'.DATA:='F';
PATHEAD:=PATPNT;
NEW(PATPNT);
PATPNT'.PREV:=PATHEAD; PATHEAD'.NEXT:=PATPNT;
PATPNT'.DATA:='0' (* DONE CONSTRUCTING PATTERN *);
READLINE (LBEG,LEND,STATUS);
IF (STATUS 0) THEN WRI
ELN(TTY,'EOF SEEN') (* HERE IF ERROR RETURN *)
ELSE BEGIN
  FIND(LBEG,PATHEAD,POS);
  IF NOT(POS = NIL) THEN WRI
ELN(TTY,'FOUND IT')
  ELSE WRI
ELN(TTY,'NOT FOUND');
END;
END.
Appendix G

"FINDA" Pattern-Matching Program
(* FILE FINDA.PAS -- ALGORITHM 'FIND' USING AN ARRAY FOR THE PATTERN STRING *)
(* note that a "'" replaces an up-arrow in this listing as that character is not on print-wheel of the diablo *)

CONST
  MAXPAT = 50;

TYPE
  NODE = PACKED RECORD
    (* DEFINE NODE STRUCTURE *)
    NEXT: 'NODE (* POINTER TO NEXT NODE *);
    PREV: 'NODE (* POINTER TO PREVIOUS NODE *);
    DATA: CHAR (* ONE CHARACTER OF LINE *)
  END;

  LINK = 'NODE (* DEFINE POINTER TO THIS TYPE *);

  PATTERN = ARRAY [1..MAXPAT] OF CHAR;
  MAXLEN = 1..MAXPAT;

VAR

  LBEG, LEND : LINK
    (* POINTERS TO BEGINNING AND END OF LIST *);

  PATARR : PATTERN;

  PATLEN : MAXLEN;

  POS, PATPNT, PATHEAD : LINK;

  STATUS : INTEGER
    (* RETURN FROM PROCEDURES *);

  CH : CHAR
    (* VARIABLE TO READ CHR INTO *);

PROCEDURE READLINE (VAR BEG, EN : LINK; VAR STAT : INTEGER);

(* READLINE CONSTRUCTS A LINKED LIST AND SETS POINTER VARIABLES BEG AND EN TO POINT TO THE FIRST AND LAST
 NODES OF A LIST IF EOF IS ENCOUNTERED BEFORE ANY CHARACTER, STATUS IS SET TO 1 AND A NIL LIST IS RETURNED,
 OTHERWISE STATUS=0. THIS ROUTINE READS PAST THE END-OF-LINE ON ENCOUNTERING IT AT THE BEGINNING OF THE
 PROCEDURE. *)

VAR

  P : 'NODE (* A LOCAL POINTER VARIABLE FOR "NEW" *);

BEGIN

  BEG := NIL; EN := NIL; STAT := 0; P := NIL;

  IF EOF (INPUT)
  THEN STAT := 1 (* RETURN EOF *)
  ELSE

  BEGIN

    IF EOLN (INPUT)
    THEN READLN (INPUT) (* GET OVER POSSIBLE EOL *);

    IF EOF (INPUT) THEN STAT := 1 (* IF WAS LAST LINE *)

  ELSE BEGIN

    WHILE NOT EOLN (INPUT) DO

    BEGIN (* CREATE A NEW NODE FOR EACH CHAR *)

      NEW (P); read (input, ch); P'.DATA := ch;

      IF (BEG = NIL) (* N.B. THERE IS NO *)
      THEN

        BEGIN (* CARRIAGE-RETURN NODE *)

          END;

        END;

      END;

    END;

  END;

END;
BEGIN := P; EN := P (* INITIALIZE LIST *)
END
ELSE
BEGIN (* ADD TO AN EXISTING LIST *)
EN'.NEXT := P;
P'.PREV := EN;
P'.NEXT := NIL;
EN := P
END
END; END(* OF EOF ELSE *);
END(* OF EOF ELSE *);

END (* OF PROCEDURE READLINE*);

PROCEDURE FIND(S: LINK; PAT: PATTERN;
VAR I: LINK; PLENGTH : MAXLEN);
(* AN ALTERNATE IMPLEMENTATION OF ALGORITHM "FIND" FROM HORROWITZ AND SAHNI.
FIND IN STRING S THE FIRST OCCURRENCE OF STRING PAT
AND
RETURN I AS A POINTER TO THE NODE IN S WHERE PAT
BEGINS;
OTHERWISE RETURN I AS NIL. *)

VAR
SAVE, P : LINK;
Q : 1 .. MAXPAT;
BEGIN
IF ((PLENGTH = 0) OR (S = NIL))
THEN I := NIL
ELSE
BEGIN
(* ATTEMPT A MATCH STARTING WITH BEGINNING OF EACH
STRING IF NO MATCH, TRY TO MATCH PAT(1) WITH
STRING(2) AND SO ON *)
I := NIL (* SET TO NIL IN CASE WE WERE PASSED A
NON-NIL VALUE *);
P := S;

REPEAT
SAVE := P; Q := 1 (*SAVE THE STARTING POSITION *);
WHILE NOT ((P = NIL) OR (Q > PLENGTH)) DO
BEGIN
IF (P'.DATA = PAT[Q])
THEN
BEGIN
P := P'.NEXT;
Q := Q + 1;
END
ELSE P := NIL;
END;

IF (Q > PLENGTH) (* CHECK TO SEE IF WE EXAUSTED
STRING OR MATCHING FAILED*)
END;
THEN
BEGIN
P:=NIL; I:=SAVE (* IF SO, WE HAVE A MATCH *)
END
ELSE
BEGIN
P:=SAVE'.NEXT (* OTHERWISE, CHECK STRING BEGINNING WITH NEXT CHAR *)
END;
UNTIL (P=NIL);
END (* OF THE ELSE ASSOCIATED WITH 'IF ((PLENGTH=0)OR(S=NIL))' *
);
END (* OF FIND *
);
BEGIN (* BEGIN THE MAIN ROUTINE *)
(" ARTIFICIALLY LOAD PATTERN ARRAY WITH A STRING ")
PATLEN:=3 (* SEARCH TARGET STRING FOR 'FOO' *
);
READLINE (LBEG,LEND,STATUS)(* GET FIRST LINE OF THE FILE 'INPUT' *
);
IF (STATUS 0)
THEN WRITELN(TTY,'EOF SEEN')(* HERE IF ERROR RETURN *)
ELSE
BEGIN
FIND(LBEG,PATARR,POS,PATLEN);
IF NOT(POS = NIL)
THEN WRITELN(TTY,'FOUND IT')
ELSE WRITELN(TTY,'NOT FOUND');
end;
END.
Appendix H

Flow of RUNOFF Justification
Appendix I

Text Processing Midterm and Final
Midterm

This is a closed book, closed notes test. Please answer all questions in the space provided on the test.

The following are short essay questions to be answered as concisely as possible. A simple "Yes" or "No" is too concise.

1. What is meant by a meta character?

2. What is the main difference between TECO and a line editor (in terms of addressing)?

3. Why is it necessary to type "271" to TECO in order to insert an alt-mode (symbolized by $)?

4. $ \langle SProcedure P'X$;0tt.-z;c$$
   What does this macro do?

5. In TECO, what is a reasonable way to call macros from other macros?

6. TECO usually requires an alt-mode to delimit commands. Please specify what genres of command require such a delimiter (in any editor).

7. In the text editor we specified, what convention would need to be changed to insert a line below the last line in the buffer if the "Append" command didn't exist?

8. In our PASCAL, why would it be necessary to implement a separate dispose routine for different record types?
9. It has been alleged that our editor is easier to implement with linked lists. How would you defend this thesis?

10. Why is it necessary to know the length of the pattern string when simple searching is implemented with an array (as opposed to a linked-list)?

11. How would you change the editor specified in class to allow the user to type multiple commands on one line?

12. Why is it so easy for TECO to allow a user to type 'N (i.e., a "control-N") before any one of its other search-string-specifying commands, when that following command may be arbitrarily complex?

13. In many searching algorithms, a pass is made over the pattern string to "compile" it into an internal representation. Why?

14. What is a major drawback of the searching algorithm implemented in TECO?

15. In many text formatters, a distinction is made between the line length and the line width of the output line. Define these terms.

16. In formatted text:
   a) What is a "river"?
   b) How is it avoided?
17. The Software Tools formatter provides a facility to output the page number either in the header or the footer, in any character position. How is this accomplished?

18. In respect to text formatters: What is a "Break"?

19. Assume you are told to install a "figure" command in a text formatter that allows enough space for a figure of a certain size.
   a) Give the exact format of the command (for the user)
   
   b) Specify how you would implement it.

20. One way to implement a control structure in the editor is to use a case statement to recognize which command has been typed.
   a) What happens if the user types an illegal command?
   
   b) How would you protect yourself from this?
21. The following is a PASCAL attempt to implement the previous question. What is wrong with it? (Give PASCAL language faults.) Note that Teletypeinputroutine is a function that returns the next character from the teletype, and that the identifiers ending in "command" are procedures. Finding 4 faults is sufficient for full credit.

```pascal
Procedure Commandloop;

Type Legalcommand = (A, B, C, D, E, Z);
Var
  LC : Legalcommand;
  CH : char;
  STATUS: integer;
Begin
  TTYSTAT:=0; STATUS=0;
  CH := Teletypeinputroutine;
  If CH in LC
    Then begin ----CASE CH OF
      A: Appendcommand;
      B: Bottomcommand;
      C: Changecommand;
      D: Downcommand;
      E: Endoflinecommand;
      Z: Quitcommand(sets "status");
      END;
    END;
End;
```
This is a closed book, closed notes test. Please answer all questions in the space provided on the test.

The following are short essay questions to be answered as concisely as possible. A simple "Yes" or "No" is too concise.

1. What is meant by a meta character?

2. What are the two most significant differences between RUNOFF and a composition program for a typesetter (like PAGE-3)?
   a) 
   b) 

3. Why is it not possible to make the following declaration in NOS PASCAL?
   TYPE Charset = SET OF CHAR;

4. The matrices of many pre-TTS typesetters did not have widths relative to each other. How was justification achieved on these machines?

5. What is meant by the "relative-width system" of type measurement?

6. What is meant by "base-line alignment" in photocomposition?

7. In regards to typeface design, what is meant by "x-height"?
8. What are two options available to a printer (or composition programmer) in eliminating a "widow" line?
   a) 
   b) 

9. What is a "discretionary hyphen" and what is its use?

10. What is meant by "TTS" and what is it descendant from?

11. a) What is "kerning"?
    
    b) How is it accomplished in either hot or cold typesetting?

12. What is "mortising" (in printing)?

13. What is the main difference between a third-generation and second-generation typesetter?

14. What distinguishes a fourth-generation typesetter from others?

15. What are two methods for sizing (making larger or smaller) characters in a phototypesetter?
   a) 
   b)
16. Many fonts are measured in parts of an "em". That is, an "em-space" may measure 18 units, while an "en" space only nine units, regardless of point size. Please give the conditions under which it is necessary to convert from this system of measurement inside a typesetter and why.

17. Give three methods of storing width values (of characters) on typesetters (not just third-generation).
   a)
   b)
   c)

18. What is meant by a perimeter character?

19. What is meant by "run length encoding" and how is it used?

20. In justifying a line, there are two ways to manipulate the amount of space occupied by the words on that line. What are they?

21. What is meant by "leading"?

22. What is vertical justification and why would it be used?
23. In hyphenation:
   a) What is meant by "cleaning up the word" and why is it done?

   b) Why would it be necessary to examine characters within a word while "cleaning it up"?

24. What are the two main methods used to accomplish hyphenation?

25. What is "vertical justification"?

26. In hyphenating a word, some procedures attempt to hyphenate by beginning from the right end of the word. Why would this be preferred to working from the left?

27. What is meant by a "homograph"?

28. What is an exception dictionary and how is it used?

29. What is a common method of ensuring the correct spelling for input to a typesetter?

30. The early composition programs set 3-column text by reading the first line for each column from a file by random-access, then setting the first line of each column with these lines.
   a) Why was this necessary?

   b) What problems were discovered in using this procedure?
31. What were two advantages of the IBM Magtape Selectric Typewriter?
   a) 
   b) 

32. What is the difference between an electronic typewriter and a word processor?

33. What is meant by a "thin window" standalone display system?

34. What are two advantages of a "dual media" station in word processing?

35. What is a "boilerplate"?

36. What is a shared-logic word processor?

37. What is the main disadvantage of a shared-logic word processor?

38. What is meant by a "data processing interface" in relation to word processing?

39. What are two advantages of shared-logic word processors?

40. How do word processing companies justify the relatively high cost of a word processor?

41. How does an Electronic Message System differ from a computer network?
42. What are two methods of managing messages in Electronic Message Systems?

43. What are four reasons that would justify an Electronic Message System?

   a) 

   b) 

   c) 

   d) 

44. What is meant by a "passive" network?