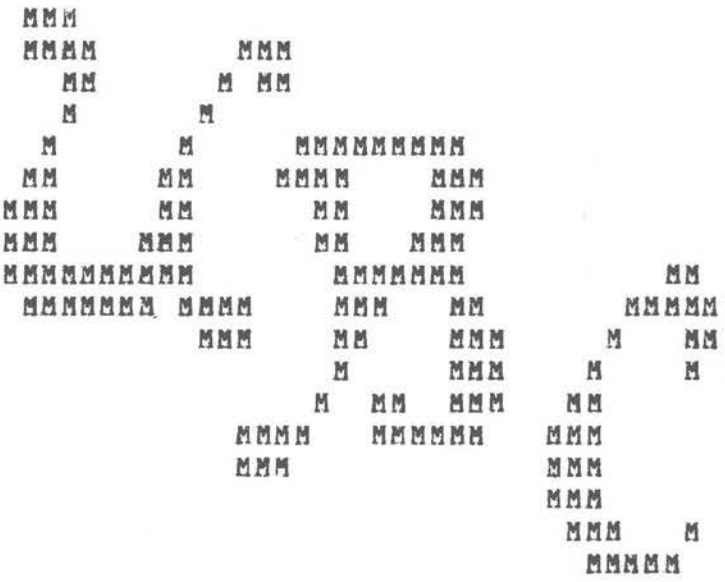


TEXTURE: A Document Processor

by

Michael Gorlick
Vince Manis
Tom Rushworth
Peter van den Bosch
Ted Venema

Technical Report 76-1

June 1976

Department of Computer Science
University of British Columbia
Vancouver, B. C.

TEXTURE: A Document Processor

Michael Gorlick, Vince Manis, Tom Rushworth,
Peter van den Bosch, and Ted Venema

Department of Computer Science
University of British Columbia

What are the benefits of a document processor? Since source documents stored within a computer system are easily changed using the system's editing programs the tedious job of re-typing a page or more of a document in order to correct a few minor mistakes is eliminated. In addition, documents produced by a computer based system are uniform in appearance. Justification, margins, page sizes and capitalization are algorithmically determined; leaving the user with complete control over the format of the final output. Finally, the computer is tireless. Human preparation and typing of long documents can lead to fatigue and a corresponding deterioration in the quality of the output documents, while the computer assures that output material is uniform in appearance and quality from beginning to end. Once a source document has been entered into a computer system, successive drafts and multiple copies can be produced easily and at low cost.

The document processor described in this paper was originally developed by Peter N. Van den Bosch [1]. This system has been further refined [2] and is currently in fairly wide use at U.B.C.

A Survey of Other Systems

Categorizing document processing facilities by the basic function they perform we can distinguish roughly four types.

- Early in the history of computing, realizing that some functions of typesetting could be automated, researchers were led to the development of typesetting software. These programs originally drove mechanical typesetters, performing little more than line justification. Today, using high-speed photo-typesetting devices, human intervention in many typesetting jobs, with the exception of data entry, can be eliminated. Manufacturers of photo-typesetters usually make available a computer typesetting system designed specifically for their equipment. The languages incorporated in such systems vary widely in sophistication and elegance. Examples of this class are PAGE-1 [3], Harris Composition System (HCS) [4] and CypherText [5]; the latter an attempt at defining a computer typesetting language without reference to specific equipment.

- A second class of document processing systems is based on the line printers and typewriter terminals available at many

computer installations. These grew out of a desire to produce and update documentation quickly, without much concern for the permanence or appearance of the final document. IBM has developed, mainly for their own large volume of documentation, several systems of this class. Examples include FORMAT [6], TEXT360 [7] and, to some extent, ATS [8]. Members of a family of programs called Runoff [9] are found on several computer systems, and FMT [10], a FORMAT-based facility, is available at installations supporting the operating system MTS.

- "Word processing systems" are making an appearance. Combining a typewriter with mass storage (cassette tape, floppy disk) and a minicomputer these devices often include rudimentary facilities for text arrangement but the concept of a document processing language is usually absent. IBM's MTST [11] and MT/SC [12] are early efforts for which documentation is available. Astrotype [13] is also typical of this rapidly evolving field.

- The fourth group of systems concerned with document processing is the highly experimental work with interactive text manipulation. A good, if erratic, survey of this work is given by Theodore H. Nelson [14]. Such systems are outside the scope of this paper and will not be discussed further.

The Criteria used in Designing TEXTURE

The following design principles are at the heart of TEXTURE:

- The Bauhaus principle (form follows function). Many computer languages and systems force the user into a framework where he must know everything before using anything. TEXTURE attempts to avoid this syndrome through, for example, the use of reasonable defaults.

- Occam's Razor (it is vain to multiply entities beyond need). TEXTURE attempts to find generalities beneath specific needs. Thus, for example, there is only one command language and only a few simple concepts.

With these principles as a foundation, the problem of what paradigm to use for various documentation questions can be confronted. As there is little documented research, or, for that matter, agreement in the area of decision models for document processing, the decision models used in the development of TEXTURE were: what a typewriter would do, what a secretary would do and what a typesetter would do according to a manual of style--not necessarily in that order.

TEXTURE DATA FLOW--THE TEXT HIERARCHY

The basic function of a document processor is to transform unformatted source text into formatted output text. In TEXTURE this is accomplished via a text hierarchy. A text hierarchy is defined in terms of six units: characters, words, segments, lines, blocks and layouts; an element of each unit (except for character) consists of one or more elements of the preceding unit. TEXTURE controls the transfer of text from the input device to the output device via the text hierarchy.

The source text which is input to TEXTURE is regarded as a stream of characters. The set of characters consists of letters, digits, punctuation, special symbols and two control characters to indicate the end of a source line or source file. Characters form the basic unit from which TEXTURE builds all other units.

At any time during the operation of TEXTURE, the set of characters can be divided into two disjoint subsets, the word terminating and word forming characters. As long as TEXTURE continues to input word forming characters, it concatenates them to form an internal word. This process is continued until a word terminating character (for example, a blank or end-of-source-line character) is encountered. In this case the word is terminated and a new one begun when the next word forming character appears. A word is indivisible, remaining together throughout all further processing.

As TEXTURE completes the building of words, it combines them into segments. A segment is the text unit in which justification occurs, where the current filler character (usually a blank) is inserted between the various words to effect the current justification method. The current justification method can be one of: flush with the left segment edge (RAGRIGHT), flush with the right segment edge (RAGLEFT), center the text within the segment (CENTERED), flush with both edges by inserting filler characters uniformly between the words of the segment (JUSTIFIED) or flush with both edges by inserting all necessary filler characters at a specified point in the segment (SPLIT).

The segments are assembled into lines. A line is a single string of text to be output, usually consisting of only one segment. In certain cases, such as tabbing to various columns, a line will consist of more than one segment since the action of tabbing terminates a segment.

Each completed line is added to a block. If a line is viewed as a piece of text with length only, then a block is a rectangular piece of text consisting of a number of lines, all of which have the same length, together with a piece of mandatory text which is associated with each block. Before any text is assembled into a block, the mandatory text (which may be

null) associated with that block is inserted at the head of the input stream.

When a block has been filled, subsequent input text is assembled into the next block of a layout. A layout is a description of how text is to be sent to the current output device and is expessed in terms of a sequence of blocks. When TEXTURE has filled one block of a layout, it begins to fill the next block in the sequence. This continues until all the blocks of the layout are filled at which point all the text within the layout is sent to the output device. The text which is output in this way is known as a page, thus a layout describes a page of text. As no restriction is placed on the order of blocks within a layout or the location of blocks with respect to each other on a page, it is possible both to fill a block near the bottom of an page before a block near the top of a page and to construct overlapping blocks.

Although the text hierarchy describes data movement through TEXTURE, it alone is insufficient to handle complete formatting. Other information (e.g. indents, spaces to be left between lines and capitalization conventions) also affects the way in which text is passed through the text hierarchy. This extra information forms a set of construction rules which determine, together with the text hierarchy, the manner in which TEXTURE processes text.

The Macro Processor

The TEXTURE system actually consists of two processors: the text hierarchy processor (TP) described above, and the macro processor (MP) which enables the user to alter the set of construction rules used by the TP. If at any time during the assembly of characters into words the TP encounters the macro-begin symbol (by default, '<'), control is passed over to the MP. The string of text from the macro-begin symbol up to the balancing macro-end symbol (by default, '>') is a macro to be evaluated by the MP. A macro which does not contain any further macros nested within it is known as a call.

The MP finds, within the macro, the leftmost call which is then evaluated and replaced by its result. This is repeated until all calls, and hence the macro, have been evaluated. A more detailed discussion of this technique is given in [15].

The following is a legal macro (using the default symbols):

```
<#EQ,<REMAINDER,<PN>,2>,1,ONE,ZERO>
```

The leftmost call is '<PN>' which returns the current page number. After evaluation of this call, the result yielded is (assuming the current page number is 19):

<#EQ,<REMAINDER,19,2>,1,ONE,ZERO>

The leftmost call is now '<REMAINDER,19,2>' which returns the remainder upon dividing its first argument by its second. The result after evaluation of this call is:

<#EQ,1,1,ONE,ZERO>

The leftmost call is now the final macro and, as '#EQ' compares its first and second arguments numerically, returning the third argument if they are equal and the fourth otherwise. The result is:

ONE

Since there are no further macros this string is passed on to the TP.

The MP calls can be broken down into four categories: those which affect the environment in which the TP constructs text units in accordance with the text hierarchy (c.f., Figure 1), those which affect the current values of the various text hierarchy units (c.f., Figure 2), those which are arithmetic in nature (c.f., Figure 3) and those which manipulate strings of text (c.f., Figure 4).

User Defined Functions

Calls also exist for the user definition of new macros. The two calls which accomplish this are 'STRING' and 'SEGMENT'. The 'STRING' call defines a new macro whose name and value are specified by the parameters passed to the 'STRING' call. Thus:

<STRING,STR,The_value_is_x>

associates with the name 'STR' the string 'The_value_is_x'. The call <STR> is now defined and returns the value:

The_value_is_x

The 'SEGMENT' call allows the user to convert a macro defined by the 'STRING' call into a macro with arguments. Consider the following example:

<SEGMENT,STR,value,x>

The string of text associated with 'STR' is scanned from left to right for any occurrence of the substring 'value'. Each occurrence of this substring is removed from the string and the

<LI,5> set the left indent to 5

<SPLIT> mark the current point in the line as a point at which to use SPLIT justification

<DOWN> from this point on, shift all upper-case characters to their lower-case equivalents

<BLOCK,TEXT,5,58,5,68,xxx> define 'TEXT' to be a block extending from line 5 to line 58, from column 5 to column 68 and with mandatory text 'xxx'

<LAYOUT,LYT,B1,TEXT,B2> define 'LYT' to be a layout consisting of the three blocks 'B1', 'TEXT' and 'B2' in that order

<INVOKE,LYT> beginning with the next page, use 'LYT' as the page layout

Figure 1

<L> end the current line, justifying the last segment of that line using the current global justification method

<L,,RAGLEFT> end the current line, justifying the last segment of that line using RAGLEFT

<TAB,n> tab to column n relative to the left edge of the block currently being built, justifying the previous segment on the same line using RAGRIGHT

Figure 2

<REMAINDER,m,n> return the remainder on dividing 'm' by 'n'

<#LT,m,n,true,false> if 'm' is numerically less than 'n' then return 'true', otherwise return 'false'

<SUM,m,n> return the sum of 'm' and 'n'

Figure 3

<LT,a,b,true,false> if 'a' is less than 'b' in a character by character comparison under a standard collating sequence return 'true', otherwise return 'false'

<STEM,str,n> return the first 'n' characters of the string 'str'

<LENGTH,str> return the length of the string 'str'

Figure 4

number 1 is associated with the locations in the string from which the substring was removed. This process is repeated for 'x' as well, associating the number 2 with each of the locations where the substring 'x' occurred.

At this point the original string of text associated with 'STR' has become a function with two parameters. The results of calling this function are illustrated as follows:

```
<STR>                                          'The__is_'
<STR,name,Peter>                       'The_name_is_Peter'
<STR,name>                                  'The_name_is_'
<STR,,Peter>                             'The__is_Peter'
<STR,name,Peter,extra>                 'The_name_is_Peter'
```

In each case, the i-th argument is inserted at any locations which have the number i associated with them. Any missing arguments in the call are assumed to be null strings.

An Event in TEXTURE

An event is an occurrence within the TP which could be of interest to the user (e.g. the completion of a line, block or page). The user is able to make use of an event by associating a string of characters with it. A copy of this string is inserted at the head of the input stream whenever the event occurs. The call which associates a string of text with an event is:

```
<HANG,event-name,text>
```

For example, the LINE event occurs when a line has been filled up, with the text of the event starting the next line. Thus:

```
<HANG,LINE,|>
```

causes each subsequent line to be prefixed with the character '|'. The text associated with an event can also be discarded by using the call:

```
<EMPTY,event-name>
```

A TEXTURE Example

The following example illustrates the method in which macros are used in TEXTURE and provides a description of the default layout (braces are used to delay detection of a call):

```
<STRING,LEFT-TITLE,><STRING,RIGHT-TITLE,{<PN>}>
<STRING,TITLE,{<LEFT-TITLE><SPLIT><RIGHT-TITLE>}>
<STRING,FOOTER,>
<BLOCK,STANDARD-HEADER,5,68,1,1,{<TITLE><NEXT>}>
<BLOCK,STANDARD-TEXT,5,68,5,58>
<BLOCK,STANDARD-FOOTER,5,68,60,60,{<FOOTER><NEXT>}>
```

```
<LAYOUT,STANDARD-LAYOUT,STANDARD-HEADER,
                        STANDARD-TEXT,
                        STANDARD-FOOTER>
<INVOKE,STANDARD-LAYOUT>
```

The above layout is active at the time TEXTURE begins processing. For many documents, this layout is adequate.

Implementation of the Text Hieracchy Processor

In order to assemble text into the hierarchy, the TP has a working element of each of the six units: character, word, segment, line, block and layout.

Each working element (except for the working layout) is destined to become part of the current or the next working element of the succeeding type. Thus the working word, when it is finished, will become part of the current working segment (if there is room), or it will become part of the next. The act of adding a lower order element to a higher one may in turn cause the higher to be completed. For example, when a line is done it may fill up a block, which may in turn fill up a layout. This upwards control path is implicit in the process of formatting text. It is internal to the TP and inaccessible to the MP.

There is a second, downward path accessible to the MP and external to the TP which is used to cause the termination of a text hierarchy element. This path is distinct from the internal one since to end a line the working segment must be ended (and to do that, the working word must be ended) before control can resume up the internal path.

There are two types of information which the TP must have in order to operate. The top two text hierarchy elements (blocks and layouts) have in addition to a working element, a definition which is used to set it up. For example, the definition of a layout is a vector of its block definitions. When a working layout is needed, the list of working blocks is created one at a time from the definitions in the vector. The TP also needs information about such matters as spacing, justification method and capitalization conventions.

Implementation of the Macro Processor

The MP is a stream oriented one, similar to those described in [15], with one major difference. A normal stream oriented macro processor prints the result of evaluation, while the MP passes its output to the TP for further processing. The TP usually disposes of all the text it receives, but may return some if an event occurs.

Some of the primitive functions can trigger events, leading to another complication in the macro evaluation: the TP must be

dispose of all MP output together with any text generated by an event, before evaluation of a macro can begin. If this were not done, the text from an event triggered in the process of evaluating a macro could appear ahead of some of the source text which was in front of the macro.

Communication

The MP and TP are implemented as co-routines, with two communication paths: the neutral-active string and the routines which make up the external control path of the TP.

To the MP the TP looks like a subroutine that disposes of evaluated text together with a number of routines for teminating elements of the various types in the text hierarchy. These routines may perform only the action requested, or they may return some text which must be evaluated before the action can be performed. <PAGE> is an example. If there is a footer, the text in it must be evaluated and placed on the page before the layout can be completed.

To the TP the MP looks like a subroutine which returns text to be broken into words and placed on the page, or which evaluates event texts, together with a number of routines which make requests for various actions.

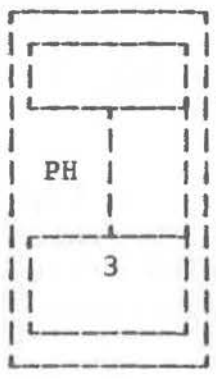
There are synchronization problems between the TP and the MP. When the MP starts there may be a number of definitions and changes to the environment to be made before starting the TP. For example, the standard layout may not be the one desired for the first page, but once the TP has been started the first layout has been set up and it can not be changed until the next page. In order to make this seem reasonable to the user, the MP simply throws away any leading blanks and end of source line characters which would ordinarily go to the TP, until a non-blank text character appears before starting the TP (and thus setting up the first layout).

The worst synchronization problem comes from events. If a call occurrs while the MP is evaluating arguments to some other macro and triggers an event, the question of when to evaluate the event and where to put the resulting text arises. Also, the event must be evaluated as if it were at the top level (i.e., text which has been evaluated goes to the TP, not into the argument list of some pending call). The solution adopted was to stack the neutral-active string and restart the MP at the top level on the event text, unstacking the neutral-active string and returning to the previous level of evaluation only when the event text has been processed.

Changes to a Layout

The text hierarchy as described so far is adequate for positioning text on a page. Nevertheless, many documents include photographs or figures which must be accomodated by temporary changes to an otherwise constant layout. There are two types of temporary changes.

Suppose the user wishes to leave space on the page to place a photograph or draw a figure. A change of this sort can be made by cutting blocks (either horizontally or vertically) in the existing layout into smaller blocks and removing some of them from the layout (e.g. in the diagram, making the cuts 1, 2 and 3 and removing block PH). Of course, this must be done before the blocks to be cut have been filled with text. Since this is done to the working layout, the current page is changed; but as a new working layout is created from the definition of the layout for the next page, the changes are temporary.



If, instead of leaving space, the user wishes to insert a block of text such as a footnote[1], cutting and removing blocks from the layout is not enough. In order to fill the block to be inserted without terminating the block currently being filled, the TP must be able to suspend the process of filling one block, fill another (or several others) and then resume filling the original. Once the block to be inserted has been filled, it must be placed in the working layout. Space on the page for the block being inserted is obtained automatically by the TP in the same way the user obtains space for a photograph (described above). However, there may be blocks in the layout that are already full of text (and so cannot be cut) which overlap the block being inserted. For example, two footnotes on the same page where both would normally appear at the bottom of the page would overlap. In this case one of the blocks is moved out of the way of the other by changing its position on the page and re-inserting it into the working layout.

Assessment of the Design and Implementation

TEXTURE appears to be fairly easy to learn--even for the non-programmers who have tried it. Over the last few months since its introduction, the TEXTURE community at UBC has grown to over 100 users, many of whom had made little or no use of

---

[1] The actual algorithmn followed by the TP when processing a footnote is very complicated, possibly because footnotes are intrinsically more difficult to automate than straight text formatting.

computers before. Of course, there are the usual troublesome areas inherent in any text processor. For example, there is little possibility of ever making footnotes completely automatic, as the correct use of footnotes requires a keen typographic eye. The average user, of course, does not use such features.

Paradoxically, TEXTURE's main strength and its primary weakness are identical: the use of the EUREKA macro processor as a command language. Our system does not suffer from the plethora of command languages found in some other processors. For example, FMT (the most commonly used document processor at UBC) has five different command languages. Such a simplification must be one of the main factors in the ease of learning TEXTURE.

The set of commands available in EUREKA is complete in the sense that all reasonable actions may be specified (though not always easily). This would not be profound if it were not for the fact that most document processors, no matter how many different command modes they have, are incomplete--often, for example, there are poor string definition facilities. One of TEXTURE's main strengths is that the command language is a full scale programming language. Often a problem which would require great ingenuity from the user of a "traditional" document processor is easily done in TEXTURE.

One of the unfortunate aspects of the EUREKA processor is that writing complex code is difficult. One can become quite proficient in the use of EUREKA macros; still, EUREKA programming is a convoluted process. This phenomenon is familiar to anyone who has made much use of macro processors.

Communication problems between the MP and the TP are more complex than one might think. The problem is that TEXTURE cannot operate in the manner that the user might reasonably expect; efficiency and internal consistency dictate otherwise. As an example, the system appears to the user to operate character by character, while, in actuality, input is line-buffered.

Another area of difficulty is the basic orientation of TEXTURE. The system of blocks and layouts described above can, in principle, describe any page format; however, in practice, some desired text formats are very difficult to implement. Although it is unlikely that any user will ever want an extremely intricate layout, there are some structures which can be used only with great difficulty. Consider, for example, the "parallel text" problem [14]: there are many uses for documents containing two texts which run in parallel from page to page. An example of this is the Instructor's Edition of a textbook. Such a document contains the actual text of the Student's Edition, but each page contains, in addition, another column of text with items keyed to corresponding sections of the main

text. TEXTURE is not designed to merge multiple input streams; thus such a document is remarkably difficult to produce via TEXTURE.

Although TEXTURE is most conveniently used from a terminal in a time-sharing system, in no sense can it be called "interactive". Once TEXTURE is initiated, it runs to completion in a strict "batch" mode. Documents are entered and updated by means of the text editor provided with the host operating system. It is a matter for further research to determine whether a TEXTURE-like system may be made interactive in any useful sense of the word. It is quite likely that a fully interactive document processor might resemble TEXTURE a lot less than, for example, NLS[16].

There are a number of problems due to the implementation: a trial version was written in PL/I, but due to the PL/I implementation available at UBC (an obsolete issue of PL/I (F)), this version was too expensive to use. Therefore, a new version was written in PL360. Whatever the merits of PL360, portability is not one of them--thus, the current TEXTURE system will only run on a 360 or 370. One day, the current version may be transcribed into a higher level, portable language such as optimising PL/I, BCPL[17], or C[18]. The user's manual is another defect: while it might gladden the heart of a computer scientist, it is hard going for a non-technical user. A primer is currently in preparation.

From this litany of complaints, the reader might assume that TEXTURE is riddled with defects. A fairer assessment might run as follows: TEXTURE is an attempt to push forward the scope and usage of document processor systems--its defects generally are not those that a user of a traditional Runoff program would encounter. We feel that the merits of TEXTURE far outweigh its faults.

[1] Peter N. van den Bosch. The design and implementation of a document processor. Master's thesis, The University of British Columbia, 1974.

[2] TEXTURE Support Group. TEXTURE User's Manual. Department of Computer Science, The University of British Columbia, 1975.

[3] PAGE-1 Composition Language--Reference Manual. Form Rep. 73-06-003P. RCA, January 1971.

[4] Harris Composition System--Language Manual. Harris Intertype Corporation. March 1970.

[5] C.G.Moore and R.P.Mann. "Cyphertext: An extensible composing and typesetting language". Proceedings Fall Joint Computer Conference Vol. 37(1970); pp. 555-561.

[6] G.M. Berns. "Description of FORMAT, a Text-Processing Program". Communications of the ACM, Vol. 12, No. 3 (March 1969); pp. 141-146.

[7] TEXT-360 - Introduction and Reference Manual. IBM Form C35-0002-0. March 1969.

[8] System/360 Administrative Terminal System (ATS): Terminal Operations Manual. IBM Form GM20-0589-2. April 1970.

[9] L. Wade. PDP-11 Runoff. Digital Equipment Corporation Users Society. October 1971.

[10] W. Webb. UBC FMT. University of British Columbia Computing Centre. April, 1976.

[11] Magnetic Tape Selectric Typewriter (MTST). IBM Forms 543-0510-1, 543-0515, 549-0204 and 549-0700.

[12] "The IBM Selectric Composer (MT/SC)". IBM Journal of Research and Development. Vol. 12, No. 1 (January 1968); pp. 3-91.

[13] Astrotype. Form #30, Automatic Office Division. Information Control Systems, Inc.

[14] T.H. Nelson. Computer Lib/Dream Machines. Hugo's Book service, Chicago p.o.b. 2622, 1974.

[15] P. Wegner. Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, 1968.

[16] D.C. Engelbart and W.K. English. "A Research Centre for Augmented Human Intellect". Proceedings Fall Joint Computer Conference Vol 33(1968); pp. 395-410.

[17] M. Richards. "BCPL--A Tool for Compiler Writing and System Programming". Proceedings Spring Joint Computer Conference Vol 35(1969); pp. 557-566.

[18] D.M. Ritchie. C Reference Manual. Bell Laboratories (Murray Hill, New Jersey), 1974.