Texture User's Manual

BY

The Texture Support Group

Technical Manual 75-08

1978 January 19

DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF BRITISH COLUMBIA
VANCOUVER, BRITISH COLUMBIA V6T 1W5

ABSTRACT

Texture is a document formatting program designed for a wide variety of applications ranging from form letters to technical reports. It features an exceptionally versatile command language, and permits great freedom in page layout.

This manual contains a description of Texture, together with three supplements: an update newsletter describing new features as of 1977 July, and descriptions of two convenient libraries, Eurekalib and Drawlib, developed by Mark Scott Johnson.

This document was printed on an IBM 3800 printer, courtesy of Block Brothers Industries, Vancouver.

Terms of Usage

The Texture package was developed by:

Michael Gorlick
Vincent Manis
Thomas Rushworth
Peter van den Bosch
Tjeerd Venema

under the auspices of the Department of Computer Science, University of British Columbia.

The Texture package consists of copyrighted material, and is distributed under a License Agreement available from the Department. Any reproduction, in whole or in part, of the package without permission, other than that explicitly provided for under that Agreement, is strictly prohibited.

NEITHER THE TEXTURE SUPPORT GROUP NOR THE UNIVERSITY MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO THE TEXTURE PACKAGE. Neither the Texture Support Group nor the University shall be held liable for errors contained therein, or for incidental or consequential damages in conjunction with the furnishing, performance, or use of Texture.

Copies may be made of this manual, but in such a case, a signed copy of the Document Agreement on the next page must be given to the Licensee.

Document Reproduction Agreement

I agree that the copy of the Texture User's Manual which I receive is for my own use in connection with a licensed copy of the Texture package. I understand that the Texture package consists of copyrighted material, and agree to conform to all provisions of the Program License Agreement, a copy of which is appended to this manual.

Date ....

Signed ....

<T,n,justification-method> Relative Tabbing

This will cause a tab to column 'n' relative to the sum of the left block edge and the left indent, justifying the last piece of text according to 'justification-method'. The value of 'justification-method' must be one of 'JUSTIFIED', 'RAGRIGHT', 'RAGLEFT', or 'CENTRED'. If 'justification-method' is omitted, RAGRIGHT is used. If 'n' is omitted, then the tab is made to the next column set by TABSET where column 1 is defined to be the sum of the left block edge and the left indent.

<TAB,n,justification-method> Absolute Tabbing

This will cause a tab to column 'n' relative to the left block edge, justifying the last piece of text according to 'justification-method'. The value of 'justification-method' must be one of 'JUSTIFIED', 'RAGRIGHT', 'RAGLEFT', or 'CENTRED'. If 'justification-method' is omitted, RAGRIGHT is used. If 'n' is omitted, then the tab is made to the next column set byTABSET where column 1 is defined to be the left block edge.

<LINESPACING,n> Setting the line spacing

On typewriters there is usually a switch which controls what is known as 'single', 'double' or 'triple' spacing. This means that a carriage return will result in the paper being advanced one, two or three lines, respectively. In Texture this command controls the number of blank lines between any two typed lines. Thus single spacing is set by <LINESPACING,0>, double spacing by <LINESPACING,1> and so on, for any positive value.

<PARAGRAPH-SPACING,n> Setting the paragraph spacing

This command sets the number of blank lines to be left between paragraphs by the 'P' command to 'n'.

<LI,n> and <RI,n> Indentation

On typewriters there are usually a pair of 'margin' buttons which set the left and right margins. In Texture it is these commands which act as 'margin' buttons. <LI,n> resets the left margin to column 'n', and <RI,n> resets the right margin 'n' columns to the left of the right printing edge.

It is worth knowing that they act much the same way as the equivalent buttons on a typewriter. If you have already typed a line, and then reset the left margin, the effect of your reset will not be seen until the next line. Similarly if you are past column 'n', and reset the right margin to column 'n', the effect will not be seen until the next line.

Indentation of the left margin has no effect on tab columns. <T,n> will have the same effect after a <LI,m> as before (for n>m). The expected occurs for a <P> command, if we think of paragraphing as a new-line followed by spacing over some number of columns; that is, the paragraph does not begin at a given column, it is over some number of columns from the left margin.

<BREAK-WORD-ON-EOL> and <NOTBREAK-WORD-ON-EOL>

After any occurrence of 'BREAK-WORD-ON-EOL' the end of source line always causes a break between words until the next 'NOTBREAK-WORD-ON-EOL' occurs. 'BREAK-WORD-ON-EOL' is equivalent to appending a trivial blank onto the end of each source line.

2.2 Adjusting text within a line

Texture normally adjusts all text so that both margins are flush (like this paragraph). If a line is suddenly terminated by the equivalent of a carriage return (<L> for example, or <P>), the line will only have the left-hand margin flush. Sometimes, however, it is desirable to centre some text within a line (as is, for instance, the title of this manual), or to have several lines coming out with a ragged right margin, as though they were typed on an ordinary typewriter. For this, there are other modes of adjustment.

<JUSTIFIED>

This is the normal mode of adjustment. It causes both margins to be flush.

<RAGRIGHT>

This is the sort of adjustment you get when you type something using a typewriter. Only the left-hand margin is set flush; the right-hand margin is set exactly where it was when the line filled up and the processor had to go to the next line. This paragraph is an example of ragged right text.

<RAGLEFT>

This is the opposite of RAGRIGHT, and causes text to be set with a ragged left and a flush right margin. There is little use for such an adjustment mode, except in paragraphs like this, and in setting concrete poetry.

<CENTERED> or <CENTRED>

This mode causes each line to be centred as much as possible. An almost equal number of blanks is put on each side of the line to centre it. A good use of this mode is in titles, and the title of this chapter is, indeed, centred. A poor use is in normal documents, unless one is demonstrating centred text, as in this paragraph.

<SPLIT>

This is not so much an adjustment mode as a command which causes a line to be split into two parts, one set flush against the left-hand margin, the other flush against the right. It is useful in tables of contents, and for other similar effects. For example, the line

<L> Chapter 3 ..... <SPLIT> ..... 42 <L>

would come out

```
Chapter 3 .....                                    ..... 42
```

<CHAR,x> Special characters

Some line printer print chains have a number of special characters on them which are very useful for producing handsome documents. These include braces ({}), square brackets ([]), superscript numbers (^{0123456789}) and other graphic characters. Since these characters do not have an equivalent on most data entry equipment (e.g., Terminals and, heaven forbid, keypunches), it is difficult to get at them without a facility like CHAR. The value 'x' is the decimal equivalent of the bit-sequence of the particular character. Thus, |, which is not included on some teletypes, has the bit-representation 01001111 in EBCDIC, and is therefore <CHAR,79>. Having to remember numbers is almost as annoying as not having the characters available in the first place, and it will be explained in the next section how one can define mnemonics for one's favorite special characters. All characters available on the TN print chain are given (hopefully meaningful) mnemonics by using the PAR=SYSLIB feature on the $RUN command which initiated execution of Texture (see Appendix B). To avoid confusion with the internal representation of some special markers, any value of 'x' which is less than 64 is turned into 64.

2.3 Case shifts

Just as not every input device has a full character set, so not every input device has lower-case characters. It would be very tiresome if we had to do all our lower-case letters with CHAR, and in recognition of this, Texture provides several source editing commands.

<DOWN> and <NOTDOWN>

After a DOWN, every text character from "A" through "Z" is automatically turned into its lower case equivalent until a NOTDOWN or UP is encountered. Eureka program text is not affected by this translation.

<UP> and <NOTUP>

After an UP, every text character from "a" through "z" is automatically turned into its upper-case equivalent until a NOTUP or DOWN is encountered. Eureka program text is nct affected by this translation.

<SET-SENTENCE,str1,str2>

This primitive changes when Texture is to start a new sentence. If while building a word, any of the characters of 'str1' is encountered and none of the characters of 'str2' is encountered in the remainder of the word, then the next word will begin a new sentence. The default value of 'str1' is '.?!' and of 'str2' is all the upper- and lower-case letters as well as the digits zero through nine.

<AUTOCAP> and <NOTAUTOCAP>

After an AUTOCAP the first letter of the first word of every sentence is capitalized until a NOTAUTOCAP is encountered. Eureka program text is not affected by this translation.

2.4 Some special operators

For convenience, Texture defines some single-character operators which are not treated as text, but have an effect on the character which follcws them:

1. "_" places an underscore under the next character. Thus "ba_th" becomes "bat̲h" on output.

2. "@" causes the next character to be shifted up. Thus, "@mc@phee" becomes "McPhee". This can be very useful in DOWN mode.

3. "¢" causes the next character to be shifted dcwn. Thus, "¢E.¢E.¢Cummings" becomes "e.e.cummings". This is very

TEXTURE: A USER'S MANUAL

> "It is shaped, sir, like itself, and it is as broad as it hath breadth; it is just so high as it is, and moves with its own organs; it lives by that which nourisheth it, and the elements once out of it, it transmigrates."
>
> Anthony and Cleopatra

Availability

Texture can be obtained using the following MTS RUN command (with the PAR field optional):

```
$RUN CS:TEXTURE SCARDS=sourcefile SPUNCH=outputfile
          SPRINT=listingfile PAR=SYSLIB
```

where SCARDS specifies the user input file, SPUNCH, the file on which Texture will initially begin text output, SPRINT, the file which will contain error messages and the output of the LIST command, and the PAR field indicates to Texture to use the system library of predefined Eureka functions (see Appendix B). If the 'SPUNCH=outputfile' is omitted on the RUN command, Texture will default its output to the MTS file '-TXTDOC'.

The input to Texture is a stream of text. This text may be broken across the 'boundaries' imposed on such a stream by the harsh practicalities of the real world: such boundaries as the end of a card or record in a line-file or on a tape. To the document processor, however, all this looks like one contiguous stream of text. The text will probably include the text of the document which Texture eventually outputs; but mixed with this prose there will in general be remarks to the processor in the form of Eureka calls. Eureka is a language for the definition and expansion of macros which define primitive functions for these purposes, along with some conditional evaluation, input/output and string manipulation primitives. A Eureka program is a Eureka call, which is generally enclosed in the delimitiers < and >. The input to the document processor is therefore first examined by Eureka, which either ignores the text (if it is not inside the Eureka call delimitiers) or evaluates the text if it is a call. The value of a call is either scanned anew or, if it was a neutral call (one preceded by the symbol ':'), it is ignored. When Eureka "ignores" text, it passes the text along to Texture proper (the document processor) which outputs the text in a finished form.

The form of the output text is controlled by the calling of primitives (described below) which affect the processing of subsequent text as soon as they have been evaluated. To the user, then, it appears as though Texture is actually a set of document processing primitives defined in Eureka, and that the Eureka output routine is one which formats the processed text. This manual will describe the primitives (those which are proper to Eureka and those which define Texture) as though this is in fact the case, although in reality this is somewhat of an oversimplification.

The manual is organized into four parts. The first part, entitled "Using Texture Immediately", is concerned with the basics, including a few useful commands. The second part, "Some More Texture", gives the remaining commands likely to be needed by the average user. The third part is "Eureka!" Which introduces the user to writing programs in Eureka. The fourth part is entitled "The Rest of Texture" and gives all the other commands and advanced concepts which remain.

1. Using Texture Immediately

If one knew nothing else about Texture than how to run it, it would be possible to feed some lines of raw text to Texture and to get the text printed out again, neatly formatted for a typewriter-size page.

This is seldom adequate, however. In any document it is necessary to be able to produce paragraphs, underline words and tab to desired columns. These are accomplished by means of "commands" to the processor, and it is perhaps most convenient to think of them as "asides" or "proofreader's marks". Like proper asides, they are enclosed in parentheses -- in this case angle brackets (< and >). This naturally precludes the use of angle brackets for any other purposes, such as mathematical notation, since the processor always thinks of them as command delimiters, and the user should be aware of this. In the next section it will be explained how angle brackets can be produced as ordinary text.

Commands may be inserted anywhere in the text: whatever their effect is, it will be felt at the point at which they occur. Whenever possible, Texture acts the way a typewriter would and commands are treated as though they were special buttons on a typewriter. The most important commands, and their effects on the text, are the following:

<P,justification-method> New paragraph

Advance to a new line using 'justification-method' to justify the last line and space in X spaces from the current left-hand printing edge where X is the current paragraph-indent (default is 5). The value of 'justification-method' must be one of 'JUSTIFIED', 'RAGRIGHT', 'RAGLEFT' or 'CENTRED'. If 'justification-method' is omitted, it defaults to RAGRIGHT if the current global justification method is JUSTIFIED, otherwise it defaults to the current global justification method. All the paragraphs in this document were started with a <P> before the first word in the paragraph.

<L,n,justification-method> New line

Advance to a new line 'n' times (n must be integral) justifying the lines using 'justification-method'. The value of 'justification-method' must be one of 'JUSTIFIED', 'RAGRIGHT', 'RAGLEFT' or 'CENTRED'. If 'justification-method' is omitted, it defaults to RAGRIGHT if the current global justification method is JUSTIFIED, otherwise it defaults to the current global justification method.

<U> and <NOTU> Underlining

Every word between a <U> and the next <NOTU> will be underlined. The blanks separating these words are not underlined. Letters, digits and hyphens within words are underlined, but punctuation is not. In section 4 the description of the pair of functions UND and NOTUND will explain how the set of characters which is underlined may be changed.

2. Some more Texture

2.1 Tabs

As anyone who has ever used a typewriter will know, it is convenient to be able to jump, at the touch of a single button, to a predetermined column. Almost all typewriters have a facility for setting 'stops' at such columns and so has Texture. Following the usual terminology, this facility is called tabbing.

<TABSET,n> Setting tab-stops

The value 'n' is a number corresponding to a column. A tab-stop is set at that column. Thus, <TABSET,26> will set a tab-stop at column 26. The meaning of the word "column" is dependent on the type of tabbing being done (see TAB and T).

<TABCLEAR,n> Clearing tab-stops

This has the opposite effect of TABSET. The tab-stop at column 'n' is removed. <TABCLEAR>, without a value 'n', causes all tab-stops to be removed.

useful in UP mode, to avoid having to shift down for a single character.

4. "*" causes the next character to be treated literally as text. Thus, "15*¢EACH" becomes "15¢EACH", not "15eACH" as it would have been without the asterisk. Similarly, "*<P>" does not cause a new paragraph, but simply causes the text "<P>" to come out (which is the only way this document could have been written).

5. "/" causes the next character to be overprinted on the previous. Hence, "a/"" becomes "a", and ":/<CHAR,191>" becomes ":".

6. "¬" is replaced by a blank on output. The difference between "a¬b" and "a b" is that "a b" is two words, and may be broken across lines or separated by extra spaces after justification, while "a¬b" is a single word which happens to have a blank in it. The character which replaces the ¬ is or is not underlined in underlining mode, depending on whether the character would normally be underlined. Thus a blank, which is the default character, would not be underlined.

These operators always act upon the next text character. Thus two or more operators might appear in a row and so affect the same text character. (e.g., "_@a" results in "A".)

<SET-op,c> Setting special characters

It is not necessary to consider oneself stuck with the six special characters given above. Sometimes these particular characters are inconvenient, and we'd prefer another set. In the <SET-op,c> commands, "c" becomes the special character denoted by "op", where "op" means the following:

<SET-TEXT,c> 'c' is henceforth treated as ordinary text. This can be convenient if one is not using one of the special characters and wants to be free from the bother of having to put an asterisk in front of any use of 'c' in the text.

<SET-BREAK,c> 'c' is henceforth treated, the way a blank is now, as a break between words. Hereafter, "acb" is treated as the two words "a" and "b".

<SET-NTB,c> 'c' is henceforth treated as a non-trivial blank (the way ¬ is by default). That is, from now on, 'c' is replaced by a blank on output (described above).

<SET-OPC,c> 'c' is henceforth treated as an overprint operator, the way / is by default.

<SET-LIT,c> 'c' is henceforth treated as a literal-next operator, the way * is by default.

<SET-DOWN,c> 'c' is henceforth treated as a down-shift operator, the way ¢ is by default.

<SET-UP,c> 'c' is henceforth treated as an up-shift operator, the way @ is by default.

<SET-UNDER,c> 'c' is henceforth treated as an underscore-operator, the way _ is by default.

<LIST,name> and <NOLIST> Turning the source listing on and off

The action of these commands is obvious: LIST causes the source to be listed on the file 'name'. If 'name' is absent, the listing file becomes the file attached to SPRINT. NOLIST will turn off the listing, until the next LIST command.

In LIST mode, each line of source text is echoed to the listing file in the following form:

A. The source file line number, useful for editing purposes.

B. The source line itself enclosed in vertical bars (|).

C. The mode in which the processor was operating at the time it ended processing the previous source line and began processing the current line (the point of the line break). The possible modes are:

    T - Texture. The processor was assembling document text at the point of the line break.

    F - Function. The processor was evaluating a Eureka call at the point of the line break.

    L - Literal. The processor was scanning over a

Eureka literal at the point of the line break.

I - Input. One of the Eureka primitives RC, RFN, RLN, RNC or RS was calling for input at the point of the line break.

D. The Eureka function nesting level at which the line break occurred. This is the number of < or :< symbols which have been scanned for which a matching > symbol has not yet been scanned at the point of the line break. This field is blank if the processor is not in 'E' or 'L' mode at the point of the line break.

E. The Eureka literal nesting level at which the line break occurred. This is the number of " symbols which have been scanned for which a matching ' symbol has not yet been scanned at the point of the line break. This field is blank if the processor is not in 'L' mode at the point of the line break.

F. The input-primitive mode at the point of the line break. If the processor is in 'I' mode at the point of the line break this field contains one of the following:

R - Read. The primitive is reading the text at the point of the line break. This is true for all input primitives except RFN.

F - Functionscan. The RFN primitive is scanning a function call at the point of the line break. The mode applies to RFN only, it is blank for all other input primitives.

L - Literalscan. The RFN primitive is scanning a literal at the point of the line break. This mode applies to RFN only, it is blank for all other input primitives.

If the processor is not in 'I' mode, then this field may contain one or both of the following:

> - Trim stem on. All SOURCE lines to Texture have leading blanks deleted.

< - Trim stern on. All SOURCE lines to Texture have trailing blanks deleted.

G. The RFN function nesting level at which the line break occurred. This is the number of < or :< symbols scanned by the RFN primitive for which a matching > symbol has not yet been scanned at the point of the line break. This field is blank if the processor is not in 'I' mode or if the level is 0.

H. The RFN literal nesting level at which the line break occurred. This is the number of " scanned for which a matching ' has not yet been scanned at the point of the line break. This field is blank if the processor is not in 'I' mode or if the level is 0.

When an error occurs, the line, up to the point of error, is listed and the error message is printed below it enclosed within a box to distinguish it from surrounding text. Errors are classified as fatal or non-fatal, a fatal error being one where a logical course of action cannot be undertaken (e.g., Invoking a block instead of a layout). When a fatal error occurs, the page currently being assembled is printed on the output file up to the current assembly point, the error message is given on the listing file and then processing halts. When a non-fatal error occurs, an error message is given on the listing file, or SERCOM if NOLIST is in effect, and processing continues, with the offending point either being assigned the default value if a default value exists, or ignoring the point if nothing is known about it (e.g., An undefined function).

<PN,n> Setting the page number

Texture will automatically number pages sequentially, starting with page 1. Sometimes, when a document is being assembled chapter by chapter, it is convenient to start off with a number other than 1, and for this reason, <PN,n> sets the page number to 'n'.

If one wants to know the current page number for some reason (and there are good reasons, such as setting up a backward reference), the command <PN> is always replaced by the value of the current page number. Thus the text "This is page <PN>" becomes "This is page 11" when it is printed.

2.5 Headers and footers

It is convenient to have some facility for putting headers at the tops of pages and footers at the bottoms, so that a person flipping through a document can find his place quickly by looking for the appropriate identifying text. For the user's convenience, Texture defines, by default, a header at the top of each page and a footer at the bottom. The header is divided into left and right parts, each of which may be set individually. The right header is, by default, the page number, but it may of course be reset. This should be sufficient for most uses, but if it is not, section 4 will explain how to define one's own headers.

<LTITLE,text> Setting the left header

The left header is set to 'text'. On each subsequent page, the text 'text' is put at the left top of the page. If 'text' is absent, as in <LTITLE>, the text that would appear at the top of the current page also appears in the stream at this point (that is, the value of <LTITLE> is the left title).

<RTITLE,text> Setting the right header

The right header is set to 'text'. On each subsequent page, the text 'text' is put at the right top of the page. If 'text' is absent, as in <RTITLE>, the text that would appear at the top of the current page also appears in the stream at this point (that is, the value of <RTITLE> is the right header). The value of <RTITLE> defaults to "<PN>", with the result that the page number comes out at the top right of each page.

<TITLE,text> Setting the entire header

The header is set to 'text'. On each subsequent page, the text 'text' is put at the top of the page. If 'text' is absent, as in <TITLE>, the text that would appear at the top of the current page also appears in the stream at this point (that is, the value of <TITLE> is the page header).

Initially, the page header is given the value "<LTITLE><SPLIT><RTITLE>", which is what causes the left header and right header to appear at opposite ends of the same line.

<FOOTER,text> Setting the footer

The footer is set to 'text'. On each subsequent page, the text 'text' is put at the bottom of the page. If 'text' is absent, as in <FOOTER>, the text that would appear at the bottom of the current page also appears in the stream at this point (that is, the value of <FOOTER> is the current footer).

<PAGE,justification-method> Skipping to the next page

On a typewriter, this is the equivalent of rolling the current page out after justifying the last non-blank line using 'justification-method', rolling the next page in, typing the header and page number and then moving the carriage to the first column of the first line. The value of 'justification-method' must be one of 'JUSTIFIED', 'RAGRIGHT', 'RAGLEFT' or 'CENTRED'. If 'justification-method' is omitted, it defaults to RAGRIGHT if the current global justification method is JUSTIFIED, otherwise it defaults to the current global justification method.

3. Eureka!

Eureka is a complete programming language in which Texture is embedded as a set of primitives and through which full computing power is available to the user throughout the processing of a document. The average user will not need to know very much about Eureka because he will only be giving simple commands to Texture. But if it becomes necessary to do some complex calculations and to have different commands performed depending on some condition, it is necessary to know how Eureka can be used as a programming language.

Fortunately, this is very simple. What follows then, is a Child's Garden of Eureka.

Every Eureka program looks something like this:

```
<a,b,c,...,z>
```

Each of the a,b,c,...,z may be an arbitrarily long string of characters. Eureka evaluates the prototype program above by scanning each of the a,b,c,...,z for more Eureka programs. These it evaluates first, and rescans the values they return for more Eureka programs. When it is finished scanning, it treats the value "a" as the name of a function, and 'calls' that function. The function may use the values of b through z for its own dark purposes, and finally it returns a string of characters as its value. This value replaces the program and is immediately rescanned by Eureka.

For example:

```
<EQ,5,<SUM,2,3>,YES,NO>
```

Eureka first evaluates all programs inside the main program, and replaces them by their value. There is exactly one program inside: the <SUM,2,3>, which, needless to say, adds its two parameters and returns the result. Thus, when finished scanning, Eureka faces the program:

```
<EQ,5,5,YES,NO>
```

Now Eureka evaluates the main program. EQ is a function which compares its first and second arguments and if these are equal returns the third, otherwise it returns the fourth argument. Hence the result is

```
YES
```

Of course, it isn't quite that simple. These are the precise semantics:

1. There are three strings and a stack whose elements are strings: a neutral, an active and a scanning string, and an evaluation stack. A Eureka program to be evaluated is on the active string and, at the beginning of evaluation, the neutral and scanning strings and the stack are empty.

2. Text is taken from the front of the active string and put on the end of the neutral string, character by character, until a closing bracket (>) is encountered.

3. Text is taken off the end of the neutral string, character by character, and put on the end of the scanning string until an argument separator is encountered (,). When an argument separator is encountered, the text on the scanning string is put on top of the stack. Step 3 is repeated until an opening bracket (<) is encountered, at which point the text on the scanning string is put on top of the stack as a single stack element. Scanning resumes with step 4.

4. The stack represents the name and arguments of a function call. The top element on the stack is the name of the function to be called, the remainder of the stack, in order, gives the arguments. The last character on the neutral string is examined. If this character is the neutral evaluation character (:), then this character is removed from the neutral string, and the result of the function call (a string) is put at the end of the neutral string. If this character is any character but the neutral evaluation character, the result of the function call is put at the front of the active string. Evaluation resumes at step 2.

That is all, except that the algorithm is enhanced as follows, to allow for quoting of text (which inhibits evaluation):

a. In step 2, whenever an opening quote (") is encountered, the scanning level is incremented by one; whenever a closing quote (') is encountered, the scanning level is decremented by one. Only at a scanning level of 0 (the initial level), does a closing bracket (>) terminate step 2.

b. In step 3, whenever a closing quote (') is encountered, the scanning level is raised by one; whenever an opening quote (") is encountered, the scanning level is lowered by one. Only when the scanning level is zero, does an argument separator (,) cause text to be put on the stack, and then only after being stripped of any leading " and trailing '.

3.1 A program example

The user may wish eventually to reproduce the output of Texture in a book format, by photographic means. If he does, he will probably want the page numbering to be placed differently for odd-numbered (right-hand) pages than for even-numbered (left-hand) pages. In most books, the page number on even-numbered pages is on the left, and on odd-numbered pages on the right. This might be done by the following commands:

```
<LTITLE,"<EQ,<MOD,<PN>,2>,0,"<PN>',"<MYTITLE>'>'>
<RTITLE,"<EQ,<MOD,<PN>,2>,0,"<MYTITLE>',"<PN>'>'>
<STRING,MYTITLE,text>
```

Here, MYTITLE is a string which expands as the desired header text. It is on the right top (and the page number on the left top) whenever the page number is even -- that is, when the page number modulo 2 is zero -- and vice versa when it is odd. To clarify the Eureka scanning algorithm, let us trace through the events at the top of a page.

At the top of a page, Eureka is looking at the following situation:

```
||document text ...
```

The bars represent the divisions of the strings. To the left of the first bar is the neutral string. Between the bars is the scanning string, and to the right of the second bar is the active string.

First, Texture causes the text of the title to be inserted at the front of the active string:

```
||<TITLE>document text ...
```

Eureka scans until it encounters the closing bracket (>):

```
(step 2)    <TITLE||>document text ...
```

The arguments of the call (there is only one) are scanned until an opening bracket (<) is encountered:

```
(step 3)    <|TITLE|document text ...
```

The function TITLE is called with no arguments, and the result is put on the active string (this was an active call):

```
(step 4)    ||<LTITLE><SPLIT><RTITLE>document text ...
```

The process repeats for LTITLE:

```
(steps 2-4) ||<EQ,<MOD,<PN>,2>,0,"<PN>',"<MYTITLE>'> ...
```

Here, the ellipsis (...) Represents the string "<SPLIT><RTITLE>document text ...". Again, step 2 causes the scan to proceed to the first closing bracket:

```
(step 2) <EQ,<MOD,<PN||>,2>,0,...
```

The function PN is evaluated:

```
(step 3-4) <EQ,<MOD,||8,2>,0,...
```

Again, the end of the first complete call is found:

```
(step 2) <EQ,<MOD,8,2||>,0,...
```

This time, each argument of the call is stacked:

```
(step 3) <EQ,<MOD,8,|2|,0,"<PN>',"<MYTITLE>'>...
(step 3) <EQ,<MOD,|8|,0,"<PN>',"<MYTITLE>'>...
(step 3) <EQ,<|MOD|,0,"<PN>',"<MYTITLE>'>...
```

This results in an evaluation stack which looks like:

```
MOD
 8
 2
```

and the string:

```
<EQ,||,0,"<PN>',"<MYTITLE>'>...
```

The evaluation of the stack produces 0 (8 modulo 2), which is put on the active string:

```
(step 4) <EQ,||0,0,"<PN>',"<MYTITLE>'>...
```

The scan continues. Note that modification (a) (to the scanning algorithm) causes the scan to pass over the closing brackets enclosed in quotes:

```
(step 2) <EQ,0,0,"<PN>',"<MYTITLE>'||>...
```

Again, the arguments are scanned and stacked; modification (b) (to the scanning algorithm) causes the scan to pass over the opening brackets enclosed in quotes:

```
(step 3) <EQ,0,0,"<PN>',|"<MYTITLE>'|...
(step 3) <EQ,0,0,|"<PN>'|...
(step 3) <EQ,0,|0|...
(step 3) <EQ,|0|...
(step 3) <|EQ|...
```

Finally, the stack looks like:

```
   EQ
    0
    0
  <PN>
<MYTITLE>
```

This is evaluated: it is true that 0=0, so the third argument "<PN>" is the value, which is placed on the active string:

```
(*)(step 4) ||<PN><SPLIT><RTITLE>document text ...
```

And so evaluation continues: PN will evaluate to the page number, SPLIT will cause an action to take place in Texture and will evaluate to the empty string, and RTITLE will evaluate to the text of MYTITLE. Finally, "document text ... " can be scanned and processed.

3.2 Neutral evaluation

Suppose the definition of LTITLE had been as follows:

```
<LTITLE,":<EQ,<MOD,<PN>,2>,0,"<PN>',"<MYTITLE>'>'>
```

That is, the function EQ has a neutral evaluation symbol (:) in front of it. Then everything would have been rather the same, up to the point marked "(*)" above. Just before this point, the Eureka scanning area looks like:

```
:||<SPLIT><RTITLE>document text ...
```

(that is, the neutral evaluation symbol preceded the call of EQ, which has just been stacked and evaluated). Then, when the value of the EQ is returned (<PN>), this value is put on the neutral rather than the active string, and the result is:

```
(step 4) <PN>||<SPLIT><RTITLE>document text ...
```

A careful scrutiny of the scanning algorithm will convince the reader that this "<PN>" will never be evaluated, but rather that it will be passed on to the document processor. In that case, the text at the top of the page will be the string "<PN>", and not the value of <PN>, which is a number.

This is the difference between neutral (:<fn>) and active (<fn>) evaluation: the result, or value, is not rescanned and therefore if it contains any Eureka programs, is not evaluated any further; the result of an active evaluation is rescanned. The reader will soon realize that this is one way of getting function calls to pass through Eureka (and so to the document processor) without being evaluated. Beyond this, the document processor user has little use for neutral evaluation; however, the Eureka programmer may find neutral evaluation valuable to prevent the values of calls from being themselves evaluated.

3.3 Some Eureka Functions

3.3.1 Lexical comparison

The following functions depend on lexical ordering. It is assumed here that the alphabet over which strings may be formed is ordered in some fashion. (This will differ by location: EBCDIC ordering is slightly different from ASCII ordering, but it is usually fairly logical. Usually, the blank is lowest, and the letters and digits are ordered a<b...<z <A<B...<Z <0<1...<9.)

Two strings aA and bB (where a,b are single characters and A,B are strings) are lexically related as follows:

```
if a < b then aA < bB
if a > b then aA > bB
if a = b then aA r bB if and only if A r B,
                      where r is one of <, >, or =
The empty string is lexically less than any non-empty
string and equal to itself.
```

<EQ,a,b,true,false>

If 'a' is lexically equivalent to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<NE,a,b,true,false>

If 'a' is lexically not equivalent to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<LT,a,b,true,false>

If 'a' is lexically less than 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<LE,a,b,true,false>

If 'a' is lexically less than 'b', or equivalent to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<GT,a,b,true,false>

If 'a' is lexically greater than 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<GE,a,b,true,false>

If 'a' is lexically greater than 'b', or equivalent to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

3.3.2 Numeric computation

As well as the above functions which act on a lexical ordering of the arguments being compared, the following functions are available which operate on an integral comparison of their arguments. If an argument is not integral, a warning message results and the offending argument is replaced by '1'.

<#EQ,a,b,true,false>

If 'a' is numerically equal to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<#NE,a,b,true,false>

If 'a' is numerically not equal to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<#LT,a,b,true,false>

If 'a' is numerically less than 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<#LE,a,b,true,false>

If 'a' is numerically less than or equal to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<#GT,a,b,true,false>

If 'a' is numerically greater than 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<#GE,a,b,true,false>

If 'a' is numerically greater than or equal to 'b', then the value is the value of 'true'; otherwise it is the value of 'false'.

<SUM,a,b>

The value is the integer sum of 'a' and 'b'.

<DIFF,a,b>

The value is the integer difference of 'a' and 'b'.

<PROD,a,b>

The value is the integer product of 'a' and 'b'.

<DIV,a,b>

The value is the integer quotient of 'a' and 'b'.

<MOD,a,b>

The value is the remainder after integer division of 'a' by 'b'; that is:

<DIFF,a,<PROD,<DIV,a,b>,b>>

3.3.3 String manipulation

The following primitives are provided to allow the analysis of a sequence of characters.

<STEM,string,n>

Returns the first 'n' characters of 'string'.

<STERN,string,n>

Returns the last 'n' characters of 'string'.

<TRIM-STEM,string,n>

Returns all but the first 'n' characters of 'string'.

<TRIM-STERN,string,n>

Returns all but the last 'n' characters of 'string'.

<LENGTH,string>

Returns the length of 'string'.

3.3.4 User internal function definition

The following primitives are provided to enable the user to define his own functions.

<STRING,name,text>

This functions defines 'text' to be a string with the name 'name'. Henceforth, whenever "<name>" appears on the active string (see semantics, page 13) evaluation will replace it with "text".

<SEGMENT,name,gap1,gap2,...,gapn>

This function 'segments' the string with the name 'name' on the various 'gap's, which are strings. Segmentation is done as follows:

For each 'gap', every segment of 'name's string is checked for an occurrence of 'gap' in that segment; for each such occurrence the segment is broken into two new segments, the part before the occurrence and the part after. Between these two segments there is created a numbered segment gap (if it was created by matching the k'th argument, then it is the segment gap numbered k).

For example:

<STRING,string,most people like cheese>

creates a string (named 'string') consisting of a single segment, which we will represent as:

(most pecple like cheese)

If this is followed by the command:

<SEGMENT,string,e,o>

the result will be the following string:

(m)2(st p)1 2(pl)1(lik)1(ch)1 1(s)1

Evaluation of the text "<string>" will now result in the text:

mst ppl lik chs

However, simple evaluation of the text "<string,$!>" will result in the text:

```
                    mst p$!pl$!  lik$!  ch$!$!s$!
```

and "<string,a,b>" results in:

```
                       mbst pabpla lika chaasa
```

That is, the first argument replaces the gaps numbered 1 and the second argument replaces the gaps numbered 2. This can be carried on for as many gaps as there are, of course. Missing arguments are considered as an empty string where the missing argument is a string argument and a 1 where the missing argument is a numerical argument, while excess arguments (those for which there is no gap) are ignored. Thus <SUM,a> will add 1 to a.

It is worth noting that, since the segmentation strings called 'gap' above are taken in left to right order, the pair of commands:

```
                    <STRING,foo,abcabcabcabc>
                      <SEGMENT,foo,ab,bc>
```

will result in 'foo' taking the internal form,

```
                          1(c)1(c)1(c)1(c)
```

after 'ab' has segmented it, and that 'bc' will thereafter fail to segment 'foo' any further. If the arguments had been reversed, as in:

```
                      <SEGMENT,foo,bc,ab>
```

the result would have been instead:

```
(a)2(a)2(a)2(a)2
```

The functions STRING and SEGMENT are, of course, how the Eureka programmer defines macros with parameters. We could define a simple definition function as follows:

```
    <STRING,define,"<STRING,|name|,"|text|'>
                 <SEGMENT,|name|,|parameters|>'>
    <SEGMENT,define,|name|,|parameters|,|text|>
```

which is called as follows:

```
    <define,repeat,"what,howoften',
          "<LT,0,howoften,
               "what<REPEAT,"what',<DIFF,howoften,1>>'>'>
```

The above call of 'define', incidentally, defines a useful function called 'repeat' which will concatenate 'howoften' (an

integer number) evaluations of a given string 'what' (which may in turn be a Eureka program).

For example, the call:

```
     <repeat,+,5>
```

expands as follows (these 'snapshots' of the Eureka scanning area give the situation at the start of each new application of step 2 of the algorithm):

```
   <repeat,+,5>
   <LT,0,5,"+<repeat,+,<DIFF,5,1>>'>
   +<repeat,+,<DIFF,5,1>>
   +<repeat,+,4>
   +<LT,0,4,"+<repeat,+,<DIFF,4,1>>'>
   .  .  .
   +++++<repeat,+,0>
   +++++<LT,0,0,"+<repeat,+,<DIFF,0,1>>'>
   +++++
```

3.3.5 User external function definition

Although Eureka provides a large variety of primitives, it is occasionally neccessary for the user to do something unusual which Eureka will not conveniently allow. To assist in this regard Eureka has the following primitive to allow the user to define and use his own external functions.

```
<SYSTEM,eureka-name,library,MTS-entry-name>
```

This primitive associates 'eureka-name' with a code segment which has an entry point 'MTS-entry-name'. The code segment is found by loading 'MTS-entry-name' from the MTS file 'library'. At any occurrence of the call <eureka-name,par1,par2,...parn> after the corresponding SYSTEM call, Eureka builds a string descriptor for each of 'par1', 'par2',..., 'parn' which is a half-word followed by a sequence of characters (the length of which is contained in the half-word). A sequential list of n full-words is then built to contain the addresses of the parameters with the i'th full-word containing the address of the i'th parameter. The address of the sequence of full-words is then put into general register 1 and control branches to 'MTS-entry-point'. The address of a standard 18 word save area is supplied in general register 13. On return Eureka expects that either general register 0 has the value 0 (returns the null string) or general register 0 points to a full-word containing the address of a string descriptor of the type described above (returns a non-null <u>string</u> value).

If the 'library' parameter is omitted, the file TEXT:UREFNS is used (see Appendix C for the description of its contents).

If 'MTS-entry-name' is omitted, it is assumed to be the same as 'eureka-name'.

In order to aid with communication between a SYSTEM defined function and the document processor the entry points TXTERR and IADROF are supplied. TXTERR enables the user to supply an error message to the document processor error routines. For example:

```
CALL TXTERR(code,message,length)
```

would be a valid call from a FORTRAN program. If code has the value 0, the message is treated as a warning (i.e. the call returns); otherwise the message is treated as an error (i.e. the call will not return, processing halts). The parameter 'message' specifies a character string to be printed whose length is given by the parameter 'length'.

The IADROF entry point returns the address of its parameter and is used as follows in a FORTRAN program:

```
J=IADROF(P)
```

After execution, J contains the address of P (P can be of any type).

3.3.6 Function deletion and garbage collection

The following functions allow the user to discard functions, blocks, layouts and environments which have been defined and are no longer needed.

<DELETE-STRING,string1,string2,...,stringn>

The strings 'string1', 'string2',..., 'stringn' are deleted from Eureka's definition space and will no longer be recognized.

<DELETE-BLOCK,block1,block2,...,blockn>

The blocks 'block1', 'block2',..., 'blockn' are deleted from Eureka's definition space and will no longer be recognized.

<DELETE-LAYOUT,layout1,layout2,...,layoutn>

The layouts 'layout1', 'layout2',..., 'layoutn' are deleted from Eureka's definition space and will no longer be recognized.

<DELETE-ENVIRONMENT,environment1,environment2,...,environmentn>

The environments 'environment1', 'environment2',..., 'environmentn' are deleted from Eureka's definition space and will no longer be recognized.

<DELETE-SYSTEM,name1,name2,...,namen>

The functions 'name1', 'name2',..., 'namen' (which must have been defined using the SYSTEM primitive) are deleted from Eureka's definition space and will no longer be recognized. In addition, the corresponding user supplied code segments will be unloaded.

<RECLAIM>

The garbage collector is invoked and all definitions deleted via any of the DELETE-xxx primitives are cleaned out of the definition space.

<DEFINITION-SPACE-SIZE,n>

At the next garbage collection (either user or system invoked), the new definition space will have a size of 'n' pages ('n' must be an integer). If called without any arguments, this function returns the current size of the definition space.

3.3.7 Eureka programs in Texture text

When defining a Eureka program within Texture text it is often desirable that the sequence of blanks which begin and end a source line be deleted and not be considered as part of the Eureka program. In this way, an indentation scheme which makes the Eureka program readable can be used without any worry that indentation will cause extra blanks to be included as part of the Eureka program. To do this, there are the following primitives:

<TRIM-LINE-STEM>

All source lines which are part of the SOURCE stream will have all leading blanks deleted before they are fed into Eureka.

<NOTTRIM-LINE-STEM>

All source lines which are part of the SOURCE stream will not have all leading blanks deleted before they are fed into Eureka.

<TRIM-LINE-STERN>

All source lines which are part of the SOURCE stream will have all trailing blanks deleted before they are fed into Eureka.

<NOTTRIM-LINE-STERN>

All source lines which are part of the SOURCE stream will _not_ have all trailing blanks deleted before they are fed into Eureka.

3.3.8 _Eureka input and output_

The following primitives provide Eureka with the ability to do input and output operations which are independent of the document source.

<RS,eof>

The value of this function is the next string of text, from the current input medium, up to but not including an end-of-string marker. By default, the input medium is the document source stream; the end-of-string marker is a full stop (.). If end of file is encountered before the end-of-string marker, the value of the call is the argument 'eof', which is actively evaluated.

<RC,eof>

The value of this function is the next character from the current input medium. The argument 'eof' has the same meaning as for RS.

<RNC,n>

The value of this function is the next 'n' characters from the current input medium. If there are less than 'n' characters left in the input medium, the function returns all the remaining characters together with as many alternating NEWLINE and EOF characters as are required to fill out the required 'n' characters.

<RLN>

This function returns the rest of the current source line if the current source line is non-empty; otherwise it returns the next source line. If the next source line is the end-of-file, the function returns the EOF character.

<RFN,fail>

This function reads in the next balanced Eureka function call (either active or neutral). The call being read may not be preceded by anything other than blanks, or an error condition will occur. If an end-of-file is encountered instead of a function call, 'fail' is returned; while a badly formed function generates an error.

<PRINT,string>

The argument 'string' is output to the current output medium. The value is the null string. By default, the output medium is the document processor, so that "<PRINT,string>" is equivalent to "string", while "<PRINT,abc<PRINT,def>hjk>" is equivalent to "defabchjk" (because the argument is evaluated first). The output medium may, however, be set to another file or device.

<INPUT,name>

After this call, all the read primitives will read from the file or device named by 'name'. If the argument 'name' is absent, the document source file again becomes the input medium.

The usefulness of this function may vary from operating system to operating system. It may be used to read from a file attached to one of the available input units however, and so merge input from two separate sources together in the stream. It should be remembered that RS peels off the end-of-string marker; naturally the end-of-string marker can be set to any convenient character.

<OUTPUT,name>

After this call, PRINT will print into the file or device 'name'. If 'name' is absent, the document processor becomes the current output medium.

<SET-EOS,c>

The end-of-string character for RS is set to 'c'.

For example, the user may be producing form letters. The names and addresses of the people to whom these form letters are to be sent are in a file called 'VICTIMS'.

Thus the complete Texture input might look like (in terms of the sample user function 'define', page 22):

```
<define,LETTER,"|name|,|address|',
                "<MYADDRESS>
                 <L>
                 |name|
                 <L>
                 |address|
                 Dear |name|;
                 <L>
                 <TEXT OF LETTER>'>
<define,MYADDRESS,etc...>
<Define,TEXT OF LETTER,etc...>
<INPUT,VICTIMS>
<repeat,"<LETTER,<RS>,<RS,"<PAGE><BYE>'>><PAGE>',10000>
```

3.3.9 Miscellaneous functions

The following are miscellaneous functions which are often useful in producing documents.

<TIME>

The value is the time of day in the form 'hh:mm:ss' on the twenty-four hour clock with 'hh' being the hours, 'mm' the minutes and 'ss' the seconds. (Note that all six digits are always given.)

<DATE>

The value is the date in the form 'mmm dd, 19yy' where 'mmm' is a three character month abbreviation, 'dd' is a two digit date (the first digit of which is blank if the date is a single digit) and 'yy' is the last two digits of the current year.

<SYSPARS>

This function returns the parameter list as specified by the PAR= field on the Texture run command.

<WARNING,message>

This causes the document processor to print 'message' as a warning message on whatever unit is the current LIST unit. Since the message is a warning and not an error, control will return to the point of the call. The function returns empty. All warning messages produced in this fashion are prefixed with '<><><>'.

<ERROR,message>

This causes the document processor to print 'message' as an error message on whatever unit is the current LIST unit. Since this message is an error, all processing will halt after the message has been printed. An error message produced in this fashion is prefixed with '<><><>'.

<WARN> and <NOTWARN>

The action of these two primitives is to turn the suppression of the printing of warning messages on and off. After an occurrence of NOTWARN all warning messages will be suppressed until a WARN is encountered.

<STATISTICS> and <NOTSTATISTICS>

The action of these two primitives is to control whether or not to print the six lines of statistics at the end of a run. If STATISTICS is in effect at the end of a run, the statistics are printed; if NOTSTATISTICS is in effect at the end of a run, the printing of statistics is suppressed.

<MTS-LINE>

This function returns the MTS line number of the last SOURCE line read in. The line number is returned in the format ' ddddd.ddd' with up to three trailing zeros removed and the '.' Removed if the last three digits are zeros. This primitive can be used together with LINE (see section 4.4) and MAX-BLOCKS to put line numbers out beside the blocks in the following manner:

```
<STRING,CUR-LN-#,><MAX-BLOCKS,70>
<HANG,LINE,"<NE,:<MTS-LINE>,:<CUR-LN-#>,
                     "<STRING,CUR-LN-#,:<MTS-LINE>>
                      :<LINE,:<MTS-LINE>,
                              :<SUM,:<RIGHT>>>'>'>
```

The increase in the number of blocks is necessary in that Texture retains any text given to it via each call of the LINE primitive as a separate block.

<BYE>

Execution of Eureka (and therefore Texture) terminates immediately. Since this could result in part of the current page being lost, it is a good idea to prefix the call to BYE with a call to PAGE.

3.3.10 Altering the Eureka special characters

Up to this point, Eureka has been described entirely using the symbols (< :< > ' " ,). Occasionally it is inconvenient to have these symbols as special characters, forcing the user to prefix the characters with an asterisk. In order to alleviate this problem, the following SET-op commands allow the user to change the characters which Eureka recognizes.

<SET-FNOPEN,c> 'c' is henceforth treated as the start-of-function character the way '<' is by default.

<SET-FNCLOSE,c> 'c' is henceforth treated as the end-of-function character the way '>' is by default.

<SET-ARGSEP,c> 'c' is henceforth treated as the argument separator character the way ',' is by default.

<SET-LITBEGIN,c> 'c' is henceforth treated as the start-literal character the way " is by default.

<SET-LITEND,c> 'c' is henceforth treated as the end-literal character the way ' is by default.

<SET-NTRLINDIC,c> 'c' is henceforth treated as the neutral-indicator character the way ':' is by default.

4. The Rest of Texture

4.1 Environments

Some sections of a document (such as footnotes and figures) are in their own environments. This means that the text and commands of one of these are processed with certain global switches set independently of the text surrounding them. As soon as, say, a footnote is entered, the old values of the global switches involved are saved on a stack of environments and a brand-new environment with default settings comes into force. Included in an environment are the following:

left and right indents
underlining switch
case-shift switches
paragraph indent
word, sentence and line spacing
tab-stops
fill character
AUTOCAP mode

justification method

The user is able to forcibly interchange environments by the use of the following Texture primitives:

<ENVIRONMENT,name>

The current environment is saved under the name 'name'.

<ACTIVATE,name>

The current environment is replaced by the environment named 'name'.

These two functions can be useful in setting up an environment for footnotes (for example), and recalling this environment whenever a footnote is entered.

To accomplish this, one might, for example, give the sequence of instructions:

```
<ENVIRONMENT,save>
 .
 .  Some instructions to set up the footnote environment
 .
<ENVIRONMENT,foot-environment>
<ACTIVATE,save>
<define,footnote,|end|,"<ENVIRONMENT,current-environment>
                       <ACTIVATE,foot-environment>
                       <FOOT,|end|>
                       <ACTIVATE,current-environment>'>
```

Henceforth, where 'FOOT' would have been called, 'footnote' is called instead in exactly the same way.

4.2 Layouts

One of the most important aspects of Texture is that the user can control the layout of his text. This is done by defining a layout as follows:

<LAYOUT,name,part1,part2,...,partn>

This defines a layout named 'name', consisting of n 'part's. Each 'part' is the name, either of another layout or of a block. The layout is defined as consisting of all the blocks and layouts whose names are given, in the order in which they are given. For each layout that is part of a layout definition, the blocks of which the sub-layout consists replace the sub-layout. Thus if layout A consists of (X,B,Y,Z) and layout B consists of (U,V,W), then layout A consists of (X,U,V,W,Y,Z).

To invoke a layout (that is, to make a given layout the active layout) use the INVOKE command.

```
<INVOKE,layout>
```

'layout' is the name of a layout and will become the active layout for the next output page and any subsequent pages until another call of INVOKE. Thus, if one wanted a layout to become immediately active, it would be necessary to command:

```
                    <INVOKE,layout><PAGE>
```

When the processor begins to assemble a new page of output text, a copy of the active layout called the current layout is made. Thus all text is assembled into the current layout and any modifications made to the page structure (cf. Temporary blocks) are made to the current layout only.

4.3 Blocks

4.3.1 Standard blocks

```
<BLOCK,name,left,right,top,bottom,text1,text2>
```

This defines a block. A block is a segment of the printed page, going from column 'left' to column 'right' and from line 'top' to line 'bottom'. Inside the space defined by the block there is room to put text. The first text that goes into any block is the sixth parameter, 'text1' (which may be an empty string) which is evaluated in the document processor's default environment. The next text to enter the block is the seventh parameter, 'text2' (which may be an empty string) which is evaluated within the environment which is active at block entry. After this, text from the SOURCE stream is put into the block. To put this into perspective, Texture defines a standard layout as follows:

```
     <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>
```

This results in the page you are now studying, and is probably

adequate for most documents. Notice that all these blocks go from columns 5 to 68, and that the header and footer blocks are only one line deep. This means that only one line of header and one line of footer are possible. Notice also that the standard layout could define its three constituent blocks to be in any order and it would still produce the same layout but not the same results. For example if the footer is before the text, then any change in the footer is not felt until the next page; but if it is after the text, any changes are felt on the same page. The current block is defined as that block into which Texture is currently assembling text.

```
<MAX-BLOCKS,n>
```

This primitive sets the maximum number of blocks which is allowed in a layout (default is 20) to 'n'.

In order to allow the user to determine aspects of a block, the following block enquiries are also given.

```
<LEFT,block-name>
```

The left-most column number in block 'block-name' is returned. If 'block-name' is omitted, the current block is assumed. Note that 'block-name' must be a block in the current layout.

```
<RIGHT,block-name>
```

The right-most column number in block 'block-name' is returned. If 'block-name' is omitted, the current block is assumed. Note that 'block-name' must be a block in the current layout.

```
<TOP,block-name>
```

The line number corresponding to the top of 'block-name' is returned. If 'block-name' is omitted, the current block is assumed. Note that 'block-name' must be a block in the current layout.

```
<BOTTOM,block-name>
```

The line number corresponding to the bottom of 'block-name' is returned. If 'block-name' is omitted, the current block is assumed. Note that 'block-name' must be a block in the current layout.

<NEXT,justification-method>

The current block is terminated immediately using 'justification-method' to justify the last non-blank line in the block. If 'justification-method' is omitted, it defaults to RAGRIGHT if the current global justification method is JUSTIFIED, otherwise it defaults to the current global justification method.

<LINES-LEFT>

The value of this function is the number of physical lines remaining in the current block.

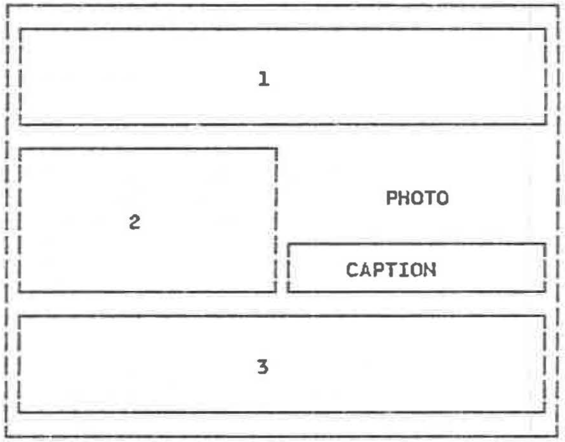
<COLS-LEFT>

The value of this function is the number of columns remaining in the current line.

4.3.2 Temporary blocks

Up to this point, all blocks and block features have been described in static terms; once the boundaries of a block have been set, they can no longer be changed in any way. This is not always a desirable situation as is shown by the following example.

Suppose a layout is desired in which a photograph is to be placed under which is to appear a caption describing the photograph, such as:



In many cases, it is not known ahead of time how long the caption for the photograph is to be, and it would be desirable if the CAPTION block were flexible (depending on the amount of text in the caption) and both the bottom of block 2 and the top of block 3 depended on where the bottom line of the caption ends up. What is needed then are two things: a method for "cutting up" larger blocks into smaller blocks and a method for specifying if this "cutting" is fixed (all dimensions known at the time of the "cut") or flexible (vertical height of "cut out" block is not known at the time the "cut" is made). These new temporary blocks could then be inserted into the current layout, thus modifying the current page dynamically.

It should be noted at the outset that the creation of these temporary blocks would only affect the current layout and that when the page is completed, printed and the new current layout for the next page is created, the new current layout is a copy of the active layout which has not been affected by any "cutting".

The following primitives enable the user to make these modifications to the current layout.

<V-CUT,bname,col,l-bname,r-bname>

This cuts the block named by 'bname' ('bname' must be a block in the current layout) vertically at column 'col' and assigns the name 'l-bname' to the block to the left of and including the the vertical cut column and 'r-bname' to the block to the right of the vertical cut column and replaces 'bname' by 'l-bname' and 'r-bname' (in that order) in the current layout.

If col is omitted, it defaults to the current column (i.e. to the value of <COLUMN>, see section 4.4) and if 'bname' is omitted, it defaults to the current block. If an attempt is made to cut through any part of a block into which Texture has already placed text, or if the value of 'col' is not within the boundaries of 'bname', a warning is given and the cut is not made.

<V-CUT-SWAP,bname,col,l-bname,r-bname>

The action of this primitive is identical to that of V-CUT with the exception of the fact that 'bname' is replaced by 'r-bname' and 'l-bname' (in that order) in the current layout which is the reverse of V-CUT.

<H-CUT-FLEX,bname,t-bname,b-bname>

This cuts the block named by 'bname' ('bname' must be a block in the current layout) horizontally and flexibly and replaces 'bname' by 't-bname' and 'b-bname' (in that order) in the current layout. The exact line at which the cut is made is not set at this time, but when 't-bname' is entered, its size is continually increased as text is entered. Any explicit use of the <NEXT> command will cause the bottom of 't-bname' and the top of 'b-bname' to be fixed at the line which Texture is assembling when the <NEXT> command is encountered. If a <NEXT> command is not encountered, 't-bname' will automatically be exited when the size of 't-bname' reaches the size of the original block ('bname') from which it was formed. In this case, 'b-bname' is regarded as a zero height block.

<H-CUT,bname,line-num,t-bname,b-bname>

This cuts the block named by 'bname' ('bname' must be a block in the current layout) horizontally at line 'line-num' and assigns the name 't-bname' to the block above and including the line of the cut and 'bname' to the block below the line of the cut and replaces 'bname' by 't-bname' and 'b-bname' (in that order) in the current layout. If 'bname' is omitted, it is assumed to be the current block; if 'line-num' is omitted it is assumed to be the number of the line which Texture is assembling. If 'bname' has more than one flexible edge, then an error will result from any attempt to cut 'bname'.

Any attempt to cut through the part of a block which contains text or any value of 'position' which is not within the boundaries of 'bname' causes a warning to be given and the cut is not made.

<REMOVE-FROM-LAYOUT,bname>

This primitive removes the block 'bname' ('bname' must be a block in the current layout) from the current layout. An error is generated if 'bname' is not a block in the current layout.

<MANDATORY-TEXT,bname,str1,str2>

The mandatory text associated with 'bname' ('bname' must be a block in the current layout) becomes 'str1' and 'str2' where 'str1' becomes the mandatory text to be evaluated in the document processor's default environment and 'str2' becomes the mandatory text to be evaluated in the environment which is active just prior to the entry of 'bname'. If 'bname' has already been entered, a warning is generated and the mandatory text of 'bname' is not altered.

Returning to the above example of the photograph, the desired effect can be achieved via the following sequence of commands (assuming the outermost block has been entered and no more than twenty lines processed):

```
<H-CUT,,20,stand-textt,stand-textb>
<H-CUT-FLEX,stand-textb,stand-textbt,stand-textbb>
<V-CUT-SWAP,stand-textbt,35,stand-textbtl,stand-textbtr>
<H-CUT,stand-textbtr,30,photo,caption>
<REMOVE-FROM-LAYOUT,photo>
```

4.4 Accessing absolute page locations

Occasionally it is useful to be able to override the notion of blocks and instruct the document processor to put a piece of text at an absolute location on the current page. Texture gives the following two primitives for doing this.

<LINE,str,col-num,line-num>

The string 'str' is put into the current page on line 'line-num' starting at column 'col-num'. The values of 'line-num' and 'col-num' need not be within the range of the current block. If 'line-num' is absent (i.e. only two parameters given) then 'line-num' defaults to the current line into which Texture is assembling text. If both 'col-num' and 'line-num' are absent (i.e. only one parameter given) then 'line-num' defaults as above and 'col-num' defaults to the current column where Texture is assembling text. If LINE is called with no parameters it returns the value of the current line number.

<COLUMN,str,line-num,col-num>

The string 'str' is put into the current page in column 'col-num' starting at line 'line-num' and working vertically downward. If 'col-num' is absent (i.e. only two parameters given) then 'col-num' defaults to the current column where Texture is assembling text. If both 'line-num' and 'col-num' are absent (i.e. only one parameter given) then 'col-num' defaults as above and 'line-num' defaults to the current line into which Texture is assembling text. If COLUMN is called with no parameters it returns the value of the current column number.

4.5 Lines and boxes

In many documents it is very useful to be able to present tables of data in a neat and orderly fashion. Often this is done through the use of boxes, setting off sections of data via dividing lines. Although the ability to just draw tables is suitable for most applications, there are cases where the user might want only half a table, or arrow pointers or any other structure which involves the concept of a line.

In this regard, Texture allows the user to make up any network of horizontal and vertical lines and will see to it, if told, that if any of these lines cross that a suitable "crossing character" will be used at this point. The following primitives provide these features:

<H-LINE,lnum,left-col,right-col,str>

A horizontal line is drawn from column 'left-col' to column 'right-col' along line 'lnum' of the current layout. The string 'str' is used to build this line by overprinting all characters of 'str' at each location along the line. The default value of 'lnum' is the current line number (the value of <LINE>), of 'left-col' is the left edge of the current block (the value of <LEFT>), of 'right-col' is the right edge of the current block (the value of <RIGHT>) and of 'str' is '—'.

<V-LINE,colnum,top-line,bottom-line,str>

A vertical line is drawn from line 'top-line' to line 'bottom-line' along column 'colnum' of the current layout. The string 'str' is used to build this line by overprinting all characters of 'str' at each location along the column. The default value of 'colnum' is the current column number (the value of <COLUMN>), of 'top-line' is the top of the current block (the value of <TOP>), of 'bottom-line' is the bottom of the current block (the value of <BOTTOM>) and of 'str' is '|'.

<POINT,col-num,line-num,str>

A point is placed at co-ordinate (line-num,col-num) of the current layout. All characters of 'str' are overprinted at this point. The default value of 'col-num' is the current column (the value of <COLUMN>), of 'line-num' is the current line (the value of <LINE>) and of 'str' is a blank unless the point coincides with an existing line in which case the corresponding join character is used (cf. JOIN).

<JOIN,joint,str>

This primitive determines what character or string of characters is to be used when lines overlap in various ways. The characters of 'str' are overprinted at any point where joint 'joint' occurs. The possible values of 'joint' and the corresponding default values for 'str' are as follows:

TL or LT. Two lines meet to form the upper-left corner of a box. The default string is '┌'.

TM or MT. Two lines meet to form an upright T joint. The default string is '—'.

TR or RT. Two lines meet to form the upper-right corner of a box. The default string is '┐'.

ML or LM. Two lines meet to form a left side T joint. The default string is '|'.

MM. Two lines intersect to form a "plus" joint. The default string is '┼'.

MR or RM. Two lines meet to form a right side T joint. The default string is '|'.

BL or LB. Two lines meet to form the lower-left corner of a box. The default string is '└'.

BM or MB. Two lines meet to form an upside down T joint. The default string is '—'.

BR or RB. Two lines meet to form the lower-right corner of a box. The default string is '┘'.

HH. Two horizontal lines overlap. The default string is '—'.

VV. Two vertical lines overlap. The default string is '|'.

H. 'str' becomes the new string for drawing horizontal lines. The default string is '—'.

V. 'str' becomes the new string for drawing vertical lines. The default string is '|'.

If the second parameter is omitted, JOIN will return the current value of the string which is being used for the joint (the second parameter). If all the parameters are missing, the automatic joining facility which will join all crossing lines as described above is turned on. The default mode of operation is with automatic joining on.

<NOTJOIN>

The automatic joining facility is turned off. All crossing joints will not be replaced by the joint string, but will just consist of the overprinting of the characters used to make up each point of the lines.

4.6 Footnotes

The details of how footnotes are handled within the Texture system are very complicated; but when using the STANDARD-LAYOUT, these details are irrelevant and misleading. For this reason, the use of footnotes within the standard layout is described first, followed by a description of what happens in the general case when an arbitray layout is being used.

<FOOT-FOOT,string>

This associates with the name 'FOOT-FOOT' the value 'string'. Any use of <FOOT-FOOT> after this point returns 'string'.

<TEXT-FOOT,string>

This associates with the name 'TEXT-FOOT' the value 'string'. Any use of <TEXT-FOOT> after this point returns 'string'.

<FOOT,end-string> For the STANDARD-LAYOUT

All text following the occurrence of the FOOT primitive up to the next occurrence of 'end-string' is treated as a footnote and is placed at the bottom of the STANDARD-TEXT block. The string which is the current value of TEXT-FOOT is used to separate the footnote from any text in the STANDARD-TEXT block. If the footnote does not fit within the STANDARD-TEXT block because Texture has already filled most of the block, the remaining text is placed at the bottom of the STANDARD-TEXT block of the next page. No more than the bottom half of the block is ever used for footnotes and if more than one footnote is encountered while building the same page, the second footnote is placed below the first footnote, any overflow going onto the next page. The string associated with FOOT-FOOT is placed between footnotes if more than one footnote occurs on the same page.

<FIRST-FOOT,yes-str,no-str,not-a-footnote-str>

If called from within a footnote, this primitive returns 'yes-str' if this is the first footnote on the page and 'no-str' otherwise. If this primitive is called but not from within a footnote, it returns 'not-a-footnote-str'.

<TEXT-AHEAD,yes-str,no-str>

This primitive returns 'yes-str' if there are footnote layouts which have been created to hold the footnote overflow from the current page, otherwise it returns 'no-str'.

In order to describe how footnotes are handled in the more complicated case of an arbitrary layout, it is first necessary to define a few terms.

The original block (defined for any point in the layout) is the block which contained that point before any cutting was done to the current layout.

Two layouts are merged if they are brought together to form a single page by removing from one of the layouts any areas which intersect with the other layout.

A layout is said to be dominant in a merge if it retains the areas of intersection of the two layouts, thus causing the other layout to lose the areas of intersection. This can be looked upon as if the non-dominant layout "shrinks away" from the dominant one; changes being made to the dimensions of the blocks in the non-dominant layout as necessary.

<FOOTNOTES-USE,layout-name>

This primitive specifies that 'layout-name' is the layout into which Texture is to build footnotes.

The following is a more complete description of what occurs within the FOOT primitive when a layout other than STANDARD-LAYOUT is being used.

<FOOT,end-string> Extended for arbitrary layouts

All text following this primitive up to the first occurrence of 'end-string' is treated as a footnote. The bottom of the original block of the point at which the FOOT primitive was encountered is found. The footnote layout is then searched for the first occurrence of a block B which would contain part of the bottom line of the original block were the two layouts overlayed.

If the line which Texture is currently assembling is further down the page than the top of B, then the top of B is decreased by setting it to be the current line + 1. The text of the footnote is then fed into B in the same manner in which text is always fed into a block, including the release of any mandatory text associated with B.

If the footnote fits entirely within B, then the current layout and B are merged with B dominant. If the footnote overflows B, then when B is exited, the footnote layout and the current layout are merged with the footnote layout dominant. The remainder of the footnote is then fed into the block of the footnote following B. This could cause a new copy of the footnote layout to be made; for instance, if B was the last block of the footnote layout, the next block of the footnote layout is the first block of the next copy. This process is continued, making as many copies of the footnote layout as necessary.

If this occurs, then when processing returns to the document and a new page is started causing a new current layout to be made from the active layout, then the new current layout is merged with the next footnote layout in the list of footnote layouts which have already been processed ahead. The footnote layout is dominant in this merging which occurs before any other processing is done in the newly created current layout.

If further footnotes are encountered, processing is done in the same manner as expessed above including re-entry of whatever block is determined for B with the exception that if another footnote has already been merged into the current layout, then the top of B is decreased to the current line + 1 + number of lines used by the previously merged footnote. The merged footnote is considered to be "floating" in that it is always located directly below the line which Texture is assembling; any new merged footnotes then are merged below any previous footnotes.

The default value for the footnote layout is described in Appendix A.

<MIN-MERGE-CUT,n>

Whever two layouts are merged, it is possible that the non-dominant layout becomes quite fragmented. The 'MIN-MERGE-CUT' primitve allows the user to specify a width 'n' such that any blocks with width less than 'n' which arise during a merging process are deleted from the current layout.

4.7 Modifying Texture's I/O stream

In order to modify the location from which the document processor is taking its source document stream, the following primitives are provided.

<SOURCE,name>

After this call, the document source file becomes 'name'. If 'name' is absent, the source file becomes the file attached to SCARDS.

<SINK,name>

After this call, the document processor output file becomes 'name'. If name is absent, the output file becomes the file attached to SPUNCH.

4.8 Events

There are a number of occurrences which it would be very useful to be warned about by the processor. There are times, for instance, when one would like to know that the processor has just finished a line and is about to start on the next one; at this point, one would like to insert some text. For this reason, Texture defines events.

<HANG,event,text>

The argument 'text' is associated with the event named by the argument 'event'. Whenever that event occurs, the text is inserted into the stream. The 'event' may be one of the following:

LINE	new line
BLOCK	new block
PAGE	new page
EOL	end of source line
EOF	end of source file

Note that <HANG,event,a><HANG,event,b> is equivalent to

<HANG,event,ab>. Since it is desirable that a source end-of-file cause completion of all processing, the string '<PAGE><BYE>' is initially hung on the EOF event. If HANG is called without the second argument, it returns the string of text currently hanging on 'event'.

<EMPTY,event>

Any text associated with 'event' is discarded. An occurrence of 'event' will not cause any text to be inserted into the stream. 'event can be any one of the events defined for HANG.

4.9 Miscellaneous functions

<UND,chars> and <NOTUND,chars>

The argument 'chars' is a string of characters. After a call to UND, the characters in 'chars' will be added to those to be underlined whenever underlining mode is active (i.e., Between a <U> and <NOTU>). After a call to NOTUND, the characters in 'chars' will be deleted from those to be underlined whenever underlining mode is active.

<WIDOW,n>

This function is called WIDOW although it actually helps prevent "widows", a typesetting term which means that some small amount of text is awkwardly left on one page when it belongs with a body of text on the next or previous page. In effect, this function will cause a jump to the next block if there are at the moment of the call no more than 'n' physical lines remaining in the current block. Notice that <WIDOW,n> is equivalent to <#LT,<LINES-LEFT>,n,"<NEXT>'>.

<MIN-WS,n>

The minimum word spacing (the least number of characters that are to separate words) is set to 'n'. By default, this value is 1, which is why most words in this document are separated by one blank. If 'n' is absent, the current minimum word spacing value is returned.

<MAX-WS,n>

The maximum word spacing (the greatest number of characters that are to separate words, after justification) is set to 'n'. By default, this value is 5.

If it is not possible to justify a given line with at most 'n' blanks between words, the justification routine gives up, issues a message to that effect, and sets the line ragged right. This is no solution to the problem of excessive spacing, of course, but it is often better than having a line come out unreadable because of unreasonable spaces between words. If the user cares more about flush right margins than about spacing, he need only set 'n' to some enormous value. If 'n' is absent, the current maximum word spacing value is returned.

<MIN-SS,n>

The minimum number of spaces between sentences is set to 'n' (this value is the standard 2 spaces, by default). A sentence is defined as ending in a full stop and a word-terminator (a full stop and a blank, usually), where a full stop is one of '.' '!' or '?'. If 'n' is absent, the current value is returned.

<PAGE-DEPTH,n>

The page depth is set to 'n'. For a normal line printer, in most installations, the default value will be about 60. If this is not so, it can be reset to the correct value by using this function. Where possible, Texture will attempt to print all 'n' lines of the page contiguously; this means that if a given installation's printer skips to a new page after 60 lines, but permits this skip to be overridden by carriage control, Texture will override the skip. This can be useful for printing, say, 100-line pages, of two columns, and photo-reducing these, for conference proceedings, etc.

<PARAGRAPH-INDENT,n>

The paragraph indentation value is set to 'n'. By default, this value is the usual, secretarial 5 columns. If 'n' is absent, the function returns the value of the current setting.

<AS-IS,end-string>

The text following this call, up to the next occurrence of 'end-string' is treated "as given". Each line is output as though each character of the input stream from the AS-IS call to the to the 'end-string' was prefixed by the literal-next character. Thus no Eureka programs in the SOURCE stream are evaluated, but events can still occur which cause Eureka to evaluate a Eureka program.

4.10 Setting Texture's special purpose characters

Occasionally in a document it is useful to be able to modify the characters which Texture uses for special purposes. The following primitives aid in this regard.

<SET-FILLER,c>

The filler character (by default, a blank) is set to 'c'. The filler character is put between words, between segments, between the left margin and the first word, and between the last word and the right margin. Thus changing the filler character (e.g., To a ".") Before a tab, will have the effect of creating a tab-drop character. Naturally the filler should be set back to a blank as soon as the tab is completed, or it will be inserted everywhere.

<SET-UNDERSCORE,c>

The underscore character (by default, _) is set to 'c'. Whenever a character is to be underscored (i.e., After the occurrence of the underscore operator, or after a <U>) it will now be overprinted with a 'c'.

<SET-NTC,c>

The non-trivial character (by default, a blank) is set to 'c'. The non-trivial character operator (by default, ¬) is henceforth replaced by a 'c'.

Appendix A

Defaults

The document processor defaults are set by feeding a stream of Eureka functions through the processor. The following is a list of those functions which are processed, given so that the document processor defaults can easily be seen.

```
<LTITLE,><RTITLE,"<PN>'>
<TITLE,"<LTITLE><SPLIT><RTITLE>'>
<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>
          <FOOT-FOOT,>
          <TEXT-FOOT,"<L>--------------------<L>'>
          <BLOCK,STANDARD-FOOTNOTE,5,68,32,58,
                  "<FIRST-FOOT,"<TEXT-FOOT>',"<FOOT-FOOT>'>'>
          <LAYOUT,STANDARD-FOOTNOTE-LAYOUT,STANDARD-FOOTNOTE>
          <FOOTNOTES-USE,STANDARD-FOOTNOTE-LAYOUT>
          <AUTOCAP>
          <DEFINITION-SPACE-SIZE,3>
          <DOWN>
          <HANG,EOF,"<PAGE><BYE>'>
          <JOIN>
          <JUSTIFIED>
          <LI,0><RI,0>
          <LINESPACING,0>
          <MAX-BLOCKS,20>
          <MAX-WS,5><MIN-WS,1>
          <MIN-MERGE-CUT,20>
          <MIN-SS,2>
          <NOLIST>
          <NOTBREAK-WORD-ON-EOL>
          <NOTTRIM-LINE-STEM><NOTTRIM-LINE-STERN>
          <NOTUND,"! ,;:?.'>
          <PAGE-DEPTH,60>
          <PARAGRAPH-INDENT,5>
          <PARAGRAPH-SPACING,0>
          <PN,1>
          <SET-SENTENCE,.!?,{upper- and lower-case  letters  and
          digits}>
          <STATISTICS>
          <TABCLEAR>
          <WARN>
          <EQ,:<STEM,:<SYSPARS>,6>,SYSLIB,
     "<STRING,#*#,"<NE,<RFN,FAIL>,FAIL,"<#*#>'>'>
```

```
<INPUT,TEXT:URELIB><#*#><INPUT>'>
```

Appendix B

The SYSLIB library

When Texture is run with 'PAR=SYSLIB' on the MTS run command, the system library of Eureka functions is read in and processed. The following is a list of the functions which are defined in this manner.

All characters available on the TN print chain at UBC are given the following (hopefully meaningful) mnemonics:

<LBRACE>	{
<RBRACE>	}
<LBRAK>	[
<RBRAK>	]
<LTOREQ>	≤
<GTOREQ>	≥
<¬EQ>	≠
<OPENBOX>	¤
<SOLIDBOX>	■
<OPENCIRCLE>	$^{\circ}$
<SOLIDCIRCLE>	•
<SUP(>	$^{(}$
<SUP)>	$^{)}$
<SUP+>	$^{+}$
<SUP+->	$^{\pm}$
<SUP->	$^{-}$
<SUP0>	^{0}
<SUP1>	^{1}
<SUP2>	^{2}
<SUP3>	^{3}
<SUP4>	^{4}
<SUP5>	^{5}
<SUP6>	^{6}
<SUP7>	^{7}
<SUP8>	^{8}
<SUP9>	^{9}
<BOXLL>	└
<BOXUL>	┌
<BOXLR>	┘
<BOXUR>	┐
<BOX->	─
<BOXSIDE>	│
<BOX+>	┼

The macro DEFINE which is used as an example in several places in this user's manual is also available.

```
<STRING,DEFINE,
       "<STRING,|NAME|,"|TEXT|'>
        <SEGMENT,|NAME|,|PARAMETERS|>'>
<SEGMENT,DEFINE,|NAME|,|PARAMETERS|,|TEXT|>
```

Two very useful external functions which are part of the SYSTEM function library (see Appendix C) are the INDEX function (for creating an index) and the CONTENTS function (for handling tables of contents). To assist the user in using these two facilities, the necessary SYSTEM commands to access the INDEX and CONTENTS functions have been accumulated as follows:

```
<STRING,LOAD-CONTENTS,"<SYSTEM,START-CONTENTS,,CTNSTART>
                      <SYSTEM,CONTENTS>
                      <SYSTEM,PRINT-CONTENTS,,CTNPRINT>'>


<STRING,LOAD-INDEX,"<SYSTEM,START-INDEX,,NDXSTART>
                   <SYSTEM,INDEX>
                   <SYSTEM,PRINT-INDEX,,NDXPRINT>'>
```

Another Eureka primitive available is INLIB which takes one parameter, an MTS file name from which it will read all Eureka functions (useful for loading in a file of the user's own macros).

```
<DEFINE,INLIB,LIBRARY,
   "<STRING,#*#,"<NE,<RFN,F>,F,"<#*#>'>'>
    <INPUT,"LIBRARY'><#*#><INPUT>
    <DELETE-STRING,#*#>'>
```

Appendix C

The SYSTEM library

The system function library contains several functions which may be desirable for various phases of document construction. Each of these functions is initially set up via the SYSTEM primitive (see page 23) using the default library and the entry point specified for each function. Setting up these functions can be done quickly by using the pre-defined Eureka functions specified in the SYSLIB library (see Appendix B). The following are the functions currently available in the system library.

Constructing a Table of Contents

A table of contents can be represented in Texture as a contents list, a sequence of quintuples of the form:

(before string,section name,between string,
page number,after string)

where 'before string' specifies some action which is to occur before printing the section name (such as line indentation), 'section name' is the name of the item being put into the table of contents, 'between string' specifies some action which is to occur between printing 'section name' and 'page number'(such as printing a row of '.'), 'page number' is the page number on which section 'section name' starts and 'after string' specifies some action which is to occur after printing the page number (such as advancing to a new line).

In these terms, a table of contents is just a sequential list of these quintuples which grows whenever a new entry is made into the table of contents. This total string can then be processed whenever the user wishes to print the table of contents.

It would quickly become tedious if every time the user were to make an entry into the table of contents he would have to supply a full quintuple to specify the entry. It would be more useful to be able to specify a number of

(before string,between string,after string)

triples initially and then indicate to the table of contents constructor the 'section name' and an indication of which triple to use. The page number would not have to be specified since this is always the current page number at the time of the call to add a new entry to the table of contents.

The above method is the one used by the table of contents constructor found in the system library. The following external functions are available to perform these various tasks:

```
<START-CONTENTS,num-pages,beforestr1,betweenstr1,afterstr1,...,
                            beforestrn,betweenstrn,afterstrn>
```

This function must be called once before it is possible to do any form of contents construction. The value of 'num-pages' must be integer and is the number of memory pages allocated to contain the table of contents. The START-CONTENTS function allows the specification of up to eight triples of the before, between and after strings as described above. The default parameter values are as follows:

'num-pages'	1
'before-stri'	<LI,<PROD,<DIFF,i>,3>>
'between-stri'	¬<SPLIT><SET-FILLER,.>
'after-stri'	<LI,0><L><SET-FILLER, >

The MTS entry point for this function is 'CTNSTART'.

```
<CONTENTS,name,triple-num>
```

The string 'name' is added to the contents list by catenating the 'triple-num'th before string, 'name', the 'triple-num'th between string, the value of <PN> and the 'triple-num'th after string onto the contents list thus far constructed. The default value of 'triple-num' is 1. The MTS entry point for this function is 'CONTENTS'.

```
<PRINT-CONTENTS,before-str,after-str>
```

The table of contents is printed by returning a catenation of 'before-str', the contents list and 'after-str' as input to the document processor. The defualt value of 'before-str' is '<LI,0>' and of 'after-str' is empty. The MTS entry point for this function is CTNPRINT.

A problem which often occurs with producing tables of contents is that the table must be constructed dynamically as the document is processed while the final location of the table of contents is usually at the beginning of the document. Thus most documents are of the form:

Title page
Table of Contents
Document body

This effect can be achieved in Texture as follows (assuming that the output document is to go into the MTS file OUTDOC).

```
              Title page source
              <SINK,OUTDOC(400)>
               <LOAD-CONTENTS>
               <START-CONTENTS>
             Document body source
              <SINK,OUTDOC(200)>
<PRINT-CONTENTS,before-string,after-string>
```

Constructing an Index

An index can be represented in Texture as an index list, a sequence of quintuples of the form:

```
(before string,index entry,between string,
                page number,after string)
```

where 'before string' specifies some action which is to occur before printing the indexed entry (such as line indentation), 'index entry' is the item being indexed, 'between string' specifies some action which is to occur between printing 'index entry' and 'page number' (such as printing a row of '.'), 'page number' is the page number on which the 'index entry' was indexed and 'after string' specifies some action which is to occur after printing the page number (such as advancing to a new line).

In these terms, an index is just a sequential list of these quintuples which grows whenever a new entry is made into the index. This total string can then be processed whenever the user wishes to print the index.

It would quickly become tedious if every time the user were to make an entry into the index he would have to supply a full quintuple to specify the entry. It would be more useful to be able to specify a number of

```
(before string,between string,after string)
```

triples initially and then indicate to the index constructor the 'index entry' and an indication of which triple to use. The page number would not have to be specified since this is always the current page at the time of the call to add a new entry to the index.

The above method is the one used by the index constructor found in the system library. The following external functions are available to perform these various tasks:

```
<START-INDEX,num-pages,beforestr1,betweenstr1,afterstr1,...,
                        beforestrn,betweenstrn,afterstrn>
```

This function must be called once before it is possible to do any form of index construction. The value of 'num-pages' must be integer and is the number of memory pages allocated to contain the index. The START-INDEX function allows the specification of up to eight triples of the before, between and after strings as described above. The default parameter values are as follows:

'num-pages'	1
'before-stri'	empty
'between-stri'	¬<LI,10><SPLIT><MIN-WS,0> <RAGLEFT><SET-FILLER,.>¬
'after-stri'	<LI,0><L><MIN-WS,1><RAGRIGHT> <SET-FILLER, >

The MTS entry point for this function is 'NDXSTART'.

```
<INDEX,name,triple-num>
```

The string 'name' is added to the index list by catenating the 'triple-num'th before string, 'name', the 'triple-num'th between string, the value of <PN> and the 'triple-num'th after string onto the index list thus far constructed. The default value of 'triple-num' is 1. The MTS entry point for this function is 'INDEX'.

```
<PRINT-INDEX,before-str,after-str>
```

The index is printed by returning a catenation of 'before-str', the index list and 'after-str' as input to the document processor. The default parameter values are:

'before-str'	<HANG,LINE,"<SET-FILLER, >'> <RAGRIGHT><LI,0><MIN-WS,1>
'after-str'	<EMPTY,LINE><JUSTIFIED>

Converting numbers to Roman Numeral and English form

The following function is supplied to convert from a string decimal representation of an integer to its equivalent capital Roman numeral or full English form.

```
<CONVERT,num,type>
```

The value of 'num' must be integer and is the decimal number which is to be converted (0 <= 'num' <= 9999). The value of 'type' is either 'R', in which case the Roman numeral equivalent of 'num' will be returned, or is 'E' in which case the full English form of the number will be returned (e.g. '84' becomes 'eighty-four'). The default value of 'type' is 'R'. The MTS entry point for this function is 'CONVERT'.

Making your own date

Occasionally it is desirable to have the date given in a form other than that given by the Texture <DATE> function. The following functions have been supplied for that purpose.

```
<DAY>
```

This function returns the numerical value of the current day, for example, '3' on the third of the month or '22' on the twenty-second of the month. The MTS entry point for this function is 'DAY'.

```
<MONTH>
```

This function returns the full alphabetic representation of the current month, for example, 'SEPTEMBER' or 'MAY'. The MTS entry point for this function is 'MONTH'.

```
<YEAR>
```

This function returns the four digit representation of the current year, for example, '1975'. The MTS entry point for this function is 'YEAR'.

INDEX

```
 MMM
 MMMM       MMM
   MM      M MM
   M      M
  M      M     MMMMMMMMM
 MM     MM     MMMM    MMM
MMM     MM       MM     MMM
MMM    MMM      MM    MMM
MMMMMMMMMM       MMMMMMM          MM
 MMMMMMM MMMM    MMM   MM        MMMMM
          MMM    MM    MMM     M   MM
                 M     MMM    M    M
                M  MM  MMM   MM
            MMMM   MMMMMM   MMM
            MMM             MMM
                            MMM
                             MMM   M
                              MMMMM
```

```
***************************
*                         *
*  Texture User's Manual  *
*         Updates         *
*                         *
***************************
```

by

The Texture Support Group

Technical Note 77-__

July 1977

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

"It is vain to multiply entities beyond need."

Sir William of Okham

In the following, annotations in braces (e.g., {23}) are page references to the Texture User's Manual, Technical Manual 75-08 (Dec 1975).

1. Running Texture

The parameters field on the Texture run command {1,49-50} need no longer explicitly ask for the system library by starting off PAR=SYSLIB... . The system library is automatically read in. If the system library is not desired, the notation PAR=NOSYSLIB... is to be used.

The parameter string may contain any Eureka programs. These are evaluated at least once (and every time the call <SYSPARS> is evaluated actively), but any resulting text is not included in the document. For example, PAR=NOSYSLIB<LIST> will preclude the system library and turn listing of source lines on until a call of <NOLIST> is encountered in the source.

2. New Primitives

The primitives described in this section are available whenever the Texture system is run.

2.1. Segment and String Manipulation

<ERASE-SEGMENTS, namelist>

All segments in the named strings are restored. Recall that a call to SEGMENT {21-23} creates "gaps" in a string. The text originally in these gaps may be restored by use of ERASE-SEGMENTS. Thus,

```
        <STRING, Alpha, gnus are gnice>
        <SEGMENT, Alpha, gn>
        "<Alpha, N>"  results in  "Nus are Nice"
        <ERASE-SEGMENTS, Alpha>
        "<Alpha, N>"  results in  "gnus are gnice"
```

<PART, string name, type, n>

The 'n'th part (of type 'type') of the string is returned. 'Type' indicates either text segment (0) or the segment gap of a given ordinal (1,2,...). Thus,

```
        <STRING, Beta, some people hate cheese>
        <SEGMENT, Beta, ee, e>
        (<PART, Beta, 0, 3>) results in (opl)
        (<PART, Beta, 1, 1>) results in (ee)
```

```
(<PART, Beta, 2, 3>) results in (e)
(<PART, Beta, 3, 1>) results in ()
(<PART, Beta, 2, 0>) results in ()
```

If any item does not exist, the null string is returned.

2.2. Function Tracing

The following primitives cause the calls of functions (macro, primitive or SYSTEMed functions) to be traced; it is intended as a debugging aid to serious Eureka programmers.

<TRACE, function name, what>

Whenever the given function is called, a trace is printed out. 'What' governs what is traced: 'what' is a string containing any of "T" (type), "A" (arguments) or "V" (value); if it is absent, it defaults to "ATV".

<NOTTRACE, function name list>

The listed functions are no longer traced.

2.3. Input

The SOURCE primitive {43} as described in the user's manual changed the input file. It was like a GOTO. If called without argument (i.e., <SOURCE>) the system input file was restored. Now SOURCE stacks files (up to 10 levels) so that a call without arguments restores the previous file. Thus,

```
<SOURCE, X>  X is active
<SOURCE, Y>  Y is active
<SOURCE>     X is active
<SOURCE>     system input is active
```

3. The SYSTEM Library

This section updates Appendix C of the manual {51-55}. Some functions have been added and several have been improved. Most of the functions in the SYSTEM library need no longer be loaded explicitly (see section 4 for further information).

3.1. New Functions

<EMPTEE, filename>

Accepts an MTS file name and empties the file. This function is not included in any OS releases of Texture.

<COMMAND, MTS command>

Accepts an MTS command and causes it to be executed. Note that any command like $RUN, $LOAD or $UNLOAD will cause Texture to disappear. This function will not be included in any OS releases of Texture.

<SUP, string>

Translates any of the numerical characters or '()+-' in the string to corresponding superscript characters on the TN-chain. E.g., "<SUP, (-125)>" produces "$^{(-125)}$"

<HYPHENATE, word>

Checks the space left on the current line; if there is not enough space for 'word' then it returns 'word' broken, if possible, into two strings, separated by a blank. A break is possible if there is a hyphen (-) or a discretionary hyphen (|) in 'word' in the portion of 'word' that will fit on the current line. The leftmost hyphen or discretionary hyphen that will fit on the line is chosen, and a blank inserted just after it. This discretionary hyphen is turned into a hyphen. After the above is completed, all remaining discretionary hyphens are removed whether or not hyphenation has taken place.

Thus, "<HYPHENATE,hy|phen|ate>" might result in "hyphen- ate" if there were 7 or 8 but not 9 spaces left; if there were 9, it would result in "hyphenate".

<.HYPHENATE, word>

This is the same as HYPHENATE, but it is used whenever 'word' is the first word in a sentence (that is, it takes into account the difference between sentence spacing and word spacing).

<#, expression>

The expression is an arithmetic expression involving any of the following:

integer numbers

+	(Addition)
-	(Subtraction or Negation)
*	(Multiplication)
/	(Integer Division)
%	(Remainder after Integer Division)
( )	(Parentheses to affect order of evaluation)

All evaluation is from left to right, except that expressions inside parentheses are treated as a single unit, and multiplication operators (*, /, %) take precedence over addition operators (+, -); negation takes highest precedence. Thus, A+B*C is equivalent to A+(B*C).

<BOLDFACE, string>

BOLDFACE simulates boldface font by causing the argument 'string' to be overprinted three times. The quality of the font is thus dependent on the accuracy of the printer and the condition of the ribbon when the document is printed. For example,

```
        <BOLDFACE,hello>
```

will print as 'hello'.

The maximum length of the argument 'string' is 100 characters. An argument exceeding this maximum will be truncated without warning.

Note: The handling of the special Texture operators (e.g., @ and _) by BOLDFACE deviates slightly from the norm. The operators *, ¢ and @ are assumed to refer to the succeeding character, regardless of whether or not it is another operator or special character (e.g., < and >). Also, the blank and the operators ¬, / and _ are returned unchanged. For example,

```
        <BOLDFACE,@HELLO>
```

will print as 'Hello',

```
        <BOLDFACE,@O/@I>
```

will print as 'O'

```
        <BOLDFACE,**HI>
```

will print as '*hi'.

Warning: In the past, excessive use of BOLDFACE on a given line or a given page caused the workspace to overflow (every boldfaced character takes up 7 times the space of an ordinary character), and Texture to crash mysteriously; this bug appears to have been fixed for the standard layout, but boldfacing long lines or large pages will almost certainly cause it to recur.

3.2. Contents and Index Functions

These have been souped up so that multiple copies may be loaded (for multiple, Lord of the Rings style indexes and multiple, thesis style tables of contents, figures and tables). There is now only one function to load to get a table of contents (or index) and it takes an additional argument, inserted just after the function name, which tells it what to do. Thus,

```
        <SYSTEM, YOUR-NAME,, CONTENTS> loads contents function
        <YOUR-NAME, START, options-as-before> starts it
            .  .  .
        <YOUR-NAME, ADD, entry> adds an entry
            .  .  .
        <STRING, PRINT-CONTENTS, "<YOUR-NAME>'>
        <YOUR-NAME, PRINT, strings-as-before> prints it
```

The definition of PRINT-CONTENTS is necessary (unless YOUR-NAME = PRINT-CONTENTS) because during printing, YOUR-NAME will call <PRINT-CONTENTS> but really wants to call itself

(except that it doesn't know you called it YOUR-NAME). Note that the default for the first argument is PRINT.

If all this seems a little hard to follow, see section 4.2 for a simplified method.

4. The SYSLIB Library

This section updates Appendix B of the user's manual {49-50}.

4.1. Automatic Loading

All functions in the SYSTEM library (except the contents and index functions) are pre-defined as macros which cause the function to be loaded on first reference. Thus it is no longer necessary to say,

```
        <SYSTEM, BOLDFACE>
            .  .  .
        <BOLDFACE, foobar>
```

The second call by itself will cause the loading of BOLDFACE (which will then be called with the argument 'foobar').

4.2. Contents and Index

These must still be loaded explicitly. The macros:

```
        <LOAD-CONTENTS>
        <LOAD-FIGURES>
        <LOAD-TABLES>
        <LOAD-INDEX>
        <LOAD-INDEX1>
```

will still be pre-defined to make it more convenient. Note that Figures and Tables are just like a Table of Contents, except that the three macros used for them have FIGURES or TABLES where the Contents macros have CONTENTS. Similarly, INDEX1 replacing INDEX uniformly in calls gives a second index. The user wishing to create other tables or other indexes might look at these macros to see how it is done.

Example:

```
        <LOAD-TABLES>    loads a table of tables function
        <START-TABLES,2>    sets up a workspace
            .  .  .
        <TABLES,Fly Populations in South America>   an entry
            .  .  .
        <PRINT-TABLES>
```

5. Other Macro Libraries

Most things that a Texture user may need to do can be done directly in Eureka. Texture users are also constantly re-inventing the wheel because they all use the same basic set of Eureka programs. For this reason, the Texture Support Group encourages the development of macro libraries and their documentation for general use.

Note: Since libraries are generally laid out prettily on the page, and since Eureka treats blanks as just more text, it is best to use the INLIB macro (defined in SYSLIB) to load a library. I.e., <INLIB, library name>. INLIB will delete all blanks.

5.1. General Eureka Library

A major macro library (developed by M.S. Johnson, initially for his own use) is available in the MTS file

CS:EUREKALIB

It contains many useful Eureka macros for manipulating strings, creating and manipulating counters, putting line numbers in the margins of documents for editing, putting version bars in the margins of documents for updates of documentation, doing structured Eureka programming, etc.

Documentation for this library is available from the Computer Science Documentation archive ($RUN CS:DOC); an up-to-date version will also be kept in the file

TEXT:EUREKALIB.W

ready for copying to the TN-printer.

5.2. Diagram Library

A number of functions for drawing diagrams in Texture documents were developed by T. Venema and upgraded by M.S. Johnson. These functions draw boxes and arrows with a minimum of effort. The library is available in the file

CS:DRAWLIB

Documentation for this library is available from the Computer Science Documentation archive ($RUN CS:DOC); an up-to-date version will also be kept in the file

TEXT:DRAWLIB.W

ready for copying to the TN-printer.

5.3. Bracket Counter

Anyone who has written a large Eureka program and subsequently tried to decide whether the brackets (< and >) and evaluation delays (" and ') were well-balanced, will have been driven to the edge of a nervous breakdown. There is now a Eureka program which will do the work for you. It is in the file

TEXT:BRAX

and may be called as follows:

<COUNT, Texture source file>

The file will be printed out line by line and the bracket and delay-quote nesting printed out with each line. The output appears in the Texture listing life (not in the Texture document file).

5.4. Including Files in the Input

It is frequently convenient to have a file reference another file, which is to be included in the source of the document. This way, one master file can serially include several files making up parts of a document; also, files may include a common subsection (copyright notice, etc.)

A function INCLUDE is defined in the file

TEXT:INCLUDE

and may be called as follows:

<INCLUDE, file name>

Such a call has the effect of the text of the file being inserted into your document source at the point of the call.

```
 MMM
 MMMM       MMM
   MM      M MM
   M      M
  M      M     MMMMMMMMM
 MM     MM    MMMM    MMM
MMM     MM      MM    MMM
MMM    MMM      MM   MMM
MMMMMMMMMM       MMMMMMM          MM
 MMMMMMM MMMM    MMM   MM       MMMMM
          MMM    MM    MMM     M   MM
                 M     MMM    M    M
                M  MM  MMM   MM
            MMMM   MMMMMM   MMM
            MMM             MMM
                            MMM
                             MMM   M
                              MMMMM
```

```
**********************************
*                                *
*           EUREKALIB:           *
*  A Library of Eureka Functions *
*                                *
**********************************
```

by

Mark Scott Johnson

Technical Note 75-6

1975 November
Revised 1976 September
and 1977 July

Department of Computer Science
The University of British Columbia
2075 Wesbrook Mall
Vancouver, British Columbia V6T 1W5

0. Preface to the second revision

The file CS:EUREKALIB now contains a new version of the Eureka library. The old library has been retained as CSLB:EUREKALIB, but it will be maintained only for a limited time under that id. Please use the new version, or make your own copy of the old library. The differences between the two versions are summarized in the next paragraph, and new or substantially revised portions of this document are indicated by vertical bars in the right-hand margin.

BOLDFACE has been deleted from EUREKALIB since it is now available in the default Texture system library. It is no longer necessary to load it via a call of the function SYSTEM; it can be treated much like a primitive. Documentation for BOLDFACE can now be found in the "Texture User's Manual Updates".

The following functions now take optional last arguments: FIGURE, PHYSICAL-L, and REVISION-BARS. Current uses of these functions need not be changed. The function LINE-NUMBERING has been changed internally to gain efficiency. A minor change in the definition of LP has occurred, but it is upward compatible. The following are new functions now contained in EUREKALIB: CONCAT, DELETE-COUNTER, INI, MAKE-COUNTER, OUTI, RESET-COUNTER, and SET-COUNTER. A new string ALPHABET has also been defined.

1. Introduction

This document describes a library of functions (written in Eureka) which Texture users may find handy. Before using this library, it must be loaded via the call <INLIB,CS:EUREKALIB>.

The user is cautioned to read this document carefully before using any of the functions described. Failure to do so may result in disaster. The implementor has tried to insure that these functions are useful, useable, general-purpose, and fully debugged. Nevertheless, the implementor absolves himself of all responsibility for problems which may arise out of the use of EUREKALIB and, further, no obligation to maintain the library is assumed.

Due to the nature of the Eureka string processor, the user is obliged to know something of the internal structure of the functions in EUREKALIB. In particular, it is important that the user not redefine any of the functions or strings described below since the functions in the library are highly interdependent. In addition, several strings are defined internally in the library and must not be used as the names of functions or strings defined by the user; the names of all such strings both begin and end with a pair of octothorpes (##).

Two other precautions are in order. First, all of the functions in EUREKALIB assume that the "standard" function evaluation environment is in effect. For this reason, the start-of-function (<), end-of-function (>), argument-separator (,), start-literal ("), end-literal ('), and neutral-indicator (:) characters must not be changed. Likewise, no primitive function should be redefined. Second, few of the functions in EUREKALIB check their arguments for validity. Thus, an erroneous call may result in an error message being initiated by one of the Eureka or Texture system functions.

Some users may be concerned with the growing size of EUREKALIB and the resulting increase in initialization overhead each time it is INLIBed. Such users are free to make their own copies of CS:EUREKALIB and to pare it down by discarding unneeded functions. If doing so, however, be certain to retain all functions and strings which are used in the definitions of the top-level functions whose retention is desired.

2. Functions defined in EUREKALIB

<COMMENT,string>

This function simply causes its argument(s) to be ignored by the document processor. It can be used to insert comments into Texture source programs without affecting the output document.

<COMPRESS,string>

COMPRESS returns its argument with all leading and trailing blanks trimmed off and with all multiple embedded blanks reduced to a single blank. For example,

```
    <COMPRESS,  THIS  IS A STRING >
```

returns the string 'THIS IS A STRING'.

<CONCAT,name,value>

This function appends the string 'value' to the current value of the string called 'name'. For example, after execution of

```
    <STRING,FOO,HELLO>
    <CONCAT,FOO, WORLD>
```

the value of <FOO> is 'HELLO WORLD'.

<COUNT,string,pattern>

This function returns a count of the number of occurrences of 'pattern' in 'string'. 'pattern' cannot be the null string. For example,

```
    <COUNT,ABABABA,ABA>
```

returns the value 3.

<DECR,name>

DECR causes the value of the string called 'name' to be decremented by one. Prior to a call of DECR, 'name' must be defined via the STRING function and must have a numeric value. For example, after execution of

```
    <STRING,COUNTDOWN,100>
    <DECR,COUNTDOWN>
```

the value of <COUNTDOWN> is 99.

<DELETE-COUNTER,name>

This function causes the counter 'name' initiated via a call of MAKE-COUNTER to be discarded and its storage freed. See the description of MAKE-COUNTER below.

<END-VERSION,n>

This function delimits the scope established by a preceding call of <VERSION,n> to specify the bounds of a document revision. See the description of REVISION-BARS below.

<EXPLODE,string1,string2>

This function returns 'string1' with 'string2' appended before each character of 'string1'. For example,

```
    <EXPLODE,BLOWUP,..>
```

returns the string '..B..L..O..W..U..P'.

Warning: Because of the way in which this function is implemented, neither of the argument strings should contain the start-of-function (<) or the neutral-indicator (:) character.

<EXTRACT-NUMBER,string,pattern,default>

EXTRACT-NUMBER returns the number following an equal sign (=) following the first occurrence of 'pattern' in 'string'. For example,

```
    <EXTRACT-NUMBER,NUM=12 OPT=3,OPT,400>
```

returns the value 3. If 'OPT=' had not been contained in the argument 'string', then the optional third argument (400 in this example) would have been returned.

The number extracted is delimited by the equal sign on the left, and by either a blank or the end of 'string' on the right.

Altho not designed to, EXTRACT-NUMBER will actually extract any string following 'pattern' which is properly delimited, whether or not it is numeric. For example,

```
     <EXTRACT-NUMBER,P=4PAGES T=5S,P,10PAGES>
```

returns the string '4PAGES'.

<FIGURE,string,length,flag,block>

The FIGURE function can be used to cause a string to be treated as a figure (i.e., the entire figure must appear on one text page). The first argument is the string representing the figure, and 'length' is the number of physical lines which 'string' will consume when printed by the document processor. The user must supply this length since Eureka functions are unable to predict how many lines a given Texture string will consume.

If the figure will fit on the current text page, FIGURE merely passes 'string' on to the document processor. However, if the figure will not fit on the current page, FIGURE returns the null string and causes the figure to appear at the top of the next text page using the page event.

The optional third argument is used to determine whether or not 'string' is being held over onto a new page. If the figure fits on the current page, the value of the string named 'flag' is set to the string 'T'; otherwise it is set to 'F'.

The optional fourth argument specifies the block of text in which the figure is to appear if it will not fit in the current block. The default block is STANDARD-TEXT. In general, this argument will only need to be specified when user-defined blocks and layouts are being employed. See the "Texture User's Manual" for an explanation of blocks and layouts.

Note: Altho it is possible for more than one figure to be saved over onto another page, the expansion of all such figures must not exceed a total of one page of output text. In other words, no figure can be held over to a second page. This is an implementation restriction which appears difficult to overcome.

<FOR,name,from,to,by,what>

This function acts like an Algol FOR loop. 'name' (which can be any string) is given the initial integer value 'from'. Next, the string 'what' is evaluated with all occurrences of 'name' being replaced by the value 'from'. Finally, the value of 'name' is incremented by 'by' and 'what' is evaluated again, but with this new value substituted. Execution terminates when the value of 'name' is one greater than the value of the integer 'to' (i.e., 'what' is executed max(0,(to-from)-by+1) times). If 'from', 'to', or 'by' is the null string, its value defaults to 1. For example,

```
     <STRING,TOTAL,0>
     <FOR,|I|,1,5,,"<STRING,TOTAL,<SUM,<TOTAL>,|I|>>'>
     <TOTAL>
```

returns the value 15, and

```
     <FOR,|X|,,5,2,"<STEM,ABCDE,|X|>'>
```

returns the string 'AABCABCDE'.

The user should note that the argument 'name' is handled via the SEGMENT system function. This means that a textual substitution of 'name' is made in 'what'.

<FULL-DATE>

The value of this function is the current date in the form 'yy mm dd', where the month is written out in full. For example, if <DATE> returns 'oct 12, 1492', then <FULL-DATE> returns '1492 October 12'.

<HI,n>

This function causes a left indentation to column 'n' (i.e., <LI,n>) to occur *after* completion of the current line of text. This is known as a hanging indentation.

HI is implemented by hanging '<LI,n>' onto the new line event. Unfortunately, if line termination is the result of one of the system functions (e.g., <L>, <P>, and <PAGE>), the line event does *not* occur. (This is an implementation restriction.) Therefore, under some circumstances the user may find the indentation occurring later than desired. This can be compensated for by calling <HANG,LINE> to flush the line event, or by calling <EMPTY,LINE> to discard the line event, before calling the system function which bypasses the normal line event mechanism.

<INCR,name>

INCR causes the value of the string called 'name' to be incremented by one. Prior to a call of INCR, 'name' must be defined via the STRING function and must have a numeric value. For example, after execution of

```
     <STRING,COUNTER,0>
     <INCR,COUNTER>
```

the value of <COUNTER> is 1.

<INI,n>

This function causes the left indentation to increase from its current value by 'n'. Since it acts as a relative LI, the indentation takes effect only at the beginning of the next line of output text. The left indentation can again be decreased via the function OUTI or reset using LI.

<LINE-NUMBERING>

This function causes subsequent document processor lines printed to contain, in the right-hand margin, the number of the approximate input source line corresponding to each output line. This feature should facilitate subsequent editing of the Texture source file by the user.

Line numbering will appear to the right of the right-most perforation on 8X11 forms for positive line numbers less than 10,000. Thus, the numbering can normally be discarded when the edges of the continuous forms are removed.

Line numbering can be discontinued by calling the function NOTLINE-NUMBERING. Since use of LINE-NUMBERING incurs about a ten percent increase in document processing time, the user is encouraged to employ this facility selectively for large documents. All line numbering can be suppressed by including the string 'NONUMBER' in the PAR= field of the Texture run command.

<LOWER,string>

LOWER returns 'string' with the lower-case operator (¢) appended before each character. Thus, when output via the document processor, 'string' will be shifted to lower-case. For example,

    <LOWER,HELLO>

will print as 'hello'.

Since LOWER is implemented simply as <EXPLODE,string,¢>, 'string' should not contain the characters < or :.

<LP>

This function causes a new paragraph to be started with a blank line separating the new paragraph from the old and forces the first word of the new paragraph to be capitalized. It is defined as '<L><WIDOW,3><P>@'.

<MAKE-COUNTER,name,type,case>

MAKE-COUNTER causes a counter called 'name' to be created of type 'type' and case 'case'. The acceptable values of 'type' are 'ALPHA' (for alphabetic), 'ROMAN' (for roman numerals), and 'ARABIC' (for arabic numerals). The default 'type' is 'ARABIC'. The acceptable values of 'case' are 'UPPER' (for upper-case), 'LOWER' (for lower-case), and 'SUPER' (for superscript). The default is to return the counter without any particular case and to let the current Texture environment determine the output case.

MAKE-COUNTER works by defining two string: 'name' and 'INCR-name'. Executing <name> returns the current value of the counter in the proper type and case. Executing <INCR-name> first increments the value of the counter by one and then returns this new value in the proper type and case. For example, after execution of

    <MAKE-COUNTER,ALPHA-CTR,ALPHA,UPPER>

executing <INCR-ALPHA-CTR> once returns the string '@A' and executing it a second time returns '@B'. It is important to note that the two calls <name> and <INCR-name> return translations of the real numeric value of the counter 'name'; the real value is hidden from view. Thus, it is possible for the user to think of ALPHA-CTR as a counter which increments alphabetically rather than numerically, even tho this is an oversimplification.

Not all combinations of 'type' and 'case' are allowed: types 'ALPHA' and 'ROMAN' cannot have case 'SUPER', and type 'ARABIC' cannot have cases 'UPPER' or 'LOWER'. All other combinations have the expected attributes. Specifying a case for 'ALPHA' will force that counter to always print in the desired case, regardless of the current global case environment (e.g., UP and DOWN). Otherwise, the output case of an alphabetic counter will depend on the global case environment and whether or not it begins a sentence.

Warning: Since the counters initiated by MAKE-COUNTER have a special internal representation, it is not possible to treat them as ordinary strings. In particular, a counter must be deleted via a call of DELETE-COUNTER (rather than the system function DELETE), its value can only be reset to zero via RESET-COUNTER, and its value can only be set to some arbitrary numeric value via SET-COUNTER (rather than via STRING).

One of the nicest featues of MAKE-COUNTER is that it allows counters to be completely transparent. For example, after execution of

    <DEFINE,HEADING,|STRING|,
       "<L,2><WIDOW,5><INCR-HEAD-CTR>. |STRING|'>
    <MAKE-COUNTER,HEAD-CTR>

calls can be made to HEADING without the need of passing a heading number; headings will automatically be numbered

1,2,3,.... Similarly, it is possible to define a footnote function such that superscripted reference numbers are completely hidden.

<MAX,x0,...,x9>

MAX accepts from one to ten integral arguments and returns the numerically largest of them.

<MIN,x0,...,x9>

MIN accepts from one to ten integral arguments and returns the numerically smallest of them.

<NOTLINE-NUMBERING>

This function causes the line numbering initiated via a call of LINE-NUMBERING to be discontinued.

<OUTI,n>

This functions causes the left indentation to decrease from its current value by 'n'. Since it acts as a relative LI, the indentation takes effect only at the beginning of the next line of output text. The left indentation can again be increased via the function INI or reset using LI.

<PAD-LEFT,string,char,pad-size>

PAD-LEFT returns 'string' padded on the left with sufficient copies of 'char' to make the result of length at least max(pad-size,<LENGTH,string>). For example,
      <PAD-LEFT,HELLO,.,10>
returns the string '.....HELLO' while
      <PAD-LEFT,WORLD,X,4>
simply returns the string 'WORLD' unchanged. Also,
      <PAD-LEFT,WORLD,+.,10>
returns the string '+.+.+.WORLD', which is of length 11.

<PAD-RIGHT,string,char,pad-size>

PAD-RIGHT is equivalent to PAD-LEFT, except that padding occurs on the right of 'string'.

<PHYSICAL-L,n,justification-method>

This function advances to a new line 'n' times, where 'n' must be integral. It differs from the system function L in that L advances lines with respect to the current LINESPACING value, while PHYSICAL-L ignores it and advances 'n' physical (as opposed to logical) new lines. 'justification-method' is the optional second argument of L; see the "Texture User's Manual" for an explanation of its utility.

<REAL-DATE>

The value of this function is the current date in the form 'yy mmm dd'. For example, if <DATE> returns 'oct 12, 1492', then <REAL-DATE> returns '1492 Oct 12'.

<REMOVE,string,pattern>

REMOVE returns 'string' with all occurrences of 'pattern' removed. For example,
      <REMOVE, B L A H, >
returns the string 'BLAH'.

<REMOVE-FROM-EVENT,event,text>

This function deletes all occurrences of 'text' from event 'event' leaving unchanged any other text associated with the event. 'event' can be any one of the events defined for the system function HANG.

<REPEAT,what,howoften>

This function concatenates 'howoften' (an integer number) evaluations of the string 'what'. If 'howoften' is omitted, its value defaults to 1. For example,
      <REPEAT,"<STEM,ABCD,2>',4>
returns the string 'ABABABAB'.

<REPLACE,string,pattern,replacement>

REPLACE returns 'string' with all occurrences of 'pattern' replaced by 'replacement'. For example,
      <REPLACE,--..-.-..,.,+>
returns the string '--++-+-++'.

<RESET-COUNTER,name>

This function causes the value of the counter called 'name' initiated by a call of MAKE-COUNTER to be reset to zero. See the description of MAKE-COUNTER above.

<REVERSE,string>

This function returns 'string' with its characters reversed. For example,

```
     <REVERSE,UVWXYZ>
```

returns the string 'ZYXWVU'.

<REVISION-BARS,position,version,block>

REVISION-BARS initializes a set of functions used to indicate versions of a document by selectively printing vertical bars (|) in the margins of the output document. This is exemplified by the bars in the right-hand margin of this document.

To use this facility, the user must supply two major pieces of information. First, revisions of the Texture source document must be delimited by calls of the pair of functions VERSION and END-VERSION. All text between matching calls of <VERSION,n> and <END-VERSION,n> are assigned the version number 'n'. It is the user's responsibility to insure that the VERSION -- END-VERSION pairs are correctly nested and balanced. Second, the user must specify which version number groups are to be indicated by vertical bars. This can be accomplished in two ways. If the PAR= field on the Texture run command contains the string 'VERSION=n' (where 'n' is integral), then all version groups whose number is greater than or equal to 'n' will be barred. Alternatively, the 'version' argument of REVISION-BARS will be used if the PAR= field is not. If 'version' is also missing, no version groups will be barred.

REVISION-BARS must be called once before any calls of VERSION or END-VERSION. The argument 'position' indicates which margin of the output document is to contain the vertical bars. 'position' can be either the string 'RIGHT' or the string 'LEFT', or the null string. The default is to place vertical bars in the right-hand margin. REVISION-BARS can be called more than once to cause the value of 'position' to be changed. For example,

```
     <TITLE,"<REVISION-BARS,<#EQ,<MOD,<PN>,2>,0,LEFT,RIGHT>,2>'
        :<TITLE>>
```

at the beginning of document processing causes vertical bars to appear in the right-hand margin of odd numbered pages and in the left-hand margin of even numbered pages. In addition, the bars will only appear for version groups numbered greater than 1.

The values of 'version' and 'block' are only significant during the first call of REVISION-BARS. If supplied on subsequent calls, they will be ignored.

The optional third argument specifies the block of text in which the revision bars are to appear. The default block is STANDARD-TEXT. In general, this argument will only need to be specified when user-defined blocks and layouts are being employed. See the "Texture User's Manual" for an explanation of blocks and layouts.

<SEARCH,string,pattern>

SEARCH returns the position of the start of the first occurrence of 'pattern' in 'string'. It returns zero if 'pattern' is the null string, if 'pattern' is not contained within 'string', or if the length of 'pattern' is greater than that of 'string'. For example,

```
     <SEARCH,UVWXY,WX>
```

returns the value 3 and

```
     <SEARCH,ABC,CD>
```

returns the value 0.

<SET-COUNTER,name,n>

This function causes the value of the counter called 'name' initiated by a call of MAKE-COUNTER to be set to the numeric value 'n'. See the description of MAKE-COUNTER above.

<SUBSTR,string,index,length>

This function returns the substring of 'string' of length 'length' beginning at position 'index'. 'index' must be greater than zero and less than or equal to <LENGTH,string>. Also, 'length' and index+length-1 must be less than or equal to <LENGTH,string>. If 'index' is omitted, its value defaults to 1. If 'length' is omitted, the entire substring beginning at position 'index' of 'string' is returned. For example,

```
     <SUBSTR,PAPER,2,3>
```

returns the string 'APE' and

```
     <SUBSTR,PAPER,3>
```

returns the string 'PER'.

<UNDER,string>

UNDER returns 'string' with the underline operator (_) appended before each character. Thus, when output via the document processor, 'string' will be underlined. For example,

```
     <UNDER,hello>
```

will print as 'hello'.

Since UNDER is implemented simply as <EXPLODE,string,_>, 'string' should not contain the characters < or :.

<UPPER,string>

UPPER returns 'string' with the upper-case operator (@) appended before each character. Thus, when output via the document processor, 'string' will be shifted to upper-case. For example,

```
    <UPPER,hello>
```

will print as 'HELLO'.

Since UPPER is implemented simply as <EXPLODE,string,@>, 'string' should not contain the characters < or :.

<VERSION,n>

This function establishes the beginning of a document revision group which must be terminated subsequently by a call of <END-VERSION,n>. See the description of REVISION-BARS above.

<WHILE,condition,what>

This function causes the string 'what' to be evaluated repeatedly until 'condition' becomes false. 'condition' must be in the form of one of the built-in predicate functions (EQ, NE, LT, LE, GT, GE, #EQ, #NE, #LT, #LE, #GT, or #GE) but without the last two arguments being present (e.g., <EQ,a,b> rather than <EQ,a,b,true,false>). For example,

```
    <STRING,LETTER,ABCX>
    <WHILE,"<NE,<LETTER>,X>',
       "<LETTER>
        <STRING,LETTER,<TRIMSTEM,<LETTER>,1>>'>
```

returns the string 'ABCX BCX CX'.

3. Strings defined in EUREKALIB

<ALPHABET>

The value of this string is the twenty-six letters of the English alphabet in upper-case.

<MAX-NUMBER>

The value of this string is 2147483647, the maximum integer representable by the hardware of the IBM 360/370s.

<S>

The value of S is a single blank. It can be used in functions read via the INLIB function. Regular blanks are removed by INLIB.

4. Additional actions initiated when using EUREKALIB

Before defining the functions and strings listed above, EUREKALIB causes the DEFINITION-SPACE-SIZE to be increased to accommodate the new functions. The only additional action initiated is the call <BREAK-WORD-ON-EOL> to cause the end of each subsequent line read by Eureka to be treated as a break between text words. This is essentially equivalent to appending a blank to the end of each subsequent line of input text.

5. Summary of functions by categories

case shifting: LOWER, UNDER, UPPER
control structures: FOR, REPEAT, WHILE
counters: DECR, DELETE-COUNTER, INCR, MAKE-COUNTER, RESET-COUNTER, SET-COUNTER
dates: FULL-DATE, REAL-DATE
figures: FIGURE
indentations: HI, INI, OUTI
line numbering: LINE-NUMBERING, NOTLINE-NUMBERING
line spacing: LP, PHYSICAL-L
miscellaneous: COMMENT, REMOVE-FROM-EVENT
numeric computation: MAX, MIN
revision bars: END-VERSION, REVISION-BARS, VERSION
string manipulation: COMPRESS, CONCAT, COUNT, EXPLODE, EXTRACT-NUMBER, PAD-LEFT, PAD-RIGHT, REMOVE, REPEAT, REPLACE, REVERSE, SEARCH, SUBSTR

```
 MMM
 MMMM       MMM
   MM      M MM
   M      M
  M      M     MMMMMMMMM
 MM     MM    MMMM    MMM
MMM     MM      MM    MMM
MMM    MMM      MM   MMM
MMMMMMMMMM       MMMMMMM         MM
 MMMMMMM MMMM    MMM   MM       MMMMM
          MMM    MM    MMM     M   MM
                 M     MMM    M    M
                M  MM  MMM   MM
            MMMM   MMMMMM   MMM
            MMM             MMM
                            MMM
                             MMM   M
                              MMMMM
```

```
**********************************************
*                                            *
*                  DRAWLIB:                  *
*      A Library of Texture Functions        *
*  for Drawing Figures Directly onto a Page  *
*                                            *
**********************************************
```

by

Mark Scott Johnson

Technical Note 77-1

1977 May 6

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

This document describes a library of functions (written in Eureka) for drawing line figures directly onto a page of text. Before using this library, it must be loaded via the call <INLIB,CS:DRAWLIB>. Also, since drawing involves Texture blocks, it will generally be necessary to up the maximum block size (which defaults to 20) via the call <MAX-BLOCKS,100>. Intricate diagrams may require an even larger block specification.

The functions described below are implemented using the Texture primitives H-LINE, V-LINE, POINT, and LINE (see sections 4.4 and 4.5 of the "Texture User's Manual") to cause lines and strings to be drawn at absolute locations within the current page. Thus, drawing is done without regard to text blocks and layouts. In other words, it is possible to draw figures on the page _on top of normal text_. It is the user's responsibility to insure (using the L and PAGE primitives) that this does not happen. Examples will be given below.

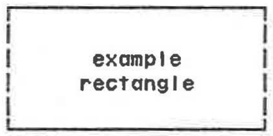
The drawing functions are divided into two classes: those involving rectangular shapes (RECTANGLE, SQUARE, and OPTION-BOX) and those involving lines with arrowed tips (DOWN-ARROW, LEFT-ARROW, RIGHT-ARROW, UP-ARROW, and ZIG-ZAG).

<RECTANGLE,col#,line#,width,height,label1,label2>

'col#' and 'line#' indicate the absolute column and line numbers of the top leftmost corner of the rectangle to be drawn. The rectangle will be 'width' characters wide and 'height' characters high. 'label1' and 'label2', which are optional, are strings which will be centered within the rectangle as labels. For example, execution of the sequence of calls:

```
     <RECTANGLE,10,<SUM,<LINE>,2>,21,6,example,rectangle>
     <L,9>
```

produces the figure:



Note in particular that the <L,9> is necessary to prevent text from being assembled on top of the rectangle.

<SQUARE,col#,line#,width,label1,label2>

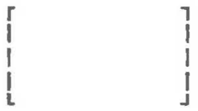
SQUARE is identical to RECTANGLE except that the height of the figure is the same as its width and, thus, only 'width' needs to be specified.

<OPTION-BOX,col#,line#,width,height>

OPTION-BOX produces an unlabeled rectangle in which the top and bottom lines are not drawn. Thus, the resulting figure approximates a set of large square brackets. For example,

```
     <OPTION-BOX,15,<SUM,<LINE>,2>,15,5>
     <L,9>
```

produces the figure:

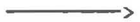


<direction-ARROW,col#,line#,length>

'direction' is one of the words: DOWN, LEFT, RIGHT, or UP. Each of these four functions operates in the same manner. 'col#' and 'line#' indicate the absolute column and line numbers of the origin of the line to be drawn. A line is drawn from the origin in the 'direction' and for the 'length' specified. An arrow head (V, <, >, A) is drawn at the end of the line. For example,

```
     <RIGHT-ARROW,20,<SUM,<LINE>,2>,10>
     <L,5>
```

produces the figure:

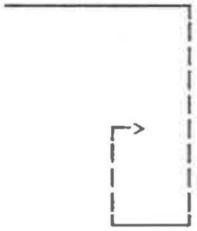


<ZIG-ZAG,col#,line#,direction^{1},length^{1},...,direction^{9},length^{9}>

ZIG-ZAG causes a zig-zagged line of up to nine components to be drawn starting at the origin (col#,line#). 'direction' is one of the words: DOWN, LEFT, RIGHT, or UP, and 'length' is the length of one of the component lines. An arrow head is drawn at the end of the last component. For example,

```
     <ZIG-ZAG,5,<SUM,<LINE>,2>,RIGHT,15,DOWN,10,
        LEFT,7,UP,5,RIGHT,3>
     <L,14>
```

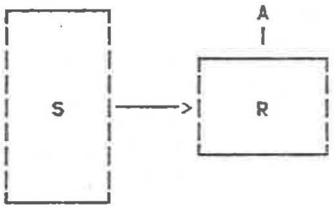
produces the figure:



When it is desired to draw a figure composed of several subfigures, it is suggested that the origin for each subfigure be specified using two user-defined numeric strings, and that the values of these strings be passed as the 'col#' and 'line#' arguments to the DRAWLIB functions rather than passing absolute integers. This facilitates modification of the subfigures by translation relative to each other and in positioning the entire figure on the page. For example,

```
     <STRING,X0,30>
     <STRING,Y0,<SUM,<LINE>,2>>
     <STRING,X1,45>
     <STRING,Y1,<SUM,<Y0>,2>>
     <SQUARE,<X0>,<Y0>,9,S>
     <RIGHT-ARROW,<SUM,<X0>,8>,<SUM,<Y0>,4>,
        <DIFF,<X1>,<SUM,<X0>,8>>>
     <RECTANGLE,<X1>,<Y1>,11,5,R>
     <UP-ARROW,<SUM,<X1>,5>,<Y1>,3>
     <L,12>
```

produces the figure:



where (X0,Y0) is the origin of the square S and (X1,Y1) is the origin of the rectangle R. Now the entire figure can be centered by changing X0 to 24 and X1 to 39 producing: