The method provides an efficient technique for modifying the visual display characteristics of a body of text as a function of the context of the text. The method involves determining a location within a body of text, and identifying a parse node associated with that location in the text. Using the parse node identifier, determine the display characteristic, and then adjust the display of the text accordingly.
FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Austria</td>
<td>GB</td>
<td>United Kingdom</td>
<td>MR</td>
<td>Mauritania</td>
</tr>
<tr>
<td>AU</td>
<td>Australia</td>
<td>GE</td>
<td>Georgia</td>
<td>MW</td>
<td>Malawi</td>
</tr>
<tr>
<td>BB</td>
<td>Barbados</td>
<td>GN</td>
<td>Guinea</td>
<td>NE</td>
<td>Niger</td>
</tr>
<tr>
<td>BE</td>
<td>Belgium</td>
<td>GR</td>
<td>Greece</td>
<td>NL</td>
<td>Netherland</td>
</tr>
<tr>
<td>BF</td>
<td>Burkina Faso</td>
<td>HU</td>
<td>Hungary</td>
<td>NO</td>
<td>Norway</td>
</tr>
<tr>
<td>BG</td>
<td>Bulgaria</td>
<td>IE</td>
<td>Ireland</td>
<td>NZ</td>
<td>New Zealand</td>
</tr>
<tr>
<td>BJ</td>
<td>Benin</td>
<td>IT</td>
<td>Italy</td>
<td>PL</td>
<td>Poland</td>
</tr>
<tr>
<td>BR</td>
<td>Brazil</td>
<td>JP</td>
<td>Japan</td>
<td>PT</td>
<td>Portugal</td>
</tr>
<tr>
<td>BY</td>
<td>Belarus</td>
<td>KE</td>
<td>Kenya</td>
<td>RO</td>
<td>Romania</td>
</tr>
<tr>
<td>CA</td>
<td>Canada</td>
<td>KG</td>
<td>Kyrgyzstan</td>
<td>RU</td>
<td>Russian Federation</td>
</tr>
<tr>
<td>CF</td>
<td>Central African Republic</td>
<td>KP</td>
<td>Democratic People's Republic of Korea</td>
<td>SD</td>
<td>Sudan</td>
</tr>
<tr>
<td>CG</td>
<td>Congo</td>
<td></td>
<td></td>
<td>SE</td>
<td>Sweden</td>
</tr>
<tr>
<td>CH</td>
<td>Switzerland</td>
<td>KR</td>
<td>Republic of Korea</td>
<td>SI</td>
<td>Slovenia</td>
</tr>
<tr>
<td>CI</td>
<td>Côte d'Ivoire</td>
<td>KZ</td>
<td>Kazakhstan</td>
<td>SK</td>
<td>Slovakia</td>
</tr>
<tr>
<td>CM</td>
<td>Cameroon</td>
<td>LI</td>
<td>Liechtenstein</td>
<td>SN</td>
<td>Senegal</td>
</tr>
<tr>
<td>CN</td>
<td>China</td>
<td>LK</td>
<td>Sri Lanka</td>
<td>TD</td>
<td>Chad</td>
</tr>
<tr>
<td>CS</td>
<td>Czechoslovakia</td>
<td>LU</td>
<td>Luxembourg</td>
<td>TG</td>
<td>Togo</td>
</tr>
<tr>
<td>CZ</td>
<td>Czech Republic</td>
<td>LV</td>
<td>Latvia</td>
<td>TJ</td>
<td>Tajikistan</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
<td>MC</td>
<td>Monaco</td>
<td>TT</td>
<td>Trinidad and Tobago</td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
<td>MD</td>
<td>Republic of Moldova</td>
<td>UA</td>
<td>Ukraine</td>
</tr>
<tr>
<td>ES</td>
<td>Spain</td>
<td>MG</td>
<td>Madagascar</td>
<td>US</td>
<td>United States of America</td>
</tr>
<tr>
<td>FI</td>
<td>Finland</td>
<td>ML</td>
<td>Mali</td>
<td>UZ</td>
<td>Uzbekistan</td>
</tr>
<tr>
<td>FR</td>
<td>France</td>
<td>MN</td>
<td>Mongolia</td>
<td>VN</td>
<td>Viet Nam</td>
</tr>
</tbody>
</table>
METHOD AND APPARATUS FOR CONTEXT SENSITIVE TEXT DISPLAYS

BACKGROUND OF THE INVENTION

1. Field of Invention

This invention relates to the field of context sensitive text displays. In particular, this invention relates to determining the display characteristics of text efficiently from the parse node containing the text in a parse derived from the text.

2. Description of Background Art

Computers have long been used to display text to humans to facilitate editing and understanding. Typically, a human is using a computer to manipulate
or view a block of text. For purposes of illustration here, a block of text will be assumed to be simply an array of characters.

In a modern computer text editor or text display tool, displaying text on a computer screen involves several steps. First, the tool needs to identify where in the text array the display should begin. This amounts to having an integer that indicates the starting position in the array. This information can be provided to the tool from the human being by keystrokes indicating a line number or a scroll bar indicating the relative position in the file. Next, the tool needs to "paint" the screen with the characters. Conceptually, the tool gets a character at the indexed point, determines the screen display characteristics, such as color, font, size, or format, associated with that character at that point in the text array, and provides that information to the graphical display hardware. The computer then increments the index and repeats the process with the next character.

Conventional text editors require the user to specify the display characteristics associated with particular points in the text without regard to what the particular text is or "means." However, in certain applications, the text may have context that the computer could use to determine the display characteristics. For example, computer programs are written in computer languages like C, Pascal, or Cobol. A program written in computer language has text strings within it that have a particular meaning or significance, such as keywords like "if", "else", or "while". A more sophisticated text display tool could use this context-sensitive information within the text to define the display characteristics of the text. Borland's C++ development environment for DOS machines provides a text editor that, for example, displays keywords with one color and comments in another color.
Using the context of the text to determine the display characteristics presents computational performance problems. The user wants to see the text displayed on the computer screen displayed promptly. However, setting the display characteristics as a function of the context requires that the computer compute the context. That computation can be done at the time the display is requested, or, alternatively, the context computation could be performed at some other, earlier time and the result stored. At the time the display is requested, the computer has to perform some computation either directly or by finding a previous result. This computation takes time. If this time is too long, then the text will not be displayed promptly, thus frustrating the user. Economically viable context-sensitive text displays therefore must efficiently store and compute the context.

A review of the process of extracting context out of certain kinds of text will clarify the problems a context-sensitive display has. One place that context extraction occurs is in the process of generating executable machine code from the text of a computer program written in a high level language. This process is called compilation. Compilation involves several steps: lexical analysis, parsing, code generation, and optimization. Only lexical analysis and parsing are relevant for purposes of displaying context sensitive information. Lexical analysis involves breaking the text into distinct, non-overlapping text strings in accordance with the rules of a specified language. These non-overlapping objects are referred to as tokens. The lexical analyzer can characterize some of these text strings because the text matches a keyword, or is a number or a symbol such as "&". Lexical analysis is computationally fast because it relies only on "local" information to determine where the next token.
After the lexical analyzer breaks the text into tokens, a parser converts the entire sequence of tokens into a parse tree that describes the computer program's structure. From this parse tree, the code generator can create blocks of executable code corresponding to the nodes of the parse tree. The code generator can then link the blocks together to make an executable program. An optimizer can then look for improvements to make in that block to reduce the amount of memory required to store the executable code or to decrease the run time of the executable code. The nature and mathematical structure of compilers is described in Compilers Principles, Techniques and Tools, by Alfred Aho, Ravi Sethi, and Jeffrey Ullman, 1986, ISBN 0-201-10088-6 which is hereby incorporated by reference. The steps for building a compiler are shown in Introduction to Compiler Construction with UNIX, by Axel T. Schreiner and H. George Friedman, Jr., published by Prentice-Hall, 1985, ISBN 0-13-474396-2, which is hereby incorporated by reference.

If it is only desired to modify the display characteristics of the text depending solely on information that can be obtained "locally" from a lexical analyzer, then a text display tool could perform lexical analysis at the time it scanned the text for the display. For common computer languages, this would allow the display tool to paint different keywords in different colors without a noticeable performance problem. However, this may not produce enough information for the user.

Changing the display characteristics of the text in accordance with information associated with the corresponding parts of a parse tree can be very useful. However, because constructing a parse tree requires processing all of the text, a tool using the parse tree to determine the display characteristics of the text
will probably not be able to compute the entire parse tree every time the display needs to be changed. Therefore, such a display tool will generally require that the parse tree be generated once and stored, and then accessed when the need arises. However, conventional parse trees occupy much memory, and require a substantial amount of time to build.

The memory requirements of a conventional parse tree are straight-forward to compute. Consider a text block consisting of $N$ characters. Experience has shown that a file of $N$ characters will generate a parse tree with $N/4$ nodes. A conventional parse tree has nodes with a data structure in C could reasonably declared as follows:

```c
typedef tree_node_struct {
    struct pt_node_struct *parent, **children;
    int child_count;
    char *start_position, *end_position;
    int parse_tree_id;
    /* other stuff */
} tree_node_struct, *pt_node;
```

This data type provides a pointer to the parent node, a pointer to an array of pointers to the children of the node, an integer for counting the number of children, pointers for indicating where, in the text block, the text that generated this node begins and ends, and an integer to indicate what node this is. In a conventional, 32 bit machine, each node will require one word for the parent pointer, one word to point at the array of children, one word for the child_count, one word each for start_position and end_position, and one word for parse_tree_id. In addition, each node will require an additional word in its parent's array of children. Therefore, each parse node will require 7 words of memory. Therefore, this conventional tree structure will require $7N/4$ words to
store the parse tree for an $N$ character file. In a conventional machine, there are 4 bytes per word, thus making the conventional data structure consume $7N$ bytes.

**SUMMARY OF INVENTION**

This invention provides an compact and easily accessed data structure for representing a parse tree associated with a block of text, and efficiently allowing text associated with particular parse nodes to be displayed in manner specified by the parse node. Linking the display characteristics of text with the parse tree associated with the text is particularly useful in viewing the results of circuit analysis for circuits created by synthesis from the same text.

A parse tree consists of one or more parse nodes in a tree configuration. Each parse node has associated with it subordinate parse nodes or a particular contiguous group of characters within the text block. The data structure of this invention represents the parse tree as an array of elements, with each element represented by two bits. The index to this array is related to the position in the text block.

Using two bits for each element provides for four kinds of elements. One element represents the beginning of parse node, called TREE_START. Another element represents the end of a parse node, called TREE_END. Another element represents the existence of a single character, called TREE_CHAR. To make the parse array more compact, the fourth element represents a fixed number of sequential characters, called the block size. The fourth element is called TREE_BLOCK. In this invention, using the fourth element to represent four characters works well.
Generating the parse array representation from a conventional tree representation can be done by recursively traversing the conventional tree. A TREE_START is written every time a child node is accessed. A TREE_END every time a parent node is returned to after accessing a child node. TREE_BLOCKS and TREE_CHARS are written to reflect the size of the text string found at the node. (See the function "ve_write_paren_tree" in Appendix A for an example.)

Constructing a conventional tree representation from the parse array can be done by sequentially marching through the parse array, keeping a stack of parse nodes, and keeping a current index into the text array. When a TREE_START symbol is encountered, a new node is created, made a child of the current node on the top of the stack, and pushed onto the stack. When a TREE_CHAR or TREE_BLOCK symbol is encountered, the appropriate number of characters beginning with the current index in the text array is assigned to the node at the top of the stack. When a TREE_END symbol is encountered, merely pop the stack. (See the function "tree_convert_to_full" in Appendix A for an example.)

Estimating the size of the parse array offers insight into the value of the invention. Each node in the parse tree will require a TREE_START symbol and a TREE_END symbol. These symbols each require 2 bits, so, with 8 bits per byte, each node will therefore require 1/2 byte. If the file has $N$ characters, and tends to have $N/4$ nodes, then the node symbols will require $(1/2)\times N/4 = N/8$ bytes. If no compression was used, then there would need to be storage space for an additional $N$ symbols to store the representation of the characters. In this situation, each character would 2 bits to store, thus requiring $N/4$ bytes to store the character representation in an uncompressed form. However, experience has
shown that using the TREE_BLOCK symbol to represent 4 characters halves the character storage requirement to \( \frac{N}{8} \) bytes. Therefore, the parse array requires \( \frac{N}{8} \) bytes for the node markers, and \( \frac{N}{8} \) bytes for the text, thus requiring \( \frac{N}{4} \) bytes for the entire parse array, compared to the \( 7N/8 \) bytes for the conventional approach. The parse array therefore reduces the memory required by a factor of 28 over a conventional parse tree. The reduce memory usage also leads to a reduced access times, and therefore a faster performing display tool. In addition, this data structure can be loaded directly from permanent storage, such as a file, because it does not contain pointers. This makes the loading time very small, and initialization very fast.

One disadvantage to using any parse tree to guide the selection of display characteristics is that is conceptually necessary to traverse the parse tree to get to the point in the parse tree that corresponds to the portion of text that needs to be displayed. However, this invention helps reduce this computation by creating a table that is keyed to the line number in the text file that stores a copy of the stack created by traversing the parse tree to the point in the text file corresponding to the end of the particular line. To further reduce storage requirements, a stack only needs to be saved every several text lines. (Every 16 lines works well.) When it comes time to get information from the parse tree, it is only necessary to get a copy of the stack corresponding to the nearest preceding recorded line, and continue traversing the parse tree from there.
-9-

BRIEF DESCRIPTION OF THE FIGURES

Figure 1. A parse tree showing the text corresponding to the nodes.

Figure 2. A textual representation of the parse tree of Figure 1 in accordance with the present invention.

Figure 3. A binary representation of the parse tree of Figure 1 in accordance with the present invention.

Figure 4. A flow chart showing a method of setting the display characteristic of the text as a function of the parse node containing that text.

Figure 5. A flow chart showing a method of obtaining the initial parse node corresponding to a particular point in the text array.

Figure 6. A generalized diagram showing relationships between parse node, program text and electronic structures created from the text.

Figure 7. A diagram of a general purpose computer system.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT

The present invention comprises a novel method and provides a novel structure for compactly storing a parse tree and using that parse tree to change the display characteristics of the associated text efficiently. The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the preferred embodiment will be
readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the invention. Thus, the present invention is not intended to be limited to the embodiment shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

Figure 7 is a simplified block diagram illustrating a general purpose programmable computer system, generally indicated at 200, which may be used in conjunction with a first embodiment of the present invention. In the presently preferred embodiment, a Sun Microsystems SPARC Workstation is used. Of course, a wide variety of computer systems may be used, including without limitation, workstations running the UNIX system, IBM compatible personal computer systems running the DOS operating system, and the Apple Macintosh computer system running the Apple System 7 operating system. Figure 7 shows one of several common architectures for such a system.

Referring to Figure 7, such computer systems may include a central processing unit (CPU) 202 for executing instructions and performing calculations, a bus bridge 204 coupled to the CPU 202 by a local bus 206, a memory 208 for storing data and instructions coupled to the bus bridge 204 by memory bus 210, a high speed input/output (I/O) bus 212 coupled to the bus bridge 204, and I/O devices 214 coupled to the high speed I/O bus 212. As is known in the art, the various buses provide for communication among system components. The I/O devices 214 preferably include a manually operated keyboard and a mouse or other selecting device for input, a CRT or other computer display monitor for output, and a disk drive or other storage device for non-volatile storage of data and program instructions. The operating system typically controls the above-identified components and provides a user interface. The
user interface is preferably a graphical user interface which includes windows and menus that may be controlled by the keyboard or selecting device. Of course, as will be readily apparent to one of ordinary skill in the art, other computer systems and architectures are readily adapted for use with embodiments of the present invention.

The present invention advantageously uses the fact that parse node information has been linked to automated design display technology. For example, netlist data structures may retain links to parse nodes generated from HDL code used to produce such structures. Moreover, graphical display tools used to illustrate features or performance of such structures also may contain links to the parse nodes generated from HDL code. See, for example, commonly assigned U.S. Patent Application Serial No. 08/417,147, filed April 3, 1995, which is expressly incorporated herein by this reference.

Referring to Figure 6, there is provided an overview which illustrates where the mechanisms of the present invention fit within the overall process of using parse nodes to create electronic or software-based structures. HDL text 2000 is created by a user. At compile time, a parse tree 2002 is created. Nodes on the parse tree correspond to portions of the HDL text. Nodes of the parse tree also correspond, for example, to cells 2004-2016 within a netlist 2018 and, for example, to portions 2020, 2022, of a display image 2024 used to analyze the netlist. Arc 2003 indicates that cell 2004 was created from parse node 2005. A netlist may retain references, such as arc 2026, to a parse node used to create netlist nodes and nets. Similarly, electronic structures within the display tool also may have references, such as arc 2028, to a parse node used to create netlist cells or nets analyzed in the display.
The mechanism of the present invention advantageously provides a two-way link between parse node information and text. See, for example, two-way link 2030 which can be created through the mechanism of the present invention. It should be appreciated that a parse tree, once created, often is not saved. Nevertheless, as explained above, electronic structures such as a netlist cell or computer display information may retain references to the parse nodes of the parse tree after the tree has been discarded. The mechanism of the present invention permits linkages between text and electronic structures based upon retained parse node information.

Appendix A contains a C program listing describing the data structure of the present invention and methods of manipulating the data structure to obtain display characteristics of the associated text. The procedures beginning with "xm" or "Xt" or "topLevelShellWidgetClass" involve the graphical user interface, and are described in "The X Window System: Programming and Applications with XT, COSF/Motif edition", by Douglas Young, published by Prentice-Hall in 1990 with ISBN 013-497074-8, which is hereby incorporated by reference. Basic data structure manipulations such as hash tables and stacks are described in "Introduction to Algorithms" by Thomas Cormen, Charles Leiserson and Ronald Rivest, by the MIT Press in 1990 with ISBN 0-07-013143-0, which is hereby incorporated by reference.

The routine "os_show_stat" merely gathers computer performance statistics and is not functionally related to the parse array operations. "Error_if" is based on the "assert" macro commonly used in C programs. "Error_if" causes the program to terminate if a specified condition is true, and continue if the condition is false.
The program listing in appendix A along with its comments and the library references are hereby incorporated by reference into the specification.

The Array Data Structure

Figure 1 illustrates a parse tree associated with some text. The parse tree consists of nodes 100, 101, 102, 103, 104, 105, and 106. The characters 1 through 13 represent generic characters. Characters 1, 2, 3, 4, 5 and 6 are associated with node 102. Characters 7 and 8 are associated with node 103. Characters 9, 10 and 11 are associated with node 105 and characters 12 and 13 are associated with node 106.

Figure 2 illustrates a text representation of the parse tree using "{" to mark the beginning of a node and "}" to mark the end of a node. For example, left brace 30 and right brace 40 together contain all of the text and nodes associated with node 100. Left brace 31 and right brace 41 demarcate the text and nodes associated with node 101. Left brace 32 and right brace 42 demarcate the text associated with node 102. Left brace 33 and right brace 43 demarcate the text associated with node 103. Left brace 34 and right brace 44 demarcate the text associated and nodes with node 104. Left brace 35 and right brace 45 demarcate the text associated with node 105. Left brace 36 and right brace 46 demarcate the text associated with node 106. One alternative embodiment of the present invention is to rewrite the text data structure with a brace or equivalent symbol inserted into the text.

The left braces serve as begin markers which denote the beginning of information encompassed by a parse node. The right braces serve as end markers which denote the end of information encompassed by a parse node.
Begin and end markers are grouped in sets: a left brace in a set denotes the beginning of information encompassed by a parse node. A right brace in the same set denotes the end of information encompassed by a parse node.

Figure 3 illustrates the use of a parse array 150 to store a representation of the parse tree using the text representation of Figure 2 as a guide. Parse array 150 is really an array of bits divided into bit pairs 200 through 223. Each bit pair is used to represent a symbol. In one embodiment, the beginning of node is denoted with the TREE_START symbol, and that symbol has a value of "00". The TREE_START symbol is a begin marker. The TREE_END symbol has value "01" and is used to demarcate the end of a node. The TREE_END symbol is an end marker. The TREE_CHAR symbol has value "10" and is used to demarcate a single character. The TREE_CHAR symbol is a character marker. The TREE_BLOCK symbol has value "11" is used to demarcate a group of 4 characters.

Using this encoding, the parse tree of Figure 1 begins with the "00" to denote the beginning of the node 100, as shown with bit pair 200. The next two bits are set to "00" to denote the beginning of node 101, as shown with bit pair 201. Bit pair 202 holds "00" to denote the beginning of node 103.

Note that node 103 has 6 characters associated with it. One embodiment of the invention would use the value "11" in bit pair 203 to note the presence of characters 1, 2, 3, and 4, while bit pairs 204 and 205 each hold a "10" to mark a place for characters 5 and 6 respectively. Other compression schemes could be used to store the character count.
Bit pair 205 holds the value "01" to mark the end of node 103.

**Determining Node Identification (Direct Computation)**

It can be very useful to associate an unique identification number with a particular node in a parse tree. A conventional parse tree often allocates an entire word in the data structure to hold that information. However, the parse array provides that information indirectly. The identification number of the node at a particular point in the parse array is the number of TREE_START symbols to the "left" of the TREE_START symbol corresponding to the particular point. For example, if it is desired to determine the node identification number for the node containing bit pair 215 of Figure 3, merely back up to the TREE_START symbol demarcating the boundary of the node containing bit pair 215. This would be bit pair 213 and count the number of TREE_START symbols preceding bit pair 213. In this case, there are 5, namely bit pairs 200, 201, 202, 207, and 212. Therefore, the node corresponding to bit pair 213 is uniquely identified as node number 5.

Of course, if the request for bit pair identification is made at a point where a TREE_START symbol is already located, it is not necessary to back up.
Constructing the Parse Array from a Conventional Parse Tree

Constructing the parse array from a conventional parse tree can be done as follows. Define an index and a tree pointer. Set the index to zero, and the tree pointer to the root node. At a particular node, write all unwritten characters up to the start of the node, write the TREE_START symbol into the parse array, and increment the index. Recursively apply this start symbol/character rule for each child node. If there are no more children, write all the unwritten characters up to the end of the node, then write the TREE_END symbol into the parse array, increment the index, and return to the processing the parent node. See "ve_write_paren_tree" in Appendix A for an example.

Constructing A Conventional Parse Tree from a Parse Array

Constructing a conventional parse tree from the parse array can be done as follows. Define a stack pointer, a text index, and a parse array index. Set the text index and the parse array index to zero.

If the symbol in the parse array at the parse array index is the TREE_START symbol, create a new parse node. The new parse node should be made a child of the node at the current stack pointer, if there is anything on the stack pointer. Push the new node onto the top of the stack, and increment the parse array index, and process the next symbol.

If the symbol in the parse array at the parse array index is a TREE_BLOCK or TREE_CHAR, then associate the appropriate number of characters in text block beginning with the text index with the parse node on...
top of the stack. Increment the text index and the parse array index appropriately.

If the symbol in the parse array at the parse array index is the TREE_END symbol, then pop the parse node on the top of the stack, and increment the parse array index. See "tree_convert_to_full" in Appendix A for an example.

**Associating Display Characteristics with a Parse Tree**

A variety of information can be attached to a parse tree that would be useful to guide the selection of the display characteristics. One way to do this is to allow another component of a system to define the display characteristic for every parse node. By using an unique identification number associated with every parse node, the display tool could look up or compute the appropriate display characteristic for that node.

For example, in conjunction with circuit analysis and as described in U.S. application 08/226,147 by Gregory et al filed on April 12, 1994, it is possible to determine the circuit characteristics associated with a particular piece of source text. For example, it might be useful to change have the text corresponding to the critical path of the circuit displayed in red while all other parts are displayed in black. A database in conjunction with the analysis tools could identify the parts of the parse tree that are on the critical path. The display tool could then inquire about the status of a particular node, and use that to select the color for the text corresponding to that node.
Using the Parse Array to Determine Display Characteristics (Simplified Version)

The process for determining the display characteristics of the text associated with a particular node is shown in Figure 4. This section provides a general and simple explanation of the process. Later sections identify modifications that can be made to decrease execution time. The process begins at step 1001 where the human interface portion of the software identifies the place in the text where the user wants the display to begin. This amounts to computing an index into the text array.

Step 1002 finds the parse node containing the starting point of the text. In particular, step 1002 involves traversing the parse tree and keeping track of how many characters in the text have been covered as well as keeping track of the current parse node. When the number of characters covered equals or exceeds the index value, then the current parse node is the parse node containing the particular text point. Figure 5 shows a conceptually straight-forward approach to this process.

Step 1101 of Figure 5 involves initializing a character count, the parse array index, and the current node id to zero and creating a stack.

Step 1102 involves identifying the bit pair in the parse array at the location specified by the parse array index.

If the symbol identified in step 1102 is the TREE_START symbol, then push the current node id onto the stack and increment the value of the current node id as shown in step 1103.
If the symbol identified in step 1102 is the TREE_END symbol, then pop the value on top of the stack into the current node id as shown in step 1103.

If the symbol identified in step 1102 is the TREE_BLOCK symbol, then increment the character count by the number of characters represented by the TREE_BLOCK symbol as shown in step 1104. In one embodiment, this is four characters.

If the symbol identified in step 1102 is the TREE_CHAR symbol, then increment the character count by 1, as shown in step 1105.

After incrementing the character count in steps 1104 or 1105, determine if the parse tree has been parsed far enough by comparing the character count with the index into the text array as shown in step 1107. If the tree has been parsed far enough, stop this process and go to step 1002 of Figure 4. To proceed efficiently in the process of Figure 4, it useful to keep the variables and stack computed in Figure 5 intact.

If the parse tree has not been traversed far enough, increment the parse array index as indicated by step 1108, and return to step 1102.

After identifying the appropriate parse node from the process in Figure 5, then it is necessary to obtain the display characteristic associated with the current parse node. This is done through a data base or a look-up table in step 1003.
In step 1004, the character is painted onto the screen with the characteristic established in step 1003.

In step 1005, the next character is identified by incrementing the character index by one. In step 1006 it is determined whether the display block has been completed. If it has, then stop the process. Otherwise, determine if the next character belongs to the current parse node, as shown in step 1008. If the current bit pair referred to by the parse array index is a TREE_BLOCK, and the new character is part of that block, then the character is within the same parse node, and the painting continues in step 1004. If the block has been exceeded or the current bit pair is a TREE_CHAR, then the parse array index needs to be incremented to obtain a new current bit pair. If the new current bit pair is a TREE_BLOCK or a TREE_CHAR, then the new character is part of the current parse node, and painting continues in step 1004. If the new current bit pair is a TREE_END, then the parse node changes in step 1009.

Adjusting the current parse node requires adjusting the stack formed in step 1002 and the current parse node and incrementing the parse array index until the parse array index refers to bit pair that is a TREE_BLOCK or a TREE_CHAR. Adjusting the stack involves popping the top of the stacking into the current parse node when a TREE_END is encountered and pushing the current parse node onto the stack when a TREE_START is encountered. After a new parse node is identified, the display characteristic is determined in step 1003.
Alternate Text Representation

The processes shown in Figure 4 and Figure 5 assumed that the text was stored as an array. The techniques shown can be trivially modified for other text data structures. For example, another approach to storing text is to have an array of pointers with each pointer referring to a block of text with each block of text ending a new line character. Using this another data structure requires modifying the character incrementing step 1005 of Figure 4, and step 1105 and step 1106 of Figure 5.

Efficient Initialization

The processes shown in Figure 4 and Figure 5 require traversing the parse tree from the beginning of the text block to identify current parse node. This can require much computational effort. One way to improve performance in step 1002 is to create a storage mechanism that can retrieve the parse tree traversal state as a function of the text display index. This can be done efficiently by observing a few characteristics of the text display and the parse tree traversal state.

The process in Figure 5 manipulates several variables, namely the character count, the parse array index, the current node, and the stack as the computer marches through the parse array. Together, this information specifies the state of traversing the parse tree. Conceptually, the process in Figure 5 specifies a parse tree traversal state for every character in the text array. For example, see the type definition for "tree_gen" in Appendix A. In principle, with some adjustments for the compaction performed using the TREE_BLOCK symbol, one could store this state in step 1107, and create a look-up table that would hold the state for each character in the text array.
Then the process of Figure 5 could be replace by looking data up. This would result in a very fast access time. Unfortunately, a table for every character would occupy a great deal of memory and take a long time to create.

However, an intermediate approach could work where the state at step 1107 is stored for every fixed number of characters. For example, the state could be stored in a table every 256 characters. At the initialization step 1101, a table index could be computed by dividing the text index by 256 and truncating, and the various variables initialized to the value stored in the corresponding table entry. This way, the process in Figure 5 would only need to parse the tree for no more than an additional 255 characters.

The state storing scheme could also be modified to deal with line numbers instead of the number of characters. In step 1001, the user interface could provide a line number in the file where the display is to start. The parse tree traversal state of step 1107 could be stored every several lines in a table. Experience has indicated that storing every 16 lines works well. At the initialization step 1101, a table index could be computed from the line index, and the initial parse tree traversal state set to the values found in the table.

**Efficient Traversal**

The parse tree traversal method shown in Figure 5 proceeds with one symbol in the parse array at a time. Determining the correct symbol from the parse array directly would require many machine instructions for massaging the bits in the array to identify the symbol, and process it accordingly. An alternate method that takes advantage of the compact nature of the representation processes four symbols concurrently by using a look-up table to determine the
appropriate actions required to process the symbols contained in a byte of storage. In particular, the function "tree_build_expanded_array" on data input converts a byte's worth of parse array to a short list of symbols. The function "tree_gen_next" uses this converted representation to proceed through the parse array efficiently. A similar approach could be used where the parse tree is traversed in step 1009 of Figure 4.
WHAT IS CLAIMED IS:

1. An electronic memory including a mapping between computer program text and parse node information about a parse tree produced from such computer program text, the memory comprising:
   an array which includes,
   respective sets of begin and end markers that correspond to respective parse nodes of the parse tree wherein respective sets of begin and end markers are ordered in the array such that respective corresponding begin and end markers for any respective given parent parse node encompass respective corresponding begin and end markers for every respective child parse node of such respective given parent node; and
   respective character markers that correspond to respective characters of the computer program text wherein respective character markers are ordered in the array such that any respective given character marker in the array is encompassed by every respective set of begin and end markers that correspond to a parse node that covers such character.

2. An electronic memory including a mapping between computer program text and parse node information about a parse tree produced from such computer program text, the memory comprising:
   an ordered array which includes,
   respective sets of begin and end markers that correspond to respective parse nodes of the parse tree wherein respective begin markers are ordered in the array such that a respective corresponding begin marker for any respective given parse node is placed before each respective begin marker of each respective parse node beneath such parent node in the parse tree and wherein a respective corresponding end marker for such respective given parse
node is placed after each respective end marker of each respective parse node
beneath such parent node in the parse tree; and
respective character markers that correspond to respective characters
of the computer program text wherein respective character markers are ordered
in the array such that any respective given character marker in the array is
encompassed by every respective set of begin and end markers that correspond
to a parse node that covers a given character that corresponds to such given
character marker.

3. The memory of claim 2 wherein:
respective character markers are ordered in the array such that any
respective given character marker that corresponds to a respective given
character that is covered by a parse node in a higher level of the parse tree and
by another parse node in a lower level of the parse tree is placed in the array
between a set of begin and end markers that correspond to the parse node in
the higher level of the parse tree and is placed in the array between another set
of begin and end markers that correspond to the parse node in the lower level
of the parse tree.

4. The memory of claim 2 wherein:
respective character markers are ordered in the array such that any
respective given character marker that corresponds to a respective given
character that is covered by a first parse node in a higher level of the parse tree
and by a second parse node in a lower level of the parse tree is placed in the
array between first a set of begin and end markers that correspond to the first
parse node and is placed in the array between second set of begin and end markers that correspond to the second parse node; and

such respective given character marker is outside a third set of begin and end markers that correspond to a third parse node.
Figure 1
SUBSTITUTE SHEET (RULE 26)
1001 Identify beginning part of text block to display

1002 Find parse node corresponding to beginning of display block

1003 Determine display characteristic from parse node

1004 Paint character

1005 Advance character pointer

1006 Within displayed block?

1007 End

1009 Determine next parse node

1008 Within same parse node?

Figure 4
Initialize a stack, character count, parse array index, and current node id

Identify symbol type in parse array at parse array index

TREE_START

TREE_END
Push current node id value onto stack. Increment current node id

TREE_CHAR
Pop node id on top of stack into current node id value

Increment character count by one

Increment character count by four

Advance parse array index

character count exceed index?

Yes

End

No

SUBSTITUTE SHEET (RULE 26)
INTERNATIONAL SEARCH REPORT

I. CLASSIFICATION OF SUBJECT MATTER
IPC G  G06F17/22

According to International Patent Classification (IPC) or to both national classification and IPC.

II. FIELDS SEARCHED
Minimum documentation searched (classification system followed by classification symbol)
IPC G  G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>EP, A, 0 423 959 (APPLE COMPUTER) 24 April 1991 see the whole document</td>
<td>1-4</td>
</tr>
<tr>
<td>A</td>
<td>EP, A, 0 361 737 (IBM) 4 April 1990 see abstract</td>
<td>1,2</td>
</tr>
<tr>
<td>A</td>
<td>IBM TECHNICAL DISCLOSURE BULLETIN, vol. 34, no. 5, October 1991 NEW YORK, US, pages 332-337, ANONYMOUS 'Symbol Table Based Program Editor.' see the whole document</td>
<td>1,2</td>
</tr>
</tbody>
</table>

Further documents are listed in the continuation of box C.

Patent family members are listed in annex.

Date of the actual completion of the international search: 8 September 1995

Name and mailing address of the ISA
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040, Tx. 31 651 epc nl, Fax (+31-70) 340-3016

Authorized officer
Suendermann, R

Form PCT/ISA/210 (second sheet) (July 1993)
<table>
<thead>
<tr>
<th>Patent document cited in search report</th>
<th>Publication date</th>
<th>Patent family member(s)</th>
<th>Publication date</th>
</tr>
</thead>
</table>

Form PCT/ISA/218 (patent family annex) (July 1992)