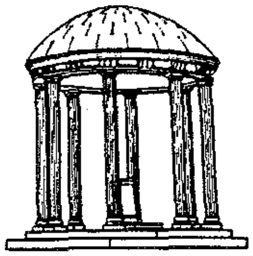
Formatting Texts Accessed Randomly

TR85-031

November, 1985

John B. Smith and Stephen F. Weiss

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175



A TextLab Report

Formatting Texts Accessed Randomly

Introduction

Storing, retrieving, and displaying text are increasingly important computing activities. Commercial full-text databases now range from the research literature for chemistry [American Chemical Society, 1982] to clippings from the popular press [New York Times, 1981], from legal codes [Menanteaux, 1982] to literary works [Morrissey & Del Vigna, 1983]. Because of the novelty of this new resource, users have accepted relatively primitive forms of textual output. Some services provide data in only uppercase; others offer output that includes upper and lower case and paragraph indentation. But, no full-text system makes effective use of today's more sophisticated low-cost output devices, such as laser printers and graphic terminals.

One could argue that enhanced formatting of data extracted from a full-text database is not essential—that the new service is so valuable, the user should be glad to have *any* form of output. However, full, accurate formatting is not *just* aesthetic. Consider the following hypothetical situation. Congressmen frequently read into the *Congressional Record* comments made by others. While they often quote those who agree with their position, they sometimes quote those who disagree in order to rebutt or ridicule that person or that point of view. A full-text search of the *Congressional Record* for passages containing certain combinations of words could locate a passage quoted by a Congressman that represented a position opposite his own. If the display did not signal through formatting that the extracted portion was a quotation, instead of the Congressman's own words, the user could be badly misled. Format information can, thus, contribute to the *substance* of a text as well as to its appearance. However, storing and using format information for full-text systems that employ random access methods presents several problems.

At the time of display, format information is manifest in the pixel image of the text or in some other analog form. One could store the actual pixel image of the original text in the database, but the volume of data required makes this option impractical as well as undesirable from the standpoint of search. Normally, format information is stored in the form of commands interspersed through the character stream that represents the text. Those commands, in turn, activate various functions in a formatting program that operate on the data and/or the display device. That is, they may activate a shift to an alternative font, such as Italics, to mark titles or they may activate a shift to the right to mark long quotations.

Other problems arise from the way in which format information is used. When a text is processed *sequentially*, as presumed by virtually all formatting systems, the system "knows" that it has processed or sent to the output device all formatting commands active for the current segment of text. However, when a text is processed *randomly*, using some form of indexed or inverted file structure, the system would not process the entire text sequentially but, instead, would jump directly to the passage identified and begin processing the text from that point. Thus, it does not "know" what format commands may be in effect at that point (e.g., in the middle of a long quotation, an Italicized passage, or a heading). The system could be instructed to "back-up" to some predefined checkpoint, such as the beginning of a section, for which no format command would be permitted to span. But, in general, this approach is restrictive and unpredictable in terms of performance.

For full-text systems that support random access, a more efficient and more predictable approach is needed. We are currently developing a system, called MICROARRAS, to provide high-performance search, retrieval, and analysis of textual data. We also want to support sophisticated output devices. The approach we are taking is to view the set of format commands as the symbols in a format grammar. The string of format commands can then be parsed using the grammar to build a data structure that serves both as a parse tree and as a search tree. While processing a retrieved text segment, the system follows the rather shallow search tree and pipes out the format commands encountered at each node to accumlulate the format commands active for that segment. Below, we describe each of these steps in more detail. To illustrate our approach, we use this article as a sample text and show the application of these methods to it.

Format Commands

For practical as well as theoretical reasons, we view individual format commands as symbols identifying generic classes of information in a document. That is, a command that identifies a long quotation signals that fact, not the indentation or other formatting convention by which the quotation is marked on the printed or displayed page. Thus, the set of format commands can be viewed as the architectural principles that give physical form to the textual substance. This perspective has been voiced most strongly in IBM's General Markup Language in its concept of ***document architecture*** [IBM, 1980], but a similar view is also found in Scribe [Reid, 1980], Microsoft's Word [Microsoft, 1985], and other more recent formatting systems [Futura, Scofield, & Shaw, 1982].

Below is the set of high-level generic commands necessary to format this paper plus a few extras. In most cases, commands come in pairs: the first defines the beginning of the ***domain*** of the operation; the second defines its end. Commands are signalled by a reserved symbol—in this case the backslash (\).

- Sections:

\body-b	body of text begin
\body-e	body of text end
\section-b	section begin
\section-e	section end
\para	paragraph
\header-b	section header begin
\header-e	section header end
\backmatter-b	backmatter begin
\backmatter-e	backmatter end

- Footnotes

\footnote-r	reference to footnote
\footnote-b	footnote begin
\footnote-e	footnote end

- Figures:

\figure-b	figure begin
\figure-e	figure end
\figure-body-b	figure body begin
\figure-body-e	figure body end
\figure-caption-b	figure caption begin
\figure-caption-e	figure caption end

- List

\list-num-b	numbered list begin
\list-num-e	numbered list end
\list-bul-b	bulleted list begin
\list-bul-e	bulleted list end
\list-b	list begin
\list-e	list end
\item-l	item left
\item-r	item right
\item	item

- Quotes

\quote-long-b	long quote begin
\quote-long-e	long quote end
\quote-short-b	short quote begin
\quote-short-e	short quote end

- Production Rules (Special-purpose):

\production-b	production begin
\production-e	production end
\production-l	left component
\production-m	middle component
\production-r	right component

- Title Page

\titlepage-b	title-page begin
\titlepage-e	title-page end
\title-b	title begin
\title-e	title end
\author-b	author begin
\author-e	author end
\address-b	address begin
\address-e	address end
\date-b	date begin
\date-e	date end

- Emphasis

\emphasis-b	emphasis begin
\emphasis-e	emphasis end

From one perspective, these commands are simply macro names and could be implemented that way using a number of different formatting systems. However, from a different perspective no symbol says anything directly about physical appearance. Each simply identifies, within the text sequence, a category of information or the end of that category.

One restriction we have imposed is that format domains may be nested but they may not overlap. The result, then, is a hierarchy of format functions and domains.

Format Grammar

In this section, we introduce a context-free grammar that specifies the set of well-formed formatted texts and formally captures the hierarchical structure imposed by the format operations. Format operations are the nonterminals; word tokens are the terminals. (In practice, the parser ignores the text words except to note the position of a format mark within the numerical sequence of word tokens.) Note that we use the generic word marker w rather than actual words in the grammar.

This grammar differs from a traditional context-free grammar in that the right-hand sides of the productions may contain regular expressions made up of terminals, nonterminals, and special operators.

$[x]$	Material inside the brackets is optional
$x + y$	Choice operator: $x + y$ means either x or y. The $+$ operator has lower recedance than concatenation. Thus $a + bc$ means either a or bc.
x^*	Kleene star: operand may appear 0 or more times
x^{*2}	Modified Kleene star: operand may appear 0, 2 or more times.
x^+	Shorthand for x followed by x^*: one or more occurrence of x
x^{+2}	Shorthand for xx followed by x^*: two or more occurrences of x
()	Parentheses used to define grouping.

While strictly speaking these modified productions are not context-free, they are actually just a notational shorthand for a much larger set of context-free productions that could have been specified (see Appendix A). Additionally, the modified production rules allow the grammar to produce a parse tree whose structure more accurately reflects the true structure of the document. For example, production 9 specifies that a section of the text can consist of, among other things, an arbitrary number of paragraphs. Figure 1 shows the structure derived for a section comprised of five paragraphs.

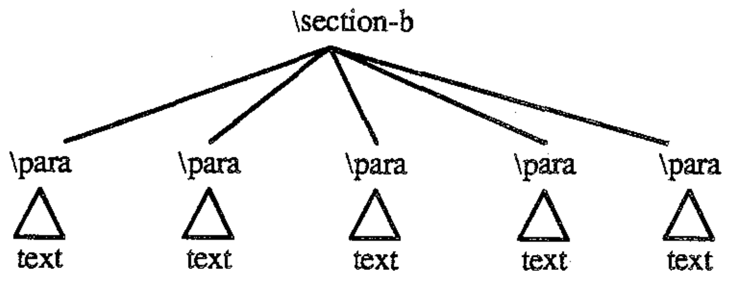


Figure 1
Five-Paragraph Section

Generating the same terminal string with purely context-free rules would require productions such as

$\backslash$section-b $\longrightarrow$ para-seq

para-seq $\longrightarrow$ $\backslash$para para-seq

para-seq $\longrightarrow$ $\backslash$para

Such rules require new nonterminals not directly associated with format operations and introduce an artificial and spurious hierarchical relationship among the paragraphs that can be seen in Figure 2.

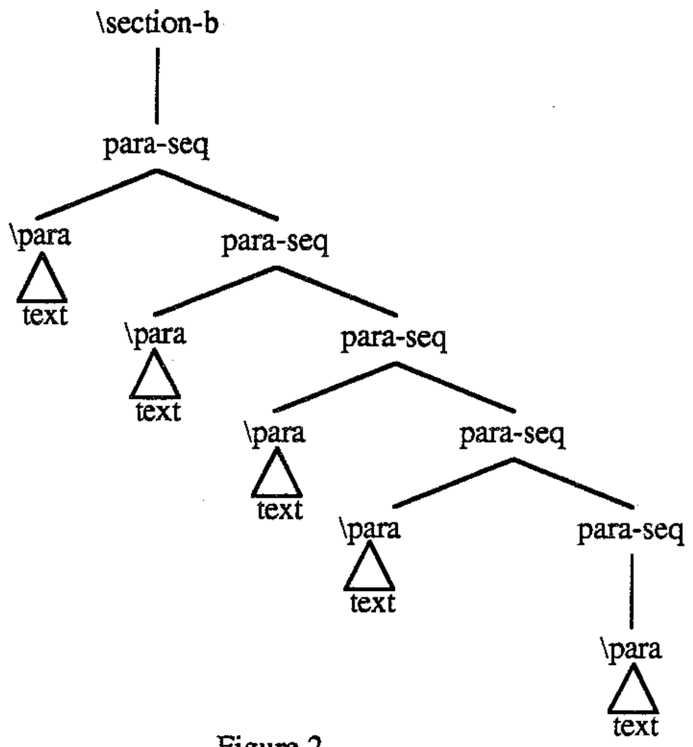


Figure 2
Five paragraph section from pure context-free rules

Following is a format grammar, using the format commands listed above, adequate to parse this article.

0. root → \text-b \text-e

1. \text-b → [\titlepage-b \titlepage-e] \body-b \body-e [\backmatter-b \backmatter-e]

2. \titlepage-b → \title-b \title-e
   (\author-b \author-e)*
   (\address-b \address-e)*
   [\date-b \date-e]

3. \title-b → t

4. \author-b → t

5. \address-b → t

6. \date-b → t

7. \body-b → \para* (\section-b \section-e)^{*2}

8. \para → t

9. \section-b → [\header-b \header-e] \para* (\section-b \section-e)^{*2}

10. \header-b → t

11. t → (*w*+
    \emphasis-b \emphasis-e +
    \quote-short-b \quote-short-e +
    \quote-long-b \quote-long-e +
    \list-b \list-e +
    \list-num-b \list-num-e +
    \list-bul-b \list-bul-e +
    \footnote-b \footnote-e +
    \footnote-r +
    \figure-b \figure-e +
    \production-b \production-e
)*

12. \emphasis-b → t

13. \quote-short-b → t

14. \quote-long-b → t \para*

15. \list-b → \item^{+2}

16. \list-num-b → \item^{+2}

17. \list-bul-b → \item^{+2}

18. \item ⟶ t + \production-b \production-e
+ \item-l \item-r

19. \item-l → t + \production-b \production-e

20. \item-r → t + \production-b \production-e

21. \footnote-r → literal

22. \footnote-b → t \para*

23. \figure-b → \figure-body-b \figure-body-e \figure-caption-b \figure-caption-e

24. \figure-body-b → literal

25. \figure-caption-b → t

27. \production-b ⟶ \production-l \production-m \production-r

28. \production-l ⟶ w^+

29. \production-m ⟶ "⟶"

30. \production-l ⟶ t

31. \backmatter-b → t

32. {any operation}-e → e (each scope terminating operator is replaced by the empty string)

The nonterminal t is not associated with any format operator, but is instead a notational shorthand for the right-hand-side of production 11. In the actual parse tree, t's are eliminated; the children of each t are lifted and become the children of t's parent node.

The Parse/Search Tree

Figure 3 shows the parse tree associated with the second paragraph of the next section (beginning "Assume that we are searching ..."). Contiguous strings of word tokens are indicated using elipses. The parse tree serves four distinct and useful roles. First, the tree indicates the structure and well-formedness of the text with respect to the format operations. Second, a left to right scan of the leaves of the tree yields the text without format operations. Third, a pre-order traversal of the tree yields the complete text with format operations in place. And finally, with the addition of some search information at each node, a top to bottom scan of a path in the tree from the root to an arbitrary word *w* will recover all the formatting operations that apply to *w*. Since the height of the parse tree is typically proportional to the logarithm of its number of leaves, recovering the formatting environment in this way is far faster than a sequential scan from the beginning of the text.

Of the four capabilities, only the first and fourth are important. The text file is a far more efficient source of running text (with or without format operations), and we will assume the availability of such a file. Thus, we will use the parse tree only for determining well-formedness and for recovering the formatting environment for an arbitrary token in the text. For these applications we need not actually store the word tokens in the parse tree. Instead, we represent individual word tokens by a pointer into the sequential text file. Strings of contiguous tokens are represented by a pointer to the beginning of the string in the sequential text file and an integer indicating the length of the string. This provides for a very compact representation of text strings and greatly reduces the size of the parse tree. Figure 4 shows the tree from Figure 3 represented this way. (Appendix B shows the paragraph printed in list form.)

Figure 5 shows the simple parse tree into which search information has been added. Two numbers are associated with each node. The first is the linear position in the text of the first token within the domain of the operation associated with that node. The second is the number of tokens within the domain. Empty domains, for operations such as \emphasis-e, have zero length. Thus, in the example, the domain of \para is from token 1 through token 187; the domain of the first \emphasis-b is token 16; and \emphasis-e has an empty domain.

Since the parse/search tree will not be used for text reconstruction, it need not actually contain contiguous blocks of text nor operations with empty domains. Figure 6 shows the tree from previous figures as it is actually stored.

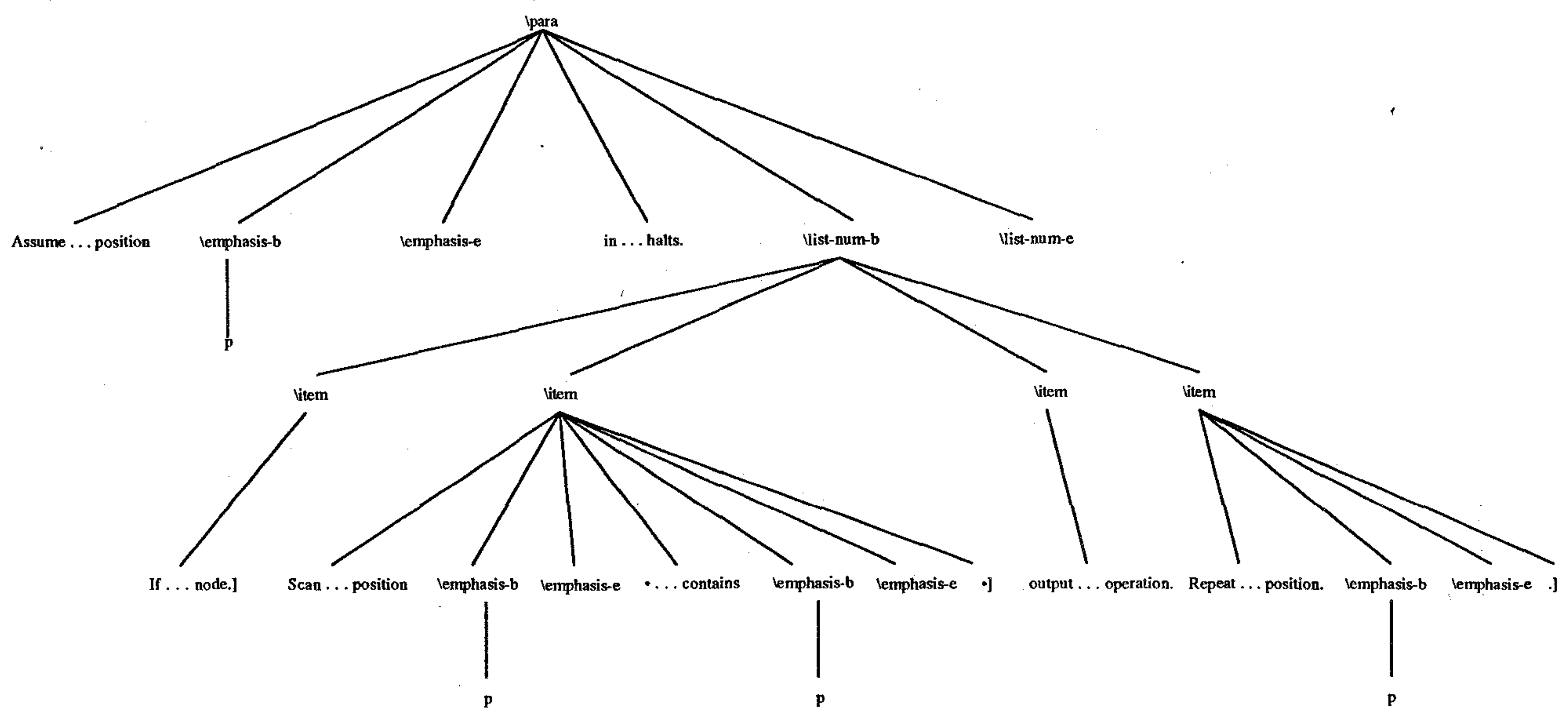


Figure 3
Parse of a paragraph with embedded emphasis and numbered list

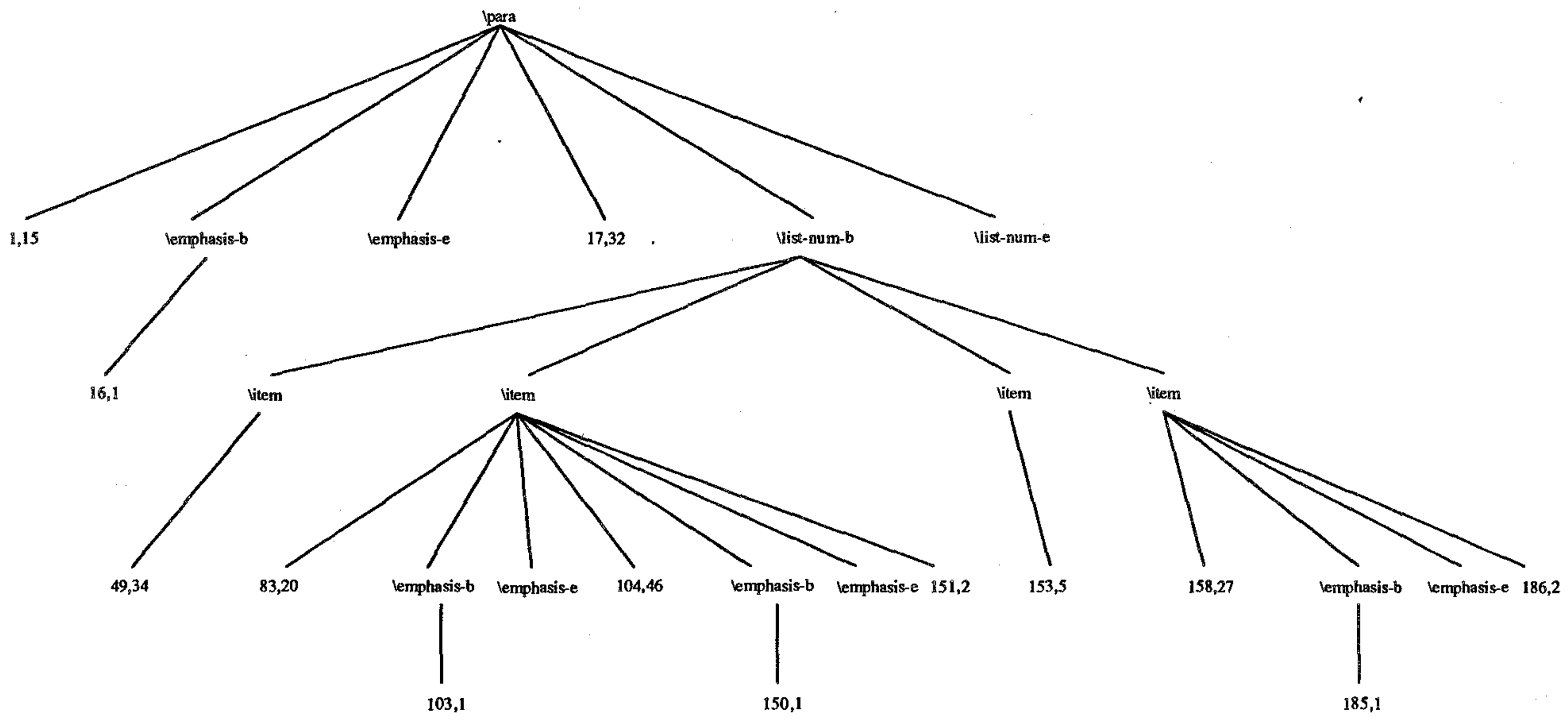


Figure 4
Parse of a paragraph with text replaced by pointers.

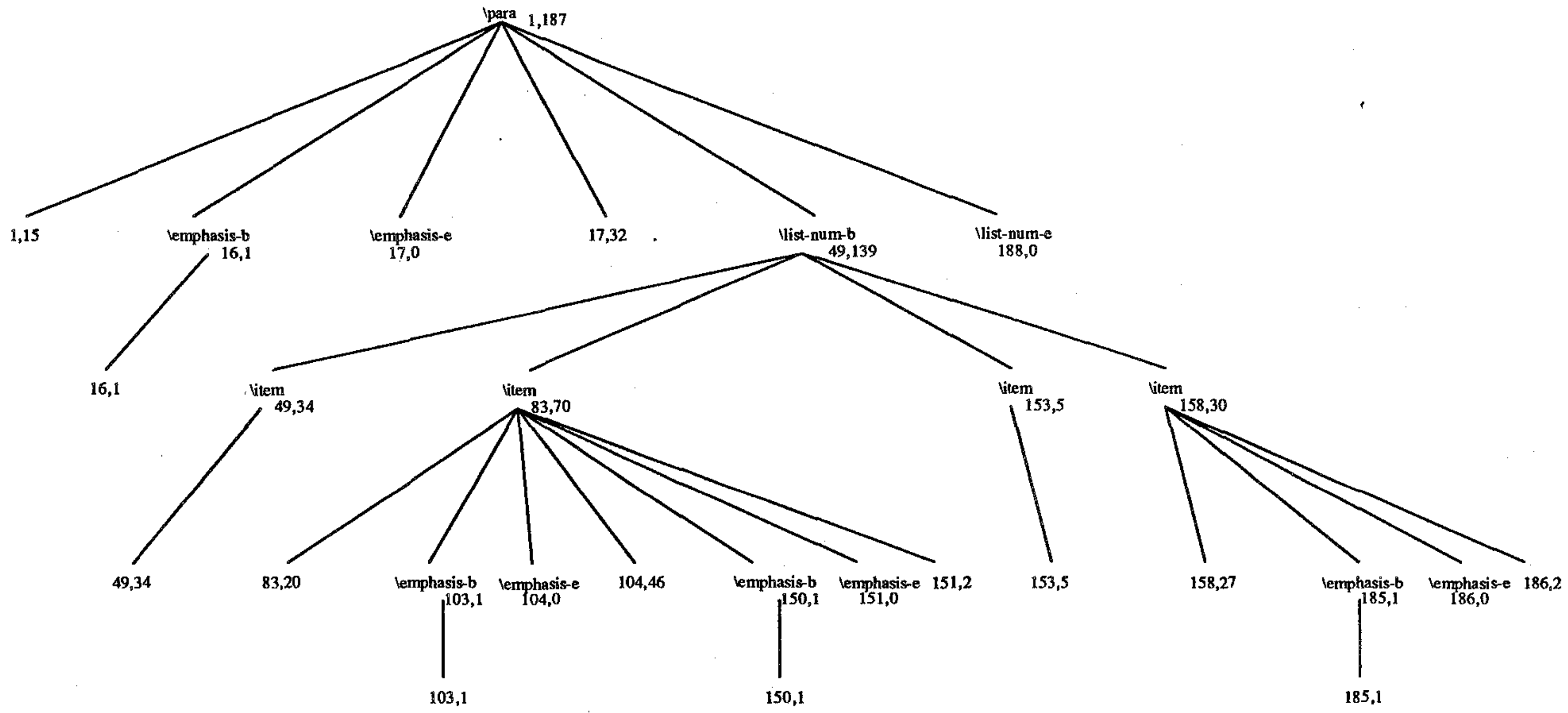


Figure 5
Parse of a paragraph with search information added.

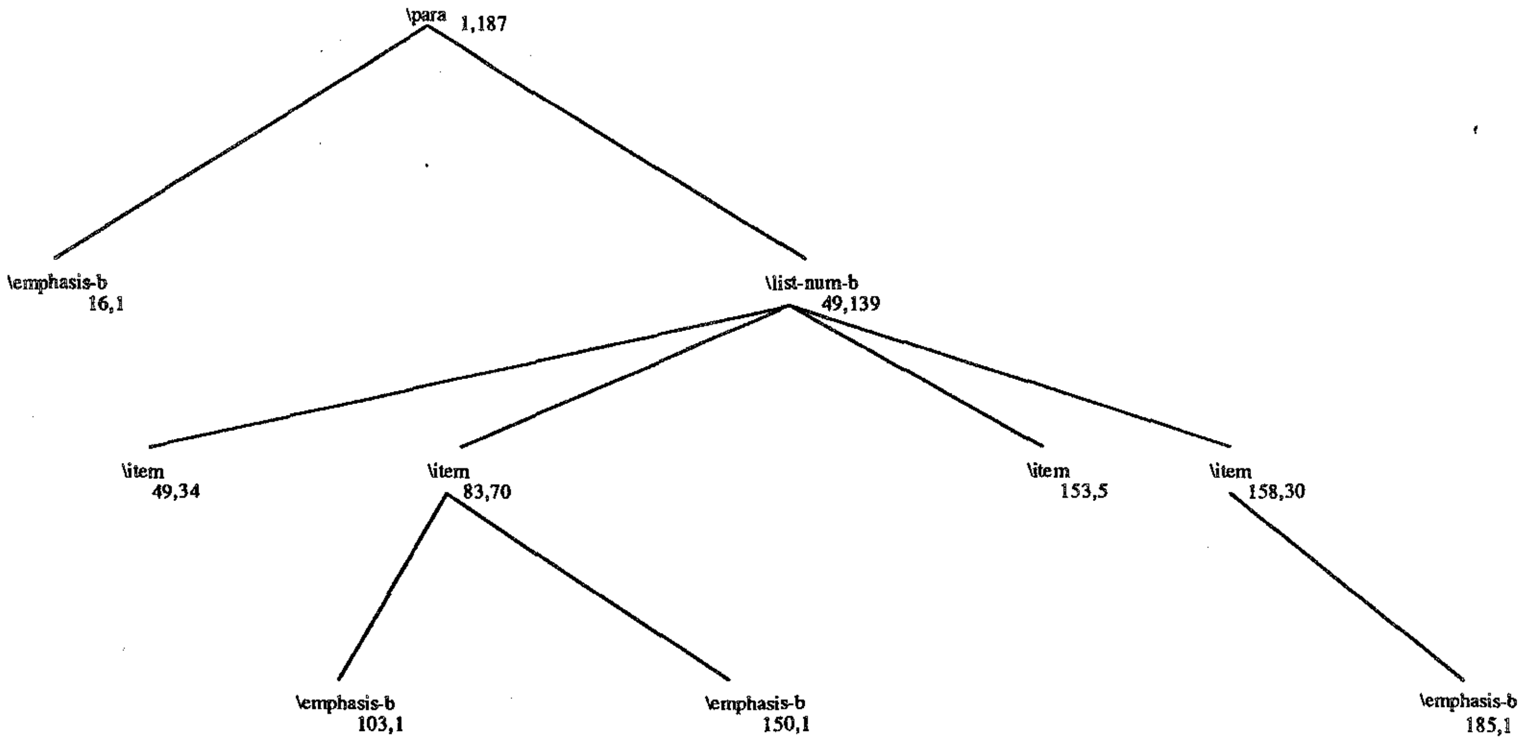


Figure 6
Parse tree as it is actually stored.

Search Algorithm

Recovering the format environment for a particular token in the text requires a top to bottom scan of the tree. In describing the algorithm, we use the term *current node* to refer to the node currently being examined; the *current operation* is the format operation associated with the current node.

Assume that we are searching for the format environment for the token at position p in the text. Initially, the current node is the root of the search tree. We search by repeating steps 1, 2, and 3 until the algorithm halts.

1. If the current node has no children, halt. If the current node has children, then descend to its eldest child. [This child now becomes the current node.]
2. Scan the current node and its siblings, left to right, looking for an operation whose domain includes position p. If such a node is found, make it the current node; if not, stop. [Operations have been defined so that domains can be nested but cannot overlap; therefore the domain of at most one of the siblings will contain p.]
3. Output the current operation.
4. Repeat the process. [When the algorithm stops, the string of operations sent to the output comprise the format environment for the word at position p.]

If, for example, we search Figure 6 for the formatting environment of the token at position 103, we would visit six nodes of the tree: the root; the two nodes at the next level; the first two children of \list-num-b; and the first child of the second item. The operations encountered along the direct path from the root to token 103 comprise the formatting operations that apply to this token. Specifically, the token at position 103 is emphasized within the second item of a numbered list within a paragraph.

The Parse Algorithm

The parsing algorithm constructs the appropriate parse/search tree from a well-formed sequential string of tokens (words and format operators). For non-well-formed strings, it provides appropriate diagnostics. This process differs from the general context-free parsing problem in that the input string contains both terminals (words) and nonterminals (format operators) rather than just terminals. The input is in essence the linearized preorder traversal of the parse tree; the parser's job is, thus, to reconstruct that tree.

We will give only general information about the parser here. Essentially, the parser is a deterministic push down automaton (PDA) with two states, p and q. The machine's stack holds a set of goals that must be satisfied in order for the text to be well-formed. As the text is scanned, each token satisfies the current goal and/or causes new subgoals to be added to the stack.

The PDA starts in state p and puts the initial goal on the stack (root). The machine then enters state q where it stays for the remainder of the parse. The operation of the parser is governed by an $n \times n$ table, where n is the number of symbols (terminals plus non-terminals) in the grammar. At any point in its operation, the parser is looking at a token t from the text and has a goal symbol s at the top of the stack. From the way the grammar is constructed we know that the table contains exactly one row for t and at most one column for s. The intersection of row t and column s contains instructions as to what to do next. These instructions include:

1. An indication as to whether t can legally satisfy (or partially satisfy) s. If not, issue an error message and stop or perform one of several fix-ups and continue. The possible fix-ups include adding a token that was expected but not found, deleting a token that was found but not expected, or a combination of the two. For example, we could convert an unexpected token into one that was expected. The most common errors can each be given their own entry in the table. Thus we can perform at each point the particular fix-up that is most appropriate to the error encountered. Less common errors produce a general purpose error message.

2. A stack instruction: remove the symbols $s_1, s_2, \ldots, s_i, i \geq 0$, from the stack and replace them with the (possibly empty) set of symbols $s'_1, s'_2, \ldots, s'_j$.

3. Instructions for constructing the parse/search tree. These consist of traversal instructions (e.g. create a new child node and descend to it; create a new sibling node and go to it; or ascend to the parent node) as well as instructions for setting the start pointer and length field.

4. After the instructions have been executed, move to the next token and repeat the process.

If we reach the end of the input text and at the same time satisfy all goals (the stack is empty), then we have a successful parse. If either the input or the stack run out before the other, then the input is not well-formed. Since there is at most one entry for each token/goal pair, parsing can be done in a single left to right pass of the input with parse time proportional to the input length.

Data Structures

The data structure for a node of the tree, shown in Figure 7, contains five fields: the format operation, two integer fields that define the domain of that operation: its starting point in the sequential text file and its length; and two pointers that define the tree structure: one to the node's leftmost child and one to its sibling on the immediate right. (Pointers are stacked to permit reverse traversal of the tree from any given point.) This data structure permits very efficient implementation of the operations necessary for the search: descend to the left-most child and traverse through siblings, left and right. The data structure also minimizes the number of pointers necessary to implement this n-ary tree.

operation			
domain		pointer	
start	length	oldest child	next sibling

Figure 7:

Tree Node

Since the parser table is quite sparce, it is implemented as a case statement. Each token/goal pair is a separate case. This structure can be easily modified and expanded to incorporate new format operations, and it facilitates a simple default error message for token/goal pairs that are not in the table.

Conclusion

To get a sense of the efficiency of this approach, consider the format tree for this document. It contains four hundred forty-nine nodes, approximately one node for each operation used to format the paper. However, it is not really representative of most texts since three hundred eighty-one nodes come from just two paragraphs containing long lists. The remainder of the document, more typical of coventional texts, requires only eighty-eight nodes. The maximum path length from the root of the tree to a leaf is nine. And while the number of children of a node is as high as thirty-two in one case (in the list of thirty-two productions), the average number of children per interior node is 2.5. This means that recovering the format environment for a particular word requires, on the average, looking at fewer than twenty nodes. This represents at least two orders of magnitude improvement over a sequential scan of the text.

This paper is roughly equivalent to one chapter of a book. We expect that for each order of magnitude increase in text size (for example, from chapter to book, and from book to collection of books), the format tree will increase by one level. Since the points of major complexity in the tree tend to be at the lowest levels (for example, in a paragraph containing a complex list), we expect the average number of children per node to remain about the same even for large text collections. Hence, the format tree will allow extremely fast recovery of the format environment for any point, even in a very large collection of texts.

References

American Chemical Society (1981).*User's Guide: American Chemical Society Experimental Full-text Primary Journal Database*. Columbus, Ohio: American Chemical Society.

Futura, R., Scofield, J., & Shaw, A. (1982). Document Formatting Systems. *Computing Surveys* 14(3), pp. 417–72.

IBM (1980). *Document Composition Facility Generalized Markup Language: Starter Set Reference*. Tucson, Arizona: IBM Corporation, General Products Division, #SH20–9187–0.

Microsoft Corporation (1985).*Microsoft Word*. Bellevue, WA: Microsoft Corporation.

Menanteaux, A.R. (1982). A User's Companion to Westlow and Lexis. *Legal Reference Services Quarterly* 2(2), pp. 19–23.

Morrissey, R. & Del Vigna, C. (1983). A Large Natural Language Date Base: American and French Research on the Treasury of the French Language. *Educom* 18(1), pp. 10–13.

New York Times (1981). *The Information Bank II: BRS/SEARCH Protocol User Guide*. New York: The New York Times.

Reid, B.K. (1980). *Scribe: A Document Specification Language and Its Compiler*. Pittsburg, PA: Carnegie-Mellon University Tech. Rep. CMU–CS–81–100.

Appendix A

We show in this appendix that the modified context-free rules used in the format grammar are in fact context-free by showing for each modified production a corresponding set of context-free productions that has precisely the same effect.

1. $A \longrightarrow x[y]z$ is equivalent to $A \longrightarrow xyz$

   $A \longrightarrow xz$

2. $A \longrightarrow x + y$ is equivalent to $A \longrightarrow x$

   $A \longrightarrow y$

3. $A \longrightarrow x^*$ is equivalent to $A \longrightarrow e$

   $A \longrightarrow xA$

4. $A \longrightarrow x^{*2}$ is equivalent to $A \longrightarrow e$

   $A \longrightarrow xx$

   $A \longrightarrow xxB$

   $B \longrightarrow xB$

   $B \longrightarrow x$

5. $A \longrightarrow x^+$ is equivalent to $A \longrightarrow x$

   $A \longrightarrow xA$

6. $A \longrightarrow x^{+2}$ is equivalent to $A \longrightarrow xx$

   $A \longrightarrow xxB$

   $B \longrightarrow x$

   $B \longrightarrow xB$

Appendix B

```
  1 Assume
    that
    we
    are
    searching
    for
    the
    format
    environment
 10 for
    the
    token
    at
    position
    p
    in
    the
    text
    .
 20 Initially
    ,
    the
    current
    node
    is
    the
    root
    of
    the
 30 search
    tree
    .
    We
    search
    by
    repeating
    steps
    1
    ,
 40 2
    ,
    and
    3
    until
    the
    algorithm
    halts
    .
    If
 50 the
    current
    node
    has
    no
    children
    ,
    halt
    .
    If
 60 the
    current
    node
    has
    children
    ,
    then
    descend
    to
    its
 70 eldest
    child
    .
    [
    This
    child
    now
    becomes
    the
    current
 80 node
    .
    ]
    Scan
    the
    current
    node
    and
    its
    siblings
 90 ,
    left
    to
    right
    ,
    looking
    for
    an
    operation
    whose
100 domain
    includes
    position
    p
    .
    If
    such
    a
    node
    is
110 found
    ,
    make
    it
    the
    current
    node
    ;
    if
    not
120 ,
    stop
    .
    [
    Operations
    have
    been
    defined
    so
    that
130 domains
    can
    be
    nested
    but
    cannot
    overlap
    ;
    therefore
    the
140 domain
    of
    at
    most
    one
    of
    the
    siblings
    will
    contain
150 ,
    .
    ]
    Output
    the
    current
    operation
    .
    Repeat
    the
160 process
    .
    [
    When
    the
    algorithm
    stops
    ,
    the
    string
170 of
    operations
    sent
    to
    the
    output
    comprise
    the
    format
    environment
180 for
    the
    word
    at
    position
    p
    .
    ]
```