A Generalized Approach to Document Markup

Dr. C. F. Goldfarb

IBM Corporation
San Jose, California, U.S.A

The Markup Process

Text processing and word processing systems typically require users to intersperse additional information in the natural text of the document being processed. This added information, called "markup," serves two purposes:

1. it separates the logical elements of the document; and

2. it specifies the processing functions to be performed on those elements.

Consider how the user of such a system marks up a document. There are three distinct steps, although he may not perceive them as such.

1. First, he analyzes the information structure and other attributes of the document; that is, he identifies each meaningful separate element, and characterizes it as a paragraph, heading, ordered list, footnote, or some other element type.

2. He then determines, from memory or a style book, the processing instructions ("controls") that will produce the format desired for that type of element.

3. Finally, he inserts the chosen controls into the text.

Here is how the start of this paper looks when marked up with controls in the Script formatting language (1):

```
.sk 1
Text processing and word
processing systems typically
require users to intersperse
additional information in
the natural text of the document
being processed.
This added information, called
'markup,' serves two purposes:
.tb 4
.of 4
.sk 1
1.¬it separates the logical
elements of the document; and
.of 4
.sk 1
2.¬it specifies the processing
functions to be
performed on those elements.
.of 0
.sk 1
```

The .SK, .TB, and .OF controls, respectively, cause the skipping of vertical space, the setting of a tab stop, and the offset, or "hanging indent," style of formatting. (The not sign (¬) in each list item represents a tab code, which would otherwise not be visible.)

Procedural markup like this has a number of disadvantages. For one thing, information about the document's attributes is usually lost. If the user, for example, decides to center both headings and figure captions when formatting, the "center" control will not indicate whether the text on which it operates is a heading or a caption. Therefore, if he wishes to use the document in an information retrieval application, search programs will be unable to distinguish headings—which might be very significant in information content—from the text of anything else that was centered.

Procedural markup is also inflexible. If the user decides to change the style of his document (perhaps because he is using a different output device), he will need to repeat the markup process to reflect the changes. This will prevent him, for example, from producing double-spaced draft copies on an inexpensive computer line printer, while still obtaining a high quality finished copy on an expensive photocomposer. And if he wishes to accept competitive bids for the typesetting of his document, he will be restricted to those vendors that use the identical text processing system, unless he is willing to pay the cost of repeating the markup process.

Moreover, markup with control words can be time-consuming, error-prone, and require a high degree of operator training, particularly when complex typographic results are desired. This is true (albeit less so) even when a system allows defined control procedures ("macros"), since these must be added to the user's vocabulary of primitive controls. The elegant and powerful TEX system (2), for example, which the American Mathematical Society is considering for standard use by its authors, includes some 300 primitive controls and macros in its basic implementation.

These disadvantages of procedural markup are avoided by a markup schema due to C. F. Goldfarb, E. J. Mosher, and R. A. Lorie (3, 4). It is called the "Generalized Markup Language" (GML) because it does not restrict documents to a single application, formatting style, or processing system. GML is based on two novel postulates:

1. Markup should describe a document's structure and other attributes, rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.

2. Markup should be rigorous, so the techniques available for processing rigorously-defined objects like programs and data bases can be used for processing documents as well.

These postulates will be developed intuitively by examining the properties of GML.

Descriptive Markup

With GML, the markup process stops at the first step: the user locates each significant element of the document and marks it with the mnemonic tag ("generic identifier") that he feels best characterizes it. The processing system associates the markup with processing instructions in a manner that will be described shortly.

Marked up in GML, the start of this paper looks like this:

```
:p.
Text processing and word
processing systems typically
require users to intersperse
additional information in
the natural text of the document
being processed.
This added information, called
:q.markup::q., serves two purposes:
:ol.
:li.it separates the logical
elements of the document; and
:li.it specifies the processing
functions to be
performed on those elements.
::ol.
```

Each generic identifier (GI) is delimited by a colon (:) if it is at the start of an element, or by two colons (::) if it is at the end. A period (.) separates a GI from any text that follows it. The mnemonics P, Q, OL, and LI stand, respectively, for the element types paragraph, quotation, ordered list, and list item.

This example has some interesting properties. Note that there are no quotation marks in the text; the processing for the quotation element generates them, and will distinguish between opening and closing quotes if the output device permits. The comma that follows the quotation element is not actually part of it, but is brought inside the quotation marks during formatting. Similarly, sequence numbers are generated for the ordered list items. The source text, in other words, contains only information; characters whose sole role is to enhance the presentation are generated during processing.

If, as postulated, descriptive markup like this suffices for all processing, it must follow that the processing of a document is a function of its attributes. The way text is composed offers intuitive support for this premise. For example, techniques like beginning chapters on a new page, italicizing emphasized phrases, and indenting lists, are employed to assist the reader's comprehension by emphasizing the structural attributes of the document and its elements.

From this we can construct a 3-step model of document processing:

1. Recognition: An attribute of the document is recognized; for example, an element with a generic identifier of "footnote."

2. Mapping: The attribute is associated with a processing function. The footnote GI, for example, could be associated with a procedure that prints footnotes at the bottom of the page, or one that collects them at the end of the chapter.

3. Processing: The chosen processing function is executed.

Text formatting programs conform to this model. They recognize such elements as words and sentences, primarily by interpreting spaces and punctuation as implicit markup. Mapping is usually via a branch table. Processing for words typically involves determining the word's width and testing for an overdrawn line, while processing for sentences might cause space to be inserted between them.[1]

In the case of low-level elements such as words and sentences, the user is normally given little control over the processing, and almost none over the recognition. Some formatters offer more flexibility with respect to higher-level elements like paragraphs, while those with powerful macro languages can go as far as to support descriptive markup. In terms of the document processing model, the advantage of descriptive markup is that it permits the user to define attributes—and therefore element types—not known to the formatter, and to specify the processing for them.

For example, the GML markup sample just described includes the element types "ordered list" and "list item," in addition to the more common "paragraph." Built-in recognition and processing of such elements is unlikely. Instead, each will be recognized by its explicit markup, and mapped to a control procedure associated with it for the particular processing run. Both the control procedure itself, and the association with a GI, would be expressed in the system's macro language. On other processing runs, or at different times in the same run, the association could be changed. The list items, for example, might be numbered in the body of a book, but lettered in an appendix.

So far the discussion has addressed only a single attribute, the generic identifier, whose value characterizes an element's semantic role or purpose. Some descriptive markup schemes refer to markup as "generic coding," because the GI is the only attribute they recognize (5). In generic coding schemes, recognition, mapping, and processing can be accomplished all at once by the simple device of using GIs as control procedure names. Different formats can then be obtained from the same markup by invoking a different set of homonymous procedures. This approach is effective enough that one notable implementation, the SCRIBE system, is able to prohibit procedural markup completely (9).

Generic coding is a considerable improvement over procedural markup in practical use, but it is conceptually unsound. This is because documents are complex objects, and they have other attributes that a markup language must be capable of describing. For example, suppose the user decides that his document is to include elements of a type called "figure," and that it must be possible to refer to individual figures by name. The markup for a particular figure element known as "angelfig" could begin like this:

```
:fig id=angelfig
```

"Fig," of course, stands for "figure," and is the value of the generic identifier attribute. The GI identifies the element as a member of a set of elements having the same role. In contrast, the "unique identifier" (ID) attribute distinguishes the element from all others, even those with the same GI. (It was unnecessary to say "GI=fig," as was done for ID, because in GML it is understood that the first piece of markup for an element is the value of its GI).

The GI and ID attributes are termed "primary" because every element can have them. There are also "secondary" attributes that are possessed only by certain element types. For example, if the user wanted some of the figures in his document to contain illustrations to be produced by an artist and added to the processed output, he could define an element type of "artwork." Because the size of the externally-generated artwork would be important, he might define artwork elements to have a secondary attribute, "depth."[2] This would result in the following markup for a piece of artwork 24 picas deep:

```
:artwork depth=24p
```

The markup for a figure would also have to describe its content. "Content" is, of course, a primary attribute, and is the one that the secondary attributes of an element describe. The content consists of an arrangement of other elements, each of which in turn may have other elements in its content, and so on until further division is impossible.[3] One way in which GML differs from generic coding schemes is in the conceptual and notational tools it provides for dealing with this hierarchical structure. These

[1] The model need not be reflected in the program architecture; processing of words, for example, could be built into the main recognition loop to improve performance.

[2] "Depth=" is not simply the equivalent of a vertical space control word. Although a full-page composition program could produce the actual space, a galley formatter might print a message instructing the layout artist to leave it. A retrieval program might simply index the figure and ignore the depth entirely.

[3] This explains why we can speak of documents and elements almost interchangeably: the document is simply the element that is at the top of the hierarchy for a given processing run. A technical report, for example, could be formatted both as a document its own right, or as an element of a journal.

are based on the second GML hypothesis, that markup can be rigorous.

Rigorous Markup

Assume that the content of the figure "angelfig" consists of two explicitly marked up elements ("explicit elements"), a figure body and a figure caption. The figure body in turn contains an artwork element, while the content of the caption is text with no explicit markup (an "implicit element"). The markup for this figure could look like this:[4]

```
:fig id=angelfig
:figbody
:artwork depth=24p
::artwork
::figbody
:figcap.Three Angels Dancing
::figcap
::fig
```

The markup rigorously expresses the hierarchy by identifying the beginning and end of each element in classical left list order. No additional information is needed to interpret the structure, and it would be possible to implement support by the simple scheme of macro invocation discussed earlier. The price the user pays for this simplicity, though, is that he must enter explicit markup for the end of every explicit element.

This price is totally unacceptable in practice. The user knows that the start of a paragraph, for example, terminates the previous one, so he will be reluctant to go to the trouble and expense of entering an explicit ending tag for every single paragraph just to share his knowledge with the system. He will have equally strong feelings about other element types he may define for himself, if they occur with any great frequency.

With GML markup, though, it is possible to advise the system about the attributes of any type of element the user creates. This is done by creating a formal definition, or "model," using a notation derived from the BNF notation used for formal grammars. While the markup in a document consists of descriptions of individual elements, a GML model defines the set of all possible valid descriptions of a type of element.

As an example, suppose the user extends his definition of "figure " to permit the figure body to contain certain kinds of textual elements instead of artwork. The GML model might look like this:[5]

```
1.  :fig id=? &def. figbody, figcap?

2.  :figbody &def. artwork / (p / ol / ul)*

3.  :artwork depth= &def. EXTERNAL

4.  :figcap &def. word, (PUNCTUATION, word)*?

5.  word &def. CHARACTER*
```

Line 1 means that a figure may optionally have the ID attribute specified, and that it can contain a figure body and, optionally, a figure caption following the figure body. Line 2 says the body may contain either artwork or an intermixed collection of paragraphs, ordered lists, and unordered lists. Line 3 defines artwork as requiring specification of the depth attribute, and having an externally generated content. Line 4 defines a figure caption's content as 1 or more words, separated by punctuation. Line 5 defines a word as containing 1 or more characters. EXTERNAL, PUNCTUATION, and CHARACTER are treated as terminals, incapable of further division. (It is assumed that P, OL, and UL have been defined elsewhere.)[6]

With this formal definition of figure elements available, the following markup for "angelfig" is now acceptable:

```
:fig id=angelfig
:figbody
:artwork depth=24p
:figcap.Three Angels Dancing
::fig
```

There has been a 40% reduction in markup, since three ending generic identifiers are no longer needed. As the model defines the figure caption as part of the contents of a figure, terminating the figure automatically terminated the caption. Since the figure caption itself is on the same level as the figure body, the :figcap tag implicitly terminated the figure body, and therefore the artwork element contained in it.

GML models have uses in addition to markup minimization. They can be used to validate the markup in a document before going to the expense of processing it, or to

[4] Like "GI=," "content=" can safely be omitted. It is unnecessary when the content is externally generated, it is understood when the content consists solely of explicit elements, and for implicit elements it is implied by the period that separates markup at the start of an element from the following text.

[5] The question mark (?) means an element is optional, the comma (,) that it follows the preceding element in sequence, and the asterisk (*) that the element can occur 1 or more additional times. The stroke (/) is used to separate alternatives. Parentheses are used for grouping as in mathematics.

[6] Some complete, practical definitions may be found in (4).

drive prompting dialogues for users unfamiliar with a document type. For example, a document entry application could read the model of a figure element and invoke control procedures for each element type. The control procedures would issue messages to the terminal prompting the user to enter the figure ID, the depth of the artwork, and the text of the caption. They would also enter the markup itself into the document being created.

Even in systems not sophisticated enough to read a model directly, knowledge of the model can be built into the control procedures. For example, in the case of our figure elements, the figure caption control procedure would terminate the figure body processing, and set a switch. If the end-of-figure procedure found the switch not set it would recognize that this instance of a figure had no caption, and would invoke the end-of-body procedure itself. The result is as if the system had read the model, although at the penalty of increasing the complexity of the control procedures.

Conclusion

Regardless of the degree of accuracy and flexibility in document description that GML makes possible, the concern of the user who prepares documents for publication is still this: can GML, or any descriptive markup scheme, achieve typographic results comparable to procedural markup? A recent publication by Prentice-Hall International (6) represents empirical corroboration of the GML hypotheses in the context of this demanding practical question.

It is a textbook on software development containing hundreds of formulas in a symbolic notation devised by the author. Despite the typographic complexity of the material (many lines, for example, had a dozen or more font changes), no procedural markup was needed anywhere in the text of the book. It was marked up using the standard GML supplied with a program called the "Document Composition Facility " (DCF) (4). To this the author added some element types required for textbooks (such as "exercise"), and mnemonic names for "constant" elements like mathematical and logical symbols.

The available control procedures supported only computer output devices, which were adequate for the book's preliminary versions that were used as class notes. No consideration was given to typesetting until the book was accepted for publication, at which point its author balked at the time and effort required to re-keyboard and proofread some 350 complex pages. He began searching for an alternative at the same time the author of this paper sought an experimental subject to validate the applicability of GML to commercial publishing.

In due course both searches were successful, and an unusual project was begun. As the GML processor, DCF, does not support photocomposers directly, procedures were written that created a source file with procedural markup for a separate typographic composition program, TERMTEXT/Format (7). Formatting specifications were provided by the publisher, and no concessions were needed to accommodate the use of GML, despite the markup having existed before the specifications.[7]

The experiment was completed on time, and the publisher considers it a complete success (8).[8] The control procedures, with some modification to the formatting style, have found additional use in the production of a variety of in-house publications.

GML, then, has both practical and academic benefits. In the publishing environment, it reduces the cost of markup, cuts lead times in book production, and offers maximum flexibility from the text data base. At the same time, its rigorous descriptive markup makes text more accessible for computer analysis. While procedural markup (or no markup at all) leaves a document as a character string that has no form other than that which can be deduced from analysis of the document's meaning, GML markup reduces a document to a regular expression in a known grammar. This permits established techniques of computational linguistics and compiler design to be applied to natural language processing and other document processing applications.

Acknowledgments

The author is indebted to E. J. Mosher, R. A. Lorie, T. I. Peterson, and A. J. Symonds—his colleagues during the early development of GML—for their many contributions to the ideas presented in this paper, to N. R. Eisenberg for his collaboration in the design and development of the control procedures used to validate the applicability of GML to commercial publishing, and to C. B. Jones and Ron Decent for risking their favorite book on some new ideas.

References

1 **Document Composition Facility: User's Guide**, Form No. SH20-9161-0, IBM Corporation, White Plains, 1978.
2 Donald E. Knuth, **TAU EPSILON CHI, a system for technical text**, American Mathematical Society, Providence, 1979.

[7] On the contrary, the publisher took advantage of GML by changing some of the specifications after he saw the page proofs.
[8] This despite some geographical complications: the publisher was in London, the book's author in Brussels, and this paper's author in California. Almost all communication was done via an international computer network, and the project was nearly completed before all the participants met for the first time.

3 C. F. Goldfarb, E. J. Mosher, and T. I. Peterson, "An Online System for Integrated Text Processing," **Proceedings of the American Society for Information Science**, 7, 147-150 (1970).

4 Charles F. Goldfarb, **Document Composition Facility Generalized Markup Language: Concepts and Design Guide**, Form No. SH20-9188-0, IBM Corporation, White Plains, 1980.

5 Charles Lightfoot, **Generic Textual Element Identification—A Primer**, Graphic Communications Computer Association, Arlington, 1979.

6 C. B. Jones, **Software Development: A Rigorous Approach**, Prentice-Hall International, London, 1980.

7 **TERMTEXT/Format Language Guide**, Form No. SH20-1372-1, IBM Corporation, White Plains, 1976.

8 Ron Decent, **personal communication to the author** (September 7, 1979).

9 B. K. Reid, "The Scribe Document Specification Language and its Compiler," **Proceedings of the International Conference on Research and Trends in Document Preparation Systems**, 59-62 (1981).