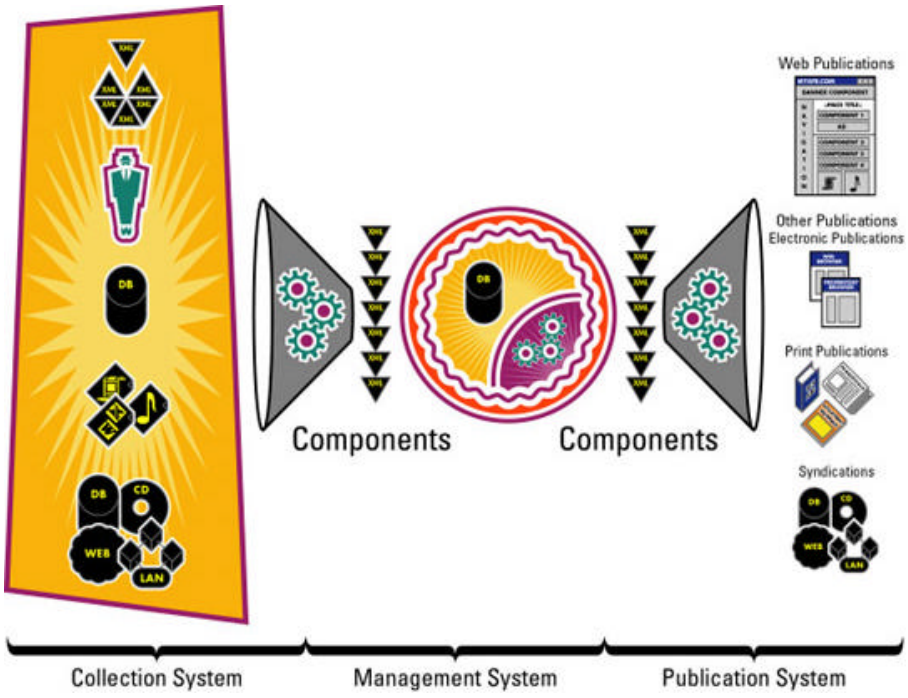
What Are Content Markup Languages?

A CM Domain White Paper

By Bob Boiko



This white paper is produced from the Content Management Domain which features the full text of the book "Content Management Bible," by Bob Boiko. Owners of the book may access the CM Domain at www.metatorial.com.

Table of Contents

Almost all text that is authored and most that you want to acquire is provided in some sort of markup language (ML). A markup language wraps the content with formatting and structural codes. To understand markup languages is to understand how format and structure can be represented in text. (It also goes a long way toward helping you understand how these qualities are represented in other media.)

In this white paper I illustrate the concepts of markup languages, drawing primarily from the most familiar markup language, HTML. I also present examples from XML and the Microsoft markup language RTF to give you a full picture of what markup languages are and what they do.

A Brief and Selective History of Markup Languages

Most Web-savvy folks know that *HTML* stands for *H* yper *T* ext *M* arkup Language. Fewer know that *XML* stands for e *X* tensible *M* arkup *L* anguage. Few have anything more than a rudimentary understanding of what the markup language part of these abbreviations means. Markup languages have been around about as long as computers have. The word *markup* is an old editorial term. Editors would "mark up" manuscripts to show what revisions needed to happen and how they should be laid out for printing.

In the computer world, you can easily trace the development of markup languages and get the basic idea of what markup languages are by looking at the development of word processors. At first, you had no word processors - only text editors. (Does anyone remember Edlin or VI?) These programs enabled you to open a text file, type or delete characters (and sometimes even backspace over them), and then save the file again. That was about it. You got no formatting -

just letters and punctuation, kind of like a typewriter. After word processors were invented, they added a whole slew of new possibilities, and simple text files became documents.

Word processors added bold, italic, underline, strikethrough, and any other formatting that the early printers could print. To represent these new features, developers of word processors invented special codes that you could type before and after the words you wanted to affect. WordStar, my first PC-based word processor, used what they called "dot commands." To make a word bold, you'd literally type **^B** before and after it. The first ^B turned on the bold, and the second turned it off. A document of any complexity would have a bevy of strange codes surrounding text. The set of dot commands constitutes a markup language - you mark up your document with formatting commands to tell the printer how to print your document.

The next and latest turn in the march of word processing was WYSIWYG (what you see is what you get). Thankfully, for most writers, dot commands disappeared. Rather than typing special codes, you'd use formatting commands to show on your computer screen what your document would look like printed, more or less. Actually, the dots didn't go away; they simply went underground. Behind the screen, text is still surrounded by special codes, but the software was now advanced enough to make formatting happen on-screen.

Markup languages got more elaborate as the types and quantities of formatting you could do in a document increased steeply. Today, word processors have a vast array of codes that mark up all the advanced features you've come to expect in your documents.

Underground markup is an advantage for word processor developers - as well as writers - because it frees them from needing to make their markup bulky and readable. If it doesn't need to be understandable to the human, you can squeeze a lot of markup information into a few bytes of data. These binary-based markup languages lie behind all of today's top word processors.

In another corner of the computing world, people were getting fed up trying to use word processors to work with very large documents and document sets. The problem was that word processors were designed to enable you to format a fancy document but did extremely little to enable you to organize and access your information. The stuff that comes out of word processors isn't easily read by programs (other than the word processor that created it) and almost impossible to do anything with besides print.

Companies with large catalogs or complicated document-production systems cared less that they could see what the printer would produce and more that each element within their documents conformed to a set of content rules that any well-written program could read and interpret. It was the structure, not the format, of the information they were after.

Some gave up on word processors and turned to databases instead. Others invented the *S*tandard *G*eneralized *M*arkup *L*anguage (*SGML*). SGML was an attempt to create a markup language that could be used anywhere by anyone to create and enforce document-structure rules. SGML could create elaborate schemas that, if imposed upon a document, could turn it into well structured data. SGML was universally hailed (among the very few who cared about markup languages) as *the* big advance in text-management technology. People believed (and many still do) that if everyone would just accept SGML, communication between text applications would become easy, and all computer-generated text would become accessible and capable of manipulation by computer programs - just like other data.

Unfortunately, the strength of SGML was also its weakness. Ever since driving markup underground, makers of word processors knew that people would choose ease-of-use over consistency and organization. Even the limited attempts that they made to enable you to really organize documents (heading levels, paragraph styles, auto-TOCs, and so on) were infrequently used and rarely used correctly. Any attempt that developers made to enforce structure was shot down in the marketplace by lazy or naive users. People voted vehemently with their wallets against complexity, constraint, and rigidity. Thus, although SGML held the key to truly organizing information, it was consigned to industries where it could be made obligatory (the military, for

example) or where the problems of text management were too complex for any other approach (huge parts catalogs and the like).

Enter the Web. Imagine the surprise for those of us who'd long since gotten used to markup languages being about as boring, arcane, and esoteric as you could get when a markup language became an overnight sensation. HTML, which shares much more with dot commands than with modern invisible markup languages, became chic and stylish. Whereas before it would be hard to find someone who'd admit to knowing a markup language, now that was a symbol of status. People actually sought out and treasured knowledge of HTML. It caught on like wildfire and fueled the blazing growth of the World Wide Web.

HTML could do this, despite the fact that it was a derivative of SGML, because it was easy to learn, very permissive, and concerned with formatting not structure. To the chagrin of the SGML community, SGML was used to create its very antithesis. SGML became like those backwoods blues players of old to whom the pop stars give honor but no money. SGML took considered study, careful analysis, and a set of very expensive, hard-to-use tools. HTML took a $15.99 book and a small amount of free software. For a while, it looked as if HTML might actually become the universal markup language that SGML always dreamed of being.

Unfortunately, the strength of HTML is also its weakness. Not at all unlike with word processing, people are becoming fed up trying to use HTML to work with very large Web sites. The problem is the same. HTML is designed to enable you to format a fancy Web page but does extremely little to enable you to organize and access your information. HTML isn't easily read by computer programs and is almost impossible to do anything with besides display in a browser.

Re-enter SGML, renamed XML. The two are so closely related that you'd be hard pressed to say what really is the difference. I always think of XML as SGML, version 2. The difference between the two isn't how they look, how they work, or the advantages that they offer you. The difference is the world into which they were launched.

In the days of SGML, the vast majority of organizations produced single-purpose documents that were targeted to the printer. It was the rare organization that had a system complex enough to warrant the expense and effort involved in SGML. Today (the XML days), a large percentage of organizations need to produce multipurpose documents that can be printed as well as displayed on an already large and still growing Web site. Just as does SGML, XML enables you to organize information and make it easily accessible to computer programs. Just as does SGML, XML marks up structure, not format. Just as does SGML, XML gives you power over large bodies of content. As does SGML, XML requires a characteristically nonhuman adherence to constraint and complexity.

You should be expecting by now that the strength of XML is also its weakness. And for all its hype, XML is really just as hard to use and just as rigid as SGML was. I have yet to see how this is going to play out, but you can certainly suspect that the need to be structured and organized doesn't play well among the computer-user masses. On the other hand, as the world's organizations become more entangled and co-dependent, the need to have structure and organization is only going to increase. So can XML triumph where SGML failed or must people's desire for ease-of-use and freedom of expression doom XML as it did its predecessor?

I hope that, now that you know the story of markup languages, you can help everyone transcend that history rather than simply repeat it. In your CMS, can you combine ease of use for staff and audiences with the rigorous structure that you need to manage your content?

What Is a Markup Language?

Most simply, a *markup language* is a set of codes or tags that surrounds content and tells a person or program what that content is (its structure) and/or what it should look like (its format). I use the word content here rather than text. The history of markup languages lies in the text world. Its future lies in any kind of content (text, images, sound, and motion) that needs to be organized

and accessed. Precious little has yet been done to mark up nontext media, but the industry is still young. So, although it's most convenient to use text examples of markup languages, please understand that the same ideas apply to other types of content.

Markup tags must have a distinct syntax that sets them apart from the content that they surround. The tag syntax is the rules of tag formation that any person or tool must follow for their markup codes to be recognized. Most people who create Web sites are familiar with the tag syntax of HTML, where a tag begins with an opening angle bracket, followed by the tag name, followed by a closing bracket. Luckily for these Web types, XML and SGML follow exactly the same rules.

This set of tags forms a metalanguage that adds context to the content that it surrounds. Context is a funny thing; it can completely change the meaning of the thing it acts upon. Consider the following example:

```
The Sky Is Falling
```

Now compare it to the following:

```
<LIE> The Sky Is Falling</LIE>
```

Tags are rarely used to negate the meaning of the content that they enclose. More commonly, they add nuance or interpretation to the content. Consider the following:

```
The Sky Is Falling
```

And compare it to this example:

```
<H1>The Sky Is Falling</H1>
```

In the first line, you have a statement. In the second line, that statement is turned into a first-level heading (as denoted by the <H1> tag in HTML). This means that it's an important statement and, furthermore, that it can stand for all the content that follows it until the next <H1>. The tag puts the context of headings around the content that it surrounds. It adds nuance and interpretation to the statement.

So markup languages are metalanguages. Truth be told, some are metalanguages and others are "meta-metalanguages." HTML is a metalanguage. It has a fixed number of tags, the meaning of which is commonly known. It works just as I've said previously. XML is a meta-metalanguage. It's a system for creating metalanguages. As you soon see, you have no particular XML tags. You just have a way to define other metalanguages, such as HTML.

A Taxonomy of Markup Languages

All markup languages add codes to text to mark format and structure distinctions in the text. Each markup language, however, does the same task a bit differently. In addition, markup languages differ in other aspects of their approach, as follows:

- Some use plain text and others use proprietary binary code as tags.
- Some use tags to represent structure. Others use tags to represent formatting only. Still others represent both format and structure.
- Some have a fixed tag set while others enable you to create your own tags.
- Some have a very limited range of structure or format constructs that you can represent while others offer wider coverage.

Table 28-1 compares HTML, XML and your average word processing markup language on the preceding aspects of markup languages.

Table 28-1 Comparing HTML, XML, and Word Processor Markups

	HTML	XML	Word Processing Markup
ASCII vs. Binary	ASCII	ASCII	Binary
Format vs. Structure	Format	Structure	Format and Structure
Extendable vs. Nonextendable	Nonextendable	Extendable	Nonextendable
Range of coverage	Low	High	Medium

ASCII vs. binary

Both HTML and XML are ASCII-based markup languages (strictly speaking, they're Unicode-based, but the point is the same). ASCII is that stuff you can read in a program such as Microsoft Notepad. In other words, both HTML and XML are constructed in such a way that they're readable without any special decoders. More important, they were designed to be viewed directly and understood by humans. In a word, they're verbose.

Word processors store their markup information in a binary format, unreadable to any program but the word processor itself. Why? Recall that, after word processing markup went underground, it became more densely packed. To see how much more dense, try the following experiment:

The content is the same in both versions. In the binary version, however, the markup information is packed into optimized small chunks. This approach makes the document more efficient (smaller) and also makes it faster to parse and display. On the other hand, it requires a tight linkage between the display software (the word processor) and the markup syntax. You could say that the markup is hard-wired into the display software.

The ASCII markup formats, on the other hand, assume an open, agnostic attitude toward display software. No proprietary knowledge is needed to parse and display them. That's why HTML and XML are termed "open standards," why they're so verbose, and why they don't load terribly quickly.

Format vs. structure

Format describes how content is intended to look as it's displayed. Qualities such as line spacing, kerning, font face, horizontal and vertical positioning, and indentation are all aspects of formatting.

Structure describes the purpose or meaning of content. Names such as Title, Heading 1, Sidebar, FAQ Question, and Price all say what a piece of content is for, but nothing about how it looks.

HTML is designed mostly for formatting, and XML is designed mostly for structure. I say *mostly* because you have structural tags in HTML, and you can use XML to specify formatting options. The overall intent of HTML, however, is to add tags to content that tell the Web browser how to display the content. The overall intent of XML is to add tags to content that specify the meaning or

use of the content. Formatting markup is needed to determine presentation. Structural markup is needed to determine use.

Now, neither type of markup is better or worse. They have different but highly intertwined purposes, and any good content system covers both. You may need the title to be bold and centered for the purposes of standard presentation. You may also need the title to be tagged as the title so that you can find it and present it in lists with other titles. The more you intend to manage content, the more important structural markup becomes.

Management is the process of collecting, categorizing, selecting, and modifying content. It becomes impossible to manage content if you have no way to specify what each piece of content "is." That's why content management has been so difficult to date on the Web. HTML doesn't give you enough clues about the function of the content that it marks up. For smaller or more static applications, you can often get away with format only (as any "Webmeister" can tell you). It's not that there's no structure to the content on these sites. Rather, it means that the structure of the content is learned and remembered by a person who can manually collect, categorize, select and modify content.

People usually think of formatting as serving some aesthetic purpose; however, formatting is there mostly to provide structural cues to the human eye. Text that's big and centered at the top of a page, for example, is very likely to be the title (a structural part of the content). Similarly, as content moves from being managed to being displayed, the structural tags need to be exchanged for format tags that the display software can understand. Your title may be tagged as a title, but as it's displayed, it must somehow give rise to the tags for *big* and *centered*.

Although HTML is basically for format and XML is basically for structure, the word processor markup is for both format and structure. Because the markup of word processors is proprietary, it must include anything that you may ever need. Thus, even though the major thrust of word processors is formatting related, as they've advanced, they've needed to include more structural components. In Microsoft Word, for example, you can create a style called "title" that enables you to locate the title and group it with other titles. Word can also tag a paragraph as a heading and then create a document outline; it can tag text as index entries, and it can include author and other metadata with a document. These are all structure tags. In Word, however, they do double duty by enabling you to apply formatting consistently to structural elements.

Extendable vs. nonextendable

An extendable markup language enables you to create new tags for special purposes. Here, XML is clearly distinguished from HTML and word processor markup. In HTML, as in a word processor, you could say it's *WTGIAYG* (What They Give is All You've Got). A finite number of tags must do for any task that you may have. In HTML, the World Wide Web Consortium (W3C) sets the standards for HTML tags and adds and removes tags with each version to try to cover the majority of needs that the committee's seen. In word processors, the developers look at what functionality needs to be offered in the next version, and then they invent tags to represent it and user interfaces for adding those tags to the content.

XML, remember, is a meta-metalanguage. It has no particular tags but rather gives you the capability to create whatever tags you may need. That's a benefit and a curse. With freedom comes responsibility. You have the freedom to design a custom tag set that perfectly describes your content. But you also have the responsibility to ensure the following:

- That the tags you create correctly and adequately describe your content.
- That all users and programs know what each tag is intended to do.
- That, if necessary, the tags can be translated to format tags as the content is displayed.
- That the tags are consistent with any emerging standards for XML tagging in your industry.

It's very easy in XML to devise an overly complex, incomprehensible, and tangled tag set. It's very hard to set up all human and computer systems around any tag set that you create.

In the nonextendable languages, you have a different problem. Anyone who's worked with HTML knows that you create a page by making tags designed for one purpose do something else (using table tags, for example, to line things up correctly). Your problem in an extendable markup language such as XML is how to create and manage your own tag set. Your problem in a nonextendable language such as HTML is how to serve the needs of your particular content by using very general tags.

Range of coverage

Each markup language has a range of the formatting and structural elements that it can represent.

HTML has a particularly low range of coverage. That is, of the world of tags that you may like to have, the HTML language offers relatively few. This makes HTML easy to learn but also easy to grow out of. I remember how gratifying it was to learn the first versions of the HTML tag set in one sitting. I remember, too, how quickly I stopped finding tags that I needed to make my pages work.

Word processors (and desktop publishing programs as well) have substantially more tags than HTML. They need a lot tags to cover the range of needs of professional publishers who may use them. As HTML's continued to develop, it's come ever closer to covering the same mid-range selection of tags that most word processors support. With the extensions offered by Cascading Style Sheets (CSS), HTML can today support a large fraction of the formatting and structure of word processor markup.

The range of coverage of XML, of course, is infinite. It can have any tag that you can dream of - and lots of them. Given that you have the wherewithal to support the tags that you create, you can represent any formatting or structure that you can imagine.

Working with Markup

Working directly with markup code can be trying at best and completely baffling at worst. It takes a certain attitude and a few key concepts to get past the complexity and to work successfully with markup code.

Anatomy of a tag

The tag is the basic unit of markup, as shown in Figure 1.

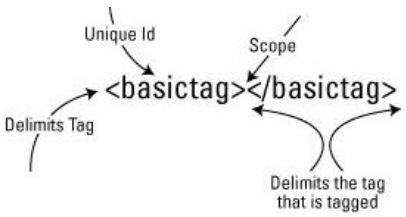


Figure 1*: The structure of a typical HTML tag*

This illustration shows the following qualities that tags must always have:

- **Delimiters:** Delimiters show you "de limits" of something. In the case of tags, delimiters show you where the tag itself begins and ends as well as where the content that's tagged begins and ends. In HTML, the angle brackets (<>) delimit the tag and <basictag></basictag> delimit the content being tagged. (Notice that no such tag as <basictag> exists in HTML.)

- **Unique identifier:** With all tags, you must have some way to distinguish one tag from another. Usually, but not necessarily, it's a unique name within the delimiters of the tag. In Figure 1, the tag ID is "basictag".
- **Scope:** All tags have a scope of application. That is, something (in this case, delimiters) must say what part of the content the tag applies to. The preceding scope is actually empty; the tags here aren't applied to any content. This tag stands alone, with no content tagged. Although not the most usual circumstance, this isn't unknown. The <HR> tag in HTML that creates a horizontal rule (a line) is an example of a tag that doesn't have any content scope.

Tags may also have the following additional qualities (see Figure 2):

- **Attributes:** Aside from simply the name that identifies the tag, other properties or quantities may go along with the tag to specify what or how much. In the preceding example, COLSPAN is an attribute that has a value of 2. In HTML, these attributes are always enclosed within the opening tag.
- **Operators and internal delimiters:** If tags have attributes, they must also have operators and internal delimiters. The operators say how the value relates to the attribute. In the preceding figure, the equal sign relates COLSPAN to 2. Internal delimiters separate the tag name from any attributes and their values. In the example, you can see one of the rare occasions where white space (a blank space) is used as a delimiter. It separates the tag name from the attribute. It also separates multiple attributes from each other. The quotation marks are also delimiters. They delimit attribute values. Because HTML browsers are so permissive, quotes are often, but not always, unnecessary.
- **Insides:** Within the scope of a tag can be content (no tags) and other tags. Remember that content doesn't need to be text but usually is. If other tags exist within the scope of a tag, the meaning of the relationship needs to be understood by the program that interprets the tag. I talk more about this in the section "Nesting," later in this white paper For now, notice in the preceding example the use of table row tags (<TR>). Because of the rules of HTML, these tags make no sense outside of the context of a <TABLE> tag.

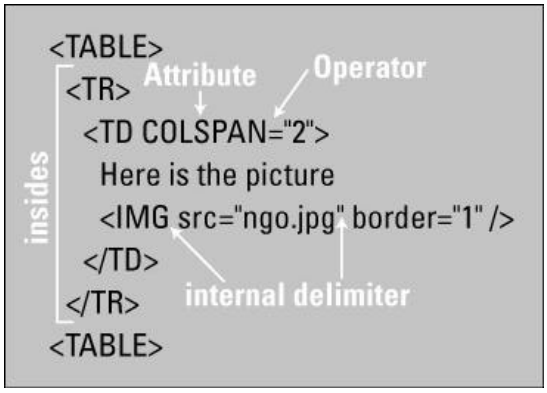


***Figure 2**: An HTML table tag showing additional tag qualities*

The language vs. the interpreter

Markup is a language. By itself, it doesn't do anything. It requires a speaker of the language, an interpreter, for it to be read and written. This is such a simple point that it always surprises me that people confuse the two. Word processors present the simplest case. Here, you have one language, one interpreter, and a very tight coupling. So tight is the coupling that it's unlikely that you've ever thought about the markup code behind word processors.

HTML is more complex. The language of HTML is spoken by browsers. Each variety of browser speaks a slightly different dialect. It's the interaction of the HTML language with the particular interpreter that finally decides how the markup is used. Different browsers make different interpretations of HTML tags as they turn them into formatting on your screen. HTML interpreters then are more loosely coupled to the language than word processors. The result is that their interpretations are less predictable.

In XML, tools are available that read and write XML, but developing the interpreter is up to you (or someone with a problem substantially similar to yours). Only you know the correct interpretation of the tags that you create. Again, XML shows itself not as a markup language but as a metamarkup language.

Representing representation

A Zen master and her student were sitting on a park bench contemplating the universe:

> *"Master, what holds up the world?" asked the student. "The world, my student, rides on the back of a giant turtle." "Yes," said the student, "but what holds the turtle up?" "You are very clever, my friend, very clever," calmly replied the master. "But I'm afraid it is turtles all the way down."*

How do you represent bold text in HTML? As follows:

```
<B>
```

How do you represent <B> in HTML? As follows:

```
&LT;B&GT;
```

How do you represent <B> in HTML?

In markup, as in life, "it's turtles all the way down." No matter which method you choose to represent markup, you always face the chance that someone wants to type those same characters and not have them interpreted as markup. The problem is that people use the same symbols to represent both text and markup. This turns out to be a minor inconvenience to all but those who are writing about the markup that they're using. Still, it points out a deep issue in the study of markup: A receding recursive set of definitions on how to interpret markup always ends in either a human making the final interpretation or a simplifying assumption that stops the recursion from being endless and sending processors into infinite regression.

Don't be baffled by syntax

Because of the problem of representing representation, and to keep markup from becoming even more verbose, the makers of markup choose unusual characters or character combinations to create tags. Even XML, whose creators deemed that "Terseness in XML markup is of minimal importance" (see www.w3.org/TR/1998/REC-xml-19980210), is full of shorthand and strange character combinations that are useful to the initiated but obscure to the neophyte. If you're new to markup, the way to get past this barrier is to first understand that the normal rules of reading don't apply and then to begin slowly to unpack each statement, paying careful attention to the meaning (or lack of meaning) of white space and nesting.

The following code, for example, shows the HTML table syntax from the earlier section:

```
<TABLE>
   <TR>
            <TD COLSPAN="2">
            Here is the picture
```

```
            <IMG SRC="ngo.jpg" BORDER="1">
            </TD>
        </TR>
    <TABLE>
```

How can you turn this code into English? As follows:

- **Shorthand tag names stand for real words**. TABLE is a table, TR is a table row, TD is table data (better known as a column), and IMG is an image. So this sample is about tables, their rows and columns, and an image.
- **Every tag "inside" is contained by that tag**. In this case, the image is contained in the column, which is contained in the row, which is contained in the table.
- **Parameters tell you what the tag has**. In the preceding example, the table column has a column span of 2 (COLSPAN), and the image has a source file name of ngo.jpg as well as a border of 1.

The only thing left to know is what things such as column span are and what effect they have in the markup. I hope that the example I used is a simple one for you. I chose it to illustrate the simple rules that still apply after the markup looks more as follows:

```
\trowd \trgaph108\trleft-108\trbrdrt\brdrs\brdrw10 \trbrdrl\brdrs\brdrw10
\trbrdrb\brdrs\brdrw10 \trbrdrr\brdrs\brdrw10 \trbrdrh\brdrs\brdrw10 \trbrdrv
\brdrs\brdrw10
\trftsWidth1\trautofit1\trpaddl108\trpaddr108\trpaddfl3\trpaddfr3
\clvertalt\clbrdrt\brdrs\brdrw10 \clbrdrl\brdrs\brdrw10 \clbrdrb\brdrs\brdrw10
\clbrdrr\brdrs\brdrw10 \cltxlrtb\clftsWidth3\clwWidth8856
\cellx8748\pard\plain \ql
\li0\ri0\widctlpar\intbl\aspalpha\aspnum\faauto\adjustright\rin0\lin0
\fs24\lang1033\langfe1033\cgrid\langnp1033\langfenp1033 { {\*\shppict
{\pict {\*\picprop\shplid1025 {\sp {\sn shapeType} {\sv 75} } {\sp {\sn fFlipH}
{\sv 0} } {\sp {\sn fFlipV} {\sv 0} } {\sp {\sn pibName} {\sv C:\'5cDocuments
and Settings\ '5cbob\'5cMy Documents\'5cMy Pictures\'ngo.jpg} } {\sp {\sn
pibFlags} {\sv 2} } {\sp {\sn fLine} {\sv 0} } {\sp {\sn fLayoutInCell} {\sv 1} } }
\picscalex100\picscaley100\piccropl0\piccropr0\piccropt0\piccropb0
\picw7488\pich5609\picwgoal4245\pichgoal3180\jpegblip\bliptag-
693656786 {\*\blipuid many pages of numbers go here} } }\cell }\pard \ql
\li0\ri0\widctlpar\intbl\aspalpha\aspnum\faauto\adjustright\rin0\lin0 { \trowd
\trgaph108\trleft-108\trbrdrt\brdrs\brdrw10 \trbrdrl\brdrs\brdrw10
\trbrdrb\brdrs\brdrw10 \trbrdrr\brdrs\brdrw10 \trbrdrh\brdrs\brdrw10
\trbrdrv\brdrs\brdrw10
\trftsWidth1\trautofit1\trpaddl108\trpaddr108\trpaddfl3\trpaddfr3
\clvertalt\clbrdrt\brdrs\brdrw10 \clbrdrl\brdrs\brdrw10 \clbrdrb\brdrs\brdrw10
\clbrdrr\brdrs\brdrw10 \cltxlrtb\clftsWidth3\clwWidth8856 \cellx8748\row }
```

This is the same content but now represented in Microsoft Word's RTF markup code.

The concept of nesting

Nesting is the official name for what I previously called "insides." It's tags within tags. Look, for example, at the following sentence:

```
<H1>This is a <FONT SIZE=+10>Heading</FONT></H1>
```

The entire line is tagged H1. (Notice the opening <H1> tag at the beginning and the closing </H1> tag at the end.) Only the word Heading is subject to the <FONT> tag, which turns up the

font size by 10 points. The question is 10 points bigger than what? The answer, of course, is 10 points bigger than an H1 (whatever that happens to be for your browser). In other words, the <FONT> tag is interpreted within the context of the <H1> tag. If it were inside a different tag, it would be interpreted differently. This is a fairly trivial example of nesting, but it shows the basic concept.

In HTML, nesting is of moderate importance. Some tags - notably, the <TABLE> tags - make no sense at all outside of the context of other tags. Following HTML's format emphasis, nesting serves the purpose of creating compound or complex formatting options such as bold, italic, or doubly indented lists.

In RTF, which I describe earlier as an ASCII equivalent of Microsoft Word markup, nesting is extreme, often going 10 levels deep! And unlike HTML, where end tags carry the name of the tag they're ending, RTF ends all tags with a single curly bracket ("}"), turning the structure of many RTF files into a rat's nest. (The very ugly markup example in the preceding section is RTF.)

Nesting in XML is central. As you may expect, nesting represents the structural components of an XML file. A common way of referring to the relationships in nested (or hierarchical) code is to use the analogy of the *parent-child relationship*. Each tag in an XML document can have one or more "parents" and one or more "children." Without paying attention to a tag's parents, it's often impossible to determine its meaning. Consider the following XML fragment that, if you look at it at closely, you see is obviously part of a larger XML structure:

```
<PR RESOURCES>
<SPEAKERS>
<NAME>Anna Emelia</NAME>
</SPEAKERS>
<HARDWARE>
<MICROPHONES/>
<SPEAKERS/>
<HARDWARE>
</PR RESOURCES>
```

What does the <SPEAKERS> tag mean? Clearly, it depends on its parents and possibly its grandparents. XML files are like outlines. Their interpretation depends largely on your ability to travel up and down the branches of the file's hierarchy. This quality more than any other, I believe, defines XML.

Note

The namers of XML focused on its eXtensibility. They may have been better off focusing on its nesting qualities by calling it HML for Hierarchical Markup Language (but, of course, *X* is sexier these days than *H*, and *H* was already taken).

The benefits of white space

White space is spaces, tabs, and paragraph marks (carriage returns and line feeds). All the markup languages that I've discussed (HTML, XML and RTF) ignore them completely. This is a strange thing. For humans, white space is essential to understanding the distinction and relationship between content elements. People are visually driven. Look at this page. On both paper or on-screen, it's the white space that tells you where one thing ends and another begins. The more white space around an object, generally, the more important it is. Indentation is essential to understanding any kind of hierarchical relationship. So linked are white space and

meaning that many conversion programs actually translate white-space qualities into structural tags for a document.

Remember the mess of RTF markup that I showed you earlier? Compare it to the following version, in which I add appropriate white space. The syntax is no less imposing, but at least you know where to start to unpack it.

```
\trowd \trgaph108\trleft-108\trbrdrt\brdrs\brdrw10
\trbrdrl\brdrs\brdrw10 \trbrdrb\brdrs\brdrw10
\trbrdrr\brdrs\brdrw10 \trbrdrh\brdrs\brdrw10 \trbrdrv
\brdrs\brdrw10
\trftsWidth1\trautofit1\trpaddl108\trpaddr108\trpaddfl3\trpaddfr3
\clvertalt\clbrdrt\brdrs\brdrw10 \clbrdrl\brdrs\brdrw10
\clbrdrb\brdrs\brdrw10 \clbrdrr\brdrs\brdrw10
\cltxlrtb\clftsWidth3\clwWidth8856 \cellx8748\pard\plain
\ql
\li0\ri0\widctlpar\intbl\aspalpha\aspnum\faauto\adjustright\rin0\li
n0
\fs24\lang1033\langfe1033\cgrid\langnp1033\langfenp1033
{
   {\*\shppict
            {\pict
                     {\*\picprop\shplid1025
                           {\sp
                                    {\sn shapeType}
                                                {\sv 75}
                                    }
                           {\sp
                                    {\sn fFlipH}
                                    {\sv 0}
                           }
                           {\sp
                                    {\sn fFlipV}
                                    {\sv 0}
                           }
                           {\sp
                                    {\sn pibName}
                                    {\sv C:\'5cDocuments and
Settings\
                                    '5cbob\'5cMy Documents\'5cMy
Pictures\'ngo.jpg}
```

```
                                }
                                {\sp
                                        {\sn pibFlags}
                                        {\sv 2}
                                }
                                {\sp
                                        {\sn fLine}
                                        {\sv 0}
                                }
                                {\sp
                                        {\sn fLayoutInCell}
                                        {\sv 1}
                                }
                        }
   \picscalex100\picscaley100\piccropl0\piccropr0\piccropt0\piccrop
b0
   \picw7488\pich5609\picwgoal4245\pichgoal3180\jpegblip\bliptag-
693656786
                        {\*\blipuid many pages of numbers go here}
                }
                }\cell
   }\pard \ql
\li0\ri0\widctlpar\intbl\aspalpha\aspnum\faauto\adjustright\rin0\li
n0
{
\trowd \trgaph108\trleft-108\trbrdrt\brdrs\brdrw10
\trbrdrl\brdrs\brdrw10 \trbrdrb\brdrs\brdrw10
\trbrdrr\brdrs\brdrw10
\trbrdrh\brdrs\brdrw10 \trbrdrv\brdrs\brdrw10
\trftsWidth1\trautofit1\trpaddl108\trpaddr108\trpaddfl3\trpaddfr3
\clvertalt\clbrdrt\brdrs\brdrw10 \clbrdrl\brdrs\brdrw10
\clbrdrb\brdrs\brdrw10 \clbrdrr\brdrs\brdrw10
\cltxlrtb\clftsWidth3\clwWidth8856 \cellx8748\row
}
```

For markup interpreters, it's the tags - and only the tags - that matter. Everyone who's just started writing in HTML is baffled by spaces that she thinks that she's put in but that somehow don't show up in the browser. Eventually, you learn to use the code   to ensure that a space isn't ignored. Markup can look totally jumbled and still be interpreted perfectly. Similarly, markup that's all nicely aligned and indented is for your eyes only; the interpreter couldn't care less.

People play in the margins

A general rule of life says that, if presented with a rule, industrious people find a loophole. As goes life, so goes markup. Where you find loopholes in the markup syntax, you can pack in some of your own information that goes unnoticed by the markup's interpreters.

You can't, for example, make up new HTML tags - they have no meaning to the standard interpreters (such as Web browsers). If you want to put something into an HTML file for your own purposes, however, you can use one of the following syntax loopholes to get nonstandard markup into a standard HTML file:

- **Metatags:** Because they have a somewhat open format, metatags are often tweaked to provide space to pack editorial, tracking, access, and other miscellaneous information into HTML files.
- **Comments:** Intended for commentary, the comment tag (<!- - comment - ->) gives developers a convenient place to type in little extras.
- **Bad syntax:** If you misspell a tag name in HTML, you may never know it (or you have a heck of a time trying to figure out why your page doesn't look right). Browsers simply ignore the misspelled tag and go on unhindered. Knowing this, creative Webmeisters often do make up tags that their own custom components understand. They know that these new tags are safely ignored by the permissiveness of the browsers. Be careful with this technique: You never know when a less permissive browser may come along and show the world your "mistake."
- **XML is open:** You can make up as many tags as you want. You usually have no need to trick the interpreter into ignoring your transgressions. Your problem is more one of developing the interpreter that can correctly identify and deal with your tags.

Summary

Markup was never glamorous until HTML came along. I'm sure that it's going to fade back into the obscurity that it came from someday. For now, however, HTML and especially XML are great conversation starters at parties. These markup languages and a host of others do the following:

- Represent content format.
- Represent content structure.
- Consist of tags that surround and add context to the content inside them.
- Give you either a simple but limited set of tags (HTML) or an unlimited set of tags but a lot of responsibility for how those tags are interpreted (XML).