The Association for Computers and the Humanities (ACH)

The Association for Computational Linguistics (ACL)

The Association for Literary and Linguistic Computing (ALLC)

Guidelines for Electronic Text Encoding and Interchange

Edited by

C. M. Sperberg-McQueen and Lou Burnard

TEI P2, Chapter 2 SG
Text Encoding Initiative
Chicago, Oxford

May 28, 1993

Draft Version 2, May 28, 1993

Note

This is the current draft (as of May 28, 1993) of one section of the *Guidelines for Electronic Text Encoding and Interchange*, version 2 (also known as TEI P2), produced by the Text Encoding Initiative. This fascicle contains:

- a table of contents for TEI P2, indicating the status of each section as of the release date of this fascicle
- a draft for one section of TEI P2
- entries derived from this section for the reference list of tags of TEI P2 (if any)
- a User Response and Comment Form

Please feel free to send us your comments on the contents of this draft and their applicability to your needs. Comments may be sent to the TEI-L list server for general discussion, or directly to either of the TEI editors (adresses on the User Response and Comment form). Bear in mind in reading, however, that TEI P2 is a reference manual, not a tutorial in the use of electronic text, and unavoidably assumes either a certain amount of prior knowledge or patience with unfamiliar technical material. Note also that some matters which may appear obscure in one section may well be treated in more detail elsewhere in the full text of TEI P2.

TEI P2, like P1, is a working draft. The observant will find several typographic and presentational flaws in the current text: for example, cross-references are not properly resolved in the LaTeX versions and some parts of the reference material are not yet complete. We hope to rectify as many as possible of these flaws in the coming months, but have decided not to allow them to delay any further the release of the draft. We hope that those interested in the Guidelines will find the current state of the text readable and usable enough for the purpose of substantive comment. As far as possible, all comments received on this section will be taken into account in the preparation of the next version of the Guidelines, which is due by the end of 1993.

Updates on the status of the draft as a whole will be distributed automatically to all subscribers of the Listserv list TEI-L. To subscribe to TEI-L, send electronic mail to the address LISTSERV@UICVM (or Listserv@uicvm.uic.edu) containing the single line

```
subscribe tei-l J. Smith
```

(substituting your name for “J. Smith”). TEI-L is also the appropriate place to pose questions or offer public comments on the TEI guidelines and other relevant issues. The TEI-L file server contains all the sections of TEI P2 thus far released, as well as other TEI materials. Its contents are shadowed at various other sites around the world; for more information consult one of the following documents:

- TEI ED J8, “Obtaining the Second Version of the TEI Guidelines (TEI P2)” (describes how to retrieve electronic copies of TEI P2 and the various formats they are available in)
- TEI ED J9, “Obtaining Paper Copies of the Second Version of the TEI Guidelines (TEI P2)” (describes how to request paper copies of TEI P2, for those without electronic mail access)

These documents are available from the TEI-L file server or through the editors at the addresses given on the User Response and Comment Form.

Overall Table of Contents for TEI P2

The following list shows the expected contents of TEI P2 as a whole and indicates the current status of each chapter.

Contents

Part 1

Introduction

Chapter 2

A Gentle Introduction to SGML

The encoding scheme defined by these Guidelines is formulated as an application of a system known as the Standard Generalized Markup Language (SGML).[1] SGML is an international standard for the definition of device-independent, system-independent methods of representing texts in electronic form. This chapter presents a brief tutorial guide to its main features, for those readers who have not encountered it before. For a more technical account of TEI practice in using the SGML standard, see chapter 30, "TEI Conformance," [in separate fascicle]; for a more technical description of the subset of SGML used by the TEI encoding scheme, see chapter 39, "Formal Grammar for the TEI-Interchange-Format Subset of SGML," [in separate fascicle].

SGML is an international standard for the description of marked-up electronic text. More exactly, SGML is a *metalanguage,* that is, a means of formally describing a language, in this case, a *markup language*. Before going any further we should define these terms.

Historically, the word *markup* has been used to describe annotation or other marks within a text intended to instruct a compositor or typist how a particular passage should be printed or laid out. Examples include wavy underlining to indicate boldface, special symbols for passages to be omitted or printed in a particular font and so forth. As the formatting and printing of texts was automated, the term was extended to cover all sorts of special *markup codes* inserted into electronic texts to govern formatting, printing, or other processing.

Generalizing from that sense, we define markup, or (synonymously) *encoding*, as any means of making explicit an interpretation of a text. At a banal level, all printed texts are encoded in this sense: punctuation marks, use of capitalization, disposition of letters around the page, even the spaces between words, might be regarded as a kind of markup, the function of which is to help the human reader determine where one word ends and another begins, or how to identify gross structural features such as headings or simple syntactic units such as dependent clauses or sentences. Encoding a text for computer processing is in principle, like transcribing a manuscript from *scriptio continua*, a process of making explicit what is conjectural or implicit, a process of directing the user as to how the content of the text should be interpreted.

By *markup language* we mean a set of markup conventions used together for encoding texts. A markup language must specify what markup is allowed, what markup is required, how markup is to be distinguished from text, and what the markup means. SGML provides the means for doing the first three; documentation such as these Guidelines is required for the last.

---

[1] International Organization for Standardization, *ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML),* ([Geneva]: ISO, 1986). Although widely said to be short for the surnames of its progenitors, the official expansion of this abbreviation is "Standard Generalized Markup Language."

The present chapter attempts to give an informal introduction—much less formal than the standard itself—to those parts of SGML of which a proper understanding is necessary to make best use of these Guidelines.

2.1 What's Special about SGML?

There are three characteristics of SGML which distinguish it from other markup languages: its emphasis on descriptive rather than procedural markup; its *document type* concept; and its independence of any one system for representing the script in which a text is written. These three aspects are discussed briefly below, and then in more depth in sections 2.3, "SGML Structures," on page 6 and 2.7, "SGML Entities," on page 21.

2.1.1 Descriptive Markup

A descriptive markup system uses markup codes which simply provide names to categorize parts of a document. Markup codes such as "**<para>**" or "\end{list}" assert simply of the part of a document which they identify "the following item is a paragraph," "this is the end of the most recently begun list," etc. By contrast, a procedural markup system defines what processing is to be carried out at particular points in a document: "call procedure *para* with parameters 1, b and x here" or "move the left margin 2 quads left, move the right margin 2 quads right, skip down one line, and go to the new left margin," etc. In SGML, the instructions needed to process a document for some particular purpose (for example, to format it) are sharply distinguished from the descriptive markup which occurs within the document. Usually, they are collected outside the document in separate procedures or programs.

With descriptive instead of procedural markup the same document can readily be processed by many different pieces of software, each of which can apply different processing instructions to those parts of it which are considered relevant. For example, a content analysis program might disregard entirely the footnotes embedded in an annotated text, while a formatting program might extract and collect them all together for printing at the end of each chapter. Different sorts of processing instructions can be associated with the same parts of the file. For example, one program might extract names of persons and places from a document to create an index or database, while another, operating on the same text, might print names of persons and places in a distinctive typeface.

2.1.2 Types of Document

Secondly, SGML introduces the notion of a *document type,* and hence a *document type definition* (DTD). Documents are regarded as having types, just as other objects processed by computers do. The type of a document is formally defined by its constituent parts and their structure. The definition of a report, for example, might be that it consisted of a title and possibly an author, followed by an abstract and a sequence of one or more paragraphs. Anything lacking a title, according to this formal definition, would not formally be a report, and neither would a sequence of paragraphs followed by an abstract, whatever other report-like characteristics these might have for the human reader.

If documents are of known types, a special purpose program (called a *parser*) can be used to process a document claiming to be of a particular type and check that all the elements required for that document type are indeed present and correctly ordered. More significantly, different documents of the same type can be processed in a uniform way. Programs can be written which take advantage of the knowledge encapsulated in the document structure information, and which can thus behave in a more intelligent fashion.

2.1.3 Data Independence

A basic design goal of SGML was to ensure that documents encoded according to its provisions should be transportable from one hardware/software environment to another without loss of information. The two features discussed so far both address this requirement at an abstract level: the third feature addresses it at the level of the strings of bytes (characters) of which documents are composed. SGML provides a general purpose mechanism for *string substitution*, that is, a simple machine-independent way of stating that a particular string of characters in the document should be replaced by some other string when the document is processed. One obvious application for this mechanism is to ensure consistency of nomenclature; another, more significant, is to counter the notorious inability of different computer systems to understand each other's character sets, or of any one system to provide all the graphic characters needed for a particular application, by providing descriptive mappings for non-portable characters. The strings defined by this string-substitution mechanism are called *entities* and they are discussed below in section 2.7, “SGML Entities,” on page 21.

2.2 Textual Structure

A text is not an undifferentiated sequence of words, much less of bytes. For different purposes, it may be divided into many different units, of different types or sizes. A prose text such as this one might be divided into sections, chapters, paragraphs, and sentences. A verse text might be divided into cantos, stanzas, and lines. Once printed, sequences of prose and verse might be divided into volumes, gatherings, and pages.

Structural units of this kind are most often used to identify specific locations or reference points within a text (“the third sentence of the second paragraph in chapter ten”; “canto 10, line 1234”; “page 412,” etc.) but they may also be used to subdivide a text into meaningful fragments for analytic purposes (“is the average sentence length of section 2 different from that of section 5?” “how many paragraphs separate each occurrence of the word *nature*?” “how many pages?”). Other structural units are more clearly analytic, in that they characterize a section of a text. A dramatic text might regard each speech by a different character as a unit of one kind, and stage directions or pieces of action as units of another kind. Such an analysis is less useful for locating parts of the text (“the 93rd speech by Horatio in Act 2”) than for facilitating comparisons between the words used by one character and those of another, or those used by the same character at different points of the play.

In a prose text one might similarly wish to regard as units of different types passages in direct or indirect speech, passages employing different stylistic registers (narrative, polemic, commentary, argument, etc.), passages of different authorship and so forth. And for certain types of analysis (most notably textual criticism) the physical appearance of one particular printed or manuscript source may be of importance: paradoxically, one may wish to use descriptive markup to describe presentational features such as typeface, line breaks, use of white space and so forth.

These textual structures overlap with each other in complex and unpredictable ways. Particularly when dealing with texts as instantiated by paper technology, the reader needs to be aware of both the physical organization of the book and the logical structure of the work it contains. Many great works (Sterne's *Tristram Shandy* for example) cannot be fully appreciated without an awareness of the interplay between narrative units (such as chapters or paragraphs) and page divisions. For many types of research, it is the interplay between different levels of analysis which is crucial: the extent to which syntactic structure and narrative structure mesh, or fail to mesh, for example, or the extent to which phonological structures reflect morphology.

2.3 SGML Structures

This section describes the simple and consistent mechanism for the markup or identification of structural textual units which is provided by SGML. It also describes the methods SGML provides for the expression of rules defining how combinations of such units can meaningfully occur in any text.

2.3.1 Elements

The technical term used in the SGML standard for a textual unit, viewed as a structural component, is *element*. Different types of elements are given different names, but SGML provides no way of expressing the meaning of a particular type of element, other than its relationship to other element types. That is, all one can say about an element called (for instance) **<blort>** is that instances of it may (or may not) occur within elements of type **<farble>**, and that it may (or may not) be decomposed into elements of type **<blortette>**. It should be stressed that the SGML standard is entirely unconcerned with the semantics of textual elements: these are application dependent.[2] It is up to the creators of SGML conformant tag sets (such as these Guidelines) to choose intelligible names for the elements they identify and to document their proper use in text markup. That is one purpose of this document. From the need to choose element names indicative of function comes the technical term for the name of an element type, which is *generic identifier*, or GI.

Within a marked up text (a *document instance*), each element must be explicitly marked or tagged in some way. The standard provides for a variety of different ways of doing this, the most commonly used being to insert a tag at the beginning of the element (a *start-tag*) and another at its end (an *end-tag*). The start- and end-tag pair are used to bracket off the element occurrences within the running text, in rather the same way as different types of parentheses or quotation marks are used in conventional punctuation. For example, a quotation element in a text might be tagged as fol-

[2] Work is currently going on in the standards community to create (using SGML syntax) a definition of a standard “document style semantics and specification language” or DSSSL.

lows:

```
...  Rosalind's remarks <quote>This is the silliest stuff
 that ere I heard of!</quote> clearly indicate ...
```

As this example shows, a start-tag takes the form **<name>**, where the opening angle bracket indicates the start of the start-tag, “name” is the generic identifier of the element which is being delimited, and the closing angle bracket indicates the end of a tag. An end-tag takes an identical form, except that the opening angle bracket is followed by a solidus (slash) character, so that the corresponding end-tag would be **</name>**.[3]

2.3.2 Content Models: An Example

An element may be *empty*, that is, it may have no content at all, or it may contain simple text. More usually however, elements of one type will be *embedded* (contained entirely) within elements of a different type. This is one reason why an end-tag needs to specify which element it is closing.

To illustrate this, we will consider a very simple structural model. Let us assume that we wish to identify within an anthology only poems, their titles, and the stanzas and lines of which they are composed. In SGML terms, our document type is the *anthology,* and it consists of a series of *poem*s. Each poem has embedded within it one element, a title, and several occurrences of another, a stanza, each stanza having embedded within it a number of line elements. Fully marked up, a text conforming to this model might appear as follows:[4]

```
<anthology>
  <poem><title>The SICK ROSE</title>
     <stanza>
          <line>O Rose thou art sick.</line>
          <line>The invisible worm,</line>
          <line>That flies in the night</line>
          <line>In the howling storm:</line>
     </stanza>
     <stanza>
          <line>Has found out thy bed</line>
          <line>Of crimson joy:</line>
          <line>And his dark secret love</line>
          <line>Does thy life destroy.</line>
     </stanza>
  </poem>
```

---

3 The actual characters used for the delimiting characters (the angle brackets, exclamation mark and solidus) may be redefined, but it is conventional to use the characters used in this description.

4 The example is taken from William Blake's *Songs of innocence and experience* (1794). The markup is designed for illustrative purposes and is not TEI conformant.

```
          <!-- more poems go here    -->

</anthology>
```

It should be stressed that this example does *not* use the same names as are proposed for corresponding elements elsewhere in these Guidelines: the above is *not* a valid TEI document. It will however serve as an introduction to the basic notions of SGML. White space and line breaks have been added to the above example for the sake of visual clarity only; they have no particular significance in the SGML encoding itself. Also, the line

```
          <!-- more poems go here    -->
```

is an SGML *comment* and is not treated as part of the text.

This example makes no assumptions about the rules governing, for example, whether or not a title can appear in places other than preceding the first stanza, or whether lines can appear which are not included in a stanza: that is why its markup appears so verbose. In such cases, the beginning and end of every element must be explicitly marked, because there are no identifiable rules about which elements can appear where. In practice, however rules can usually be formulated to reduce the need for so much tagging. For example, considering our greatly over-simplified model of a poem, we could state the following rules:

1. An anthology contains a number of poems and nothing else.
2. A poem always has a single title element which precedes the first stanza and contains no other elements.
3. Apart from the title, a poem consists only of stanzas.
4. Stanzas consist only of lines and every line is contained by a stanza.
5. Nothing can follow a stanza except another stanza or the end of a poem.
6. Nothing can follow a line except another line or the start of a new stanza.

From these rules, it may be inferred that we do not need to mark the ends of stanzas or lines explicitly. From rule 2 it follows that we do not need to mark the end of the title—it is implied by the start of the first stanza. Similarly, from rules 3 and 1 it follows that we need not mark the end of the poem: since poems cannot occur within poems but must occur within anthologies, the end of a poem is implied by the start of the next poem, or by the end of the anthology. Applying these simplifications, we could mark up the same poem as follows:

```
<anthology>
     <poem><title>The SICK ROSE
     <stanza>
          <line>O Rose thou art sick.
          <line>The invisible worm,
          <line>That flies in the night
          <line>In the howling storm:
     <stanza>
          <line>Has found out thy bed
          <line>Of crimson joy:
          <line>And his dark secret love
          <line>Does thy life destroy.
```

```
          <poem>
               <!-- more poems go here    -->

     </anthology>
```

The ability to use rules stating which elements can be nested within others to simplify markup is a very important characteristic of SGML. Before considering these rules further, you may wish to consider how text marked up in the form above could be processed by a computer for very many different purposes. A simple indexing program could extract only the relevant text elements in order to make a list of titles, or of words used in the poem text; a simple formatting program could insert blank lines between stanzas, perhaps indenting the first line of each, or inserting a stanza number. Different parts of each poem could be typeset in different ways. A more ambitious analytic program could relate the use of punctuation marks to stanzaic and metrical divisions.[5] Scholars wishing to see the implications of changing the stanza or line divisions chosen by the editor of this poem can do so simply by altering the position of the tags. And of course, the text as presented above can be transported from one computer to another and processed by any program (or person) capable of making sense of the tags embedded within it with no need for the sort of transformations and translations needed to move word processor files around.

2.4 Defining SGML Document Structures: The DTD

Rules such as those described above are the first stage in the creation of a formal specification for the structure of an SGML document, or *document type definition*, usually abbreviated to *DTD*. In creating a DTD, the document designer may be as lax or as restrictive as the occasion warrants. A balance must be struck between the convenience of following simple rules and the complexity of handling real texts. This is particularly the case when the rules being defined relate to texts which already exist: the designer may have only the haziest of notions as to an ancient text's original purpose or meaning and hence find it very difficult to specify consistent rules about its structure. On the other hand, where a new text is being prepared to an exact specification, for example for entry into a textual database of some kind, the more precisely stated the rules, the better they can be enforced. Even in the case where an existing text is being marked up, it may be beneficial to define a restrictive set of rules relating to one particular view or hypothesis about the text—if only as a means of testing the usefulness of that view or hypothesis. It is important to remember that every document type definition is an interpretation of a text. There is no single DTD which encompasses any kind of absolute truth about a text, although it may be convenient to privilege some DTDs above others for particular types of analysis.

At present, SGML is most widely used in environments where uniformity of document structure is a major desideratum. In the production of technical documentation, for example, it is of major importance that sections and subsections should be properly nested, that cross references should be properly resolved and so forth. In such situ-
&@fn@sep

5 Note that this simple example has not addressed the problem of marking elements such as sentences explicitly; the implications of this are discussed below in section 2.5.2, “Concurrent Structures,” on page 14.

ations, documents are seen as raw material to match against pre-defined sets of rules. As discussed above, however, the use of simple rules can also greatly simplify the task of tagging accurately elements of less rigidly constrained texts. By making these rules explicit, the scholar reduces his or her own burdens while also being forced to make explicit an interpretation of the text being encoded.

2.4.1 An Example DTD

A DTD is expressed in SGML as a set of declarative statements, using a simple syntax defined in the standard. For our simple model of a poem, the following declarations would be appropriate:

```
<!ELEMENT anthology      - -  (poem+)>
<!ELEMENT poem           - -  (title?, stanza+)>
<!ELEMENT title          - O  (#PCDATA) >
<!ELEMENT stanza         - O  (line+)   >
<!ELEMENT line           O O  (#PCDATA) >
```

These five lines are examples of formal SGML element declarations. A declaration, like an element, is delimited by angle brackets; the first character following the opening bracket must be an exclamation mark, followed immediately by one of a small set of SGML-defined keywords, specifying the kind of object being declared. The five declarations above are all of the same type: each begins with an “ELEMENT” keyword, indicating that it declares an element, in the technical sense defined above. Each consists of three parts: a name or group of names, two characters specifying “minimization rules,” and a “content model.” Each of these parts is discussed further below. Components of the declaration are separated by white space, that is one or more blanks, tabs or newlines.

The first part of each declaration above gives the generic identifier of the element which is being declared, for example *poem*, *title*, etc. It is possible to declare several elements in one statement, as discussed below.

2.4.2 Minimization Rules

The second part of the declaration specifies what are called *minimization rules* for the element concerned. These rules determine whether or not start- and end-tags must be present in every occurrence of the element concerned. They take the form of a pair of characters, separated by white space, the first of which relates to the start-tag, and the second to the end-tag. In either case, either a hyphen or a letter O (for “omissible” or “optional”) must be given; the hyphen indicating that the tag must be present, and the letter O that it may be omitted. Thus, in this example, every element except **<line>** must have a start-tag. Only the **<poem>** and **<anthology>** elements must have end-tags as well.

2.4.3 Content Model

The third part of each declaration, enclosed in parentheses, is called the *content model* of the element, because it specifies what element occurrences may legitimately contain. Contents are specified either in terms of other elements or using special reserved words. There are several such reserved words, of which by far the most commonly encountered is #PCDATA, as in this example. This is an abbreviation for "parsed character data," and it means that the element being defined may contain any valid character data. If an SGML declaration is thought of as a structure like a family tree, with a single ancestor at the top (in our case, this would be the **<anthology>**), then almost always, following the branches of the tree downwards (for example, from **<anthology>** to **<poem>** to **<stanza>** to **<line>** and **<title>**) will lead eventually to #PCDATA. In our example, **<title>** and **<line>**s are so defined. Since their content models say #PCDATA only and name no embedded elements, they may not contain any embedded elements.

2.4.4 Occurrence Indicators

The declaration for **<stanza>** in the example above states that a stanza consists of one or more lines. It uses an *occurrence indicator* (the plus sign) to indicate how many times the element named in its content model may occur. There are three occurrence indicators in the SGML syntax, conventionally represented by the plus sign, the question mark, and the asterisk or star. The plus sign means that there may be one or more occurrences of the element concerned; the question mark means that there may be at most one and possibly no occurrence; the star means that the element concerned may either be absent or appear one or more times. Thus, if the content model for **<stanza>** were "(LINE*)," stanzas with no lines would be possible as well as those with more than one line. If it were "(LINE?)," again empty stanzas would be countenanced, but no stanza could have more than a single line. The declaration for **<poem>** in the example above thus states that a **<poem>** cannot have more than one title, but may have none, and that it must have at least one **<stanza>** and may have several.

2.4.5 Group Connectors

The content model "(TITLE?, STANZA+)" contains more than one component, and thus needs additionally to specify the order in which these elements (**<title>** and **<stanza>**) may appear. This ordering is determined by the *group connector* (the comma) used between its components. There are three possible group connectors, conventionally represented by comma, vertical bar, and ampersand.[6] The comma means that the components it connects must both appear in the order specified by the content model. The ampersand indicates that the components it connects must both appear but may appear in any order. The vertical bar indicates that only one of the components it connects may appear. If the comma in this example were replaced by an ampersand, a title could appear either before the stanzas of a **<poem>** or at the end (but not between stanzas). If it were replaced by a vertical bar, then a **<poem>** would consist of either a title or just stanzas—but not both!

---

[6] Like the occurrence indicators, these are assigned formal names by the standard and may be redefined with an appropriate SGML declaration.

2.4.6 Model Groups

In our example so far, the components of each content model have been either single elements or #PCDATA. It is quite permissible however to define content models in which the components are lists of elements, combined by group connectors. Such lists, known as *model groups,* may also be modified by occurrence indicators and themselves combined by group connectors. To demonstrate these facilities, let us now expand our example to include non-stanzaic types of verse. For the sake of demonstration, we will categorize poems as one of *stanzaic, couplets,* or *blank* (or *stichic*). A blank-verse poem consists simply of lines (we ignore the possibility of verse paragraphs for the moment)[7] so no additional elements need be defined for it. A couplet is defined as a **<line1>** followed by a **<line2>**.

```
<!ELEMENT couplet O O (line1, line2) >
```

The elements **<line1>** and **<line2>** (which are distinguished to enable studies of rhyme scheme, for example) have exactly the same content model as the existing **<line>** element. They can therefore share the same declaration. In this situation, it is convenient to supply a *name group* as the first component of a single element declaration, rather than give a series of declarations differing only in the names used. A name group is a list of GIs connected by any group connector and enclosed in parentheses, as follows:

```
<!ELEMENT (line | line1 | line2) O O (#PCDATA) >
```

The declaration for the **<poem>** element can now be changed to include all three possibilities:

```
<!ELEMENT poem - O (title?, (stanza+ | couplet+ | line+) ) >
```

That is, a poem consists of an optional title, followed by several stanzas, or several couplets, or several lines. Note the difference between this definition and the following:

```
<!ELEMENT poem - O (title?, (stanza | couplet | line)+ ) >
```

The second version, by applying the occurrence indicator to the group rather than to each element within it, would allow for a single poem to contain a mixture of stanzas, couplets or blank verse.

Quite complex models can easily be built up in this way, to match the structural complexity of many types of text. As a further example, consider the case of stanzaic verse in which a refrain or chorus appears. A refrain may be composed of repetitions of the line element, or it may simply be text, not divided into verse lines. A refrain can appear at the start of a poem only, or as an optional addition following each stanza. This could be expressed by a content model such as the following:

---

[7] It will not have escaped the astute reader that the fact that verse paragraphs need not start on a line boundary seriously complicates the issue; see further section 2.5.2, "Concurrent Structures," on page 14.

```
<!ELEMENT refrain - - (#PCDATA | line+)>
<!ELEMENT poem    - O (title?,
                      ( (line+)
                      | (refrain?, (stanza, refrain?)+ ) )) >
```

That is, a poem consists of an optional title, followed by either a sequence of lines, or an un-named group, which starts with an optional refrain, followed by one of more occurrences of another group, each member of which is composed of a stanza followed by an optional refrain. A sequence such as “refrain - stanza - stanza - refrain” follows this pattern, as does the sequence “stanza - refrain - stanza - refrain.” The sequence “refrain - refrain - stanza - stanza” does not, however, and neither does the sequence “stanza - refrain - refrain - stanza.” Among other conditions made explicit by this content model are the requirements that at least one stanza must appear in a poem, if it is not composed simply of lines, and that if there is both a title and a stanza they must appear in that order.

2.5 Complicating the Issue: More on Element Declarations

In the simple cases described so far, it has been assumed that one can identify the immediate constituents of every element defined in a textual structure. A poem consists of stanzas, and an anthology consists of poems; stanzas do not float around unattached to poems or combined into some other unrelated element; a poem cannot contain an anthology. All the elements of a given document type may be arranged into a hierarchic structure, arranged like a family tree with a single ancestor at the top and many children (mostly the elements containing #PCDATA) at the bottom. This gross simplification turns out to be surprisingly effective for a large number of purposes. It is not however adequate for the full complexity of real textual structures. In particular, it does not cater for the case of more or less freely floating elements that can appear at almost any hierarchic level in the structure, and it does not cater for the case where different elements overlap or several different trees may be identified in the same document. To deal with the first case, SGML provides the *exception* mechanism; to deal with the second, SGML permits the definition of “concurrent” document structures.

2.5.1 Exceptions to the Content Model

In most documents, there will be some elements that can occur at any level of its structure. Annotations, for example, might be attached to the whole of a poem, to a stanza, to a line of a stanza or to a single word within it. In a textual critical edition, the same might be true of variant readings. In this simple case, the complexity of adding an annotation element as an optional component of every content model is not particularly onerous; in a more realistically complex model perhaps containing some ten or twenty levels such an approach can become much more difficult.

To cope with this, SGML allows for any content model to be further modified by means of an *exception* list. There are two types of exception: *inclusions,* that is, additional elements that can be included at any point in the model group or any of its constituent elements; and *exclusions,* that is, elements that cannot be included within the current model.

To extend our declarations further to allow for annotations and variant readings, which we will assume can appear anywhere within the text of a poem, we first need to add declarations for these two elements:

```
<!ELEMENT (note | variant) - - (#PCDATA)>
```

The note and variant elements must have both start- and end-tags, since they can appear anywhere. Rather than add them to the content model for each type of poem, we can add them in the form of an inclusion list to the poem element, which now reads:

```
<!ELEMENT poem - O (title?, (stanza+ | couplet+ | line+) )
                                        +(note | variant) >
```

The plus sign at the start of the (NOTE | VARIANT) name list indicates that this is an inclusion exception. With this addition, notes or variants can appear at any point in the content of a poem element—even those (such as **<title>**) for which we have defined a content model of #PCDATA. They can thus also appear within notes or variants!

If we wanted for some reason to prevent notes or variants appearing within titles, we could add an exclusion exception to the declaration for **<title>** above:

```
<!ELEMENT title  - O  (#PCDATA)  -(note | variant) >
```

The minus sign at the start of the (NOTE | VARIANT) name list indicates that this is an exclusion exception. With this addition, notes and variants will be prohibited from appearing within titles, notwithstanding their potential inclusion implied by the previous addition to the content model for **<poem>**.

In the same way, we could prevent notes and variants from nesting within notes and variants by modifying the definition above to read

```
<!ELEMENT (note | variant) - - (#PCDATA)  -(note | variant) >
```

The meticulous reader will note that this precludes both variants within notes and notes within variants. Inclusion and exclusion exceptions should be used with care as their ramifications may not be immediately apparent.

2.5.2 Concurrent Structures

All the structures we have so far discussed have been simply hierarchic: that is, at every level of the tree, each node is entirely contained by a parent node. The figure below represents the structure of a document conforming to the simple DTD we have so far defined as a tree (drawn on its side through exigencies of space). We have already seen how Blake's poem can be divided into a title and two stanzas, each of four lines. In this diagram, we add a second poem, consisting of and a title, to make up an instance of an anthology:

```
                         |-------------------title
                         |
                         |              |----line1
                         |              |----line2
          |------POEM1---|----stanza1---|----line3
          |              |              |----line4
          |              |
          |              |              |----line5
          |              |----stanza2---|----line6
          |                             |----line7
          |                             |----line8
anthology-|
          |
          |              |-------------------title
          |              |
          |              |              |----line1
          |              |              |----line2
          |------POEM2---|----stanza1---|----line3
                                        |----line4
                                        |----line5
```

Clearly, there are many such trees that might be drawn to describe the structure of this or other anthologies. Some of them might be representable as further subdivisions of this tree: for example, we might subdivide the lines into individual words, since no word crosses a line boundary. But equally clearly, there are many other trees that might be drawn which do *not* fit within this tree. We might, for example be interested in syntactic structures which rarely respect the formal boundaries of verse. Or, to take a simpler example, we might want to represent the pagination of different editions of the same text.

One way of doing this would be to group the lines and titles of our current model into pages. A declaration for such an element is simple enough:

```
<!ELEMENT page - - ((title?, line+)+)   >
```

That is, a page consists of one or more unnamed groups, each of which contains an optional title, followed by a sequence of lines. (Note, incidentally, that this model prohibits a title appearing on its own at the foot of a page). However, simply inserting the element **<page>** into the hierarchy already defined is not as easy as it might seem. Some poems are longer than a single page, and other pages contain more than one poem. We cannot therefore insert the element **<page>** between **<anthology>** and **<poem>** in the hierarchy, nor can it go between **<poem>** and **<stanza>**, nor yet in both places at once! What is needed is the ability to create a separate hierarchy, with the same elements at the bottom (the stanzas, lines and titles), but combined into a different superstructure. This is the ability which the CONCUR feature of SGML gives.

A separate document type definition must be created for each hierarchic tree into which the text is to be structured. The definition we have so far built up for the anthology looks, in full, like this:

```
<!DOCTYPE anthology [
<!ELEMENT anthology      - -  (poem+)             >
<!ELEMENT poem           - -  (title?, stanza+)   >
<!ELEMENT stanza         - O  (line+)             >
<!ELEMENT (title | line) - O  (#PCDATA)           >
]>
```

As this example shows, the name of a document type must always be the same as the name of the largest element in it, that is the element at the top of the hierarchy. The syntax used is discussed further below (see section 2.8.2, “The DTD,” on page 23). Let us now add to this declaration a second definition for a concurrent document type, which we will call a paged anthology, or **<p.anth>** for short:

```
<!DOCTYPE p.anth [
<!ELEMENT p.anth         - -  (page+)               >
<!ELEMENT page           - -  ((title?, line+)+)    >
<!ELEMENT (title|line)   - O  (#PCDATA)             >
]>
```

We have now defined two different ways of looking at the same basic text—the PCDATA components grouped by both these document type definitions into lines or titles. In one view, the lines are grouped into stanzas and poems; in the other they are grouped into pages only. Notice that it is exactly the same text which is visible in both views: the two hierarchies simply allow us to arrange it in two different ways.

To mark up the two views, it will be necessary to indicate which hierarchy each element belongs to. This is done by including the name of the document type (the view) within parentheses immediately before the identifier concerned, inside both start- and end-tags. Thus, pages (which are only visible in the **<p.anth>** document type) must be tagged with a **<(p.anth)page>** tag at their start and a **</(p.anth)page>** at their end. In the same way, as poems and stanzas appear only in the **<anthology>** document type, they must now be tagged using **<(anthology)poem>** and **<(anthology)stanza>** tags respectively. For the line and title elements, however, which appear in both hierarchies, no document type specification need be given: any tag containing only a name is assumed to mark an element present in every active document type.

As a simple example, let us assume that Blake's poem appears in some paged anthology, with the page break occurring half way through the first stanza. The poem might then be marked up as follows:

```
<(anthology)anthology>
<(p.anth)p.anth>
<(p.anth)page>

<!--      other titles and lines on this page here -->

     <(anthology)poem><title>The SICK ROSE
     <(anthology)stanza>
          <line>O Rose thou art sick.
          <line>The invisible worm,
</(p.anth)page>
<(p.anth)page>
```

```
          <line>That flies in the night
          <line>In the howling storm:
     <(anthology)stanza>
          <line>Has found out thy bed
          <line>Of crimson joy:
          <line>And his dark secret love
          <line>Does thy life destroy.
     </(anthology)poem>

<!--      rest of material on this page here    -->
</(p.anth)page>

</(p.anth)p.anth)
</(anthology)anthology>
```

It is now possible to select only the elements concerned with a particular view from the text, even though both are represented in the tagging. A processor concerned only with the pagination will see only those elements whose tags include the P.ANTH specification, or which have no specification at all. A processor concerned only with the ANTHOLOGY view of things will not see the page breaks. And a processor concerned to inter-relate the two views can do so unambiguously.

A note of caution is appropriate: CONCUR is an optional feature of SGML, and not all available SGML software systems support it, while those which do, do not always do so according to the letter of the standard. For that reason, if for no other, wherever these Guidelines have identified a potential application of CONCUR, they also invariably suggest alternative methods as well. For fuller discussion of these issues, see chapter 34, “Markup for Non-Hierarchical Phenomena,” [in separate fascicle].

Note also that we cannot introduce a new element, a page number for example, into the **<p.anth>** document type, since there is no existing data in the **<anthology>** document type which could be fitted into it. One way of adding that extra information is discussed in the next section.

2.6 Attributes

In the SGML context, the word “attribute,” like some other words, has a specific technical sense. It is used to describe information which is in some sense descriptive of a specific element occurrence but not regarded as part of its content. For example, you might wish to add a “status” attribute to occurrences of some elements in a document to indicate their degree of reliability, or to add an “identifier” attribute so that you could refer to particular element occurrences from elsewhere within a document. Attributes are useful in precisely such circumstances.

Although different elements may have attributes with the same name, (for example, in the TEI scheme, every element is defined as having an *id* attribute), they are always regarded as different, and may have different values assigned to them. If an element has been defined as having attributes, the attribute values are supplied in the document instance as *attribute-value pairs* inside the start-tag for the element occurrence. An end-tag may not contain an attribute-value specification, since it would be redundant.

For example

```
<poem id=P1 status="draft"> ... </poem>
```

The **<poem>** element has been defined as having two attributes: *id* and *status.* For the instance of a **<poem>** in this example, represented here by an ellipsis, the *id* attribute has the value "P1" and the *status* attribute has the value "draft." An SGML processor can use the values of the attributes in any way it chooses; for example, a formatter might print a poem element which has the status attribute set to "draft" in a different way from one with the same attribute set to "revised"; another processor might use the same attribute to determine whether or not poem elements are to be processed at all. The *id* attribute is a slightly special case in that, by convention, it is always used to supply a unique value to identify a particular element occurrence, which can be used for cross reference purposes, as discussed further below.

Like elements, attributes are declared in the SGML document type declaration, using rather similar syntax. As well as specifying its name and the element to which it is to be attached, it is possible to specify (within limits) what kind of value is acceptable for an attribute and a default value.

The following declarations could be used to define the two attributes we have specified above for the **<poem>** element:

```
<!ATTLIST poem
          id       ID                                #IMPLIED
          status   (draft | revised | published)    draft        >
```

The declaration begins with the symbol "ATTLIST," which introduces an *attribute list specification*. The first part of this specifies the element (or elements) concerned. In our example, attributes have been declared only for the **<poem>** element. If several elements share the same attributes, they may all be defined in a single declaration; just as with element declarations, several names may be given in a parenthesized list. Following this name (or list of names), is a series of rows, one for each attribute being declared, each containing three parts. These specify the name of the attribute, the type of value it takes, and a default value respectively.

Attribute names ("id" and "status" in this example) are subject to the same restrictions as other names in SGML; they need not be unique across the whole DTD, however, but only within the list of attributes for a given element.

The second part of an attribute specification can take one of two forms, both illustrated above. The first case uses one of a number of special keywords to declare what kind of value an attribute may take. In the example above, the special keyword *ID* is used to indicate that the attribute ID will be used to supply a unique identifying value for each poem instance (see further the discussion below). Among other possible SGML keywords are

CDATA: The attribute value may contain any valid character data, including tags

IDREF: The attribute value must contain a pointer to some other element (see further the discussion of ID below)

NMTOKEN: The attribute value is a *name token,* that is, (more or less) any string of alphanumeric characters
NUMBER: The attribute value is composed only of numerals

In the example above, a list of the possible values for the “status” attribute has been supplied. This means that a parser can check that no **<poem>** is defined for which the *status* attribute does not have one of “draft,” “revised,” or “published” as its value. Alternatively, if the declared value had been either CDATA or NAME, a parser would have accepted almost any string of characters (status=awful or status=12345678 if it had been a NMTOKEN; status="anything goes" or status = "well, *almost* anything" if it were CDATA). Sometimes, of course, the set of possible values cannot be pre-defined. Where it can, as in this case, it is generally better to do so.

The last piece of each information in each attribute definition specifies how a parser should interpret the absence of the attribute concerned. This can be done by supplying one of the special keywords listed below, or (as in this case) by supplying a specific value which is then regarded as the value for every element which does not supply a value for the attribute concerned. Using the example above, if a poem is simply tagged **<poem>**, the parser will treat it exactly as if it were tagged **<poem status=draft>**. Alternatively, one of the following keywords may be used to specify a default value for an attribute:

#REQUIRED: A value must be specified.
#IMPLIED: A value need not be supplied (as in the case of *ID* above).
#CURRENT: If no value is supplied in this element occurrence, the last specified value should be used.

For example, if the attribute definition above were rewritten as

```
<!ATTLIST poem
          id       ID                               #IMPLIED
          status   (draft | revised | published)    #CURRENT      >
```

then poems which appear in the anthology simply tagged **<poem>** would be treated as if they had the same status as the preceding poem. If the keyword were #REQUIRED rather than #CURRENT, the parser would report such poems as erroneously tagged, as it would if any value other than “draft,” “published,” or “revised” were supplied. The use of #CURRENT implies that whatever value is specified for this attribute on the first poem will apply to all subsequent poems, until altered by a new value. Only the status of the first poem need therefore be supplied, if all are the same.

It is sometimes necessary to refer to an occurrence of one textual element from within another, an obvious example being phrases such as “see note 6” or “as discussed in chapter 5.” When a text is being produced the actual numbers associated with the notes or chapters may not be certain. If we are using descriptive markup, such things as page or chapter numbers, being entirely matters of presentation will not in any case be present in the marked up text: they will be assigned by whatever processor is operating on the text (and may indeed differ in different applications). SGML therefore provides a special mechanism by which any element occurrence may be given a special identifier, a kind of label, which may be used to refer to it from anywhere else within the same text. The cross-reference itself is regarded as an element occurrence of a specific kind, which must also be declared in the DTD. In each case, the identifying label (which may be arbitrary) is supplied as the value of a special attribute.

Suppose, for example, we wish to include a reference within the notes on one poem that refers to another poem. We will first need to provide some way of attaching a label to each poem: this is done by defining an attribute for the **<poem>** element, as suggested above.

```
<!ATTLIST poem
          id       ID     #IMPLIED >
```

Here we define an attribute *id*, the value of which must be of type ID. It is not required that any attribute of type ID have the name “ID” as well; it is however a useful convention almost universally observed. Note that not every poem need carry an *id* attribute and the parser may safely ignore the lack of one in those which do not. Only poems to which we intend to refer need use this attribute; for each such poem we should now include in its start-tag some unique identifier, for example:

```
<POEM ID=Rose>
     Text of poem with identifier 'ROSE'
</POEM>

<POEM ID=P40>
     Text of poem with identifier 'P40'
</POEM>

<POEM>
     This poem has no identifier
</POEM>
```

Next we need to define a new element for the cross reference itself. This will not have any content—it is only a pointer—but it has an attribute, the value of which will be the identifier of the element pointed at. This is achieved by the following declarations:

```
<!ELEMENT poemref - O EMPTY                  >
<!ATTLIST poemref     target IDREF #REQUIRED >
```

The **<poemref>** element needs no end-tag because it has no content. It has a single attribute called *target*. The value of this attribute must be of type IDREF (the keyword used for cross reference pointers of this type) and it must be supplied.

With these declarations in force, we can now encode a reference to the poem with id “Rose” as follows:

```
Blake's poem on the sick rose <POEMREF TARGET=Rose> ...
```

When an SGML parser encounters this empty element it will simply check that a poem exists with the identifier “Rose”. Different SGML processors could take any number of additional actions: a formatter might construct an exact page and line reference for the location of the poem in the current document and insert it, or just quote the poem's title or first lines. A hypertext style processor might use this element as a signal to activate a link to the poem being referred to. The purpose of the SGML markup is simply to indicate that a cross reference exists: it does not determine what the processor is to do with it.

2.7 SGML Entities

The aspects of SGML discussed so far are all concerned with the markup of structural elements within a document. SGML also provides a simple and flexible method of encoding and naming arbitrary parts of the actual content of a document in a portable way. In SGML the word *entity* has a special sense: it means a named part of a marked up document, irrespective of any structural considerations. An entity might be a string of characters or a whole file of text. To include it in a document, we use a construction known as an *entity reference*. For example, the following declaration

```
<!ENTITY tei "Text Encoding Initiative">
```

defines an entity whose name is “tei”[8] and whose value is the string “Text Encoding Initiative.” This is an instance of a *general entity declaration*; so is the following *system entity declaration*:

```
<!ENTITY ChapTwo SYSTEM "sgmlmkup.txt">
```

This defines a system entity whose name is “ChapTwo” and whose value is the text associated with the system identifier—in this case, the system identifier is the name of an operating system file and the replacement text of the entity is the contents of the file.

Once an entity has been declared, it may be referenced anywhere within a document. This is done by supplying its name prefixed with the ampersand character and followed by the semicolon. The semicolon may be omitted if the entity reference is followed by a space or record end.

When an SGML parser encounters such an *entity reference,* it immediately substitutes the value declared for the entity name. Thus, the passage “The work of the &tei has only just begun” will be interpreted by an SGML processor exactly as if it read “The work of the Text Encoding Initiative has only just begun.” In the case of a system entity, it is, of course, the contents of the operating system file which are substituted, so that the passage “The following text has been suppressed: &ChapTwo;” will be expanded to include the whole of whatever the system finds in the file sgmlmkup.txt.

This obviously saves typing, and simplifies the task of maintaining consistency in a set of documents. If the printing of a complex document is to be done at many sites, the document body itself might use an entity reference, such as “&site;”, wherever the name of the site is required. Different entity declarations could then be added at different sites to supply the appropriate string to be substituted for this name, with no need to change the text of the document itself.

---

[8] By convention case is significant in entity names, unlike element names.

This *string substitution* mechanism has many other applications. It can be used to circumvent the notorious inadequacies of many computer systems for representing the full range of graphic characters needed for the display of modern English (let alone the requirements of other modern scripts or of ancient languages). So-called “special characters” not directly accessible from the keyboard (or if accessible not correctly translated when transmitted) may be represented by an entity reference.

Suppose, for example, that we wish to encode the use of ligatures in early printed texts. The ligatured form of “ct” might be distinguished from the non-ligatured form by encoding it as “&ctlig;” rather than “ct.” Other special typographic features such as leafstops or rules could equally well be represented by mnemonic entity references in the text. When processing such texts, an entity declaration would be added giving the desired representation for such textual elements. If, for example, ligatured letters are of no interest, we would simply add a declaration such as

```
<!ENTITY ctlig "ct" >
```

and the distinction present in the source document would be removed. If, on the other hand, a formatting program capable of representing ligatured characters is to be used, we might replace the entity declaration to give whatever sequence of characters such a program requires as the expansion.

A list of entity declarations is known as an *entity set*. Standard entity sets are provided for use with most SGML processors, in which the names used will normally be taken from the lists of such names published as an annex to the SGML standard and elsewhere, as mentioned above.

The replacement values given in an entity declaration are, of course, highly system dependent. If the characters to be used in them cannot be typed in directly, SGML provides a mechanism to specify characters by their numeric values, known as *character references*. A character reference is distinguished from other characters in the replacement string by the fact that it begins with a special symbol, conventionally the sequence “&#”, and ends with the normal semicolon. For example, if the formatter to be used represents the ligatured form of ct by the characters c and t prefixed by the character with decimal value 102, the entity declaration would read:

```
<!ENTITY ctlig "&#102;ct" >
```

Note that character references will generally not make sense if transferred to another hardware or software environment: for this reason, their use is only recommended in situations like this.

Useful though the entity reference mechanism is for dealing with occasional departures from the expected character set, no one would consider using it to encode extended passages, such as quotations in Greek or Russian in an English text. In such situations, different mechanisms are appropriate. These are discussed elsewhere in these Guidelines (see chapter 4, “Characters and Character Sets,” [in separate fascicle]).

A special form of entities, *parameter entities,* may be used within SGML markup declarations; extensive examples may be found in chapter 3, “Structure of the TEI Document Type Definitions,” [in separate fascicle].

2.8 Putting It All Together

An SGML conformant document has a number of parts, not all of which have been discussed in this chapter, and many of which the user of these Guidelines may safely ignore. For completeness, the following summary of how the parts are interrelated may however be found useful.

An SGML document consists of an SGML *prolog* and a *document instance*. The prolog contains an *SGML declaration* (described below) and a *document type declaration,* which contains element and entity declarations such as those described above. Different software systems may provide different ways of associating the document instance with the prolog; in some cases, for example, the prolog may be “hard-wired” into the software used, so that it is completely invisible to the user.

2.8.1 The SGML Declaration

The SGML declaration specifies basic facts about the dialect of SGML being used such as the character set, the codes used for SGML delimiters, the length of identifiers, etc. Its content for TEI-conformant document types is discussed further in chapters 39, “Formal Grammar for the TEI-Interchange-Format Subset of SGML,” [in separate fascicle] and 30, “TEI Conformance,” [in separate fascicle]. Normally the SGML declaration will be held in the form of compiled tables by the SGML processor and will thus be invisible to the user.

2.8.2 The DTD

The document type declaration specifies the document type definition against which the document instance is to be validated. Like the SGML declaration it may be held in the form of compiled tables within the SGML processor, or associated with it in some way which is invisible to the user, or requires only that the name of the document type be specified before the document is validated.

At its simplest the document type declaration consists simply of a base document type definition (possibly also one or more concurrent document type definitions) which is prefixed to the document instance. For example:

```
<!DOCTYPE my.dtd [
    <!-- all declarations for MY.DTD go here -->
    ...
]>
<my.dtd>
    This is an instance of a MY.DTD type document
</my.dtd>
```

More usually, the document type definition will be held in a separate file and invoked by reference, as follows:

```
<!DOCTYPE tei system "tei.dtd" [
]>
<tei>
     This is an instance of an unmodified TEI type document
</tei>
```

Here, the text of the TEI.1 document type declaration is not given explicitly, but the SGML processor is told that it may be read from the file with the system identifier given in quotation marks. The square brackets may still be supplied, as in this example, even though they contain nothing.

The part enclosed by square brackets is known as the *document type subset*. Its purpose is to specify any modification to be made to the DTD being invoked, thus:

```
<!DOCTYPE TEI SYSTEM "tei.dtd" [
     <!ENTITY tla "Three Letter Acronym">
     <!ELEMENT my.tag - - (#PCDATA)>
     <!-- any other special-purpose declarations or
          re-definitions go in here -->
]>
<tei>
     This is an instance of a modified TEI.1 type document,
     which may contain <my.tag>my special tags</my.tag> and
     references to my usual entities such as &tla;.
</tei.1>
```

In this case, the document type definition in force is the concatenation of the document type subset and the contents of the file specified. The order is important, because in SGML only the first declaration of an entity counts. In the above example, therefore, if the document type definition also declared an entity called “tla” the declaration within brackets would take precedence. (This does not apply to elements, which may not be declared more than once; if a declaration for **<my.tag>** already existed in the document type definition, the above example would be flagged as erroneous). Combining and extending document type definitions is discussed further in chapter 3, “Structure of the TEI Document Type Definitions,” [in separate fascicle].

2.8.3 The Document Instance

The document instance is the content of the document itself. It contains only text, markup and general entity references, and thus may not contain any new declarations. A convenient way of building up large documents in a modular fashion might be to use a DTD like the following:

```
<!DOCTYPE tei [
     <!ENTITY chap1 system "chap1.txt">
     <!ENTITY chap2 system "chap2.txt">
     <!ENTITY chap3 "-- not yet written --">
]>
<tei.1>
     Some text to go before chapter 1
     &chap1;
     A bit of text meant to go between chapters 1 and 2.
```

```
        &chap2;
        &chap3;
        ...
    </tei.1>
```

In this example, a DTD called TEI (the example assumes that this is available without formality to the processor being used) has been extended by entity declarations for each chapter of the work. The first two are system entities referring to the file in which the text of particular chapters is to be found; the third a dummy, indicating that the text does not yet exist (alternatively, an entity with a null value could be used). In the document instance, the entity references &chap1; etc. will be resolved by the parser to give the required contents. The chapter files themselves will not, of course, contain any markup declarations—just tagged text.

2.9 Using SGML

A variety of software is available to assist in the tasks of creating, validating and processing SGML documents. Only a few basic types can be described here. At the heart of most such software is an SGML *parser*: that is, a piece of software which can take a document type definition and generate from it a software system capable of validating any document invoking that DTD. Output from a parser, at its simplest, is just "yes" (the document instance is valid) or "no" (it is not). Most parsers will however also produce a new version of the document instance in *canonical form* (typically with all end-tags supplied and entity references resolved) or formatted according to user specifications. This form can then be used by other pieces of software (loosely or tightly coupled with the parser) to provide additional functions, such as structured editing, formatting and database management.

A *structured editor* is a kind of intelligent word-processor. It can use information extracted from a processed DTD to prompt the user with information about which elements are required at different points in a document as the document is being created. It can also greatly simplify the task of preparing a document, for example by inserting tags automatically.

A *formatter* operates on a tagged document instance to produce a printed form of it. Many typographic distinctions, such as the use of particular typefaces or sizes, are intimately related to structural distinctions, and formatters can thus usefully take advantage of descriptive markup. It is also possible to define the tagging structure expected by a formatting program in SGML terms, as a concurrent document structure.

Text-oriented database management systems typically use inverted file indexes to point into documents, or subdivisions of them. A search can be made for an occurrence of some word or word pattern within a document or within a subdivision of one. Meaningful subdivisions of input documents will of course be closely related to the subdivisions specified using descriptive markup. It is thus simple for textual database systems to take advantage of SGML-tagged documents. Much research work is also currently going into ways of extending the capabilities of existing (non-text) database systems to take advantage of the structuring information made explicit by SGML markup.

Hypertext systems improve on other methods of handling text by supporting associative links within and across documents. Again, the basic building block needed for such systems is also a basic building block of SGML markup: the ability to identify and to link together individual document elements comes free as a part of the SGML way of doing things. By tagging links explicitly, rather than using proprietary software, developers of hypertexts can be sure that the resources they create will continue to be useful. To load an SGML document into a hypertext system requires only a processor which can correctly interpret SGML tags such as those discussed in section 16, "Segmentation and Alignment," [in separate fascicle].

Usability Report and Comment

If you have comments of any kind on the current draft of the TEI Guidelines, please complete this form and return it to one of the following addresses:

C. M. Sperberg-McQueen, Computer Center (M/C 135), University of Illinois at Chicago, 1940 W. Taylor Room 124, Chicago IL 60612-7352, USA. Bitnet: U35395@UICVM, Internet: U35395@uicvm.uic.edu

Lou Burnard, Oxford University Computing Service, 13 Banbury Road, Oxford OX2 6NN, UK. Fax: +44 (865) 273275, Janet: LOU@UK.AC.OXFORD.VAX Internet: LOU@vax.oxford.ac.uk

Further copies of this form may be downloaded from LISTSERV @ UICVM (or listserv @ uicvm.uic.edu). Send a mail message containing the line 'GET TEIURC MEMO TEI-L'

Your name:
Your postal address:

Your e-mail address (if any):
Your occupation:
Your academic background:
Your immediate reactions to the Guidelines (please tick or cross one only):

Relevance to your present concerns or interests: high medium low
Importance to your future research plans: high medium low
Comprehensibility/usability of current draft: high medium low

Detailed technical comment: [Please feel free to comment on any part of the Guidelines or in general. Comments relating to specific parts of the text are most helpful if they include a section or chapter number]

Thank you!

Temporary Index