International Standard 8879

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION•МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ•ORGANISATION INTERNATIONALE DE NORMALISATION

Information processing — Text and office systems — Standard Generalized Markup Language (SGML)

Traitement de l'information — Systèmes bureautiques — Langage standard généralisé de balisage (SGML)

First edition — 1986-10-15

UDC 681.3.06 **Ref. No. ISO 8879-1986 (E)**

Descriptors : data processing, documentation, logical structure, programming (computers), artificial languages, programming languages.

Price based on 155 pages

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75 % approval by the member bodies voting.

International Standard ISO 8879 was prepared by Technical Committee ISO/TC 97, *Information processing systems*.

Users should note that all International Standards undergo revision from time to time and that any reference made herein to any other International Standard implies its latest edition, unless otherwise stated.

Printed in Switzerland

Contents

Annexes

Figures

This page intentionally left blank

Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML)

0 Introduction

This International Standard specifies a language for document representation referred to as the "Standard Generalized Markup Language" (SGML). SGML can be used for publishing in its broadest definition, ranging from single medium conventional publishing to multi-media data base publishing. SGML can also be used in office document processing when the benefits of human readability and interchange with publishing systems are required.

0.1 Background

A document can be viewed in the abstract as a structure of various types of element. An author organizes a book into chapters that contain paragraphs, for example, and figures that contain figure captions. An editor organizes a magazine into articles that contain paragraphs that contain words, and so on.

Processors treat these elements in different ways. A formatting program might print headings in a prominent type face, leave space between paragraphs, and otherwise visually convey the structure and other attributes to the reader. An information retrieval system would perhaps assign extra significance to words in a heading when creating its dictionary.

Although this connection between a document's attributes and its processing now seems obvious, it tended to be obscured by early text processing methods. In the days before automated typesetting, an editor would "mark up" a manuscript with the specific processing instructions that would create the desired format when executed by a compositor. Any connection between the instructions and the document's structure was purely in the editor's head.

Early computerized systems continued this approach by adding the process-specific "markup" to the machine-readable document file. The markup still consisted of specific processing instructions, but now they were in the language of a formatting program, rather than a human compositor. The file could not easily be used for a different purpose, or on a different computer system, without changing all the markup.

As users became more sophisticated, and as text processors became more powerful, approaches were developed that alleviated this problem. "Macro calls" (or "format calls") were used to identify points in the document where processing was to occur. The actual processing instructions were kept outside of the document, in "procedures" (or "macro definitions" or "stored formats"), where they could more easily be changed.

While the macro calls could be placed anywhere in a document, users began to recognize that most were placed at the start or end of document elements. It was natural, therefore, to choose names for such macros that were "generic identifiers" of the element types, rather than names that suggested particular processing (for example, "heading" rather than "format-17"), and so the practice of "generic coding" (or "generalized tagging") began.

Generic coding was a major step towards making automated text processing systems reflect the natural relationship between document attributes and processing. The advent of "generalized markup languages" in the early 1970's carried this trend further by providing a formal language base for generic coding. A generalized markup language observes two main principles:

a) Descriptive markup predominates and is distinguished from processing instructions.

Descriptive markup includes both generic identifiers and other attributes of document elements that motivate processing instructions. The processing instructions, which can be in any language, are normally collected outside of the document in procedures.

As the source file is scanned for markup and the various elements are recognized, the processing system executes the procedures associated with each element and attribute for that process. For other processes, different procedures can be associated with the same elements and attributes without changing the document markup.

When a processing instruction must be entered directly in a document, it is delimited differently from descriptive markup so that it can easily be located and changed for different processes.

b) Markup is formally defined for each type of document.

A generalized markup language formalizes document markup by incorporating "document type definitions". Type definitions include a specification (like a formal grammar) of which elements and attributes can occur in a document and in what order. With this information it is possible to determine whether the markup for an individual document is correct (that is, complies with the type definition) and also to supply markup that is missing, because it can be inferred unambiguously from other markup that is present.

NOTE — A more detailed introduction to the concepts of generic coding and the Standard Generalized Markup Language can be found in annex A.

0.2 Objectives

The Standard Generalized Markup Language standardizes the application of the generic coding and generalized markup concepts. It provides a coherent and unambiguous syntax for describing whatever a user chooses to identify within a document. The language includes:

— An "abstract syntax" for descriptive markup of document elements.

— A "reference concrete syntax" that binds the abstract syntax to particular delimiter characters and quantities. Users can define alternative concrete syntaxes to meet their requirements.

— Markup declarations that allow the user to define a specific vocabulary of generic identifiers and attributes for different document types.

— Provision for arbitrary data content. In generalized markup, "data" is anything that is not defined by the markup language. This can include specialized "data content notations" that require interpretation different from general text: formulas, images, non-Latin alphabets, previously formatted text, or graphics.

— Entity references: a non-system-specific technique for referring to content located outside the mainstream of the document, such as separately-written chapters, pi characters, photographs, etc.

— Special delimiters for processing instructions to distinguish them from descriptive markup. Processing instructions can be entered when needed for situations that cannot be handled by the procedures, but they can easily be found and modified later when a document is sent to a different processing system.

For a generalized markup language to be an acceptable standard, however, requires more than just providing the required functional capabilities. The language must have metalinguistic properties, in order to satisfy the constraints imposed by the need to use it in a multiplicity of environments. The major constraints, and the means by which the Standard Generalized Markup Language addresses them, can be summarized as follows:

a) Documents "marked up" with the language must be processable by a wide range of text processing and word processing systems.

The full form of the language, with all optional features, offers generality and flexibility that can be exploited by sophisticated systems; less powerful systems need not support the features. To facilitate interchange between dissimilar systems, an "SGML declaration" describes any markup features or concrete syntax variations used in a document.

b) The millions of existing text entry devices must be supported.

SGML documents, with the reference concrete syntax, can easily be keyboarded and understood by humans, without machine assistance. As a result:

- Use of SGML need not await the development and acceptance of a new generation of hardware — just software to process the documents on existing machines.
- Migration to such a new generation (when it comes) will be easier, as users will already be familiar with SGML.

c) There must be no character set dependency, as documents might be keyed on a variety of devices.

The language has no dependency on a particular character set. Any character set that has bit combinations for letters, numerals, space, and delimiters is acceptable.

d) There must be no processing, system, or device dependencies.

Generalized markup is predominantly descriptive and therefore inherently free of such dependencies. The occasional processing instruction is specially delimited so it can be found and converted for interchange, or when a different process renders the instruction irrelevant.

References to external parts of a document are indirect. The mappings to real system storage are made in "external entity declarations" that occur at the start of the document, where they can easily be modified for interchange.

The concrete syntax can be changed with the SGML declaration to accommodate any reserved system characters.

e) There must be no national language bias.

The characters used for names can be augmented by any special national characters. Generic identifiers, attribute names, and other names used in descriptive markup are defined by the user in element and entity declarations.

The declaration names and keywords used in markup declarations can also be changed.

Multiple character repertoires, as used in multi-lingual documents, are supported.

f) The language must accommodate familiar typewriter and word processor conventions.

The "short reference" and "data tag" capabilities support typewriter text entry conventions. Normal text containing paragraphs and quotations is interpretable as SGML although it is keyable with no visible markup.

g) The language must not depend on a particular data stream or physical file organization.

The markup language has a virtual storage model in which documents consist of one or more storage entities, each of which is a sequence of characters. All real file access is handled by the processing system, which can decide whether the character sequence should be viewed as continuous, or whether it should reflect physical record boundaries.

h) "Marked up" text must coexist with other data.

A processing system can allow text that conforms to this International Standard to occur in a data stream with other material, as long as the system can locate the start and end of the conforming text.

Similarly, a system can allow data content not defined by SGML to occur logically within a conforming document. The occurrence of such data is indicated by markup declarations to facilitate interchange.

i) The markup must be usable by both humans and programs.

The Standard Generalized Markup Language is intended as a suitable interface for keyboarding and interchange without preprocessors. It allows extensive tailoring to accommodate user preferences in text entry conventions and the requirements of a variety of keyboards and displays.

However, it is recognized that many implementers will want to take advantage of the language's information capture capabilities to provide intelligent editing or to create SGML documents from a word processing front-end environment. SGML accommodates such uses by providing the following capabilities:

- Element content can be stored separately from the markup.
- Control characters can be used as delimiters.
- Mixed modes of data representation are permitted in a document.
- Multiple concurrent logical and layout structures are supported.

0.3 Organization

The organization of this International Standard is as follows:

a) The physical organization of an SGML document as an entity structure is specified in clause 6.

b) The logical organization of an SGML document as an element structure, and its representation with descriptive markup, is specified in clause 7.

c) Processing instructions are discussed in clause 8.

d) Common markup constructs, such as characters, entity references, and processing instructions, are covered in clause 9.

e) Markup declarations with general applicability (comment, entity, and marked section) are specified in clause 10.

f) Markup declarations that are used primarily to specify document type definitions (document type, element, notation, short reference mapping, and short reference use) are defined in clause 11.

g) Markup declarations that are used primarily to specify link process definitions (link type, link attribute, link set, and link set use) are defined in clause 12.

h) The SGML declaration, which specifies the document character set, capacity set, concrete syntax, and features, is defined in clause 13.

i) The reference concrete syntax is defined in clause 14.

j) Conformance of documents, applications, and systems is defined in clause 15.

There are also a number of annexes containing additional information; they are not integral parts of the body of this International Standard.

NOTE — This International Standard is a formal specification of a computer language, which may prove difficult reading for those whose expertise is in the production of documents, rather than compilers. Annexes A, B, and C discuss the main concepts in an informal tutorial style that should be more accessible to most readers. However, the reader should be aware that those annexes do not cover all SGML constructs, nor all details of those covered, and subtle distinctions are frequently ignored in the interest of presenting a clear overview.

1 Scope

This International Standard:

a) Specifies an abstract syntax known as the Standard Generalized Markup Language (SGML). The language expresses the description of a document's structure and other attributes, as well as other information that makes the markup interpretable.

b) Specifies a reference concrete syntax that binds the abstract syntax to specific characters and numeric values, and criteria for defining variant concrete syntaxes.

c) Defines conforming documents in terms of their use of components of the language.

d) Defines conforming systems in terms of their ability to process conforming documents and to recognize markup errors in them.

e) Specifies how data not defined by this International Standard (such as images, graphics, or formatted text) can be included in a conforming document.

NOTE — This International Standard does not:

a) Identify or specify "standard" document types, document architectures, or text structures.

b) Specify the implementation, architecture, or markup error handling of conforming systems.

c) Specify how conforming documents are to be created.

d) Specify the data stream, message handling system, file structure, or other physical representation in which conforming documents are stored or interchanged, or any character set or coding scheme into or from which conforming documents might be translated for such purposes.

e) Specify the data content representation or notation for images, graphics, formatted text, etc., that are included in a conforming document.

2 Field of Application

The Standard Generalized Markup Language can be used for documents that are processed by any text processing or word processing system. It is particularly applicable to:

a) Documents that are interchanged among systems with differing text processing languages.

b) Documents that are processed in more than one way, even when the procedures use the same text processing language.

Documents that exist solely in final formatted form are not within the field of application of this International Standard.

3 References

ISO 639, *Codes for the representation of names of languages.*[1]

ISO 646, *Information processing — 7-bit coded character set for information interchange.*

ISO 9069, *Information processing — SGML support facilities — SGML Document Interchange Format (SDIF).*[1]

ISO 9070, *Information processing — SGML support facilities — Registration procedures for public text.*[1]

The following references are used in conjunction with illustrative material:

ISO 2022, *Information processing — ISO 7-bit and 8-bit coded character sets — Code extension techniques.*

ISO 3166, *Codes for the representation of names of countries.*

ISO 4873, *Information processing — ISO 8-bit code for information interchange — Structure and rules for implementation.*

ISO 6937, *Information processing — Coded character sets for text communication.*

ISO 8632/2, *Information processing systems — Computer graphics — Metafile for the storage and transfer of picture description information — Part 2: Character encoding.*[1]

ISO 8632/4, *Information processing systems — Computer graphics — Metafile for the storage and transfer of picture description information — Part 4: Clear text encoding.*[1]

4 Definitions

NOTE — The typographic conventions described in 5.1 are employed in this clause.

For the purposes of this International Standard, the following definitions apply:

4.1 abstract syntax (of SGML): Rules that define how markup is added to the data of a document, without regard to the specific characters used to represent the markup.

4.2 active document type (declaration): A document type declaration with respect to which an SGML entity is being parsed.

4.3 active link type (declaration): A link type declaration with respect to which an SGML entity is being parsed.

4.4 ambiguous content model: A *content model* for which an element or character string occurring in the document instance can satisfy more than one *primitive content token* without look-ahead.

NOTE — Ambiguous content models are prohibited in SGML.

4.5 application: Text processing application.

4.6 application convention: Application-specific rule governing the text of a document in areas that SGML leaves to user choice.

NOTE — There are two kinds: content conventions and markup conventions.

4.7 application-specific information: A parameter of the *SGML declaration* that specifies information required by an application and/or its architecture.

NOTE — For example, the information could identify an architecture and/or an application, or otherwise enable a system to determine whether it can process the document.

4.8 associated element type: An element type associated with the subject of a markup declaration by its *associated element type* parameter.

4.9 attribute (of an element): A characteristic quality, other than type or content.

4.10 attribute definition: A member of an attribute definition list; it defines an attribute name, allowed values, and default value.

4.11 attribute definition list: A set of one or more attribute definitions defined by the *attribute definition list* parameter of an attribute definition list declaration.

4.12 attribute (definition) list declaration: A markup declaration that associates an attribute definition list with one or more element types.

4.13 attribute list: Attribute specification list.

4.14 attribute list declaration: Attribute definition list declaration.

4.15 attribute specification: A member of an attribute specification list; it specifies the value of a single attribute.

4.16 attribute (specification) list: Markup that is a set of one or more attribute specifications.

[1] At present at the stage of draft.

NOTE — Attribute specification lists occur in start-tags and link sets.

4.17 attribute value literal: A delimited character string that is interpreted as an *attribute value* by replacing references and ignoring or translating function characters.

4.18 available public text: Public text that is available to the general public, though its owner may require payment of a fee or performance of other conditions.

4.19 B sequence: An uninterrupted sequence of upper-case letter "B" characters; in a string assigned as a short reference, it denotes a blank sequence whose minimum length is the length of the B sequence.

4.20 base document element: A document element whose document type is the base document type.

4.21 base document type: The document type specified by the first document type declaration in a prolog.

4.22 basic SGML document: A conforming SGML document that uses the reference concrete syntax and capacity set and the SHORTTAG and OMITTAG markup minimization features.

NOTE — It also uses the SHORTREF feature by virtue of using the reference concrete syntax.

4.23 bit: Binary digit; that is, either zero or one.

4.24 bit combination: An ordered collection of bits, interpretable as a binary number.

4.25 blank sequence: An uninterrupted sequence of ***SPACE*** and/or ***SEPCHAR*** characters.

4.26 capacity: A named limit on some aspect of the size or complexity of a document, expressed as a number of points that can be accumulated for a kind of object or for all objects.

NOTE — The set of capacities is defined by the abstract syntax, but values are assigned to them by individual documents and SGML systems.

4.27 capacity set: A set of assignments of numeric values to capacity names.

NOTE — In an SGML declaration, the capacity set identifies the maximum capacity requirements of the document (its actual requirements may be lower). A capacity set can also be defined by an application, to limit the capacity requirements of documents that implementations of the application must process, or by a system, to specify the capacity requirements that it is capable of meeting.

4.28 CDATA: Character data.

4.29 CDATA entity: Character data entity.

4.30 chain of (link) processes: Processes, performed sequentially, that form a chain in which the source of the first process is an instance of the base document type, and the result of each process but the last is the source for the next. Any portion of the chain can be iterated.

NOTE — For example, a complex page makeup application could include three document types--logical, galley, and page--and two link processes--justification and castoff. The justification process would create an instance of a galley from an instance of a logical document, and the castoff process would in turn create pages from the galleys. The two processes could be iterated, as decisions made during castoff could require re-justification of the galleys at different sizes.

4.31 character: An atom of information with an individual meaning, defined by a character repertoire.

NOTES

1 There are two kinds: graphic character and control character.

2 A character can occur in a context in which it has a meaning, defined by markup or a data content notation, that supercedes or supplements its meaning in the character repertoire.

4.32 (character) class: A set of characters that have a common purpose in the abstract syntax, such as non-SGML characters or separator characters.

NOTE — Specific characters are assigned to character classes in four different ways:

a) explicitly, by the abstract syntax (***Special***, ***Digit***, ***LC Letter***, and ***UC Letter***);

b) explicitly, by the concrete syntax (***LCNMSTRT***, ***FUNCHAR***, ***SEPCHAR***, etc.);

c) implicitly, as a result of explicit assignments made to delimiter roles or other character classes (***DELMCHAR*** and ***DATACHAR***); or

d) explicitly, by the document character set (***NONSGML***).

4.33 character data: Zero or more characters that occur in a context in which no markup is recognized, other than the delimiters that end the *character data*. Such characters are classified as data characters because they were declared to be so.

4.34 character data entity: An entity whose text is treated as *character data* when referenced and is not dependent on a specific system, device, or application process.

4.35 character entity set: A public entity set consisting of general entities that are graphic characters.

NOTES

1 Character entities are used for characters that have no coded representation in the document character set, or that cannot be keyboarded conveniently, or to achieve device independence for characters whose bit combinations do not cause proper display on all output devices.

2 There are two kinds of character entity sets: definitional and display.

4.36 character number: A *number* that represents the base-10 integer equivalent of the coded representation of a character, obtained by treating the sequence of bit combinations as a single base-2 integer.

4.37 character reference: A reference that is replaced by a single character.

NOTE — There are two kinds: named character reference and numeric character reference.

4.38 character repertoire: A set of characters that are used together. Meanings are defined for each character, and can also be defined for control sequences of multiple characters.

NOTE — When a character occurs in a control sequence, the meaning of the sequence supercedes the meanings of the individual characters.

4.39 character set: A mapping of a character repertoire onto a code set such that each character is associated with its coded representation.

4.40 (character) string: A sequence of characters.

4.41 class: Character class.

4.42 code extension: The use of a single coded representation for more than one character, without changing the document character set.

NOTE — When multiple national languages occur in a document, graphic repertoire code extension may be useful.

4.43 code set: A set of bit combinations of equal size, ordered by their numeric values, which must be consecutive.

NOTE — For example, a code set whose bit combinations have 8 bits (an "8-bit code") could consist of as many as 256 bit combinations, ranging in value from 00000000 through 11111111 (0 through 255 in the decimal number base), or it could consist of any contiguous subset of those bit combinations.

4.44 code set position: The numeric value of a bit combination in a code set.

4.45 coded representation: The representation of a character as a sequence of one or more bit combinations of equal size.

4.46 comment: A portion of a markup declaration that contains explanations or remarks intended to aid persons working with the document.

4.47 comment declaration: A markup declaration that contains only comments.

4.48 concrete syntax (of SGML): A binding of the abstract syntax to particular delimiter characters, quantities, markup declaration names, etc.

4.49 concrete syntax parameter: A parameter of the SGML declaration that identifies the concrete syntax used in document elements and (usually) prologs.

NOTE — The parameter consists of parameters that identify the syntax-reference character set, function characters, shunned characters, naming rules, delimiter use, reserved name use, and quantitative characteristics.

4.50 conforming SGML application: An SGML application that requires documents to be conforming SGML documents, and whose documentation meets the requirements of this International Standard.

4.51 conforming SGML document: An SGML document that complies with all provisions of this International Standard.

4.52 containing element: An element within which a subelement occurs.

4.53 content: Characters that occur between the start-tag and end-tag of an element in a document instance. They can be interpreted as data, proper subelements, included subelements, other markup, or a mixture of them.

NOTE — If an element has an explicit content reference, or its declared content is "EMPTY", the content is empty. In such cases, the application itself may generate data and process it as though it were content data.

4.54 content convention: An application convention governing data content, such as a restriction on length, allowable characters, or use of upper-case and lower-case letters.

NOTE — A content convention is essentially an informal data content notation, usually restricted to a single element type.

4.55 (content) model: Parameter of an element declaration that specifies the *model group* and *exceptions* that define the allowed *content* of the element.

4.56 content model nesting level: The largest number of successive ***grpo*** or ***dtgo*** delimiters that occur in a *content model* without a corresponding ***grpc*** or ***dtgc*** delimiter.

4.57 content reference (attribute): An impliable attribute whose value is referenced by the application to generate content data.

NOTE — When an element has an explicit content reference, the element's *content* in the document instance is empty.

4.58 contextual sequence: A sequence of one or more markup characters that must follow a delimiter string within the same entity in order for the string to be recognized as a delimiter.

4.59 contextually optional element: An element

a) that can occur only because it is an inclusion; or

b) whose *content token* in the currently applicable model group is a contextually optional token.

4.60 contextually optional token: A *content token* that

a) is an inherently optional token; or

b) has a ***plus*** occurrence indicator and has been satisfied; or

c) is in a model group that is itself a contextually optional token, no tokens of which have been satisfied.

4.61 contextually required element: An element that is not a contextually optional element and

a) whose *generic identifier* is the *document type name*; or

b) whose currently applicable model token is a contextually required token.

NOTE — An element could be neither contextually required nor contextually optional; for example, an element whose currently applicable model token is in an ***or*** group that has no inherently optional tokens.

4.62 contextually required token: A *content token* that

a) is the only one in its model group; or

b) is in a ***seq*** group

i) that

— is itself a contextually required token; or

— contains a token which has been satisfied;

and

ii) all preceding tokens of which

— have been satisfied; or

— are contextually optional.

4.63 control character: A character that controls the interpretation, presentation, or other processing of the characters that follow it; for example, a tab character.

4.64 control sequence: A sequence of characters, beginning with a control character, that controls the interpretation, presentation, or other processing of the characters that follow it; for example, an escape sequence.

4.65 core concrete syntax: A variant of the reference concrete syntax that has no short reference delimiters.

4.66 corresponding content (of a content token): The element(s) and/or data in a document instance that correspond to a *content token*.

4.67 current attribute: An attribute whose current (that is, most recently specified) value is its default value.

NOTE — The start-tag cannot be omitted for the first occurrence of an element with a current attribute.

4.68 current element: The open element whose *start-tag* most recently occurred (or was omitted through markup minimization).

4.69 current link set: The link set associated with the current element by a *link set use declaration* in the element content or link type definition. If the current element has no associated link set, the previous current link set continues to be the current link set.

4.70 current map: The short reference map associated with the current element by a *short reference use* declaration in the element content or document type definition. If the current element has no associated map, the previous current map continues to be the current map.

4.71 current rank: A number that is appended to a rank stem in a tag to derive the generic identifier. For a *start-tag* it is the *rank suffix* of the most recent element with the identical *rank stem*, or a *rank stem* in the same *ranked group*. For an *end-tag* it is the *rank suffix* of the most recent open element with the identical *rank stem*.

4.72 data: The characters of a document that represent the inherent information content; characters that are not recognized as markup.

4.73 data character: An *SGML character* that is interpreted as data in the context in which it occurs, either because it was declared to be data, or because it was not recognizable as markup.

4.74 data content: The portion of an element's *content* that is data rather than markup or a subelement.

4.75 data content notation: An application-specific interpretation of an element's data content, or of a non-SGML data entity, that usually extends or differs from the normal meaning of the document character set.

NOTE — It is specified for an element's *content* by a notation attribute, and for a non-SGML data entity by the *notation name* parameter of the entity declaration.

4.76 data tag: A string that conforms to the data tag pattern of an open element. It serves both as the *end-tag* of the open element and as *character data* in the element that contains it.

4.77 data tag group: A model group token that associates a data tag pattern with a target element type.

NOTE — Within an instance of a target element, the data content and that of any subelements is scanned for a string that conforms to the pattern (a "data tag").

4.78 data tag pattern: A data tag group token that defines the strings that, if they occurred in the proper context, would constitute a data tag.

4.79 declaration: Markup declaration.

4.80 declaration subset: A delimited portion of a markup declaration in which other declarations can occur.

NOTE — Declaration subsets occur only in document type, link type, and marked section declarations.

4.81 declared concrete syntax: The concrete syntax described by the *concrete syntax* parameter of the *SGML declaration*.

4.82 dedicated data characters: Character class consisting of each *SGML character* that has no possible meaning as markup; a member is never treated as anything but a *data character*.

4.83 default entity: The entity that is referenced by a general entity reference with an undeclared name.

4.84 default value: A portion of an attribute definition that specifies the attribute value to be used if there is no *attribute specification* for it.

4.85 definitional (character) entity set: A character entity set whose purpose is to define entity names for graphic characters, but not actually to display them. Its *public identifier* does not include a *public text display version*.

NOTE — During processing, the system replaces a definitional entity set with a corresponding display character entity set for the appropriate output device.

4.86 delimiter characters: Character class that consists of each *SGML character*, other than a *name character* or *function character*, that occurs in a string assigned to a delimiter role by the concrete syntax.

4.87 delimiter-in-context: A character string that consists of a delimiter string followed immediately in the same entity by a contextual sequence.

4.88 delimiter role: A role defined by the abstract syntax, and filled by a character string assigned by the concrete syntax, that involves identifying parts of the markup and/or distinguishing markup from data.

4.89 delimiter set: A set of assignments of delimiter strings to the abstract syntax delimiter roles.

4.90 delimiter set parameter: A parameter of an SGML declaration that identifies the delimiter set used in the declared concrete syntax.

4.91 delimiter (string): A character string assigned to a delimiter role by the concrete syntax.

4.92 descriptive markup: Markup that describes the structure and other attributes of a document in a non-system-specific manner, independently of any processing that may be performed on it. In particular, it uses tags to express the element structure.

4.93 device-dependent version (of public text): Public text whose *formal public identifier* differs from that of another public text only by the addition of a *public text display version*, which identifies the display devices supported or coding scheme used.

4.94 digits: Character class composed of the 10 Arabic numerals from "0" through "9".

4.95 display (character) entity set: An entity set with the same entity names as a corresponding definitional character entity set, but which causes the characters to be displayed. It is a device-dependent version of the corresponding definitional entity set.

4.96 document: A collection of information that is processed as a unit. A document is classified as being of a particular document type.

NOTE — In this International Standard, the term almost invariably means (without loss of accuracy) an SGML document.

4.97 document architecture: Rules for the formulation of text processing applications.

NOTE — For example, a document architecture can define:

a) attribute semantics for use in a variety of element definitions;

b) element classes, based on which attributes the elements have;

c) structural rules for defining document types in terms of element classes;

d) link processes, and how they are affected by the values of attributes; and/or

e) information to accompany a document during interchange (a "document profile").

4.98 document character set: The character set used for all markup in an SGML document, and initially (at least) for data.

NOTE — When a document is interchanged between systems, its character set is translated to the receiving system character set.

4.99 document element: The element that is the outermost element of an instance of a document type; that is, the element whose *generic identifier* is the *document type name*.

4.100 document instance: Instance of a document type.

4.101 document instance set: The portion of an *SGML document entity* or *SGML subdocument entity* in the entity structure that contains one or more instances of document types. It is co-extensive with the base document element in the element structure.

NOTE — When the concurrent instance feature is used, multiple instances can exist in a document, and data and markup can be shared among them.

4.102 document type: A class of documents having similar characteristics; for example, journal, article, technical manual, or memo.

4.103 (document) type declaration: A markup declaration that contains the formal specification of a document type definition.

4.104 document type declaration subset: The element, entity, and short reference sets occurring within the declaration subset of a document type declaration.

NOTE — The external entity referenced from the document type declaration is considered part of the declaration subset.

4.105 document (type) definition: Rules, determined by an application, that apply SGML to the markup of documents of a particular type. A document type definition includes a formal specification, expressed in a document type declaration, of the element types, element relationships and attributes, and references that can be represented by markup. It thereby defines the vocabulary of the markup for which SGML defines the syntax.

NOTE — A document type definition can also include comments that describe the semantics of elements and attributes, and any application conventions.

4.106 document type specification: A portion of a tag or entity reference that identifies the document type instances within which the tag or entity reference will be processed.

4.107 ds (separator): A declaration separator, occurring in declaration subsets.

4.108 DTD: Document type definition.

4.109 effective status (of a marked section): The highest priority status keyword specified on a marked section declaration.

4.110 element: A component of the hierarchical structure defined by a document type definition; it is identified in a document instance by descriptive markup, usually a start-tag and end-tag.

NOTE — An element is classified as being of a particular element type.

4.111 element declaration: A markup declaration that contains the formal specification of the part of an element type definition that deals with the content and markup minimization.

4.112 element set: A set of element declarations that are used together.

NOTE — An element set can be public text.

4.113 element structure: The organization of a document into hierarchies of elements, with each hierarchy conforming to a different document type definition.

4.114 element type: A class of elements having similar characteristics; for example, paragraph, chapter, abstract, footnote, or bibliography.

4.115 element (type) definition: Application-specific rules that apply SGML to the markup of elements of a particular type. An element type definition includes a formal specification, expressed in element and attribute definition list declarations, of the content, markup minimization, and attributes allowed for a specified element type.

NOTE — An element type definition is normally part of a document type definition.

4.116 element type parameter: A parameter of an element declaration that identifies the type of element to which the definition applies.

NOTE — The specification can be direct, in the form of an individual *generic identifier* or member of a *name group*, or indirect, via a *ranked element* or member of a *ranked group*.

4.117 empty link set: A link set in which all result element types are implied and no attributes are specified.

4.118 empty map: A short reference map in which all delimiters are mapped to nothing.

NOTE — The empty map need not (and cannot) be declared explicitly, but can be referenced by its reserved name, which is "#EMPTY" in the reference concrete syntax.

4.119 end-tag: Descriptive markup that identifies the end of an element.

4.120 entity: A collection of characters that can be referenced as a unit.

NOTES

1 Objects such as book chapters written by different authors, pi characters, or photographs, are often best managed by maintaining them as individual entities.

2 The physical organization of entities is system-specific, and could take the form of files, members of a partitioned data set, components of a data structure, or entries in a symbol table.

4.121 entity declaration: A markup declaration that assigns an SGML name to an entity so that it can be referenced.

4.122 entity end (signal): A signal from the system that an entity's replacement text has ended.

4.123 entity manager: A program (or portion of a program or a combination of programs), such as a file system or symbol table, that can maintain and provide access to multiple entities.

4.124 entity reference: A reference that is replaced by an entity.

NOTE — There are two kinds: named entity reference and short reference.

4.125 entity set: A set of entity declarations that are used together.

NOTE — An entity set can be public text.

4.126 entity structure: The organization of a document into one or more separate entities.

NOTE — The first entity is an *SGML document entity*; it contains entity references that indicate where the other entities belong with respect to it.

4.127 entity text: The entity declaration parameter that specifies the replacement text, either by including it in a parameter literal, or by pointing to it with an external identifier.

4.128 equivalent reference string: A character string, consisting of an entity reference and possibly an ***RE*** and/or ***RS***, that replaces a short reference when a document is converted from a concrete syntax that supports short references to one that does not.

4.129 escape sequence: A control sequence whose first character is escape (ESC).

4.130 exceptions: A parameter of an element declaration that modifies the effect of the element's *content model*, and the content models of elements occurring within it, by permitting inclusions and prohibiting exclusions.

4.131 exclusions: Elements that are not allowed anywhere in the content of an element or its subelements even though the applicable content model or inclusions would permit them optionally.

4.132 explicit content reference: A content reference that was specified in an *attribute specification*.

4.133 explicit link (process definition): A link process definition in which the result element types and their attributes and multiple sets of link attribute values can be specified.

4.134 external entity: An entity whose text is not incorporated directly in an entity declaration; its system identifier and/or public identifier is specified instead.

NOTE — A document type or link type declaration can include the identifier of an external entity containing all or part of the declaration subset; the external identifier serves simultaneously as both the entity declaration and the entity reference.

4.135 external identifier: A parameter that identifies an external entity or data content notation.

NOTE — There are two kinds: system identifier and public identifer.

4.136 fixed attribute: An attribute whose specified value (if any) must be identical to its default value.

4.137 formal public identifier: A public identifier that is constructed according to rules defined in this International Standard so that its owner identifier and the components of its text identifier can be distinguished.

4.138 formal public identifier error: An error in the construction or use of a *formal public identifer*, other than an error that would prevent it being a valid *minimum literal*.

NOTE — A formal public identifier error can occur only if "FORMAL YES" is specified on the SGML declaration. A failure of a *public identifier* to be a *minimum literal*, however, is always an error.

4.139 function character: A markup character, assigned by the concrete syntax, that can perform some SGML function in addition to potentially being recognized as markup. If it is not recognized as markup in a context in which data is allowed it is treated as data (unless the language dictates special treatment, as in the case of the ***RE*** and ***RS*** function characters).

4.140 function character identification parameter: A parameter of an SGML declaration that identifies the characters assigned to the ***RE***, ***RS***, and ***SPACE*** functions, and allows additional functions to be defined.

4.141 G0 set: In graphic repertoire code extension, the virtual character set that represents the document character set graphic characters whose character numbers are below 128, in their normal code set positions.

4.142 general delimiter (role): A delimiter role other than a short reference.

4.143 general entity: An entity that is referenced from within the content of an element or an attribute value literal.

4.144 general entity reference: A named entity reference to a general entity.

4.145 generic identifier: A name that identifies the element type of an element.

4.146 GI: Generic identifier.

4.147 graphic character: A character, such as a letter, digit, or punctuation, that normally occupies a single position when text is displayed.

4.148 graphic repertoire code extension: Code extension in which multiple graphic character sets are mapped onto positions of the document code set by using shift functions to invoke virtual character sets.

4.149 group: The portion of a parameter that is bounded by a balanced pair of ***grpo*** and ***grpc*** delimiters or ***dtgo*** and ***dtgc*** delimiters.

NOTE — There are five kinds: name group, name token group, model group, data tag group, and data tag template group. A name, name token, or data tag template group cannot contain a group, but a model group can contain a model group and a data tag group can contain a data tag template group.

4.150 ID: Unique identifier.

4.151 ID reference list: An attribute value that is a list of ID reference values.

4.152 ID reference value: An attribute value that is a *name* specified as an *id value* of an element in the same document instance.

4.153 ID value: An attribute value that is a *name* that uniquely identifies the element; that is, it cannot be the same as any other *id value* in the same document instance.

4.154 impliable attribute: An attribute for which there need not be an *attribute specification*, and whose value is defined by the application when it is not specified.

4.155 implicit link (process definition): A link process definition in which the result element types and their attributes are all implied by the application, but multiple sets of link attribute values can be specified.

4.156 inert function characters: Character class consisting of function characters whose additional SGML "function" is to do nothing.

4.157 included subelement: A subelement that is not permitted by its containing element's model, but is permitted by an inclusion exception.

4.158 inclusions: Elements that are allowed anywhere in the content of an element or its subelements even though the applicable model does not permit them.

4.159 inherently optional token: A model group token that:

a) has an ***opt*** or ***rep*** occurrence indicator; or
b) is an ***or*** group, one of whose tokens is inherently optional; or
c) is an ***and*** or ***seq*** group, all of whose tokens are inherently optional.

4.160 instance (of a document type): The data and markup for a hierarchy of elements that conforms to a document type definition.

4.161 interpreted parameter literal: The text of a *parameter literal*, exclusive of the literal delimiters, in which character and parameter entity references have been replaced.

4.162 ISO owner identifier: An *owner identifier*, consisting of an ISO publication number or character set registration number, that is used when a *public identifier* identifies, or is assigned by, an ISO publication, or identifies an ISO registered character set.

4.163 ISO text description: A *public text description*, consisting of the last element of an ISO publication title (without part designation, if any), that is used when a *public identifier* identifies an ISO publication.

4.164 keyword: A parameter that is a reserved name defined by the concrete syntax, as opposed to arbitrary text.

NOTE — In parameters where either a keyword or a name defined by an application could be specified, the keyword is always preceded by the reserved name indicator. An application is therefore able to define names without regard to whether those names are also used by the concrete syntax.

4.165 link attribute: An attribute of a source element type that is meaningful only in the context of a particular process that is performed on the source document instance.

4.166 link process: A process that creates a new instance of some document type (the result) from an existing instance of the same or another document type (the source). Processes can be chained, so that the result of one is the source for the next.

NOTE — Examples of link processes include editing, in which the source and result document types are usually the same, and formatting, in which they are usually different.

4.167 link process definition: Application-specific rules that apply SGML to describe a link process. A link process definition includes a formal specification, expressed in a *link type declaration*, of the link between elements of the source and result, including the definitions of source attributes applicable to the link process ("link attributes").

NOTES

1 A link process definition can also include comments that describe the semantics of the process, including the meaning of the link attributes and their effect on the process.

2 There are three kinds of link process definitions: simple, implicit, and explicit.

4.168 link set: A named set of associations, declared by a *link set declaration*, in which elements of the source document type are linked to elements of the result document type. For each element link, source link attributes and result element attributes can be specified.

4.169 link set declaration: A markup declaration that defines a link set.

4.170 link type declaration: A markup declaration that contains the formal specification of a link process definition.

4.171 link type declaration subset: The entity sets, link attribute sets, and link set and link set use declarations, that occur within the declaration subset of a link type declaration.

NOTE — The external entity referenced from the link type declaration is considered part of the declaration subset.

4.172 locking shift: A shift function that applies until another locking shift function occurs.

4.173 lower-case letters: Character class composed of the 26 unaccented small letters from "a" through "z".

4.174 lower-case name characters: Character class consisting of each additional lower-case *name character* assigned by the concrete syntax.

4.175 lower-case name start characters: Character class consisting of each additional lower-case *name start character* assigned by the concrete syntax.

4.176 LPD: Link process definition.

4.177 map: Short reference map.

4.178 mark up: To add markup to a document.

4.179 marked section: A section of the document that has been identified for a special purpose, such as ignoring markup within it.

4.180 marked section declaration: A markup declaration that identifies a marked section and specifies how it is to be treated.

4.181 marked section end: The closing delimiter sequence of a marked section declaration.

4.182 marked section start: The opening delimiter sequence of a marked section declaration.

4.183 markup: Text that is added to the data of a document in order to convey information about it.

NOTE — There are four kinds of markup: descriptive markup (tags), references, markup declarations, and processing instructions.

4.184 markup character: An SGML character that, depending on the context, could be interpreted either as markup or data.

4.185 markup convention: Application convention governing markup, such as a rule for the formulation of an entity name, or a preferred subset of allowed short reference delimiters.

4.186 (markup) declaration: Markup that controls how other markup of a document is to be interpreted.

NOTE — There are 13 kinds: SGML, entity, element, attribute definition list, notation, document type, link type, link set, link use, marked section, short reference mapping, short reference use, and comment.

4.187 (markup) minimization feature: A feature of SGML that allows markup to be minimized by shortening or omitting tags, or shortening entity references.

NOTE — Markup minimization features do not affect the document type definition, so a minimized document can be sent to a system that does not support these features by first restoring the omitted markup. There are five kinds: SHORTTAG, OMITTAG, SHORTREF, DATATAG, and RANK.

4.188 markup-scan-in characters: Character class consisting of function characters that restore markup recognition if it was suppressed by the occurrence of a markup-scan-out character.

4.189 markup-scan-out characters: Character class consisting of function characters that suppress markup recognition until the occurrence of a markup-scan-in character or entity end.

4.190 markup-scan-suppress characters: A character class consisting of function characters that suppress markup recognition for the immediately following character in the same entity (if any).

4.191 minimal SGML document: A conforming SGML document that uses the core concrete syntax throughout, no features, and the reference capacity set.

4.192 minimization feature: Markup minimization feature.

4.193 model: Content model.

4.194 model group: A component of a content model that specifies the order of occurrence of elements and character strings in an element's *content*, as modified by *exceptions* specified in the *content model* of the element and in the content models of other open elements.

4.195 multicode basic concrete syntax: A multicode variant of the basic concrete syntax in which markup is not recognized when code extension is in use.

4.196 multicode concrete syntax: A concrete syntax that allows code extension control characters to be SGML characters.

4.197 multicode core concrete syntax: A multicode variant of the core concrete syntax in which markup is not recognized when code extension is in use.

4.198 name: A name token whose first character is a name start character.

4.199 name character: A character that can occur in a name: name start characters, digits, and others designated by the concrete syntax.

4.200 name group: A group whose tokens are required to be names.

4.201 name start character: A character that can begin a name: letters, and others designated by the concrete syntax.

4.202 name token: A character string, consisting solely of name characters, whose length is restricted by the NAMELEN quantity.

NOTE — A name token that occurs in a group is also a token; one that occurs as an attribute value is not.

4.203 name token group: A group whose tokens are required to be name tokens.

4.204 named character reference: A character reference consisting of a delimited *function name*.

4.205 named entity reference: An entity reference consisting of a delimited name of a general entity or parameter entity (possibly qualified by a document type specification) that was declared by an entity declaration.

NOTE — A general entity reference can have an undeclared name if a default entity was declared.

4.206 naming rules parameter: A parameter of an SGML declaration that identifies additions to the standard name alphabet character classes and specifies the case substitution.

4.207 non-SGML character: A character in the document character set whose coded representation never occurs in an SGML entity.

4.208 non-SGML data entity: An entity whose characters are not interpreted in accordance with this International Standard, and in which, therefore, no SGML markup can be recognized.

NOTE — The interpretation of a non-SGML data entity is governed by a data content notation, which may be defined by another International Standard.

4.209 NONSGML: The class of non-SGML characters, defined by the document character set.

4.210 normalized length (of an attribute specification list): A length calculated by ignoring the actual characters used for delimiting and separating the components and counting an extra fixed number per component instead.

4.211 notation attribute: An attribute whose value is a *notation name* that identifies the data content notation of the element's *content*.

NOTE — A notation attribute does not apply when there is an explicit content reference, as the element's *content* will be empty.

4.212 notation declaration: A markup declaration that associates a name with a notation identifier.

4.213 notation identifier: An *external identifier* that identifies a data content notation in a *notation declaration*. It can be a *public identifier* if the notation is public, and, if not, a description or other information sufficient to invoke a program to interpret the notation.

4.214 notation name: The name assigned to a data content notation by a notation declaration.

4.215 number: A name token consisting solely of digits.

4.216 number token: A name token whose first character is a digit.

NOTE — A number token that occurs in a group is also a token; one that occurs as an attribute value is not.

4.217 numeric character reference: A character reference consisting of a delimited *character number*.

4.218 object capacity: The capacity limit for a particular kind of object, such as entities defined or characters of entity text.

4.219 omitted tag minimization parameter: A parameter of an element declaration that specifies whether a technically valid omission of a start-tag or end-tag is considered a reportable markup error.

4.220 open element: An *element* whose *start-tag* has occurred (or been omitted through markup minimization), but whose *end-tag* has not yet occurred (or been omitted through markup minimization).

4.221 open entity: An entity that has been referenced but whose entity end has not yet occurred.

4.222 open marked section declaration: A marked section declaration whose *marked section start* has occurred but whose *marked section end* has not yet occurred.

4.223 owner identifier: The portion of a public identifier that identifies the owner or originator of public text.

NOTE — There are three kinds: ISO, registered, and unregistered.

4.224 parameter: The portion of a markup declaration that is bounded by parameter separators (whether required or optional). A parameter can contain other parameters.

4.225 parameter entity: An entity that is referenced from a markup declaration parameter.

4.226 parameter entity reference: A named entity reference to a parameter entity.

4.227 parameter literal: A parameter or token consisting of delimited replaceable parameter data.

4.228 parsed character data: Zero or more characters that occur in a context in which text is parsed and markup is recognized. They are classified as data characters because they were not recognized as markup during parsing.

4.229 PCDATA: Parsed character data.

4.230 PI entity: Processing instruction entity.

4.231 point: A unit of capacity measurement, roughly indicative of relative storage requirements.

4.232 procedure: Processing defined by an application to operate on elements of a particular type.

NOTES

1 A single procedure could be associated with more than one element type, and/or more than one procedure could operate on the same element type at different points in the document.

2 A procedure is usually part of a procedure set.

4.233 procedure set: The procedures that are used together for a given application process.

NOTE — In SGML applications, a procedure set usually constitutes the application processing for a link process definition.

4.234 processing instruction: Markup consisting of system-specific data that controls how a document is to be processed.

4.235 processing instruction entity: An entity whose text is treated as the *system data* of a *processing instruction* when referenced.

4.236 prolog: The portion of an SGML document or SGML subdocument entity that contains document type and link type declarations.

4.237 proper subelement: A subelement that is permitted by its containing element's model.

4.238 ps (separator): A parameter separator, occurring in markup declarations.

4.239 public identifier: A minimum literal that identifies public text.

NOTES

1 The public identifiers in a document can optionally be interpretable as formal public identifiers.

2 The system is responsible for converting public identifiers to system identifiers.

4.240 public text: Text that is known beyond the context of a single document or system environment, and which can be accessed with a public identifier.

NOTES

1 Examples are standard or registered document type definitions, entity sets, element sets, data content notations, and other markup constructs (see annex D).

2 Public text is not equivalent to published text; there is no implication of unrestricted public access. In particular, the owner of public text may choose to sell or license it to others, or to restrict its access to a single organization.

3 Public text simplifies access to shared constructs, reduces the amount of text that must be interchanged, and reduces the chance of copying errors.

4.241 public text class: The portion of a text identifier that identifies the SGML markup construct to which the public text conforms.

4.242 public text description: The portion of a text identifier that describes the public text.

4.243 public text designating sequence: The portion of a text identifier, used when public text is a character set, that contains an ISO 2022 escape sequence that designates the set.

4.244 public text display version: An optional portion of a text identifier that distinguishes among public text that has a common *public text description* by describing the devices supported or coding scheme used. If omitted, the public text is not device-dependent.

4.245 public text language: The portion of a text identifier that specifies the natural language in which the public text was written.

NOTE — It can be the language of the data, comments, and/or defined names.

4.246 quantity: A numeric restriction on some aspect of markup, such as the maximum length of a name or the maximum nesting level of open elements.

NOTE — Quantities are defined by the abstract syntax, but specific values are assigned to them by the concrete syntax.

4.247 quantity set: A set of assignments of numeric values to quantity names.

4.248 ranked element: An element whose *generic identifier* is composed of a *rank stem* and a *rank suffix*. When a ranked element begins, its *rank suffix* becomes the current rank for its *rank stem*, and for the rank stems in the *ranked group* (if any) of which the *rank stem* is a member.

4.249 ranked group: A group of rank stems that share the same current rank. When any ranked element whose stem is in the group begins, its *rank suffix* becomes the current rank for all rank stems in the group.

4.250 rank stem: A name from which a generic identifier can be derived by appending the current rank.

4.251 rank suffix: A number that is appended to a rank stem to form a *generic identifier*.

NOTE — The numbers are usually sequential, beginning with 1, so the resulting generic identifiers suggest the relative ranks of their elements (for example, H1, H2, and H3 for levels of heading elements, where "H" is the rank stem).

4.252 record: A division of an SGML entity, bounded by a record start and a record end character, normally corresponding to an input line on a text entry device.

NOTES

1 It is called a "record" rather than a "line" to distinguish it from the output lines created by a text formatter.

2 An SGML entity could consist of many records, a single record, or text with no record boundary characters at all (which can be thought of as being part of a record or without records, depending on whether record boundary characters occur elsewhere in the document).

4.253 record boundary (character): The record start (***RS***) or record end (***RE***) character.

4.254 record end: A *function character*, assigned by the concrete syntax, that represents the end of a record.

4.255 record start: A *function character*, assigned by the concrete syntax, that represents the start of a record.

4.256 reference: Markup that is replaced by other text, either an entity or a single character.

4.257 reference capacity set: The capacity set defined in this International Standard.

4.258 reference concrete syntax: A concrete syntax, defined in this International Standard, that is used in all SGML declarations.

4.259 reference delimiter set: The delimiter set, defined in this International Standard, that is used in the reference concrete syntax.

4.260 reference quantity set: The quantity set defined by this International Standard.

4.261 reference reserved name: A reserved name defined by this International Standard.

4.262 registered owner identifier: An owner identifier that was constructed in accordance with ISO 9070. It is unique among registered owner identifiers, and is distinguishable from ISO owner identifiers and unregistered owner identifiers.

4.263 replaceable character data: Character data in which a *general entity reference* or *character reference* is recognized and replaced.

NOTE — Markup that would terminate *replaceable character data* is not recognized in the replacement text of entities referenced within it.

4.264 replaceable parameter data: Character data in which a *parameter entity reference* or *character reference* is recognized and replaced.

NOTE — Markup that would terminate *replaceable parameter data* is not recognized in the replacement text of entities referenced within it.

4.265 replacement character: The character that replaces a *character reference*.

4.266 replacement text: The text of the entity that replaces an entity reference.

4.267 reportable markup error: A failure of a document to conform to this International Standard when it is parsed with respect to the active document and link types, other than:

a) an ambiguous content model;

b) an exclusion that could change a group's required or optional status in a model;

c) exceeding a capacity limit;

d) an error in the SGML declaration;

e) the occurrence of a non-SGML character; or

f) an formal public identifier error.

4.268 required attribute: An attribute for which there must always be an *attribute specification* for the attribute value.

4.269 reserved name: A name defined by the concrete syntax, rather than by an application, such as a markup declaration name.

NOTE — Such names appear in this International Standard as syntactic literals.

4.270 reserved name use parameter: A parameter of the *SGML declaration* that specifies any replacement in the declared concrete syntax for a reference reserved name.

4.271 result document type (of a link): A document type, a new instance of which is created as the result of a link process.

4.272 result element type (of a link): An element that is defined in the result document type declaration.

4.273 s (separator): A separator, consisting of separator characters and other non-printing function characters, that occurs in markup and in *element content*.

4.274 satisfied token: A *content token* whose corresponding content has occurred.

4.275 SDATA entity: Specific character data entity.

4.276 separator: An *s*, *ds*, *ps*, or *ts*.

4.277 separator characters: A character class that consists of function characters that are allowed in separators and that will be replaced by ***Space*** in those contexts in which ***RE*** is replaced by ***Space***.

4.278 SGML: Standard Generalized Markup Language

4.279 SGML application: Rules that apply SGML to a text processing application. An SGML application includes a formal specification of the markup constructs used in the application, expressed in SGML. It can also include a non-SGML definition of semantics, application conventions, and/or processing.

NOTES

1 The formal specification of an SGML application normally includes document type definitions, data content notations, and entity sets, and possibly a concrete syntax or capacity set. If processing is defined by the application, the formal specification could also include link process definitions.

2 The formal specification of an SGML application constitutes the common portions of the documents processed by the application. These common portions are frequently made available as public text.

3 The formal specification is usually accompanied by comments and/or documentation that explains the semantics, application conventions, and processing specifications of the application.

4 An SGML application exists independently of any implementation. However, if processing is defined by the application, the non-SGML definition could include application procedures, implemented in a programming or text processing language.

4.280 SGML character: A character that is permitted in an SGML entity.

4.281 SGML declaration: A markup declaration that specifies the character set, concrete syntax, optional features, and capacity requirements of a document's markup. It applies to all of the SGML entities of a document.

4.282 SGML document: A document that is represented as a sequence of characters, organized physically into an entity structure and logically into an element structure, essentially as described in this International Standard. An SGML document consists of data characters, which represent its information content, and markup characters, which represent the structure of the data and other information useful for processing it. In particular, the markup describes at least one document type definition, and an instance of a structure conforming to the definition.

4.283 SGML document entity: The SGML entity that begins an SGML document. It contains, at a minimum, an SGML declaration, a base document type declaration, and the start and end (if not all) of a base document element.

4.284 SGML entity: An entity whose characters are interpreted as markup or data in accordance with this International Standard.

NOTE — There are three types of SGML entity: *SGML document entity*, *SGML subdocument entity*, and *SGML text entity*.

4.285 SGML parser: A program (or portion of a program or a combination of programs) that recognizes markup in conforming SGML documents.

NOTE — If an analogy were to be drawn to programming language processors, an SGML parser would be said to perform the functions of both a lexical analyzer and a parser with respect to SGML documents.

4.286 SGML subdocument entity: An SGML entity that conforms to the SGML declaration of the SGML document entity, while conforming to its own document type and link type declarations. It contains, at a minimum, a base document type declaration and the start and end of a base document element.

4.287 SGML system: A system that includes an SGML parser, an entity manager, and both or either of:

a) an implementation of one or more SGML applications; and/or
b) facilities for a user to implement SGML applications, with access to the SGML parser and entity manager.

4.288 SGML text entity: An SGML entity that conforms to the SGML declaration of the SGML document entity, and to the document type and link type declarations to which the entity from which it is referenced conforms.

4.289 shift function: In graphic repertoire code extension, a control sequence or control character that invokes a graphic character set.

NOTE — There are two kinds: single shift and locking shift.

4.290 short reference: Short reference string.

4.291 short reference delimiter role: A delimiter role to which zero or more strings can be assigned by the concrete syntax. When a short reference string is recognized, it is replaced by the general entity to whose name it is mapped in the current map, or is treated as a separator or data if it is mapped to nothing.

4.292 (short reference) map: A named set of associations, declared by a *short reference mapping declaration*, in which each short reference delimiter is mapped to a general entity name or to nothing.

4.293 short reference mapping declaration: A markup declaration that defines a short reference map.

4.294 short reference set: A set of short reference mapping, short reference use, and entity declarations that are used together.

NOTE — A short reference set can be public text.

4.295 short reference (string): A character string assigned to the short reference delimiter role by the concrete syntax.

4.296 short reference use declaration: A markup declaration that associates a short reference map with one or more element types, or identifies a new current map for the current element.

4.297 shunned character (number): A character number, identified by a concrete syntax, that should be avoided in documents employing the syntax because some systems might erroneously treat it as a control character.

4.298 significant SGML character: A *markup character* or *minimum data* character.

4.299 simple link (process definition): A link process definition in which the result element types and their attributes are all implied by the application, and only one set of link attribute values can be specified. The source document type must be the base.

4.300 single shift: A shift function that applies to the following character only.

4.301 source document type (of a link): A document type, an existing instance of which is the source of a link process.

4.302 source element type (of a link): An element type that is defined in the source document type declaration.

4.303 space: A *function character*, assigned by the concrete syntax, that represents a space.

4.304 specific character data entity: An entity whose text is treated as *character data* when referenced. The text is dependent on a specific system, device, or application process.

NOTE — A specific character data entity would normally be redefined for different applications, systems, or output devices.

4.305 Standard Generalized Markup Language: A language for document representation that formalizes markup and frees it of system and processing dependencies.

4.306 start-tag: Descriptive markup that identifies the start of an element and specifies its generic identifier and attributes.

4.307 status keyword: A marked section declaration parameter that specifies whether the marked section is to be ignored and, if not, whether it is to be treated as character data, replaceable character data, or normally.

4.308 string: Character string.

4.309 subelement: An element that occurs in the content of another element (the "containing element") in such a way that the subelement begins when the containing element is the current element.

4.310 syntax-reference character set: A character set, designated by a concrete syntax and known to all potential users of the syntax, that contains every significant SGML character. It enables a concrete syntax to be defined without regard to the particular document or system character sets with which it might be used.

4.311 system character set: The character set used in an SGML system.

4.312 system declaration: A declaration, included in the documentation for a conforming SGML system, that specifies the features, capacity set, concrete syntaxes, and character set that the system supports, the data content notations that it can interpret, and any validation services that it can perform.

4.313 system identifier: System data that specifies the file identifier, storage location, program invocation, data stream position, or other system-specific information that locates an external entity.

4.314 tag: Descriptive markup.

NOTE — There are two kinds: start-tag and end-tag.

4.315 target element: An element whose *generic identifier* is specified in a *data tag group*

4.316 text: Characters.

NOTE — The characters could have their normal character set meaning, or they could be interpreted in accordance with a data content notation as the representation of graphics, images, etc.

4.317 text identifier: The portion of a public identifier that identifies a public text so that it can be distinguished from any other public text with the same owner identifier.

NOTE — It consists of a *public text class*, an optional *unavailable text indicator*, a *public text description*, a *public text language*, and an optional *public text display version*.

4.318 text processing application: A related set of processes performed on documents of related types.

NOTE — Some examples are:

a) Publication of technical manuals for a software developer: document types include installation, operation, and maintenance manuals; processes include creation, revision, formatting, and page layout for a variety of output devices.

b) Preparation of manuscripts by independent authors for members of an association of publishers: document types include book, journal, and article; creation is the only defined process, as each publisher has its own methods of formatting and printing.

c) Office correspondence: document types include memos, mail logs, and reports; processes include creation, revision, simple formatting, storage and retrieval, memo log update, and report generation.

4.319 token: The portion of a group, including a complete nested group (but not a *connector*), that is bounded by token separators (whether required or optional).

4.320 total capacity: A limit on the sum of all object capacities.

4.321 ts (separator): A token separator, occurring in groups.

4.322 type definition: Document type definition.

4.323 unavailable public text: Public text that is available only to a limited population, selected by its owner.

4.324 unique identifier: A *name* that uniquely identifies an element.

4.325 unregistered owner identifier: An owner identifier that can be distinguished from registered owner identifiers and ISO owner identifiers. As it is not constructed according to a registration standard, it could duplicate another unregistered owner identifier.

4.326 upper-case letters: Character class composed of the 26 capital letters from "A" through "Z".

4.327 upper-case name characters: Character class consisting of the upper-case forms of the corresponding lower-case name characters.

4.328 upper-case name start characters: Character class consisting of the upper-case forms of the corresponding lower-case name start characters.

4.329 validating SGML parser: A conforming SGML parser that can find and report a reportable markup error if (and only if) one exists.

4.330 variant concrete syntax: A concrete syntax other than the reference concrete syntax or core concrete syntax.

4.331 variant (conforming) SGML document: A conforming SGML document that uses a variant concrete syntax.

4.332 virtual character set: In graphic repertoire code extension, one of the character sets, known as G0, G1, G2, or G3, that represents the mapping of a real graphic character set, designated by an escape sequence, to a document code set position previously announced by an escape sequence.

5 Notation

NOTE — This clause describes the notation used in this International Standard to define the Standard Generalized Markup Language. This notation is not part of SGML itself (although there are some similarities between them), and therefore this clause should be considered as affecting only the presentation of this International Standard, not the substance.

The SGML abstract syntax is specified by formal syntax productions, each of which defines a "syntactic variable". A production consists of a reference number (in square brackets), the name of the syntactic variable being defined, an equals sign, and an expression that constitutes the definition.

[number] syntactic variable = expression

The expression is composed of one or more "syntactic tokens", parenthesized expressions, and symbols that define the ordering and selection among them.

5.1 Syntactic Tokens

The following list shows the syntactic token types using the typographic conventions employed for them in this International Standard, together with the source of their definitions:

syntactic variable. A syntactic token that is defined by a syntax production.

"SYNTACTIC LITERAL". A syntactic token consisting of a reserved name that is to be entered in markup exactly as it is shown in the syntax production, except that corresponding lower-case letters can be used if upper-case translation of general names is specified by the concrete syntax. Syntactic literals are defined whenever they occur in a syntax production, and the definition is applicable only in the context of that production.

delimiter role. A syntactic token that represents a delimiter string. Delimiter roles are defined in figure 3, which also lists the strings assigned to general delimiter roles by the reference concrete syntax. Strings assigned to the ***shortref*** delimiter role are shown in figure 4.

TERMINAL VARIABLE. A syntactic token that represents a character class whose members are not necessarily the same in all SGML documents. Terminal variables whose members are assigned by the concrete syntax are defined in figure 2. (The ***NONSGML*** variable, whose members are assigned by the document character set, is defined in 13.1.2.)

Terminal Constant. A syntactic token that represents either an entity end signal, or a character class whose members are the same in all SGML documents. Terminal constants are defined in figure 1.

5.2 Ordering and Selection Symbols

If there is more than one syntactic token in an expression, the ordering and selection among them is determined by symbols that connect them, as follows:

, All must occur, in the order shown.
& All must occur, in any order.
| One and only one must occur.

Each selected syntactic token must occur once and only once unless the contrary is indicated by a suffix consisting of one of the following symbols:

? Optional (0 or 1 time).
+ Required and repeatable (1 or more times).
* Optional and repeatable (0 or more times).

The occurrence suffixes are applied first, then the ordering connectors. Parentheses can be used as in mathematics to change these priorities.

6 Entity Structure

6.1 SGML Document

An *SGML document* is physically organized in an entity structure. The first entity is an *SGML document entity*; it contains entity references that indicate where the other entities belong with respect to it.

[1] SGML document = *SGML document entity*,
(SGML subdocument entity |
SGML text entity | *non-SGML data entity)**

NOTES

1 This International Standard does not constrain the physical organization of the document within the data stream, message handling protocol, file system, etc., that contains it. In particular, separate entities could occur in the same physical object, a single entity could be divided between multiple objects, and the objects could occur in any order.

2 This International Standard does not constrain the characters that can occur in a data stream outside of the SGML entities. Such characters would be interpreted according to applicable standards and conventions.

3 The SGML Document Interchange Format (SDIF) standardized in ISO 9069 allows an SGML document to be managed conveniently as a single object, while still preserving the entity structure.

6.2 SGML Entities

[2] SGML document entity = *SGML declaration*,
*s**, *prolog*, *s**, *document instance set*, ***Ee***

[3] SGML subdocument entity = *prolog*, *s**,
document instance set, ***Ee***

[4] SGML text entity = *SGML character**, ***Ee***

The characters of an SGML entity are parsed in accordance with this International Standard to recognize the *SGML declaration*, *prolog*, and *document instance set*, and their constituent parts.

Each *SGML character* is parsed in the order it occurs, in the following manner:

a) The character is tested to see if it is part of a delimiter string (see 9.6). If a general delimiter string is recognized, the appropriate delimiter action is performed. If a *short reference* is recognized, it is treated as specified in 9.4.6.

b) If the character is not part of a delimiter string, or it is part of an unmapped short reference, it is tested to see if it is a separator; if so, it is ignored.
c) If the character is not part of a delimiter or a separator, it is treated as data.

If an *SGML character* is a *function character*, its function is performed in addition to any other treatment it may receive.

6.2.1 S Separator

[5] s = ***SPACE*** | ***RE*** | ***RS*** | ***SEPCHAR***

An *SGML character* is not considered part of an *s* if it can be interpreted as some other markup. If it is considered part of an *s* it is ignored.

6.2.2 Entity End

An ***Ee*** is an entity end signal.

NOTE — An ***Ee*** is not a character and is never treated as data. It can occur only where expressly permitted.

A system can represent an ***Ee*** in any manner that will allow it to be distinguished from SGML characters.

NOTE — For example, an ***Ee*** could be represented by the bit combination of a non-SGML character, if any have been assigned.

6.2.3 Implied SGML Declaration

While a document is processed exclusively by a single system, the system can imply the *SGML declaration*. The declaration must be present explicitly, however, if the document is subsequently sent to another system.

6.3 Non-SGML Data Entity

[6] non-SGML data entity = *character**, ***Ee***

NOTE — A *non-SGML data entity* is declared with a notation parameter that identifies how the data is to be interpreted.

7 Element Structure

7.1 Prolog

[7] prolog = *other prolog**,
base document type declaration,
(*document type declaration* |
other prolog)*, (*link type declaration* |
other prolog)*

[8] other prolog = *comment declaration* |
processing instruction | *s*

[9] base document type declaration =
document type declaration

An SGML document or subdocument entity conforms to the document type and link process definitions that are specified by the document type and link type declarations, respectively, in its *prolog*. An SGML text entity conforms to the *prolog* to which the entity from which it is referenced conforms. Parsing of an SGML entity is done with respect to those document and link types that are considered to be active.

NOTE — The system normally identifies the active document and link types to the SGML parser during initialization (see clause F.2).

Document type declarations in addition to the base are permitted only if "CONCUR YES" or "EXPLICIT YES" is specified on the *SGML declaration*.

7.2 Document Element

[10] document instance set
= *base document element*

[11] base document element = *document element*

[12] document element = *element*

7.2.1 Limits

Instances of document type definitions other than the base are permitted only to the limit specified on the "CONCUR" parameter of the *SGML declaration*.

7.3 Element

[13] element = *start-tag*?, *content*, *end-tag*?

If an element has a *declared content* of "EMPTY", or an explicit content reference, the *end-tag* must be omitted.

NOTE — This requirement has nothing to do with markup minimization.

7.3.1 Omitted Tag Minimization

If "OMITTAG YES" is specified on the *SGML declaration*, a tag can be omitted as provided in this sub-sub-clause, as long as the omission would not create an ambiguity.

NOTE — A document type definition may consider a technically valid omission to be a markup error (see 11.2.2).

7.3.1.1 Start-tag Omission

The *start-tag* can be omitted if the element is a contextually required element and any other elements that could occur are contextually optional elements, except if:

a) the element type has a required attribute or *declared content*; or
b) the *content* of the instance of the element is empty.

An omitted *start-tag* is treated as having an empty *attribute specification list*.

7.3.1.2 End-tag Omission

The *end-tag* can be omitted for a *document element*, or for an element that is followed either

a) by the *end-tag* of another open *element*; or
b) by an *element* or *SGML character* that is not allowed in its *content*.

NOTE — An element that is not allowed because it is an exclusion has the same effect as one that is not allowed because no token appears for it in the model group.

7.3.2 Data Tag Minimization

If "DATATAG YES" is specified on the *SGML declaration*, the *end-tag* can be omitted for an element that is the corresponding content of a data tag group.

The data content of the (target) element and its subelements is scanned for a string that conforms to one of the data tag templates in the element's *data tag pattern*. That string, plus any succeeding characters that conform to the *data tag padding template*, are considered the element's data tag. The data tag is treated both as the *end-tag* of the target element and as *character data* in its containing element.

NOTE — A *generic identifier* that occurs as a target element in a *data tag group* could also occur in other contexts as an *element token*. In those contexts, it would not be scanned for a data tag. It could also occur in other data tag groups, possibly with different data tag patterns.

The data content of the target element is scanned a character at a time. At each character, the longest possible *data tag template* in the *data tag pattern* is matched. If more than one target element is an open element, then at each character the templates of the most recently opened target element are tested first. If there is no match, the templates of the next most recently opened target element are tested, and so on.

NOTE — A data tag will therefore terminate the most recently opened target element whose data tag pattern it satisfies.

The matching of a *data tag pattern* to the content occurs after recognition of markup and replacement of references in the content, but before any ***RE*** or ***RS*** characters are ignored.

NOTE — The *data tag pattern* can therefore contain named character references to ***RE*** and ***RS***.

A data tag cannot be recognized in any context in which end-tags are not recognized; for example, within a CDATA marked section.

In matching the *data tag padding template*, the template characters can be omitted, used once, or repeated indefinitely. The last (or only) use can stop short of the full template.

7.3.3 Quantities

The number of open elements cannot exceed the "TAGLVL" quantity.

The length of a data tag cannot exceed the "DTAGLEN" quantity.

7.4 Start-tag

[14] start-tag = (**stago**,
document type specification,
generic identifier specification,
attribute specification list, *s**, **tagc**) |
minimized start-tag

7.4.1 Minimization

[15] minimized start-tag = *empty start-tag* |
unclosed start-tag | *net-enabling start-tag*

A *start-tag* can be a *minimized start-tag* only if "SHORTTAG YES" is specified on the *SGML declaration*.

7.4.1.1 Empty Start-tag

[16] empty start-tag = **stago**,
document type specification, *s**, **tagc**

A *start-tag* can be an *empty start-tag* only if the *element* is in the base document instance, in which case the tag's *generic identifier specification* is assumed to be:

a) if "OMITTAG YES" is specified on the *SGML declaration*, the *generic identifier* of the most recently started open element in the base document type; or
b) if "OMITTAG NO" is specified on the *SGML declaration*, the *generic identifier* of the most recently ended element in the base document type; or

c) if there was no such previous applicable element, the *generic identifier* of the *document element*.

An *empty start-tag* is treated as having an empty *attribute specification list*.

7.4.1.2 Unclosed Start-tag

[17] unclosed start-tag = **stago**,
document type specification,
generic identifier specification,
attribute specification list, *s**

A *start-tag* can be an *unclosed start-tag* only if it is followed immediately by the character string assigned to the **stago** or **etago** delimiter role, regardless of whether the string begins a valid delimiter-in-context sequence.

7.4.1.3 NET-enabling Start-tag

[18] net-enabling start-tag = **stago**,
document type specification,
generic identifier specification,
attribute specification list, *s**, **net**

A *start-tag* can be a *net-enabling start-tag* only if its *element* is in the base document instance.

7.4.2 Quantities

The length of a *start-tag*, before resolution of references in the *attribute specification list* and exclusive of the **stago** and **tagc** or **net** delimiters, cannot exceed the "TAGLEN" quantity.

7.5 End-tag

[19] end-tag = (**etago**,
document type specification,
generic identifier specification, *s**, **tagc**) |
minimized end-tag

An *end-tag* ends the most recently started open *element*, within the instance of the document type specified by the *document type specification*, whose generic identifier is specified by the *generic identifier specification*.

7.5.1 Minimization

[20] minimized end-tag = *empty end-tag* |
unclosed end-tag | *null end-tag*

An *end-tag* can be a *minimized end-tag* only if "SHORTTAG YES" is specified on the *SGML declaration*.

7.5.1.1 Empty End-tag

[21] empty end-tag = **etago**, *s**, **tagc**

If an *end-tag* is an *empty end-tag*, its *generic identifier* is that of the most recently started open element in the base document instance. If there was no such element, the *end-tag* is an error.

7.5.1.2 Unclosed End-tag

[22] unclosed end-tag = **etago**,
document type specification,
generic identifier specification, *s**

An *end-tag* can be an *unclosed end-tag* only if it is followed immediately by the character string assigned to the **stago** or **etago** delimiter role, regardless of whether the string begins a valid delimiter-in-context sequence.

7.5.1.3 Null End-tag

[23] null end-tag = **net**

The **net** is recognized as a *null end tag* only if the *start-tag* of an open element in the base document type was a *net-enabling start-tag*. It is assumed to be the *end tag* of the most recently started such element.

7.6 Content

[24] content = *mixed content* | *element content* |
replaceable character data | *character data*

[25] mixed content = (*data character* | *element* |
other content)*

[26] element content = (*element* | *other content* |
s)*

[27] other content = *comment declaration* |
short reference use declaration |
link type use declaration |
processing instruction | **shortref** |
character reference |
general entity reference |
marked section declaration | **Ee**

An element's *declared content* or *content model* determines which of the four types of *content* it has, or whether it is empty, except that the *content* must be empty if the element has an explicit content reference.

The *content* of an element declared to be *character data* or *replaceable character data* is terminated only by an **etago** delimiter-in-context (which need not open a valid *end-tag*) or a valid **net**. Such termination is an error if it would have been an error had the *content* been *mixed content*.

NOTE — Content characters could be classed as data content for either of two reasons:

a) Declared character data.

The element's entire content was declared to be *character data* or *replaceable character data* by the *declared content* parameter of the *element declaration*.

b) Parsed character data.

The element was declared to have *mixed content*, and an *SGML character* within it was parsed as data because it was not recognized as markup.

7.6.1 Record Boundaries

If an ***RS*** in *content* is not interpreted as markup, it is ignored.

Within *content*, an ***RE*** remaining after replacement of all references and recognition of markup is treated as data unless its presence can be attributed solely to markup. That is:

a) The first ***RE*** in an element is ignored if no ***RS***, data, or proper subelement preceded it.

b) The last ***RE*** in an element is ignored if no data or proper subelement follows it.

c) An ***RE*** that does not immediately follow an ***RS*** or ***RE*** is ignored if no data or proper subelement intervened.

In applying these rules to an element, subelement content is ignored; that is, a proper or included subelement is treated as an atom that ends in the same record in which it begins.

NOTE — For example, in

```
record 1<outer><sub>
record 2</sub>
</outer>record 3
```

the first ***RE*** in the outer element is at the end of record 2. It is treated as data if "sub" is a proper subelement of "outer", but is ignored if "sub" is an included element, because no data or proper subelement would have preceded it in the outer element.

In either case, the first ***RE*** in the subelement is at the end of record 1; it is ignored because no data or proper subelement preceded it in the subelement.

An ***RE*** is deemed to occur immediately prior to the first data or proper subelement that follows it (that is, after any intervening markup declaration, processing instruction, or included subelement).

NOTES

1 A specific character data entity, non-SGML data entity, or SGML subdocument entity, is treated as data, while a processing instruction entity is not.

2 Although the handling of record boundaries is defined by SGML, there is no requirement that SGML documents must be organized in records.

3 No entity, including the *SGML document entity* and external entities, is deemed to start with an ***RS*** or end with an ***RE*** unless it really does.

7.7 Document Type Specification

[28] document type specification = *name group*?

Markup containing a *document type specification* is processed only if:

a) a *name* in the *name group* is that of an active document type; or

b) there is no *name group* (that is, the *document type specification* is empty).

NOTE — An effect of this requirement is that markup with an empty document type specification (that is, no *name group*) will apply to all document instances (or the only one).

A *name group* can be specified only if "CONCUR YES" or "EXPLICIT YES" is specified on the *SGML declaration*.

7.7.1 General Entity References

A general entity must have been declared in the *document type declaration* of each active document type named in the *document type specification* or, if the specification was empty or the entity was not so declared, in the base *document type declaration*.

NOTE — An effect of this requirement is that a general entity declared in the base document type (including the default entity) can be referenced in an instance of any document type in which no general entity with the same name was defined.

7.7.2 Parameter Entity References

A *name group* cannot be specified for a *parameter entity reference* within a *document type declaration* or *link type declaration*. Such an entity must have been declared within the same *document type declaration* or, in the case of the *link type declaration*, within the source *document type declaration*.

7.8 Generic Identifier (GI) Specification

[29] generic identifier specification =
generic identifier | *rank stem*

[30] generic identifier = *name*

A *generic identifier* is valid only if it was specified as an *element type* in the document type definition and:

a) it was named in the *content model* of the *element declaration* for the element in which it occurred; or
b) it was named in an applicable *inclusions* exception; or
c) it is the *document type name* and it occurred in the *start-tag* or *end-tag* of the document element.

7.8.1 Rank Feature

If "RANK YES" is specified on the *SGML declaration*, the provisions of this sub-sub-clause apply.

7.8.1.1 Full Generic Identifier

If the full *generic identifier* is specified for a ranked element, its *rank suffix* becomes the current rank for its *rank stem*, and for the rank stems in any *ranked group* of which the *rank stem* is a member.

7.8.1.2 Rank Stem

A *generic identifier specification* can be a *rank stem* if it was declared via a *ranked element* or member of a *ranked group* in an applicable *element declaration*.

Specifying a *rank stem* is equivalent to specifying the *generic identifier* that is derived by appending the current rank to it. It is an error to specify a *rank stem* if no element previously occurred to establish the current rank for that stem.

7.9 Attribute Specification List

[31] attribute specification list =
*attribute specification**

[32] attribute specification = *s**, (*name*, *s**, ***vi***,
*s**)?, *attribute value specification*

The validity of the attribute specification list is determined by the *attribute definition list* associated with the element. If there is no associated *attribute definition list*, the *attribute specification list* must be empty.

Every attribute for which there is an *attribute definition*, other than an impliable attribute, must be specified (unless markup minimization is used, as described in 7.9.1.1).

There can be only one *attribute specification* for each *attribute definition*.

The leading *s* can only be omitted from an *attribute specification* that follows a delimiter.

7.9.1 Minimization

7.9.1.1 Omitted Attribute Specification

If "SHORTTAG YES" or "OMITTAG YES" is specified on the *SGML declaration*:

a) There need be an *attribute specification* only for a required attribute, and for a current attribute on the first occurrence of any element in whose *attribute definition list* it appears. Other attributes will be treated as though specified with an *attribute value* equal to the declared *default value*.
b) If there is an *attribute value specification* for a current attribute, the specified *attribute value* will become the default value. The new default affects all elements associated with the *attribute definition list* in which the attribute was defined.

7.9.1.2 Omitted Attribute Name

If "SHORTTAG YES" is specified on the *SGML declaration*, the *name* and ***vi*** can be omitted if the *attribute value specification* is an undelimited *name token* that is a member of a group specified in the *declared value* for that attribute.

NOTE — A *name token* can occur in only one group in an *attribute definition list* (see 11.3.3)

7.9.2 Quantities

The normalized length of the *attribute specification list* is the sum of the normalized lengths of each attribute name and attribute value specified, which cannot exceed the "ATTSPLEN" quantity.

The normalized length of an attribute name is the "NORMSEP" quantity plus the number of characters in the name.

7.9.3 Attribute Value Specification

[33] attribute value specification =
attribute value | *attribute value literal*

[34] attribute value literal = (***lit***,
*replaceable character data**, ***lit***) | (***lita***,
*replaceable character data**, ***lita***)

An *attribute value literal* is interpreted as an *attribute value* by replacing references within it, ignoring ***Ee*** and ***RS***, and replacing an ***RE*** or ***SEPCHAR*** with a ***SPACE***.

7.9.3.1 Minimization

If "SHORTTAG YES" is specified on the *SGML declaration*, an *attribute value specification* can be an *attribute value* (that is, not an *attribute value literal*) provided that it contains nothing but name characters.

7.9.4 Attribute Value

[35] attribute value = *character data* | *general entity name* | *id value* | *id reference value* | *id reference list* | *name* | *name list* | *name token* | *name token list* | *notation name* | *number* | *number list* | *number token* | *number token list*

[36] id value = *name*

[37] id reference list = *name list*

[38] id reference value = *name*

[39] name list = *name*, (*s*+, *name*)*

[40] name token list = *name token*, (*s*+, *name token*)*

[41] notation name = *name*

[42] number list = *number*, (*s*+, *number*)*

[43] number token list = *number token*, (*s*+, *number token*)*

The *declared value* parameter of an *attribute definition* determines which of the fourteen types of *attribute value* must be specified.

7.9.4.1 Syntactic Requirements

An *attribute value* must conform to the *declared value*.

If the *declared value* includes a group, the *attribute value* must be a token in that group.

An empty *attribute value literal* can be specified only for an *attribute value* whose type is *character data*.

7.9.4.2 Fixed Attribute

The specified *attribute value* for a fixed attribute must be its *default value*.

7.9.4.3 General Entity Name

The value of a *general entity name* attribute must be the name of an SGML subdocument entity or non-SGML data entity. If the attribute was specified in a *start-tag*, the entity definition must apply to each document instance in which the tag occurs. If the attribute is a link attribute, the definition must apply to the source document type.

NOTE — The notation of a non-SGML data entity must apply to the same document types as the entity.

7.9.4.4 Notation

The *attribute value* of a notation attribute must be a *notation name* that was declared in the same document type declaration as the element.

It is an error to specify a value for a notation attribute if there is an explicit content reference.

NOTE — As the element's *content* will be empty, it is pointless to specify a notation for it. Even if the content reference were to a non-SGML data entity, the applicable notation would be specified by the *notation name* parameter of the entity declaration.

7.9.4.5 Quantities

The normalized length of an *attribute value*, whether specified directly or interpreted from an *attribute value literal*, is the "NORMSEP" quantity plus:

a) for an *id reference list*, *name list*, *name token list*, *number list*, or *number token list* (even if there is only one token in the list), the sum of the number of characters in each token in the list, plus the "NORMSEP" quantity for each token in the list; or

b) for all others, the number of characters in the value, plus the "NORMSEP" quantity for each reference to a "CDATA" or "SDATA" entity.

The normalized length of an *attribute value* cannot exceed the "LITLEN" quantity.

In a single *start tag*, the total number of names in *id reference value* and *id reference list* attribute values, whether defaulted or specified, cannot exceed the "GRPCNT" quantity.

8 Processing Instruction

[44] processing instruction = **pio**, *system data*, **pic**

[45] system data = *character data*

Processing instructions are deprecated, as they reduce portability of the document. If a processing instruction must be used, it should be defined as an entity, so that the *system data* will be confined to the *prolog*, where a recipient of the document can more easily locate and modify it.

A _processing instruction_ that returns data must be defined as an "SDATA" entity and entered with an entity reference. One that does not return data should be defined as a "PI" entity.

No markup is recognized in system data other than the delimiter that would terminate it.

NOTE — The characters allowed in _system data_ and their interpretation are defined by the system. If it is desired to allow non-SGML characters or the **_pic_** delimiter character in _system data_, an alternative way of entering them should be provided so that the actual characters do not occur in the document.

8.1 Quantities

The length of a _processing instruction_, exclusive of its delimiters, cannot exceed the "PILEN" quantity.

9 Common Constructs

9.1 Replaceable Character Data

[46] replaceable character data =
(_data character_ | _character reference_ |
general entity reference | **_Ee_**)*

Markup that would terminate _replaceable character data_ is not recognized in an entity that was referenced from within the same _replaceable character data_.

An **_Ee_** can occur in _replaceable character data_ only if the reference to the entity it terminates occurred in the same _replaceable character data_.

9.2 Character Data

[47] character data = _data character_*

[48] data character = _SGML character_

[49] character = _SGML character_ | **_NONSGML_**

NOTE — A non-SGML character can be entered as a data character within an SGML entity by using a _character reference_.

9.2.1 SGML Character

[50] SGML character = _markup character_ |
DATACHAR

[51] markup character = _name character_ |
function character | **_DELMCHAR_**

[52] name character = _name start character_ |
Digit | **_LCNMCHAR_** | **_UCNMCHAR_**

[53] name start character = **_LC Letter_** | **_UC Letter_**
| **_LCNMSTRT_** | **_UCNMSTRT_**

9.2.2 Function Character

[54] function character = **_RE_** | **_RS_** | **_SPACE_** |
SEPCHAR | **_MSOCHAR_** | **_MSICHAR_** |
MSSCHAR | **_FUNCHAR_**

9.3 Name

[55] name = _name start character_,
_name character_*

[56] number = **_Digit_**+

[57] name token = _name character_+

[58] number token = **_Digit_**, _name character_*

The upper-case form of each character in a _name_, _name token_, _number_, or _number token_, as specified by the "NAMECASE" parameter of the _SGML declaration_, is substituted for the character actually entered.

9.3.1 Quantities

The length of a _name_, _name token_, _number_, or _number token_, cannot exceed the "NAMELEN" quantity.

9.4 Entity References

The replacement text of an entity reference must comply with the syntactic and semantic requirements that govern the context of the reference. For purposes of this rule, a reference to an _SGML subdocument entity_ or a _non-SGML data entity_ is treated like a reference to a _data character_.

NOTE — Such entities can also be accessed by a _general entity name attribute_.

A reference to an undeclared entity is an error unless there is an applicable default entity (see 9.4.4).

A reference to an entity that has already been referenced and has not yet ended is invalid (i.e., entities cannot be referenced recursively).

9.4.1 Quantities

The number of open entities (except for the unreferenced _SGML document entity_ in which the _document_ begins) cannot exceed the "ENTLVL" quantity.

9.4.2 Limits

The number of open SGML subdocument entities cannot exceed the quantity specified on the "SUBDOC" parameter of the _SGML declaration_.

Variable	Characters	Numbers	*Description*
Digit	0 - 9	48 - 57	Digits
Ee	(system signal; not a character)		Entity end signal
LC Letter	a - z	97 - 122	Lower-case letters
Special	' ( ) + , - . / : = ?	39 - 41 43 - 47 58 61 63	Special minimum data characters
UC Letter	A - Z	65 - 90	Upper-case letters

Figure 1 — Character Classes: Abstract Syntax

9.4.3 Obfuscatory Entity References

Any use of entity references that obscures the markup is deprecated.

NOTE — Most such abuses are prohibited by the syntax of SGML. The following principles should be observed (those that say "must" are restatements of syntax rules, stated formally elsewhere, that enforce the principles):

a) The opening delimiter of a tag, *processing instruction*, declaration, literal, or other delimited text, must be in the same entity as the closing delimiter. An entity must not end in delimited text unless it began there, and an entity that begins there must end there.

b) The content of an element or marked section that was declared to be *character data* or *replaceable character data* must (and other elements should) start and end in the same entity.

c) An element's *start-tag* and *end-tag* (and a marked section's *marked section start* and *marked section end*) should be in the same entity, or they should be the replacement text of entities whose references are in the same entity.

d) In a markup declaration, a reference must be replaced either by zero or more consecutive complete parameters (with any intervening *ps* separators), or, within a group, by one or more consecutive complete tokens (with any intervening *ts* separators and/or connectors).

9.4.4 Named Entity Reference

[59] general entity reference = ***ero***,
document type specification, *name*,
reference end

[60] parameter entity reference = ***pero***,
document type specification, *name*,
reference end

An entity *name* is required to be declared by an *entity declaration* before it can be used in a named entity reference, except that an undeclared *name* can be used in a *general entity reference* to refer to the default entity, if one was defined.

9.4.5 Reference End

[61] reference end = (***refc*** | ***RE***)?

NOTE — Ending a reference with an ***RE*** has the effect of suppressing the record end.

The ***refc*** or ***RE*** can be omitted only if the reference is not followed by a *name character*, or by a character that could be interpreted as the omitted *reference end*.

9.4.6 Short Reference

If a *short reference* is mapped to a general entity name in the current map, it is treated as markup and replaced by the named entity. If the *short reference* is mapped to nothing, each character in the delimiter string is treated as an *s* separator if it can be recognized as such, and if not it is treated as data.

9.4.6.1 Equivalent Reference String

A short reference can be removed from a document by replacing it with an equivalent reference string that contains a named entity reference. The entity *name* must be that to which the *short reference* is mapped in the current map.

If the short reference contains any quantity of ***RS*** or ***RE*** characters, the equivalent reference string will include a single ***RS*** or ***RE***, or both, as shown in the following list:

Short Reference	*Equivalent Reference String*
No ***RS*** or ***RE***	***ero***, *name*, ***refc***
RS; no ***RE***	***RS***, ***ero***, *name*, ***refc***
RE; no ***RS***	***ero***, *name*, ***RE***
Both ***RS*** and ***RE***	***RS***, ***ero***, *name*, ***RE***

NOTES

1 Equivalent reference strings are used when a document is converted from a concrete syntax that supports short references to one that does not.

Variable	Characters	Numbers	*Description*
DATACHAR	(implicit)	(implicit)	Dedicated data characters
DELMCHAR	(implicit)	(implicit)	Delimiter characters
FUNCHAR	(none)	(none)	Inert function characters
LCNMCHAR	- .	45 46	Lower-case name characters
LCNMSTRT	(none)	(none)	Lower-case name start characters
MSICHAR	(none)	(none)	Markup-scan-in characters
MSOCHAR	(none)	(none)	Markup-scan-out characters
MSSCHAR	(none)	(none)	Markup-scan-suppress characters
RE		13	Record end character
RS		10	Record start character
SEPCHAR		9	Separator characters
SPACE		32	Space character
UCNMCHAR	- .	45 46	Upper-case name characters
UCNMSTRT	(none)	(none)	Upper-case name start characters

Figure 2 — Character Classes: Concrete Syntax

2 A single ***RS*** and/or ***RE*** is preserved in the equivalent reference string to prevent records from becoming joined, and possibly exceeding a system's maximum length restriction. They are not recognized as data: the ***RS*** because it never is, and the ***RE*** because it serves as the *reference end.*

9.5 Character Reference

[62] character reference = ***cro***, (*function name* | *character number*), *reference end*

[63] function name = "RE" | "RS" | "SPACE" | *name*

[64] character number = *number*

A *name* specified as a *function name* must have been specified as an *added function* in the concrete syntax.

A *character reference* should be used when a character could not otherwise be entered conveniently in the text.

A replacement character is considered to be in the same entity as its reference.

A replacement character is treated as though it were entered directly except that the replacement for a numeric *character reference* is always treated as data.

NOTES

1 A system can determine its own internal representation for a replacement character. Care should be taken to distinguish a normal *function character* (entered directly or as a replacement for a named character reference) from one that replaces a numeric character reference.

2 When a document is translated to a different document character set, the *character number* of each numeric character reference must be changed to the corresponding *character number* of the new set.

9.6 Delimiter Recognition

A delimiter string is recognized as satisfying a delimiter role only within the particular recognition mode (or modes) in which the role is meaningful and, in some cases, only if a contextual constraint is satisfied. The roles, their recognition modes, and any contextual constraints on their recognition, are listed in figure 3. Also shown are the strings assigned to general delimiters in the reference delimiter set, and the character numbers of those strings in the *translation-reference character set* of the reference concrete syntax. The strings assigned as short reference delimiters in the reference delimiter set are shown in figure 4.

9.6.1 Recognition Modes

The recognition modes are:

Mode	***Meaning***
CON	Recognized in *content* and in the *marked section* of marked section declarations that occur in *content*.
CXT	Recognized as part of the contextual sequence of a "CON" or "DS" mode delimiter-in-context. (See below.)

Name	String	Number	Mode	Constraint	*Description of Role*
AND	&	38	GRP		And connector
COM	--	45 45	CXT MD		Comment start or end
CRO	&#	38 35	CON LIT	CREF	Character reference open
DSC	]	93	DS MD		Declaration subset close
DSO	[	91	CXT MD		Declaration subset open
DTGC	]	93	GRP		Data tag group close
DTGO	[	91	GRP		Data tag group open
ERO	&	38	CON LIT	NMS	Entity reference open
ETAGO	</	60 47	CON	GI	End-tag open
GRPC	)	41	GRP		Group close
GRPO	(	40	CXT GRP MD		Group open
LIT	"	34	GRP LIT MD TAG		Literal start or end
LITA	'	39	GRP LIT MD TAG		Literal start or end (alternative)
MDC	>	62	CXT MD		Markup declaration close
MDO	<!	60 33	CON DS	DCL	Markup declaration open
MINUS	-	45	MD		Minus; exclusion
MSC	]]	93 93	CON DS	MSE	Marked section close
NET	/	47	CON TAG	ELEM	Null end-tag
OPT	?	63	GRP		Optional occurrence indicator
OR	|	124	GRP		Or connector
PERO	%	37	DS GRP MD	NMS	Parameter entity reference open
PIC	>	62	PI		Processing instruction close
PIO	<?	60 63	CON DS		Processing instruction open
PLUS	+	43	GRP MD		Required and repeatable; inclusion
REFC	;	59	REF		Reference close
REP	*	42	GRP		Optional and repeatable
RNI	#	35	GRP MD		Reserved name indicator
SEQ	,	44	GRP		Sequence connector
SHORTREF			CON		Short reference (see figure 4)
STAGO	<	60	CON TAG	GI	Start-tag open
TAGC	>	62	CXT TAG		Tag close
VI	=	61	TAG		Value indicator

Figure 3 — Reference Delimiter Set: General

DS Recognized in a declaration subset and in the *marked section* of marked section declarations that occur in a declaration subset.

GRP Recognized in a group.

LIT Recognized in a literal.

MD Recognized in a *markup declaration* (other than in a group or declaration subset).

PI Recognized in a *processing instruction*.

REF Recognized in a *general entity reference*, *parameter entity reference*, or *character reference*.

TAG recognized in a *start-tag* or *end-tag*.

9.6.2 Contextual Constraints

The most common constraint is that the delimiter string must start a delimiter-in-context in which it is followed by one of the listed contextual sequences:

CREF *name start character* or **Digit**

DCL *name start character*, **com**, **dso**, or **mdc**

GI *name start character*, or, if "SHORTTAG YES" is specified on the *SGML declaration*, **tagc**, or, if "CONCUR YES" is specified on the *SGML declaration*, **grpo**

MSE **mdc**

NMS *name start character*, or, if "CONCUR YES" is specified on the *SGML declaration*, **grpo**

Another contextual constraint is:

ELEM In "CON" mode, recognized only within an element whose *start-tag* was a *net-enabling start-tag*; in "TAG" mode, no constraints.

9.6.3 Order of Recognition

Delimiter strings (including any required contextual sequences) are recognized in the order they occur, with no overlap.

NOTE — For example, if "abc" and "bcd" are delimiter strings, and the document contains "abcde", then "abc" will be recognized and parsing will continue at "d", so "bcd" will not be recognized.

This rule holds true even if the recognized delimiter is semantically incorrect, or is a short reference that is mapped to nothing.

NOTE — For example, in the reference delimiter set, the solidus (/) will be recognized as part of an ***etago*** delimiter-in-context rather than as the *net*, even if the *end-tag* GI was not declared, or is not the GI of an open element.

9.6.4 Delimiters Starting with the Same Character

If multiple delimiter strings start with the same character, only the longest delimiter string or delimiter-in-context among them will be recognized at a given point in the document.

NOTE — For example, if "ab" and "abc" are delimiters and the document contains "abcd", then "abc" will be recognized and parsing will continue at "d", so "ab" will not be recognized.

This rule holds true even if the longer delimiter is semantically incorrect, or is a short reference that is mapped to nothing.

NOTE — If, in the previous example, "ab" and "abc" were short reference delimiters, short reference "abc" alone would be recognized and short reference "ab" would not be, even if short reference "abc" were mapped to nothing in the current map and short reference "ab" were mapped to an entity.

9.6.5 Short References with Blank Sequences

If there is a B sequence in the definition of a short reference delimiter, it will cause recognition of a blank sequence in the *content*. The minimum length of the blank sequence is the length of the B sequence.

NOTE — That is, one B means one or more blanks, two B's means two or more blanks, etc.

A string that could be recognized as more than one delimiter will be considered the string of the most specific delimiter that it satisfies.

NOTE — For example, a tab character would be recognized as "&#TAB;" in preference to "B", and three spaces would be recognized as "BBB" in preference to "BB".

9.6.5.1 Quantities

The length of a blank sequence recognized as a short reference cannot exceed the "BSEQLEN" quantity. If an actual blank sequence is longer, only the first "BSEQLEN" characters will be included in the short reference string, and parsing will resume with the following character.

9.6.6 Name Characters

If a *name start character* is assigned to a delimiter role, it will be recognized as a delimiter (in preference to recognition as a *name start character*) if a *name token* has not already begun; if found within a *name token*, it will be treated as a *name character*.

If general upper-case substitution is specified by the "NAMECASE" parameter of the *SGML declaration*, then for purposes of delimiter recognition, a *name character* assigned to a delimiter role is treated as though it were its upper-case form.

9.7 Markup Suppression

An ***MSOCHAR*** suppresses recognition of markup until an ***MSICHAR*** or entity end occurs. An ***MSSCHAR*** does so for the next character in the same entity (if any).

NOTE — An ***MSOCHAR*** occurring in *character data* or other delimited text will therefore suppress recognition of the closing delimiter. An ***MSSCHAR*** could do so if it preceded the delimiter.

If markup recognition has not been suppressed by an ***MSOCHAR***, an ***MSICHAR*** has no effect on markup recognition, but is not an error.

If markup recognition has been suppressed by an ***MSOCHAR***, a subsequent ***MSOCHAR*** or ***MSSCHAR*** has no effect on markup recognition, but is not an error.

An ***MSOCHAR*** that follows an ***MSSCHAR*** has no effect on markup recognition.

9.8 Capacity

The size and complexity of a document must not exceed the number of capacity points allowed by the document capacity set for the objects occurring in the document.

The names of the total and individual capacities, together with the values assigned to them in the reference capacity set, are shown in figure 5. The set's public identifier is:

```
ISO 8879-1986//CAPACITY Reference//EN
```

String	Number	*Description*
&#TAB;	9	Horizontal tab
&#RE;	13	Record end
&#RS;	10	Record start
&#RS;B	10 66	Leading blanks
&#RS;&#RE;	10 13	Empty record
&#RS;B&#RE;	10 66 13	Blank record
B&#RE;	66 13	Trailing blanks
&#SPACE;	32	Space
BB	66 66	Two or more blanks
"	34	Quotation mark
#	35	Number sign
%	37	Percent sign
'	39	Apostrophe
(	40	Left parenthesis
)	41	Right parenthesis
*	42	Asterisk
+	43	Plus sign
,	44	Comma
-	45	Hyphen
--	45 45	Two hyphens
:	58	Colon
;	59	Semicolon
=	61	Equals sign
@	64	Commercial at
[	91	Left square bracket
]	93	Right square bracket
^	94	Circumflex accent
_	95	Low line
{	123	Left curly bracket
|	124	Vertical line
}	125	Right curly bracket
~	126	Tilde

Figure 4 — Reference Delimiter Set: Short References

The points accumulated for each type of object cannot exceed the value of that object's individual capacity, and the total points for all objects cannot exceed the "TOTALCAP" value.

The capacity values must be sufficient for the greatest capacity requirement among the possible sets of concurrent instances or chains of link processes that could be processed at once.

Points are counted for the SGML document entity and SGML text entities referenced from it, plus the set of possible open subdocument entities and SGML text entities referenced from them that would require the largest capacity.

NOTE — As an example of capacity calculation, when a concrete syntax with 32 short references and a "NAMELEN" of 8 is used, a capacity of 30818 or more would be required to accommodate a document with 100 entities averaging 70 characters (800+7000), 200 element types (1600) with 2000 tokens in content models (16000) and 25 exceptions groups with a total of 50 names (200+400), 50 attributes with default values averaging 20 characters (400+1000) and 100 attribute name tokens (800), 5 data content notations with identifiers averaging 50 characters (40+250), 50 ID and 50 IDREF attributes (400+400), 5 short reference maps (5x(8+(8x32))=1320), and a single implicit link type declaration with 4 link sets, each containing 5 source element names (40+168).

10 Markup Declarations: General

NOTE — The declaration names and keywords in the syntax productions are reference reserved names that can be redefined for a variant concrete syntax with the *reserved name use* parameter of the *SGML declaration*.

10.1 Parts of Declarations

10.1.1 Parameter Separator

[65] ps = *s* | **Ee** | *parameter entity reference* | *comment*

A *parameter entity reference* can be used wherever a parameter could occur. The entity must consist of zero or more of the consecutive complete parameters that follow the *ps* in which the reference occurs, together with any intervening *ps* separators. The entity must end within the same declaration.

An **Ee** can occur in a *ps* only if the reference to the entity it terminates occurs in a *ps* in the same declaration.

A required *ps* that is adjacent to a delimiter or another *ps* can be omitted if no ambiguity would be created thereby.

A *ps* must begin with an *s* if omitting it would create an ambiguity.

10.1.2 Parameter Literal

[66] parameter literal = (**lit**, *replaceable parameter data*, **lit**) | (**lita**, *replaceable parameter data*, **lita**)

[67] replaceable parameter data = (*data character* | *character reference* | *parameter entity reference* | **Ee**)*

A *parameter literal* is interpreted as a parameter (or token) by replacing references while the declaration is being processed.

Except for parameter entity and character references, no markup is recognized in a *parameter literal* other than the terminal **lit** or **lita**, and those are not recognized within the replacement text of a reference.

NOTE — If the literal is in the *entity text* parameter of an *entity declaration*, markup characters in its text could be recognized when the entity is referenced.

An **Ee** can occur in *replaceable parameter data* only if the reference to the entity it terminates occurred in the same *replaceable parameter data*.

10.1.2.1 Quantities

The length of an interpreted *parameter literal* cannot exceed the "LITLEN" quantity (unless some other restriction is applied in the context in which it is used).

10.1.3 Group

[68] name token group = **grpo**, *ts**, *name token*, (*ts**, *connector*, *ts**, *name token*)*, *ts**, **grpc**

[69] name group = **grpo**, *ts**, *name*, (*ts**, *connector*, *ts**, *name*)*, *ts**, **grpc**

[70] ts = *s* | **Ee** | *parameter entity reference*

Only one type of *connector* should be used in a single *name group* or *name token group*.

NOTE — No specific type of *connector* is mandated, so that a group defined in an entity can be referenced as both a *model group* (where the specific connector is meaningful) and a *name group* (where it is not).

A token can occur only once in a single *name group* or *name token group*.

A *parameter entity reference* can be used anywhere in a group that a token could occur. The entity must consist of one or more of the consecutive complete tokens that follow the *ts* in which the reference occurs in the same group (i.e., at the same nesting level), together with any surrounding or intervening *ts* separators and any intervening connectors. The entity must end within the same group.

An **Ee** can occur in a *ts* only if:

- the *ts* follows a token (as opposed to a *connector*, **grpo**, or **dtgo**); and
- the reference to the entity the **Ee** terminates occurs in the same group (i.e., at the same nesting level).

10.1.3.1 Quantities

The number of tokens in a group cannot exceed the "GRPCNT" quantity.

10.1.4 Declaration Separator

[71] ds = *s* | **Ee** | *parameter entity reference* | *comment declaration* | *processing instruction* | *marked section declaration*

A *parameter entity reference* in a *ds* must refer to an entity that consists of zero or more complete markup declarations and/or *ds* separators.

An ***Ee*** can occur in a *ds* only if the reference to the entity it terminates occurs in a *ds* in the same parameter.

10.1.5 Associated Element Type

[72] associated element type = *generic identifier*
| *name group*

Each *name* in the *name group* must be a *generic identifier*.

10.1.6 External Identifier

[73] external identifier = ("SYSTEM" |
("PUBLIC", *ps*+, *public identifier*)), (*ps*+,
system identifier)?

[74] public identifier = *minimum literal*

[75] system identifier = (***lit***, *system data*, ***lit***) |
(***lita***, *system data*, ***lita***)

The *system identifier* can be omitted if the system can generate it from the *public identifier* and/or other information available to it.

If "FORMAL YES" is specified on the *SGML declaration*, a *public identifier* is interpreted as a *formal public identifier* (see 10.2) and a formal public identifier error can occur.

NOTE — It is still a *minimum literal*, and all requirements pertaining to minimum literals apply.

10.1.6.1 Quantities

The length of a *system identifier*, exclusive of delimiters, cannot exceed the "LITLEN" quantity.

10.1.6.2 Capacities

The number of characters of *entity text* counted towards the ENTCHCAP capacity for an *external identifier* is that of its *system identifier* component, whether specified or generated (and exclusive of delimiters).

10.1.7 Minimum Literal

[76] minimum literal = (***lit***, *minimum data*, ***lit***) |
(***lita***, *minimum data*, ***lita***)

[77] minimum data = *minimum data character**

[78] minimum data character = ***RS*** | ***RE*** | ***SPACE***
| ***LC Letter*** | ***UC Letter*** | ***Digit*** | ***Special***

A *minimum literal* is interpreted by ignoring ***RS*** and replacing a sequence of one or more ***RE*** and/or ***SPACE*** characters with a single ***SPACE***.

10.1.7.1 Quantities

The length of an interpreted *minimum literal*, exclusive of delimiters, cannot exceed the "LITLEN" quantity of the reference quantity set, regardless of the concrete syntax used.

10.2 Formal Public Identifier

[79] formal public identifier = *owner identifier*,
"//", *text identifier*

A *formal public identifier* cannot contain consecutive solidi ("//") except where expressly permitted by this sub-clause.

NOTE — As a ***public identifier*** is a *minimum literal*, ***RS*** characters are removed, and sequences of one or more ***RE*** and/or ***SPACE*** characters replaced by a single ***SPACE***, prior to interpretation as a *formal public identifier*. The *minimum literal* length limitation applies to the interpreted text (see 10.1.7.1).

10.2.1 Owner Identifier

[80] owner identifier = *ISO owner identifier* |
registered owner identifier |
unregistered owner identifier

NOTE — In formulating an *owner identifier*, standards such as ISO 3166 can be helpful.

10.2.1.1 ISO Owner Identifier

[81] ISO owner identifier = *minimum data*

The usual form of *ISO owner identifier* can be used only when the *public identifier* identifies an ISO publication or is assigned within one. It consists of the ISO publication number, without the language suffix.

NOTE — For example, the *ISO owner identifier* for public text defined in this document is "ISO 8879-1986" in all translations. If the public text is translated, that fact is indicated by specifying the appropriate *public text language* in the *text identifier*.

A special form of *ISO owner identifier* can be used only when the public text is an ISO registered character set and the *public text class* is "CHARSET". It consists of the string "ISO Registration Number ", followed by the registration number of the character set.

10.2.1.2 Registered Owner Identifier

[82] registered owner identifier = "+//",
minimum data

NOTE — A registered owner identifier could be a citation of a national or industry standard, or some other unique identifier assigned in accordance with ISO 9070.

10.2.1.3 Unregistered Owner Identifier

[83] unregistered owner identifier = "-//",
minimum data

NOTE — An unregistered owner identifier could be a (presumably unique) designation created by a trade organization or other user community, or by an individual.

10.2.2 Text Identifier

[84] text identifier = *public text class*, ***SPACE***,
unavailable text indicator?,
public text description, "//",
(*public text language* |
public text designating sequence), ("//",
public text display version)?

[85] unavailable text indicator = "-//"

If the *unavailable text indicator* is present, the text is unavailable public text; otherwise, it is available public text.

If the *public text class* is "CHARSET", the *text identifier* includes a *public text designating sequence*; otherwise, it includes a *public text language*.

A *text identifier* cannot be the same as another *text identifier* in a *formal public identifier* that has the same *owner identifier*.

NOTE — If two public texts with the same owner have the same public text description, they must be of different classes, versions, etc.

10.2.2.1 Public Text Class

[86] public text class = *name*

The *name* must be one that identifies an SGML construct in the following list:

Name	*SGML Construct*
CAPACITY	*capacity set*
CHARSET	*character data*
DOCUMENT	*SGML document*
DTD	*document type declaration subset*
ELEMENTS	*element set*
ENTITIES	*entity set*
LPD	*link type declaration subset*
NONSGML	*non-SGML data entity*
NOTATION	*character data*
SHORTREF	*short reference set*
SUBDOC	*SGML subdocument entity*
SYNTAX	*concrete syntax*
TEXT	*SGML text entity*

The *name* must be entered with upper-case letters.

NOTE — When appropriate, a system can use the *public text class* to determine strategies for converting the public text from its interchange form into a referenceable entity that uses the system character set and concrete syntax.

10.2.2.2 Public Text Description

[87] public text description = *ISO text description*
| *minimum data*

An *ISO text description* can be used only when the *public identifier* identifies an ISO publication. It consists of the last element of the publication title, without the part number designation (if any).

NOTE — For example, the *ISO text description* for ISO 8632/4 is "Clear text encoding".

10.2.2.3 Public Text Language

[88] public text language = *name*

The *public text language* must be a two-character *name*, entered with upper-case letters. The *name* must be the language code from ISO 639 that identifies the principal natural language used in the public text.

NOTES

1 The natural language will affect the usability of some public text classes more than others.

2 The portions of text most likely to be influenced by a natural language include the data, defined names, and comments.

3 A system can use the *public text language* to facilitate automatic language translation.

10.2.2.4 Public Text Designating Sequence

[89] public text designating sequence
= *minimum literal*

The *minimum literal* must be the external form of a designating escape sequence prescribed by ISO 2022 for the character set referenced by the *public identifier*. If the public text is an ISO registered character set, the designating escape sequence must be a registered escape sequence for that set.

NOTES

1 For example, the external form of the registered G0 designating sequence for the graphic characters of ISO 646 IRV (registered character set 002) is:

```
ESC 2/8 4/0
```

2 For registered character sets, the *public text designating sequence* uniquely identifies the public text. For other character sets, it uniquely identifies the public text with respect to a particular *owner identifier*.

10.2.2.5 Public Text Display Version

[90] public text display version = *minimum data*

The *public text display version* must be omitted if the *public text class* is "CAPACITY", "CHARSET", "NOTATION", or "SYNTAX". For other classes, if the public text is device-dependent, the text identifier must include a *public text display version* that describes the devices supported or coding scheme used.

When a system accesses public text for which a *public text display version* could have been specified but was not, it must substitute the best available device-dependent version for the display device in use. If there is none, no substitution occurs.

NOTE — This mechanism is particularly useful with character entity sets.

10.3 Comment Declaration

[91] comment declaration = ***mdo***, (*comment*, (*s* | *comment*)*)?, ***mdc***

[92] comment = ***com***, *SGML character**, ***com***

No markup is recognized in a *comment*, other than the ***com*** delimiter that terminates it.

10.4 Marked Section Declaration

[93] marked section declaration =
marked section start,
status keyword specification, ***dso***,
marked section, *marked section end*

[94] marked section start = ***mdo***, ***dso***

[95] marked section end = ***msc***, ***mdc***

[96] marked section = *SGML character**

The *marked section* must comply with the syntactic and semantic requirements that govern the context in which the *marked section declaration* occurs.

A *marked section end* that occurs outside of a *marked section declaration* is an error.

10.4.1 Quantities

The number of open marked section declarations cannot exceed the "TAGLVL" quantity.

10.4.2 Status Keyword Specification

[97] status keyword specification = (*ps*+, *qualified status keyword* | *status keyword* | "TEMP")*, *ps**

[98] qualified status keyword = *status keyword qualifier*, *status keyword*

[99] status keyword qualifier = *name group*

[100] status keyword = "CDATA" | "IGNORE" | "INCLUDE" | "RCDATA"

where

IGNORE specifies that the section is treated as though there were no characters in the *marked section*, except that a nested *marked section declaration* is recognized so that the correct ending can be found, but its *status keyword specification* is ignored.

INCLUDE specifies that the *marked section* is not to be ignored.

CDATA specifies that the *marked section* is treated as *character data*.

RCDATA specifies that the *marked section* is treated as *replaceable character data*.

TEMP identifies the section as a temporary part of the document that might need to be removed at a later time.

In the event of a conflicting specification, the status keywords have the following priority (highest shown first):

"IGNORE"
"CDATA"
"RCDATA"
"INCLUDE"

If none is specified, "INCLUDE" is assumed.

A *qualified status keyword* is ignored unless the *name group* includes the name of an active document type or active link type.

A *qualified status keyword* can be specified only if "CONCUR YES" or "EXPLICIT YES" is specified on the *SGML declaration*.

If the effective status is "CDATA" or "RCDATA", a nested *marked section declaration* within the *marked section* in the same entity is not allowed; the *marked section declaration* is terminated by the first *marked section end*.

If the effective status is "IGNORE", an ***Ee*** is not allowed in the *marked section*.

NOTE — The scan of an "IGNORE", "CDATA", or "RCDATA" marked section ignores virtually all markup but marked section ends. As a result, processing instructions, attribute values, literals, character data elements, and comments are not recognized to be such, so their characters are also scanned. This could cause erroneous results if the characters look like marked sections. In most cases, problems can be avoided by entering such characters with references, instead of directly.

10.5 Entity Declaration

[101] entity declaration = ***mdo***, "ENTITY", *ps*+, *entity name*, *ps*+, *entity text*, *ps**, ***mdc***

10.5.1 Entity Name

[102] entity name = *general entity name* | *parameter entity name*

[103] general entity name = *name* | (***rni***, "DEFAULT")

[104] parameter entity name = ***pero***, *ps*+, *name*

where

DEFAULT means the entity is the default entity.

An attempt to redefine an entity is ignored, but is not an error.

NOTE — This requirement allows an entity declaration in a document type declaration subset to take priority over a later declaration of the same entity in a public document type definition.

The *ps* in a *parameter entity name* is required, even though it follows a delimiter.

10.5.1.1 Quantities

A *name* in a *parameter entity name* must be at least one character shorter than the "NAMELEN" quantity.

10.5.1.2 Capacities

Points are counted towards the "ENTCAP" capacity for the default entity, and for each unique entity name that is defaulted in one or more references.

Points are counted towards the "ENTCHCAP" capacity for the default entity. They are counted for defaulted entities only if a system identifier is generated.

10.5.2 Entity Text

[105] entity text = *parameter literal* | *data text* | *bracketed text* | *external entity specification*

If a *parameter literal* alone is specified as the *entity text*, the interpreted *parameter literal* is the replacement text of the entity.

An ***Ee*** is deemed to be present at the end of the replacement text; it is not entered explicitly.

10.5.3 Data Text

Data text is treated as *character data* when referenced, regardless of the context in which the entity reference occurs. It is specified as a *parameter literal*, whose characters, after resolution of references in the usual manner, will comprise the *entity text*.

[106] data text = ("CDATA" | "SDATA" | "PI"), *ps*+, *parameter literal*

where

CDATA means the interpreted *parameter literal* is the replacement text of a character data entity.

SDATA means the interpreted *parameter literal* is the replacement text of a specific character data entity.

PI means the interpreted *parameter literal* is the replacement text of a processing instruction data entity.

"CDATA" or "SDATA" can be specified only if the *entity name* is a *general entity name*.

A *processing instruction* that returns data must be defined as an "SDATA" entity.

NOTES

1 A "CDATA" or "SDATA" entity must be referenced in a context in which *character data* can occur, and a "PI" entity in a context in which a *processing instruction* could occur.

2 "SDATA" is normally specified if the entity would be redefined for different applications, systems, or output devices; for example, if the data contained processing instructions, or characters not present in the *translation-reference character set*.

10.5.4 Bracketed Text

[107] bracketed text = ("STARTTAG" | "ENDTAG"
| "MS" | "MD"), *ps+*, *parameter literal*

where the keywords mean that the entity consists of the interpreted *parameter literal* bracketed with delimiters, as follows:

STARTTAG means preceded by ***stago*** and followed by ***tagc***.

ENDTAG means preceded by ***etago*** and followed by ***tagc***.

MS means preceded by a *marked section start* and followed by a *marked section end*.

MD means preceded by ***mdo*** and followed by ***mdc***.

NOTE — Bracketed text is simply text with delimiter characters; there is no requirement that the entities form valid start-tags or other constructs. As usual, the validity of the entity is determined in context whenever the entity is referenced.

10.5.4.1 Quantities

The length of an interpreted *parameter literal* in *bracketed text* cannot exceed the "LITLEN" quantity, less the length of the bracketing delimiters.

10.5.5 External Entity Specification

[108] external entity specification
= *external identifier*, (*ps+*, *entity type*)?

[109] entity type = "SUBDOC" | ("NDATA", *ps+*,
notation name)

where

SUBDOC means the entity is an *SGML subdocument entity*.

NDATA means the entity is a *non-SGML data entity*.

The *entity type* can be specified only if the *entity name* is a *general entity name*.

If the *entity type* is omitted, the entity is an *SGML text entity*.

The *notation name* must be declared in the same document type definition as the entity.

NOTE — It need not have been defined prior to this declaration, but must be defined prior to a reference to the entity.

A non-SGML data entity can reference (in its own notation) other non-SGML data entities and SGML subdocument entities. Such entities should be declared in the same document type definition as it is.

"SUBDOC" can be specified only if "SUBDOC YES" is specified on the *SGML declaration*.

11 Markup Declarations: Document Type Definition

11.1 Document Type Declaration

[110] document type declaration = ***mdo***,
"DOCTYPE", *ps+*, *document type name*,
(*ps+*, *external identifier*)?, (*ps+*, **dso**,
document type declaration subset, **dsc**)?,
*ps**, ***mdc***

[111] document type name = *generic identifier*

[112] document type declaration subset =
(*entity set* | *element set* |
short reference set)*

[113] entity set = (*entity declaration* | *ds*)*

[114] element set = (*element declaration* |
attribute definition list declaration |
notation declaration | *ds*)*

[115] short reference set = (*entity declaration* |
short reference mapping declaration |
short reference use declaration | *ds*)*

The *document type name* must be a *generic identifier* that does not occur as a *document type name* or *link type name* in the same *prolog*.

The *external identifer* points to an entity that is referenced at the end of the declaration subset and is considered to be part (or all) of it. The effective document type definition is the combination of the declarations entered in the subset and the external ones.

NOTE — A parameter entity declaration in the subset will have priority over another declaration for that entity in the external entity, as the external entity is parsed later.

A *document type declaration* must contain an *element declaration* for the *document type name*.

A *short reference set* is permitted only in the base document type declaration.

11.2 Element Declaration

[116] element declaration = **_mdo_**, "ELEMENT",
_ps_+, _element type_, (_ps_+,
omitted tag minimization)?, _ps_+,
(_declared content_ | _content model_), _ps_*,
mdc

The order in which elements and characters occur within an element in the document instance must comply with the element type definition specified in the element declaration.

The _omitted tag minimization_ parameter and its preceding _ps_ can be omitted only if "OMITTAG NO" is specified on the _SGML declaration_.

11.2.1 Element Type

[117] element type = _generic identifier_ |
name group | _ranked element_ |
ranked group

Within a document type definition, a _generic identifier_ can be specified only once in an _element type_ parameter, whether directly or indirectly.

If the _element type_ is a group, the group members are defined in the order their names occur, and the definition applies to each of them.

If the _element type_ is a _name group_, each _name_ is a _generic identifier_.

11.2.1.1 Ranked Element

[118] ranked element = _rank stem_, _ps_+,
rank suffix

[119] ranked group = **_grpo_**, _ts_*, _rank stem_, (_ts_*,
connector, _ts_*, _rank stem_)*, _ts_*, **_grpc_**, _ps_+,
rank suffix

[120] rank stem = _name_

[121] rank suffix = _number_

The _generic identifier_ specified by a _ranked element_ or member of a _ranked group_ is the _rank stem_ with the _rank suffix_ appended.

11.2.1.2 Quantities

The length of a _generic identifier_ cannot exceed the "NAMELEN" quantity.

11.2.2 Omitted Tag Minimization

[122] omitted tag minimization =
start-tag minimization, _ps_+,
end-tag minimization

[123] start-tag minimization = "O" | **_minus_**

[124] end-tag minimization = "O" | **_minus_**

where

O means that omission of the tag under the conditions specified in 7.3.1 is not a markup error.

minus means that omission of the tag under the conditions specified in 7.3.1 is a markup error.

minus should be specified for start-tag minimization if omission is prohibited by 7.3.1.

"O" should be specified for end-tag minimization if the element has a declared value of "EMPTY".

NOTE — Specifying "O" serves as a reminder that empty elements do not have end-tags (although this has nothing to do with markup minimization).

11.2.3 Declared Content

[125] declared content = "CDATA" | "RCDATA" |
"EMPTY"

where

RCDATA means that the _content_ is _replaceable character data_.

CDATA means that the _content_ is _character data_.

EMPTY means the _content_ is empty.

11.2.4 Content Model

[126] content model = (_model group_ | "ANY"),
(_ps_+, _exceptions_)?

[127] model group = **_grpo_**, _ts_*, _content token_, (_ts_*,
connector, _ts_*, _content token_)*, _ts_*, **_grpc_**,
occurrence indicator?

[128] content token = _primitive content token_ |
model group

[129] primitive content token = (**_rni_**, "PCDATA") |
element token | _data tag group_

[130] element token = _generic identifier_,
occurrence indicator?

where

ANY means the _content_ is _mixed content_ in which _parsed character data_ and any elements defined in the same document type definition are allowed.

PCDATA means *parsed character data* is allowed.

NOTE — The ***rni*** distinguishes this keyword from an *element token* of "PCDATA".

If "#PCDATA" or a *data tag group* is present in a *model group*, the element's *content* is *mixed content*; if not, it is *element content*. In either case, the elements and data characters of the *content* must conform to the *content model* by satisfying *model group* tokens and *exceptions* in the following order of priority:

a) a repetition of the most recent satisfied token, if it has a ***rep*** or ***plus*** occurrence indicator; or

b) some other token in a *model group*, possibly as modified by exclusion *exceptions*(see 11.2.5); or

c) a token in an inclusion *exceptions* group (see 11.2.5.1).

NOTE — For example, in an instance of the following element

```
<!element e (a+ | b)+>
```

successive "a" elements will satisfy repetitions of the element token, rather than repetitions of the model group.

All data characters occurring between successive tags are considered to satisfy a single "#PCDATA" token, even if some were declared to be character data by a marked section declaration.

11.2.4.1 Connector

[131] connector = ***and*** | ***or*** | ***seq***

If there is more than one *content token* in a *model group*, the ordering and selection among their corresponding content is determined by the *connector*, as follows:

seq All must occur, in the order entered.
and All must occur, in any order.
or One and only one must occur.

Only one kind of *connector* can occur in a single *model group* (but a *model group* nested within it could have a different *connector*).

11.2.4.2 Occurrence Indicator

[132] occurrence indicator = ***opt*** | ***plus*** | ***rep***

The corresponding content of each selected *content token* must occur once and only once unless the contrary is indicated by the token's *occurrence indicator*, as follows:

opt Optional (0 or 1 time).
plus Required and repeatable (1 or more times).
rep Optional and repeatable (0 or more times).

The "#PCDATA" content token is regarded as having an *occurrence indicator* of ***rep***.

An inherently optional token is treated as having an ***opt*** occurrence indicator if none is specified, or as having a ***rep*** occurrence indicator if ***plus*** is specified.

11.2.4.3 Ambiguous Content Model

A *content model* cannot be ambiguous; that is, an element or character string that occurs in the document instance must be able to satisfy only one *primitive content token* without look-ahead.

NOTE — For example, the content model in

```
<!element e ((a, b?), b)>
```

is ambiguous because after an "a" element occurs, a "b" element could satisfy either of the remaining tokens. The ambiguity can be avoided by using intermediate elements, as in:

```
<!element e (f, b)>
<!element f (a, b?)>
```

Here the token satisfied by "b" is determined unambiguously by whether the "f" element ends before the "b" occurs. (The theoretical basis of content models is discussed in annex H.)

11.2.4.4 Data Tag Group

[133] data tag group = ***dtgo***, *ts**, *generic identifier*, *ts**, ***seq***, *ts**, *data tag pattern*, *ts**, ***dtgc***

[134] data tag pattern = (*data tag template group* | *data tag template*), (*ts**, ***seq***, *ts**, *data tag padding template*)?

[135] data tag template group = ***grpo***, *ts**, *data tag template*, (*ts**, ***or***, *ts**, *data tag template*)*, *ts**, ***grpc***

[136] data tag template = *parameter literal*

[137] data tag padding template = *parameter literal*

A *data tag group* is interpreted as a ***seq*** group with two tokens: an element GI followed by "#PCDATA".

NOTE — For example, with the reference delimiter set, the *model group*

```
([hours, (" :" | ":"), " "], minutes)
```

is treated as though it were

```
((hours, #PCDATA), minutes)
```

A *data tag group* can only be present in the base *document type declaration*.

A *parameter literal* in a *data tag pattern* is interpreted in the normal manner, except that a numeric character reference to a non-SGML character or *function character* is prohibited.

11.2.4.5 Quantities

The content model nesting level cannot exceed the "GRPLVL" quantity.

The grand total of the tokens at all levels of a model group cannot exceed the "GRPGTCNT" quantity.

The length of an interpreted *parameter literal* in a *data tag pattern* cannot exceed the "DTEMPLEN" quantity.

11.2.5 Exceptions

[138] exceptions = (*exclusions*, (*ps*+, *inclusions*)?) | *inclusions*

The *exceptions* apply anywhere in an instance of the element, including subelements whose *content* is *mixed content* or *element content*.

At any point in a document instance, if an element is both an applicable inclusion and an exclusion, it is treated as an exclusion.

11.2.5.1 Inclusions

[139] inclusions = ***plus***, *name group*

Inclusions modify the effect of model groups to which they apply in the manner shown in the following example: given that "Q" is a generic identifier or group in the model group, "x" is its occurrence indicator (or empty if there is no occurrence indicator), and "R1" through "Rn" are applicable inclusions, then a token

```
Qx
```

is treated as though it were

```
(R1|R2|...|Rn)*,(Q,(R1|R2|...|Rn)*)x
```

An element that can satisfy an element token in the content model is considered to do so, even if the element is also an inclusion.

NOTES

1 Inclusions should not be used for contextual subelements. They should be used only for elements that are not logically part of the content at the point where they occur in the document, such as index entries or floating figures.

2 An ***RE*** that follows an inclusion will normally be ignored, while one that follows a proper subelement will be treated as data (see 7.6.1).

11.2.5.2 Exclusions

[140] exclusions = ***minus***, *name group*

Exclusions modify the effect of model groups to which they apply by precluding options that would otherwise have been available (just as though the user had chosen to leave optional elements out of the document).

It is an error if an exclusion attempts to modify the effect of a model group in any other way. In particular, it is an error if:

a) an exclusion applies to tokens other than those in *inclusions*, those having an ***opt*** or ***rep*** occurrence indicator, or those that are members of ***or*** groups; or
b) an exclusion attempts to change a token's required or optional status.

NOTE — For example, it is prohibited to exclude all members of a required *model group*, as the group would then no longer be required.

11.3 Attribute Definition List Declaration

[141] attribute definition list declaration = ***mdo***, "ATTLIST", *ps*+, *associated element type*, *ps*+, *attribute definition list*, *ps**, ***mdc***

[142] attribute definition list = *attribute definition*, (*ps*+, *attribute definition*)*

[143] attribute definition = *attribute name*, *ps*+, *declared value*, *ps*+, *default value*

An *associated element type* cannot be associated with another attribute definition list in the same declaration subset in which this list occurs.

11.3.1 Quantities

The total number of attribute names and name tokens in the *attribute definition list* cannot exceed the "ATTCNT" quantity.

11.3.2 Attribute Name

[144] attribute name = *name*

An *attribute name* can be specified only once in the same *attribute definition list*.

11.3.3 Declared Value

[145] declared value = "CDATA" | "ENTITY" |
"ID" | "IDREF" | "IDREFS" | "NAME" |
"NAMES" | "NMTOKEN" | "NMTOKENS" |
"NUMBER" | "NUMBERS" | "NUTOKEN" |
"NUTOKENS" | *notation* |
name token group

[146] notation = "NOTATION", *ps*+, *name group*

where

CDATA means the *attribute value* is *character data*.

ENTITY means the *attribute value* is a *general entity name*.

ID means the *attribute value* is an *id value*.

IDREF means the *attribute value* is an *id reference value*.

IDREFS means the *attribute value* is an *id reference list*.

NAME means the *attribute value* is a *name*.

NAMES means the *attribute value* is a *name list*.

NMTOKEN means the *attribute value* is a *name token*.

NMTOKENS means the *attribute value* is a *name token list*.

NOTATION means the *attribute value* is a *notation name* that identifies the data content notation of the element's *content*. The *name group* specifies the permissible notation names.

NUMBER means the *attribute value* is a *number*.

NUMBERS means the *attribute value* is a *number list*.

NUTOKEN means the *attribute value* is a *number token*.

NUTOKENS means the *attribute value* is a *number token list*.

"ID" and "NOTATION" can each be declared only once in the *attribute definition list*.

A token cannot occur more than once in an *attribute definition list*, even in different groups.

"NOTATION" cannot be declared for an element whose *declared content* is "EMPTY".

11.3.4 Default Value

[147] default value = ((***rni***, "FIXED", *ps*+)?,
attribute value specification) | (***rni***,
("REQUIRED" | "CURRENT" | "CONREF" |
"IMPLIED"))

where

FIXED means the attribute is a fixed attribute.

REQUIRED means the attribute is a required attribute.

CURRENT means the attribute is a current attribute.

CONREF means the attribute is a content reference attribute.

IMPLIED means the attribute is an impliable attribute.

NOTE — Specifying an empty literal is not equivalent to specifying "IMPLIED".

If an attribute value is specified in this parameter, it must conform to the syntactic requirements specified in 7.9.4.1 .

NOTE — Further testing of general entity name and notation values is performed when the default value is used in an attribute specification.

If the *declared value* is "ID", the *default value* must be "IMPLIED" or "REQUIRED".

"CONREF" cannot be declared for an element whose *declared content* is "EMPTY".

11.3.4.1 Quantities

"CONREF", "REQUIRED", and "IMPLIED" have normalized lengths of zero.

11.3.4.2 Capacities

In calculating "ATTCHCAP" requirements, the default value of a current attribute is given the length of the longest value specified for the attribute in the document.

11.4 Notation Declaration

[148] notation declaration = ***mdo***, "NOTATION",
ps+, *notation name*, *ps*+,
notation identifier, *ps**, ***mdc***

[149] notation identifier = *external identifier*

NOTE — The *notation identifier* should contain sufficient information to allow the notation interpreter to be invoked with the proper parameters.

The *notation name* cannot be specified on another *notation declaration* in the same document type definition.

If a *notation identifier* includes a *public identifier* and "FORMAL YES" is specified on the *SGML declaration*, the *public text class* must be "NOTATION".

11.5 Short Reference Mapping Declaration

[150] short reference mapping declaration = **mdo**, "SHORTREF", *ps*+, *map name*, (*ps*+, *parameter literal*, *ps*+, *name*)+, *ps*∗, **mdc**

[151] map name = *name*

The *map name* cannot be specified on another *short reference mapping declaration* in the same document type definition.

The interpreted *parameter literal* is a *short reference* delimiter that is mapped to the *name* of a general entity that is defined in the same document type definition.

NOTE — A general entity is required because the short reference will be replaced by a named entity reference if the document is sent to a system that does not support short references, and parameter entity references are not permitted in *content*.

A *short reference* delimiter can be mapped only once in a *short reference mapping declaration*.

If a *short reference* delimiter is not specified it is considered to be mapped to nothing.

11.6 Short Reference Use Declaration

[152] short reference use declaration = **mdo**, "USEMAP", *ps*+, *map specification*, (*ps*+, *associated element type*)?, *ps*∗, **mdc**

[153] map specification = *map name* | (**rni**, "EMPTY")

where

EMPTY means the map is the empty map.

11.6.1 Use in Document Type Declaration

If the declaration occurs in a *document type declaration*, the *associated element type* must be specified. The named map will become the current map whenever an element of an associated type becomes the current element.

The *map name* must be defined on a *short reference mapping declaration* in the same *document type declaration*.

NOTE — It need not have been defined prior to this declaration, but must be defined prior to becoming the current map.

Specifying an associated element type that is already associated with a map is not an error, but is ignored.

11.6.2 Use in Document Instance

If the declaration occurs in a document instance, an *associated element type* cannot be specified. The map becomes the current map for this instance of the current element.

The *map name* must have been defined on a *short reference mapping declaration* in the document type definition to which the instance conforms.

11.6.3 Current Map

A map is the current map as long as its associated element is the current element. It can become superceded for an instance of the element: either temporarily by a subelement becoming the current element, or permanently by a *short reference use declaration* occurring in an instance of the element.

If an element type has no associated short reference map, the current map for an instance of the element is the map that is current when the instance begins. If the element is a document element, the current map will be the empty map.

12 Markup Declarations: Link Process Definition

12.1 Link Type Declaration

[154] link type declaration = **mdo**, "LINKTYPE", *ps*+, *link type name*, (*simple link specification* | *implicit link specification* | *explicit link specification*), (*ps*+, *external identifier*)?, (*ps*+, **dso**, *link type declaration subset*, **dsc**)?, *ps*∗, **mdc**

[155] link type name = *name*

The *link type name* must be different from any other *link type name* or *document type name* in the same *prolog*.

The *external identifer* points to an entity that is referenced at the end of the declaration subset and is considered to be part (or all) of it. The effective link process definition is the combination of the declarations entered in the subset and the external ones.

NOTE — A parameter entity declaration in the subset will have priority over another declaration for that entity in the external entity, as the external entity is parsed later.

12.1.1 Simple Link Specification

[156] simple link specification = ***rni***, "SIMPLE",
rni, "IMPLIED"

where

SIMPLE means the link is a simple link.

IMPLIED means the *result document type name* is implied by the application.

If a simple link is specified, "SIMPLE YES" must be specified on the SGML declaration link type features parameter.

The source document type is the base document type.

12.1.2 Implicit Link Specification

[157] implicit link specification =
source document type name, ***rni***,
"IMPLIED"

where

IMPLIED means the *result document type name* is implied by the application.

If an implicit link is specified, "IMPLICIT YES" must be specified on the SGML declaration link type features parameter.

The *source document type name* must be the base document type name.

12.1.3 Explicit Link Specification

[158] explicit link specification =
source document type name,
result document type name

[159] source document type name
= *document type name*

[160] result document type name
= *document type name*

If an explicit link is specified, "EXPLICIT YES" must be specified on the SGML declaration *link type features* parameter.

The *source document type name* must be the base document type, or another document type that is a result document type in a chain of processes.

Each *document type name* must previously have been specified on a *document type declaration* in the same *prolog*.

12.1.3.1 Limits

The number of link processes in the longest chain cannot exceed the quantity specified for "EXPLICIT" on the *link type features* parameter of the *SGML declaration*.

12.1.4 Link Type Declaration Subset

[161] link type declaration subset =
link attribute set?, (*link set declaration* |
link set use declaration | *entity set*)*

[162] link attribute set =
(*attribute definition list declaration* |
entity set)*

12.1.4.1 Parameter Entities

Entity declarations in the declaration subset must define parameter entities. When this link type is active, the entity declarations are treated as if they occurred at the end of the source document type declaration subset.

A link type declaration can contain parameter entity references to entities defined in the source document type declaration, as well as in its own declaration subset.

12.1.4.2 Link Attributes

An associated element type of an attribute definition list must be a source element type.

The *declared value* of a link attribute cannot be "ID", "IDREF", "IDREFS", or "NOTATION".

"CONREF" cannot be specified for a link attribute.

12.1.4.3 Simple Link

If the declaration defines a simple link, the declaration subset must consist solely of a *link attribute set* that contains no more than one *attribute definition list declaration*. The list must be associated with the base document element type, and can define only fixed attributes.

12.2 Link Set Declaration

[163] link set declaration = ***mdo***, "LINK", *ps*+,
link set name, (*ps*+,
source element specification, *ps*+,
result element specification)+, *ps**, ***mdc***

[164] link set name = *name*

The *link set name* cannot be specified on another *link set declaration* in the same link type definition.

A *source element* can be linked to a given *result element*, or to "#IMPLIED", only once in a link set.

NOTE — That is, a given source/result pairing must be unique in a link set.

12.2.1 Source Element Specification

[165] source element specification =
associated element type,
link attribute specification?

[166] link attribute specification = *ps*+, ***dso***,
attribute specification list, *ps**, ***dsc***

An associated element type in a *source element specification* must be defined in the source *document type declaration*.

The validity of a link *attribute specification list* is determined by the attribute definition list associated with the source element type in the link type declaration subset. All element types associated with an attribute specification must be associated with the same definition.

The *link attribute specification must be omitted if its attribute specification list* is empty.

12.2.2 Result Element Specification

[167] result element specification = (***rni***,
"IMPLIED") | (*generic identifier*,
result attribute specification?)

[168] result attribute specification = *ps*+, ***dso***,
attribute specification list, *ps**, ***dsc***

where

IMPLIED means the *result element* is implied by the application.

A result element *generic identifier* must be defined in the result *document type declaration*.

The validity of the result *attribute specification list* is determined by the *attribute definition list* associated with the result element in the result element document type declaration.

The *result attribute specification* must be omitted if its *attribute specification list* is empty.

12.3 Link Set Use Declaration

[169] link set use declaration = ***mdo***, "USELINK",
ps+, *link set specification*, *ps*+,
(*associated element type* | *link type name*),
*ps**, ***mdc***

[170] link set specification = *link set name* | (***rni***,
"EMPTY")

where

EMPTY means the link set is the empty link set.

12.3.1 Use in Link Type Declaration

If the declaration occurs in a *link type declaration*, an *associated element type* must be specified. The named link set will become the current link set whenever an element of an associated type becomes the current element.

The *link set name* must be defined on a *link set declaration* in the same *link type declaration*.

NOTE — It need not have been defined prior to this declaration, but must be defined prior to becoming the current link set.

An *associated element type* must be an element type defined in the source *document type declaration*. If it is already associated with a link set in this *link type declaration*, the specification is ignored, but is not an error.

12.3.2 Use in Document Instance

If the declaration occurs in a document instance, a *link type name* must be specified. The link set becomes the current link set for this instance of the current element.

The *link type name* must be that of the *link type declaration* in which the link set was defined.

12.3.3 Current Link Set

A link set is the current link set as long as its element is the current element. It can become superceded for an instance of the element: either temporarily by a subelement becoming the current element, or permanently by a *link set use declaration* occurring in an instance of the element.

If an element type has no associated link set, the current link set for an instance of the element is the link set that is current when the instance begins. If the element is a document element, the current link set is the empty link set.

13 SGML Declaration

[171] SGML declaration = ***mdo***, "SGML", *ps*+,
"ISO 8879-1986", *ps*+,
document character set, *ps*+, *capacity set*,
ps+, *concrete syntax scope*, *ps*+,
concrete syntax, *ps*+, *feature use*, *ps*+,
application-specific information, *ps**, ***mdc***

The reference concrete syntax must be used in the *SGML declaration*, regardless of the concrete syntax used in the remainder of the document.

Only markup characters (in the reference concrete syntax) and *minimum data* characters can be used in the parameters and comments, although the replacement text of a character reference could be an SGML character other than a markup or minimum data character.

NOTES

1 The *SGML declaration* is intended for human consumption (in printed form!) as well as for machine processing, as it enables the recipient of a document to determine whether a system can process it "as is", whether character translation or other algorithmic conversion is needed (for example, if document markup features or a different delimiter set were used), or whether conversion that could require manual intervention is needed (for example, if document type features or a different quantity set were used).

2 A character reference such as "&#222;" is valid because the reference consists solely of markup and minimum data characters, even though the replacement text does not.

3 No entity references can occur in an SGML declaration (because no entities could have been declared).

13.1 Document Character Set

[172] document character set = "CHARSET",
ps+, *character set description*

The document character set must include a coded representation, as a single bit combination, for each significant SGML character.

NOTE — If the document uses two concrete syntaxes, the markup characters of both are significant SGML characters.

As part of the translation of a document to a new character set, the character numbers in this parameter and any numeric character references in the document must be changed.

NOTE — It is recognized that the recipient of a document must be able to translate it to his system character set before the document can be processed by machine. There are two basic approaches to communicating this information:

a) If the character set is standard, registered, or otherwise capable of being referenced by an identifying name or number, that identifier can be communicated to the recipient of the document. The communication must necessarily occur outside of the document; for example, in a field of the document interchange data stream, or via other (probably non-electronic) media.

b) For other character sets, a human-readable copy of the SGML declaration will provide sufficient information.

13.1.1 Character Set Description

[173] character set description =
(*base character set*, *ps*+,
described character set portion)+

The described character set portions must collectively describe each character number in the described character set once and only once.

13.1.1.1 Base Character Set

[174] base character set = "BASESET", *ps*+,
public identifier

The *public identifier* is a human-readable identifier of the *base character set*.

NOTE — For example, a standard or registered name or number, or other designation that will be understood by the expected recipients of the document.

If "FORMAL YES" is specified on the *other features* parameter, the *public identifier* must be a *formal public identifier* with a *public text class* of "CHARSET".

13.1.1.2 Described Character Set Portion

[175] described character set portion =
"DESCSET", (*ps*+,
character description)+

[176] character description =
described set character number, *ps*+,
number of characters, *ps*+,
(*base set character number* |
minimum literal | "UNUSED")

[177] described set character number
= *character number*

[178] base set character number
= *character number*

[179] number of characters = *number*

where

UNUSED means that no meaning is assigned to the specified character numbers in the described set.

The specified *number of characters* in the described character set, beginning with the specified *described set character number*, are assigned meanings as follows:

a) If a *base set character number* is specified, the meanings are those of the corresponding characters in the *base character set*, beginning with the specified *base set character number*.

   NOTE — If a base set character number is unused, no meaning is assigned to the corresponding described set character number.

b) If a *minimum literal* is specified, the meaning or meanings are as described in the literai.

   NOTE — A *minimum literal* should be specified only if no character in the *base character set* has the desired meaning.

c) If "UNUSED" is specified, no meanings are assigned.

13.1.2 Non-SGML Character Identification

Each *character number* to which no meaning is assigned by the *character set description* is added to ***NONSGML***, thereby identifying it as a non-SGML character.

NOTE — After receipt and translation of a document, the non-SGML characters may be different because the new document character set may map control characters to different coded representations.

A shunned character must be identified as a non-SGML character, unless it is a significant SGML character.

NOTES

1 For example, in figure 8, characters numbered 9, 10, and 13, which are shunned characters, are nevertheless not assigned as non-SGML characters because they are function characters.

2 If the document uses two concrete syntaxes, the shunned characters of both are subject to this requirement.

13.2 Capacity Set

[180] capacity set = "CAPACITY", *ps*+,
        (("PUBLIC", *ps*+, *public identifier*) |
        ("SGMLREF", (*ps*+, *name*, *ps*+,
        number)+))

The specified *name* is a name given to a capacity in figure 5. The capacity is assigned the value indicated by the specified *number*.

The reference capacity set value is used for any capacity for which no replacement is assigned by this parameter.

NOTE — The "SGMLREF" keyword, which is required (and therefore redundant) when a public identifier is not used, is a reminder of this rule for human readers of the SGML declaration.

The capacity values must express limits that are not exceeded by the document. They must be sufficient for the greatest capacity requirement among the possible sets of concurrent instances or chains of link processes that could be processed at once.

The value assigned to "TOTALCAP" must equal or exceed the largest individual capacity.

If "FORMAL YES" is specified on the *other features* parameter, the *public identifier* must be a *formal public identifier* with a *public text class* of "CAPACITY".

13.3 Concrete Syntax Scope

This parameter specifies whether a declared concrete syntax must be used for the entire document, or whether the reference concrete syntax can be used in the prologs.

[181] concrete syntax scope = "SCOPE", *ps*+,
        ("DOCUMENT" | "INSTANCE")

where

DOCUMENT means the declared concrete syntax is used throughout the document.

INSTANCE means the reference concrete syntax is used in prologs and the declared concrete syntax is used in document instance sets.

If "INSTANCE" is specified, the declared concrete syntax must meet the following requirements:

a) the syntax-reference character set must be the same as that of the reference concrete syntax;

b) the significant SGML characters must be such that the start of a document instance set is always distinguishable from the end of its prolog; and

c) the quantity set values must equal or exceed those of the reference quantity set.

Name	Value	Points	*Object for which capacity points are counted*
TOTALCAP	35000	(total)	Grand total of individual capacity points.
ENTCAP	35000	NAMELEN	Entity defined.
ENTCHCAP	35000	1	Character of entity text.
ELEMCAP	35000	NAMELEN	Element defined.
GRPCAP	35000	NAMELEN	Token at any level in a content model.
EXGRPCAP	35000	NAMELEN	Exclusion or inclusion exceptions group.
EXNMCAP	35000	NAMELEN	Name in an exclusion or inclusion exceptions group.
ATTCAP	35000	NAMELEN	Attribute defined.
ATTCHCAP	35000	1	Character of default attribute value (keyword counts as zero).
AVGRPCAP	35000	NAMELEN	Token defined in an attribute value name group or name token group.
NOTCAP	35000	NAMELEN	Data content notation defined.
NOTCHCAP	35000	1	Character in a notation identifier.
IDCAP	35000	NAMELEN	ID attribute specified (explicitly or by default).
IDREFCAP	35000	NAMELEN	IDREF attribute specified (explicitly or by default).
MAPCAP	35000	NAMELEN	(plus NAMELEN for each short reference delimiter in the concrete syntax) Short reference map declared.
LKSETCAP	35000	NAMELEN	Link types or link sets defined.
LKNMCAP	35000	NAMELEN	Document type or element in a link type or link set declaration.

Figure 5 — Reference Capacity Set

13.4 Concrete Syntax

[182] concrete syntax = "SYNTAX", *ps*+,
(*public concrete syntax* |
(*shunned character number identification*,
ps+, *syntax-reference character set*, *ps*+,
function character identification, *ps*+,
naming rules, *ps*+, *delimiter set*, *ps*+,
reserved name use, *ps*+, *quantity set*))

The reference concrete syntax or core concrete syntax should be used unless a variant concrete syntax is necessitated by such requirements as the keyboard, display capabilities, or characteristics of the national language.

13.4.1 Public Concrete Syntax

[183] public concrete syntax = "PUBLIC", *ps*+,
public identifier, (*ps*+, "SWITCHES", (*ps*+,
character number, *ps*+,
character number)+)?

where

SWITCHES means that markup characters in the specified concrete syntax have been switched.

The pairs of character numbers are in the *syntax-reference character set* of the *public concrete syntax*. The first of each pair is a markup character in the identified concrete syntax and the second is a character that substitutes for it in every instance in which the first character was used.

NOTE — The concrete syntax that results from the switches must meet all the usual requirements, just as if it had been declared explicitly.

If "FORMAL YES" is specified on the *other features* parameter, the *public identifier* must be a *formal public identifier* with a *public text class* of "SYNTAX".

13.4.2 Shunned Character Number Identification

[184] shunned character number identification =
"SHUNCHAR", *ps*+, ("NONE" |
(("CONTROLS" | *character number*), (*ps*+,
character number)*))

where

NONE means there are no shunned character numbers.

CONTROLS means that any character number that the document character set considers to be the coded representation of a control character, and not a graphic character, is a shunned character.

Each specified *character number* is identified as a shunned character number.

NOTE — Character numbers in this parameter need not (and should not) be changed when a document is translated to another character set.

13.4.3 Syntax-reference Character Set

[185] syntax-reference character set =
character set description

The syntax-reference character set must include a coded representation, as a single bit combination, of each significant SGML character.

13.4.4 Function Character Identification

[186] function character identification =
"FUNCTION", *ps*+, "RE", *ps*+,
character number, *ps*+, "RS", *ps*+,
character number, *ps*+, "SPACE", *ps*+,
character number, (*ps*+, *added function*,
ps+, *function class*, *ps*+,
character number)*

[187] added function = *name*

[188] function class = "FUNCHAR" | "MSICHAR" |
"MSOCHAR" | "MSSCHAR" | "SEPCHAR"

where the keywords identify the *added function*, as follows:

FUNCHAR means an inert function character.

SEPCHAR means a separator character.

MSOCHAR means a markup-scan-out character.

MSICHAR means a markup-scan-in character.

MSSCHAR means a markup-scan-suppress character.

The character with the specified *character number* in the *syntax-reference character set* is assigned to the function.

A character can be assigned to only one function.

An *added function* cannot be "RE", "RS", "SPACE", or another *added function*.

"MSICHAR" must be specified for at least one *added function* if "MSOCHAR" is specified for an *added function*.

NOTE — When code extension is used, shift characters could be assigned to markup suppression functions to avoid false delimiter recognition, but only by sacrificing the ability to use entity references to obtain device independence (see clause E.3).

13.4.5 Naming Rules

[189] naming rules = "NAMING", *ps*+,
"LCNMSTRT", *ps*+, *parameter literal*,
ps+, "UCNMSTRT", *ps*+,
parameter literal, *ps*+, "LCNMCHAR",
ps+, *parameter literal*, *ps*+,
"UCNMCHAR", *ps*+, *parameter literal*,
ps+, "NAMECASE", *ps*+, "GENERAL",
ps+, ("NO" | "YES"), *ps*+, "ENTITY",
ps+, ("NO" | "YES")

where

LCNMSTRT means each *character* in the literal (if any) is added to ***LCNMSTRT***.

UCNMSTRT Each *character* in the literal (if any) is added to ***UCNMSTRT*** as the associated upper-case form of the character in the corresponding position of ***LCNMSTRT***.

LCNMCHAR means each *character* in the literal (if any) is added to ***LCNMCHAR***.

UCNMCHAR Each *character* in the literal (if any) is added to ***UCNMCHAR*** as the associated upper-case form of the character in the corresponding position of ***LCNMCHAR***.

NAMECASE specifies whether upper-case substitution is to be performed for entity references and entity names ("ENTITY") and/or for all other names, name tokens, number tokens, and delimiter strings ("GENERAL").

YES means an ***LC Letter*** will be replaced by the corresponding ***UC Letter***, and a *character* in ***LCNMSTRT*** or ***LCNMCHAR*** will be replaced by its associated upper-case form.

NO means no upper-case substitution will take place.

The upper-case form of a name character can be the same as the lower-case.

A *character* assigned to ***LCNMCHAR***, ***UCNMCHAR***, ***LCNMSTRT***, or ***UCNMSTRT*** cannot be an ***LC Letter***, ***UC Letter*** , ***Digit***, ***RE***, ***RS***, ***SPACE***, or ***SEPCHAR***.

A *character* assigned to ***LCNMCHAR*** or ***UCNMCHAR*** cannot be assigned to ***LCNMSTRT*** or ***UCNMSTRT***.

UCNMCHAR must have the same number of characters as ***LCNMCHAR***; ***UCNMSTRT*** must have the same number of characters as ***LCNMSTRT***.

13.4.6 Delimiter Set

[190] delimiter set = "DELIM", *ps*+,
general delimiters, *ps*+,
short reference delimiters

13.4.6.1 General Delimiters

A delimiter or delimiter-in-context must differ from every other delimiter and delimiter-in-context that can be recognized in the same mode.

The use of a *name start character* or ***Digit*** in a delimiter string is deprecated.

[191] general delimiters = "GENERAL", *ps*+,
"SGMLREF", (*ps*+, *name*, *ps*+,
parameter literal)*

The specified *name* is a name given to a general delimiter role in figure 3. The interpreted parameter literal is assigned to the role.

General delimiter roles not assigned by this parameter are assigned as in the reference delimiter set.

NOTE — The "SGMLREF" keyword, which is required (and therefore redundant), is a reminder of this rule for human readers of the SGML declaration.

A general delimiter string cannot consist solely of function characters. A general delimiter string that contains such characters in combination with others is permitted, but is deprecated.

13.4.6.2 Short Reference Delimiters

[192] short reference delimiters = "SHORTREF",
ps+, ("SGMLREF" | "NONE"), (*ps*+,
parameter literal)*

where

SGMLREF means that the short references assigned by the reference delimiter set are included in this delimiter set.

NONE means that no *short reference* delimiters are assigned except for those assigned by this parameter.

The interpreted parameter literal is assigned as a short reference delimiter string.

A *parameter literal* can have a single B sequence, which cannot be preceded or followed by a blank sequence or by a reference to a character that can occur in a blank sequence.

A *short reference* string longer than a single character is deprecated unless the string is a common keyboarding convention or coding sequence.

A *short reference* string is deprecated if:

a) it contains all, or the start, of a delimiter or delimiter-in-context that is recognized in CON mode; and

b) it is likely to create the impression that the delimiter was erroneously ignored.

NOTE — In applying this requirement, remember that a short reference is recognized as a delimiter even when it is not mapped to an entity. Therefore, a general delimiter within it will never be recognized as such.

13.4.7 Reserved Name Use

[193] reserved name use = "NAMES", *ps*+,
"SGMLREF", (*ps*+, *name*, *ps*+, *name*)*

The first of each pair of names is a reference reserved name, and the second is a name that is to replace it in the declared concrete syntax.

NOTE — Reserved names that occur only in the *SGML declaration*, including delimiter role, quantity, and capacity names, cannot be replaced, as the *SGML declaration* is always in the reference concrete syntax.

The reference reserved name is used for any reserved name for which no replacement is assigned by this parameter.

NOTE — The "SGMLREF" keyword, which is required (and therefore redundant), is a reminder of this rule for human readers of the SGML declaration.

The replacement for a reference reserved name cannot be another reference reserved name, or a replacement for one.

13.4.8 Quantity Set

[194] quantity set = "QUANTITY", *ps*+,
"SGMLREF", (*ps*+, *name*, *ps*+, *number*)*

The specified *name* is a name given to a quantity in figure 6, which also shows the value assignments that constitute the reference quantity set. The designated quantity is assigned the value indicated by the specified *number*.

The reference quantity set value is used for any quantity for which no replacement is assigned by this parameter.

Name	Value	*Description of Quantity*
ATTCNT	40	Number of attribute names and name tokens in an element's *attribute definitions*.
ATTSPLEN	960	Normalized length of a start-tag's *attribute specifications*.
BSEQLEN	960	Length of a blank sequence in a short reference string.
DTAGLEN	16	Length of a data tag.
DTEMPLEN	16	Length of a data tag template or pattern template (undelimited).
ENTLVL	16	Nesting level of entities (other than primary).
GRPCNT	32	Number of tokens in a group.
GRPGTCNT	96	Grand total of tokens at all levels of a model group.
GRPLVL	16	Nesting level of model groups (including first level).
LITLEN	240	Length of a literal or delimited *attribute value* (undelimited).
NAMELEN	8	Length of a *name*, *name token*, *number*, etc.
NORMSEP	2	Used in lieu of counting separators in calculating normalized lengths.
PILEN	240	Length of a *processing instruction* (undelimited).
TAGLEN	960	Length of a *start-tag* (undelimited).
TAGLVL	24	Nesting level of open elements.

Figure 6 — Reference Quantity Set

NOTE — The "SGMLREF" keyword, which is required (and therefore redundant), is a reminder of this rule for human readers of the SGML declaration.

13.5 Feature Use

[195] feature use = "FEATURES", *ps*+,
markup minimization features, *ps*+,
link type features, *ps*+, *other features*

13.5.1 Markup Minimization Features

[196] markup minimization features = "MINIMIZE",
ps+, "DATATAG", *ps*+, ("NO" | "YES"),
ps+, "OMITTAG", *ps*+, ("NO" | "YES"),
ps+, "RANK", *ps*+, ("NO" | "YES"), *ps*+,
"SHORTTAG", *ps*+, ("NO" | "YES")

where

NO means the feature is not used.
YES means the feature is used.
DATATAG means data characters may serve simultaneously as tags.
OMITTAG means some tags may be omitted altogether.
RANK means element ranks may be omitted from tags.
SHORTTAG means short tags with omitted delimiters, attribute specifications, or generic identifiers may be used.

NOTE — The use of short references is not specified in this parameter because it is specified by the "SHORTREF" parameter.

13.5.2 Link Type Features

[197] link type features = "LINK", *ps*+, "SIMPLE",
ps+, ("NO" | "YES"), *ps*+, "IMPLICIT",
ps+, ("NO" | "YES"), *ps*+, "EXPLICIT",
ps+, ("NO" | ("YES", *ps*+, *number*))

where

NO means the feature is not used.
YES means the feature is used.
EXPLICIT means explicit link process definitions may be used and the longest chain of link processes has the specified number of links (1 or more).
IMPLICIT means implicit link process definitions may be used.
SIMPLE means simple link process definitions may be used.

13.5.3 Other Features

[198] other features = "OTHER", *ps*+, "CONCUR",
ps+, ("NO" | ("YES", *ps*+, *number*)), *ps*+,
"SUBDOC", *ps*+, ("NO" | ("YES", *ps*+,
number)), *ps*+, "FORMAL", *ps*+, ("NO" |
"YES")

where

NO means the feature is not used.
YES means the feature is used.
CONCUR means instances of the specified number of document types (1 or more) may occur concurrently with an instance of the base document type.

SUBDOC	means the specified number of SGML subdocument entities (1 or more) may be open at one time.
FORMAL	means that public identifiers are interpreted as formal public identifiers.

13.6 Application-specific Information

[199] application-specific information =
"APPINFO", *ps*+, ("NONE" |
minimum literal)

where

NONE	means that no application-specific information has been specified.

The *minimum literal* specifies application-specific information that is applicable to the document.

14 Reference and Core Concrete Syntaxes

The reference concrete syntax is defined by the SGML declaration *concrete syntax* parameter shown in figure 7. Its *public identifier* is:

```
"ISO 8879-1986//SYNTAX Reference//EN"
```

The core concrete syntax is the same as the reference concrete syntax, except that "NONE" is specified for the "SHORTREF" parameter. Its *public identifier* is:

```
"ISO 8879-1986//SYNTAX Core//EN"
```

NOTE — The *syntax-reference character set* of the reference concrete syntax is ISO 646 IRV. That set consists of characters numbered 0 through 127, which correspond to the like-numbered characters in ISO 4873 and ISO 6937. The set was chosen because it is the simplest standard character set that contains all of the significant SGML characters used in the reference concrete syntax. This choice does not restrict the document character sets that can be used, nor their size.

15 Conformance

15.1 Conforming SGML Document

If an SGML document complies with all provisions of this International Standard it is a conforming SGML document.

15.1.1 Basic SGML Document

If a conforming SGML document uses the reference concrete syntax throughout, the reference capacity set, and only the SHORTTAG and OMITTAG features, it is a basic SGML document.

NOTE — A typical SGML declaration for a basic SGML document is shown in figure 8. Only the *document character set* parameter can differ from one basic SGML document to another.

15.1.2 Minimal SGML Document

If a conforming SGML document uses the core concrete syntax, the reference capacity set, and no features, it is a minimal SGML document.

15.1.3 Variant Conforming SGML Document

If a conforming SGML document uses a variant concrete syntax, it is a variant conforming SGML document.

15.2 Conforming SGML Application

If an SGML application meets the requirements of this sub-clause it is a conforming SGML application.

15.2.1 Application Conventions

A conforming SGML application's conventions can affect only areas that are left open to specification by applications.

NOTE — Some examples are: naming conventions for elements and entities, or a content convention that data characters not in the translation-reference character set always be entered by references rather than directly.

15.2.2 Conformance of Documents

A conforming SGML application shall require its documents to be conforming SGML documents, and shall not prohibit any markup that this International Standard would allow in such documents.

NOTE — For example, an application markup convention could recommend that only certain minimization functions be used, but could not prohibit the use of other functions if they are allowed by the formal specification.

15.2.3 Conformance of Documentation

A conforming SGML application's documentation shall meet the requirements of this International Standard (see 15.5).

15.3 Conforming SGML System

If an SGML system meets the requirements of this sub-clause it is a conforming SGML system.

NOTE — An effect of this sub-clause is to require that a conforming SGML system be able to process a minimal SGML document.

```
         SYNTAX

SHUNCHAR CONTROLS 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
            18 19 20 21 22 23 24 25 26 27 28 29 30 31 127 255
BASESET  "ISO 646-1983//CHARSET
          International Reference Version (IRV)//ESC 2/5 4/0"
DESCSET  0 128 0
FUNCTION RE                13
         RS                10
         SPACE             32
         TAB      SEPCHAR   9
NAMING   LCNMSTRT ""
         UCNMSTRT ""
         LCNMCHAR "-."     -- Lower-case hyphen, period are --
         UCNMCHAR "-."     -- same as upper-case (45 46).   --
         NAMECASE GENERAL  YES
                  ENTITY   NO
DELIM    GENERAL  SGMLREF
         SHORTREF SGMLREF
NAMES    SGMLREF
QUANTITY SGMLREF
```

Figure 7 — Reference Concrete Syntax

15.3.1 Conformance of Documentation

A conforming SGML system's documentation shall meet the requirements of this International Standard (see 15.5).

15.3.2 Conformance to System Declaration

A conforming SGML system shall be capable of processing any conforming SGML document that is not inconsistent with the system's *system declaration* (see 15.6).

NOTE — As this International Standard does not define data content notations or system data, a system's inability to process such text does not affect whether it is a conforming SGML system.

15.3.3 Support for Reference Concrete Syntax

A conforming SGML system shall be able to parse documents in the reference concrete syntax in addition to any variant concrete syntax that it may support.

NOTE — This requirement can be satisfied by converting from the reference to the system concrete syntax when a document is received.

A conforming SGML system that can create or revise SGML documents shall be able to do so for SGML documents that use the reference concrete syntax.

NOTE — This requirement can be satisfied by converting from the system to the reference concrete syntax when a document is to be exported.

If a conforming SGML system allows a user to edit SGML markup directly, it must also allow the reference concrete syntax to be edited directly.

If an SGML system does not support short references in any syntax, the core concrete syntax can be used instead of the reference concrete syntax.

NOTES

1 A system can meet the requirement to support the reference concrete syntax by using separate programs or modules.

2 This requirement should not be interpreted to require that interchange be restricted to the reference concrete syntax; documents can be interchanged in variant concrete syntaxes as well.

```
                <!SGML "ISO 8879-1986"
        -- This document is a basic SGML document. --

                         CHARSET
        -- 8-bit document character set whose first 128 characters
           are the same as the syntax-reference character set. --
BASESET  "ISO 646-1983//CHARSET
          International Reference Version (IRV)//ESC 2/5 4/0"
DESCSET     0   9  UNUSED
            9   2   9
           11   2  UNUSED
           13   1  13
           14  18  UNUSED
           32  95  32
          127   1  UNUSED
BASESET  "ISO Registration Number 109//CHARSET
          ECMA-94 Right Part of Latin Alphabet Nr. 3//ESC 2/13 4/3"
DESCSET   128  32  UNUSED
          160   5  32
          165   1  "SGML User's Group logo"
          166  88  38            -- Includes 5 unused for NONSGML --
          254   1  127           -- Move 127 to unused position as --
          255   1  UNUSED        -- 255 is shunned character number --

CAPACITY PUBLIC "ISO 8879-1986//CAPACITY Reference//EN"
SCOPE    DOCUMENT
SYNTAX   PUBLIC "ISO 8879-1986//SYNTAX Reference//EN"

                        FEATURES
MINIMIZE DATATAG NO  OMITTAG  YES  RANK     NO  SHORTTAG YES
LINK     SIMPLE  NO  IMPLICIT NO   EXPLICIT NO
OTHER    CONCUR  NO  SUBDOC   NO   FORMAL   NO

                    APPINFO NONE>
```

Figure 8 — Typical SGML Declaration for Basic SGML Document

15.3.4 Support for Reference Capacity Set

A conforming SGML system shall be able to parse documents whose capacities are no greater than those of the reference capacity set. If SGML documents can be created with the system, the system shall be able to create documents whose capacities are no greater than those of the reference capacity set.

15.3.5 Consistency of Parsing

A conforming SGML system shall parse the same document identically for all applications and processes that operate on it.

NOTES

1 An application program, using normal interfaces to the SGML parser, should not be able to affect the state of the parse, such as by generating text and causing it to be parsed as though it were part of the document. Documentation for application developers should make them aware of this requirement.

2 This requirement enables a system to be tested for conformance without having to test every application.

15.3.6 Application Conventions

A conforming SGML system shall not enforce application conventions as though they were requirements of this International Standard.

NOTE — Warnings of the violation of application conventions can be given, but they must be distinguished from reports of markup errors.

15.4 Validating SGML Parser

If an SGML parser in a conforming SGML system meets the requirements of this sub-clause, it is a validating SGML parser.

NOTE — A conforming SGML system need not have a validating SGML parser. Implementors can therefore decide whether to incur the overhead of validation in a given system. A user whose text editing system allowed the validation and correction of SGML documents, for example, would not require the validation process to be repeated when the documents are processed by a formatting system.

15.4.1 Error Recognition

A validating SGML parser shall find and report a reportable markup error if one exists, and shall not report an error when none exists.

A validating SGML parser can optionally report:

a) an ambiguous content model;
b) an exclusion that could change a group's required or optional status in a model;
c) a failure to observe a capacity limit;
d) an error in the SGML declaration;
e) the occurrence of a non-SGML character; or
f) a formal public identifier error.

NOTE — This International Standard does not specify how a markup error should be handled, beyond the requirement for reporting it. In particular, it does not state whether the erroneous text should be treated as data, and/or whether an attempt should be made to continue processing after an error is found.

15.4.2 Identification of SGML Messages

Reports of SGML markup errors, including optional reports, shall be identified as SGML messages in such a manner as to distinguish them clearly from all other messages.

15.4.3 Content of SGML Messages

A report of an SGML markup error, including an optional report, shall state the nature and location of the error in sufficient detail to permit its correction.

NOTE — This requirement is worded to allow implementors maximum flexibility to meet their user and system requirements. More precise suggestions are made in clause F.4.

15.5 Documentation Requirements

The objectives of this International Standard will be met most effectively if users, at all levels, are aware that SGML documents conform to an International Standard that is independent of any application or parser. The documentation of a conforming SGML system or application shall further such awareness.

NOTE — These requirements are intended to help users apply knowledge gained from one SGML system to the use of other systems, not to inhibit a casual and friendly writing style.

15.5.1 Standard Identification

Standard identification shall be in the national language of the documentation.

Standard identification text shall be displayed prominently

a) in a prominent location in the front matter of all publications (normally the title page and cover page);
b) on all identifying display screens of programs; and
c) in all promotional and training material.

For applications, the identification text is:

```
An SGML Application Conforming to
International Standard ISO 8879 --
Standard Generalized Markup Language
```

For systems, the identification text is:

```
An SGML System Conforming to
International Standard ISO 8879 --
Standard Generalized Markup Language
```

The documentation for a conforming SGML system shall include a system declaration (see 15.6).

15.5.2 Identification of SGML Constructs

The documentation shall distinguish SGML constructs from application conventions and system functions, and shall identify the SGML constructs as being part of the Standard Generalized Markup Language.

NOTE — The objective of this requirement is for the user to be aware of which constructs are common to all SGML systems, and which are unique to this one. This will reduce the experienced user's learning time for a new system or application.

This International Standard shall be cited as a reference for supported SGML constructs that are not specifically documented for the system or application. For example, if, for simplicity's sake, only a subset of some function is presented (such as by omitting some of the options of the entity declaration), it shall be stated clearly that other options exist and can be found in this International Standard.

15.5.3 Terminology

All SGML constructs shall be introduced using the terminology of this International Standard, translated to the national language used by the publication or program.

Such standard terminology should be used throughout the documentation. If, notwithstanding, a non-standard equivalent is used for a standard term, it must be introduced in context and it shall not conflict with any standard SGML terms, including terms for unsupported or undocumented constructs.

15.5.4 Variant Concrete Syntax

If a variant concrete syntax is used, that fact shall be made clear to the user. The rules of that syntax shall not be attributed to SGML.

15.6 System Declaration

[200] system declaration = ***mdo***, "SYSTEM", *ps*+,
capacity set, *ps*+, *feature use*, *ps*+,
concrete syntax scope, *ps*+,
concrete syntaxes supported, *ps*+,
validation services, *ps**, ***mdc***

A *system declaration* must meet the same syntax requirements as an *SGML declaration* with respect to the concrete syntax used, data characters allowed, etc.

The *capacity set* parameter is specified as on the *SGML declaration*, except that the capacity of the system is being described, rather than the capacity requirements of a document.

The *feature use* parameter is specified as on the *SGML declaration*, except that the ability of the system to support a feature is being described, rather than the characteristics of a document that uses the feature.

The *concrete syntax scope* parameter is specified as on the *SGML declaration*, except that the ability of the system to support two syntaxes at once is being described, rather than whether a document uses two syntaxes.

NOTE — The *system declaration* should include comments to indicate which data content notations and types of *system data* the system can support.

15.6.1 Concrete Syntaxes Supported

This parameter specifies the concrete syntaxes that the system SGML parser can parse, the translation of their markup characters into the system character set, and any allowed variations.

[201] concrete syntaxes supported = (*ps*+,
concrete syntax, (*ps*+,
concrete syntax changes)?, (*ps*+,
character set translation)?)+

A *concrete syntax* parameter is specified, as on the *SGML declaration*, for each concrete syntax that the system can parse. One of the specified concrete syntaxes must be either the reference concrete syntax, if short references are supported for any concrete syntax, or the core concrete syntax if they are not.

15.6.1.1 Concrete Syntax Changes

This parameter describes concrete syntaxes that the system can parse, that are minor modifications of the specified *concrete syntax*. The keywords define the nature and extent of the permitted changes.

[202] concrete syntax changes = "CHANGES",
ps+, ("SWITCHES" | ("DELIMLEN", *ps*+,
number, *ps*+, "SEQUENCE", *ps*+, ("YES" |
"NO"), *ps*+, "SRCNT", *ps*+, *number*))

where

SWITCHES means that markup characters in the specified concrete syntax can be switched, provided that each replacement substitutes for its original character in every instance in which that character was used.

DELIMLEN means new strings that do not exceed the specified number of characters (1 or more) can be assigned to the delimiter roles.

SEQUENCE indicates whether a blank sequence can be used in short reference delimiters. If so, it is considered to have a length of 1 character.

SRCNT means that different short reference delimiters can be assigned, as long as they do not exceed the specified number (0 or more).

15.6.1.2 Character Set Translation

The *character set translation* parameter defines, in the same form as on the *SGML declaration*, the translation to the system character set from the *translation-reference character set* of the specified *concrete syntax*.

If a number of concrete syntaxes have the same *translation-reference character set*, this parameter must be specified for only one of them, and will apply to all.

15.6.2 Validation Services

The *validation services* parameter specifies whether a system has a validating SGML parser, and which, if any, optional validation services it provides.

```
[203] validation services = "VALIDATE", ps+,
        "GENERAL", ps+, ("NO" | "YES"), ps+,
        "MODEL", ps+, ("NO" | "YES"), ps+,
        "EXCLUDE", ps+, ("NO" | "YES"), ps+,
        "CAPACITY", ps+, ("NO" | "YES"), ps+,
        "NONSGML", ps+, ("NO" | "YES"), ps+,
        "SGML", ps+, ("NO" | "YES"), ps+,
        "FORMAL", ps+, ("NO" | "YES")
```

where:

NO means the service is not provided.

YES means the service is provided.

GENERAL means a reportable markup error will be found and reported.

MODEL means an ambiguous content model will be reported.

EXCLUDE means an exclusion that could change a group's required or optional status in a model will be reported.

CAPACITY means that exceeding a capacity limit will be reported.

NONSGML means the occurrence of at least one non-SGML character, but not necessarily all, will be reported.

SGML means an error in the SGML declaration will be reported.

FORMAL means a formal public identifier error will be reported.

Annex A

Introduction to Generalized Markup

(This annex does not form an integral part of this International Standard.)

A.1 The Markup Process

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

a) Separating the logical elements of the document; and

b) Specifying the processing functions to be performed on those elements.

In publishing systems, where formatting can be quite complex, the markup is usually done directly by the user, who has been specially trained for the task. In word processors, the formatters typically have less function, so the (more limited) markup can be generated without conscious effort by the user. As higher function printers become available at lower cost, however, the office workstation will have to provide more of the functionality of a publishing system, and "unconscious" markup will be possible for only a portion of office word processing.

It is therefore important to consider how the user of a high function system marks up a document. There are three distinct steps, although he may not perceive them as such.

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

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

c) Finally, he inserts the chosen controls into the text.

Here is how the start of this paper looks when marked up with controls in a typical text processing formatting language:

```
.SK 1
Text processing and word processing systems typically
require additional information to be interspersed among
the natural text of the document being processed.
This added information, called "markup," serves two purposes:
.TB 4
.OF 4
.SK 1
1. Separating the logical elements of the document; and
.OF 4
.SK 1
2. Specifying the processing functions to be
performed on those elements.
.OF 0
.SK 1
```

Adapted from Goldfarb, Charles F., "A Generalized Approach to Document Markup", *SIGPLAN Notices*, June 1981, by permission of the author and the Association for Computing Machinery.

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

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

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

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

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

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

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

These postulates will be developed intuitively by examining the properties of this type of markup.

A.2 Descriptive Markup

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

A notation for generalized markup, known as the Standard Generalized Markup Language (SGML), has been developed by a Working Group of the International Organization for Standardization (ISO). Marked up in SGML, the start of this paper might look like this:

```
<p>
Text processing and word processing systems typically
require additional information to be interspersed among
the natural text of the document being processed.
This added information, called <q>markup</q>, serves two purposes:
<ol>
<li>Separating the logical elements of the document; and
<li>Specifying the processing functions to be
performed on those elements.
</ol>
```

Each generic identifier (GI) is delimited by a less-than symbol (<) if it is at the start of an element, or by less-than followed by solidus (</) if it is at the end. A greater-than symbol (>) separates a GI from any text that follows it.[1] The mnemonics P, Q, OL, and LI stand, respectively, for the element types paragraph, quotation, ordered list, and list item. The combination of the GI and its delimiters is called a "start-tag" or an "end-tag", depending upon whether it identifies the start or the end of an element.

This example has some interesting properties:

a) There are no quotation marks in the text; the processing for the quotation element generates them and will distinguish between opening and closing quotation marks if the output device permits.

b) The comma that follows the quotation element is not actually part of it. Here, it was left outside the quotation marks during formatting, but it could just as easily have been brought inside were that style preferred.

c) There are no sequence numbers for the ordered list items; they are generated during formatting.

The source text, in other words, contains only information; characters whose only role is to enhance the presentation are generated during processing.

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

From this analysis, a 3-step model of document processing can be constructed:

a) Recognition: An attribute of the document is recognized, e.g., an element with a generic identifier of "footnote".

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

c) Processing: The chosen processing function is executed.

Text formatting programs conform to this model. They recognize such elements as words and sentences, primarily by interpreting spaces and punctuation as implicit markup. Mapping is usually via a branch table. Processing for words typically involves determining the word's width and testing for an overdrawn line; processing for sentences might cause space to be inserted between them.[2]

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

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

[1] Actually, these characters are just defaults. SGML permits a choice of delimiter characters.

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

So far the discussion has addressed only a single attribute, the generic identifier, whose value characterizes an element's semantic role or purpose. Some descriptive markup schemes refer to markup as "generic coding", because the GI is the only attribute they recognize (5). In generic coding schemes, recognition, mapping, and processing can be accomplished all at once by the simple device of using GIs as control procedure names. Different formats can then be obtained from the same markup by invoking a different set of homonymous procedures. This approach is effective enough that one notable implementation, the SCRIBE system, is able to prohibit procedural markup completely (1).

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

```
<fig id=angelfig>
```

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

The GI and ID attributes are termed "primary" because every element can have them. There are also "secondary" attributes that are possessed only by certain element types. For example, if the user wanted some of the figures in his document to contain illustrations to be produced by an artist and added to the processed output, he could define an element type of "artwork". Because the size of the externally-generated artwork would be important, he might define artwork elements to have a secondary attribute, "depth".[1] This would result in the following start-tag for a piece of artwork 24 picas deep:

```
<artwork depth=24p>
```

The markup for a figure would also have to describe its content. "Content" is, of course, a primary attribute, the one that the secondary attributes of an element describe. The content consists of an arrangement of other elements, each of which in turn may have other elements in its content, and so on until further division is impossible.[2] One way in which SGML differs from generic coding schemes is in the conceptual and notational tools it provides for dealing with this hierarchical structure. These are based on the second generalized markup hypothesis, that markup can be rigorous.

A.3 Rigorous Markup

Assume that the content of the figure "angelfig" consists of two elements, a figure body and a figure caption. The figure body in turn contains an artwork element, while the content of the caption is text characters with no explicit markup. The markup for this figure could look like this:[3]

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

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

[3] Like "GI=", "content=" can safely be omitted. It is unnecessary when the content is externally generated, it is understood when the content consists solely of tagged elements, and for data characters it is implied by the delimiter (>) that ends the start-tag.

```
<fig id=angelfig>
<figbody>
<artwork depth=24p>
</artwork>
</figbody>
<figcap>Three Angels Dancing
</figcap>
</fig>
```

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

This price would be totally unacceptable had the user to enter all the tags himself. He knows that the start of a paragraph, for example, terminates the previous one, so he would be reluctant to go to the trouble and expense of entering an explicit end-tag for every single paragraph just to share his knowledge with the system. He would have equally strong feelings about other element types he might define for himself, if they occurred with any great frequency.

With SGML, however, it is possible to omit much markup by advising the system about the structure and attributes of any type of element the user defines. This is done by creating a "document type definition", using a construct of the language called an "element declaration". While the markup in a document consists of descriptions of individual elements, a document type definition defines the set of all possible valid markup of a type of element.

An element declaration includes a description of the allowable content, normally expressed in a variant of regular expression notation. Suppose, for example, the user extends his definition of "figure" to permit the figure body to contain either artwork or certain kinds of textual elements. The element declaration might look like this:[1]

```
<!--      ELEMENTS    MIN  CONTENT (EXCEPTIONS) -->
<!ELEMENT fig         - -  (figbody, figcap?)>
<!ELEMENT figbody     - O  (artwork | (p | ol | ul)+)>
<!ELEMENT artwork     - O  EMPTY>
<!ELEMENT figcap      - O  (#PCDATA)>
```

The first declaration means that a figure contains a figure body and, optionally, can contain a figure caption following the figure body. (The hyphens will be explained shortly.)

The second says the body can contain either artwork or an intermixed collection of paragraphs, ordered lists, and unordered lists. The "O" in the markup minimization field ("MIN") indicates that the body's end-tag can be omitted when it is unambiguously implied by the start of the following element. The preceding hyphen means that the start-tag *cannot* be omitted.

The declaration for artwork defines it as having an empty content, as the art will be generated externally and pasted in. As there is no content in the document, there is no need for ending markup.

The final declaration defines a figure caption's content as 0 or more characters. A character is a terminal, incapable of further division. The "O" in the "MIN" field indicates the caption's end-tag can be omitted. In addition to the reasons already given, omission is possible when the end-tag is unambiguously implied by the end-tag of an element that contains the caption.

It is assumed that p, ol, and ul have been defined in other element declarations.

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

[1] The question mark (?) means an element is optional, the comma (,) that it follows the preceding element in sequence, the asterisk (*) that the element can occur 0 or more times, and the plus (+) that it must occur 1 or more times. The vertical bar (|) is used to separate alternatives. Parentheses are used for grouping as in mathematics.

```
<fig id=angelfig>
<figbody>
<artwork depth=24p>
<figcap>Three Angels Dancing
</fig>
```

There has been a 40% reduction in markup, since the end-tags for three of the elements are no longer needed.

- As the element declaration defined the figure caption as part of the content of a figure, terminating the figure automatically terminated the caption.

- Since the figure caption itself is on the same level as the figure body, the <figcap> start-tag implicitly terminated the figure body.

- The artwork element was self-terminating, as the element declaration defined its content to be empty.[1]

A document type definition also contains an "attribute definition list declaration" for each element that has attributes. The definitions include the possible values the attribute can have, and the default value if the attribute is optional and is not specified in the document.

Here are the attribute list declarations for "figure" and "artwork":

```
<!--       ELEMENTS     NAME     VALUE     DEFAULT -->
<!ATTLIST  fig          id       ID        #IMPLIED>
<!ATTLIST  artwork      depth    CDATA     #REQUIRED>
```

The declaration for figure indicates that it can have an ID attribute whose value must be a unique identifier name. The attribute is optional and does not have a default value if not specified.

In contrast, the depth attribute of the artwork element is required. Its value can be any character string.

Document type definitions have uses in addition to markup minimization.[2] They can be used to validate the markup in a document before going to the expense of processing it, or to drive prompting dialogues for users unfamiliar with a document type. For example, a document entry application could read the description of a figure element and invoke procedures for each element type. The procedures would issue messages to the terminal prompting the user to enter the figure ID, the depth of the artwork, and the text of the caption. The procedures would also enter the markup itself into the document being created.

The document type definition enables SGML to minimize the user's text entry effort without reliance on a "smart" editing program or word processor. This maximizes the portability of the document because it can be understood and revised by humans using any of the millions of existing "dumb" keyboards. Nonetheless, the type definition and the marked up document together still constitute the rigorously described document that machine processing requires.

A.4 Conclusion

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

It is a textbook on software development containing hundreds of formulas in a symbolic notation devised by the author. Despite the typographic complexity of the material (many lines, for example, had a dozen or more font changes), no procedural markup was needed anywhere in the text of the book. It was marked up using a language that adhered to the principles of generalized markup but was less flexible and complete than the SGML (4).

[1] SGML actually allows the markup to be reduced even further than this.

[2] Some complete, practical document type definitions may be found in (4), although they are not coded in SGML.

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

In due course both searches were successful, and an unusual project was begun. As the author's processor did not support photocomposers directly, procedures were written that created a source file with procedural markup for a separate typographic composition program. Formatting specifications were provided by the publisher, and no concessions were needed to accommodate the use of generalized markup, despite the marked up document having existed before the specifications.[1]

The experiment was completed on time, and the publisher considers it a complete success (7).[2] The procedures, with some modification to the formatting style, have found additional use in the production of a variety of in-house publications.

Generalized markup, then, has both practical and academic benefits. In the publishing environment, it reduces the cost of markup, cuts lead times in book production, and offers maximum flexibility from the text data base. In the office, it permits interchange between different kinds of word processors, with varying functional abilities, and allows auxiliary "documents", such as mail log entries, to be derived automatically from the relevant elements of the principal document, such as a memo.

At the same time, SGML's rigorous descriptive markup makes text more accessible for computer analysis. While procedural markup (or no markup at all) leaves a document as a character string that has no form other than that which can be deduced from analysis of the document's meaning, generalized markup reduces a document to a regular expression in a known grammar. This permits established techniques of computational linguistics and compiler design to be applied to natural language processing and other document processing applications.

A.5 Acknowledgments

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

A.6 Bibliography

1 B. K. Reid, "The Scribe Document Specification Language and its Compiler", *Proceedings of the International Conference on Research and Trends in Document Preparation Systems*, 59-62 (1981).
2 Donald E. Knuth, *TAU EPSILON CHI, a system for technical text*, American Mathematical Society, Providence, 1979.
3 C. F. Goldfarb, E. J. Mosher, and T. I. Peterson, "An Online System for Integrated Text Processing", *Proceedings of the American Society for Information Science*, 7, 147-150 (1970).
4 Charles F. Goldfarb, *Document Composition Facility Generalized Markup Language: Concepts and Design Guide*, Form No. SH20-9188-1, IBM Corporation, White Plains, 1984.
5 Charles Lightfoot, *Generic Textual Element Identification—A Primer*, Graphic Communications Computer Association, Arlington, 1979.
6 C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall International, London, 1980.
7 Ron Decent, *personal communication to the author* (September 7, 1979).

[1] On the contrary, the publisher took advantage of generalized markup by changing some of the specifications after he saw the page proofs.

[2] This despite some geographical complications: the publisher was in London, the book's author in Brussels, and this paper's author in California. Almost all communication was done via an international computer network, and the project was nearly completed before all the participants met for the first time.

Annex B

Basic Concepts

(This annex does not form an integral part of this International Standard.)

This annex describes some of the basic concepts of the Standard Generalized Markup Language (SGML). Before beginning it, the reader should consult annex A to gain an initial familiarity with generic coding and generalized markup.

NOTE — The reader should be aware that this annex does not cover all basic SGML constructs, nor all details of those covered, and subtle distinctions are frequently ignored in the interest of presenting a clear overview.

B.1 Documents, Document Type Definitions, and Procedures

The fundamental concept of generalized markup is the relationship between documents, document type definitions, and procedures.

B.1.1 Documents

In generalized markup, the term “document” does not refer to a physical construct such as a file or a set of printed pages. Instead, a document is a logical construct that contains a *document element*, the top node of a tree of elements that make up the document's *content*. A book, for example, could contain “chapter” elements that in turn contain “paragraph” elements and “picture” elements.

Eventually, the terminal nodes of this document tree are reached and the actual characters or other data are encountered. If paragraphs, for example, were terminal, their content would be characters, rather than other elements. If photographs were terminal they would contain neither elements nor characters, but some noncharacter data that represents an image.

The elements are distinguished from one another by additional information, called *markup*, that is added to the data content. A document thus consists of two kinds of information: data and markup.

B.1.2 Document Type Definitions

An element's markup consists of a *start-tag* at its beginning and an *end-tag* at its end. The tags describe the characteristic qualities of the element.

One of these characteristics is the *generic identifier*, which identifies the “type” of the element (manual, paragraph, figure, list, etc.). In addition, there can be other characteristics, called “attributes”, that further qualify the generic identifier.

An individual document's markup tags describe its structure of elements. That is, they indicate which elements occur in the document's content, and in what order. That structure must conform to rules that define the permitted structures for all documents of a given type; that is, those documents having the same generic identifier.

The rules that define the possible structures are part of a *document type definition* of that type of document. A document type definition specifies:

a) The generic identifiers (GIs) of elements that are permissible in a document of this type.

b) For each GI, the possible attributes, their range of values, and defaults.

c) For each GI, the structure of its content, including
   i) which subelement GIs can occur and in what order;
   ii) whether text characters can occur;
   iii) whether noncharacter data can occur.

A document type definition does *not* specify:

a) The delimiters that are used to indicate markup.

b) Anything about the ways in which the document can be formatted or otherwise processed.

B.1.3 Procedures

Markup tags describe a document's structure of elements; they do not say how to process that structure. Many kinds of processing are possible, one of which is to *format* the text.

Formatting can be thought of as mapping the element structure onto paper or a display screen with graphic arts conventions. For example, the element "paragraph" could be displayed by setting the text of the element off from surrounding text with blank lines. Alternatively, it could be displayed by indenting its first line.

Processing is handled by *procedures*, which are written in the language of a formatter or other processing system. When a document is processed, a procedure is associated with each generic identifier (that is, with each type of element). The procedure then processes the content of the element. In the case of formatting, for example, the procedure performs the actions that render an element into printed text or another display form.

Thus, production of a document begins when a user creates the text, marking it up as a particular document type. One of the facilities that can process that document is a formatter, which could have more than one set of procedures available.

For example, a document called "mybook", marked up as a "TechManual" document type, could be formatted in a number of ways by using a different *procedure set* each time. One set could produce output in single column CRT display style, another set in two column printed report style, and a third set in still another style.

When developing a completely new text application, then, a designer would create document type definitions, using the Standard Generalized Markup Language. Probably, he would also implement one or more procedure sets, using the languages of the systems that are to process the documents.

B.2 Markup

Markup is text that is added to the data of a document in order to convey information about it. In SGML, the markup in a document falls into four categories:

a) Descriptive Markup ("Tags")

Tags are the most frequent and the most important kind of markup. They define the structure of the document, as described above.

b) Entity Reference

Within a system, a single document can be stored in several parts, each in a separate unit of system storage, called an *entity*. (Depending on the system, an entity could be a file, a data set, a variable, a data stream object, a library member, etc.)

Separate entities are connected by *entity references* that can occur in a document's markup. An entity reference is a request for text—the entity—to be imbedded in the document at the point of the reference. The entity could have been defined either earlier within the document or externally.

The entity reference capability includes the functions commonly called *symbol substitution* and *file imbedding*.

c) Markup Declaration

Declarations are statements that control how the markup is interpreted. They can be used to define entities and to create document type definitions.

d) Processing Instructions

These are instructions to the processing system, in its own language, to take some specific action. Unlike the other kinds of markup, processing instructions are system-dependent, and are usually application-dependent as well. They normally need to be changed if the document is processed differently (for example, formatted in a different style), or on a different system.

An SGML system must recognize these four kinds of markup and handle them properly: that is, it must have an "SGML parser". The parser need not be a single dedicated program; as long as a system can perform the parsing process, it can be said to have an SGML parser.

Markup occurs in a document according to a rigid set of rules. Some of the rules are dictated by SGML; they apply to all types of document. Other rules are defined by the document type definition for the type of document being processed.

Employing the rules of SGML, the markup parser must:

a) Scan the text of each element's content to distinguish the four kinds of markup from one another and from the data. (Noncharacter content data is not scanned by the parser.)

b) Replace entity references by their entities.

c) Interpret the markup declarations.

d) Give control to the processing system to execute processing instructions.

e) Interpret the descriptive markup tags to recognize the generic identifiers ("GIs") and attributes, and, following the rules of the document type:
   i) Determine whether each GI and its attributes are valid.
   ii) Track the location in the document structure.

f) Give control to the processing system to execute the procedure associated with the GI. (Once again, there is no actual requirement for separate programs. "Giving control" means only that the ensuing processing is not defined by this International Standard.)

B.3 Distinguishing Markup from Text

The markup discussed in this section applies to all document types. The delimiter characters used are the delimiter set of the *reference concrete syntax*. (They will be discussed as if there were only one concrete syntax, although SGML allows variant concrete syntaxes to be defined.)

B.3.1 Descriptive Markup Tags

Descriptive markup tags identify the start and end of elements. There are three special character strings that are important (see figure 9):

STAGO Start-TAG Open

This is a delimiter that indicates the beginning of a start-tag. In the figure, " < " is the ***stago***.

TAGC TAG Close

The string from the ***stago*** to the ***tagc*** is called the *start-tag*. In it the generic identifier ("GI") and all the attributes are given. In the figure, "quote" was the GI and " > " is used for the ***tagc***.

ETAGO End-TAG Open

This is a two-character delimiter that indicates the beginning of an end-tag. In the figure, " </" is the ***etago***. Between the ***tagc*** of the start-tag and the ***etago*** of the end-tag is the *content* of the element, which can include data characters and subordinate elements. (Noncharacter data is kept separately; it will be discussed later.) The end-tag contains a repetition of the GI to make the markup easier to read.

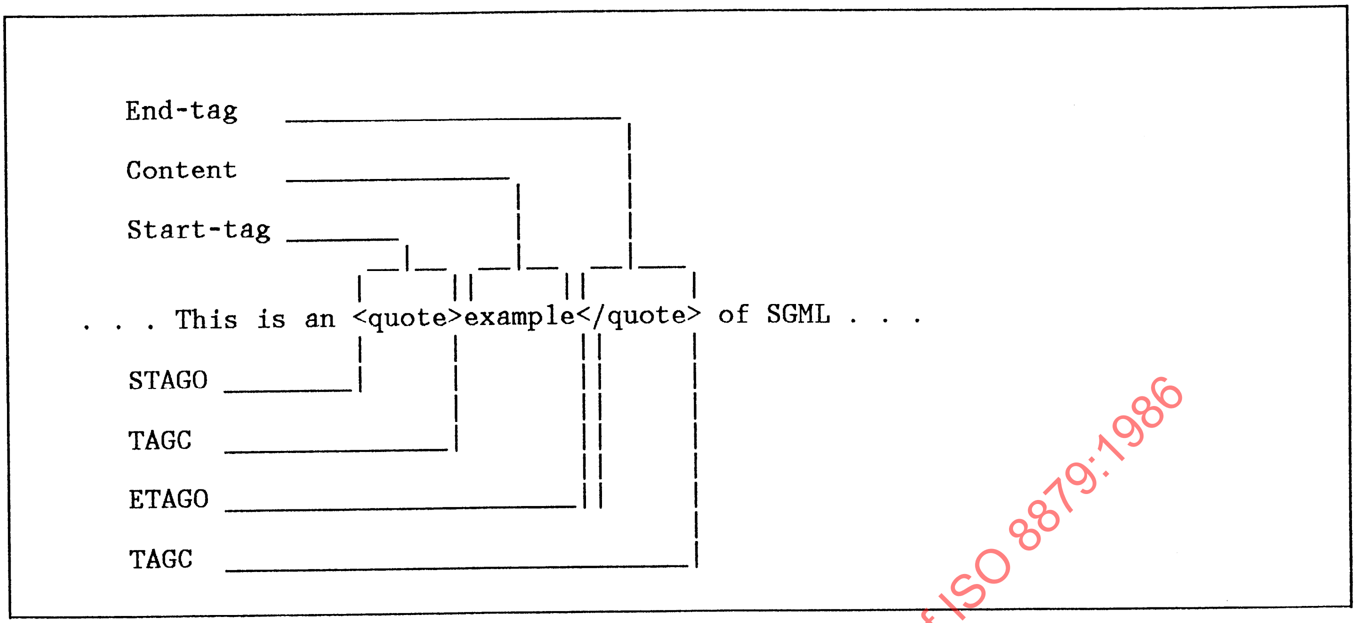

Figure 9 — Element Markup

The scheme just described is the most general way of delimiting an element. SGML, though, allows a number of techniques of *markup minimization* that allow the shortening of tags, and even their omission. These techniques, which are available as optional features, will be described later.

B.3.2 Other Markup

An entity reference begins with an "entity reference open" (***ero***) delimiter and ends with a "reference close" (***refc***). These are the ampersand and semicolon, respectively, in the following example:

```
The &SGML; supports publishing and office systems.
```

A markup declaration is delimited by "markup declaration open" (***mdo***) and "markup declaration close" (***mdc***) delimiters, and a processing instruction by a "processing instruction open" (***pio***) and a "processing instruction close" (***pic***).

```
here is an imbedded <!markup declaration> example
and an imbedded <?processing instruction> example
```

To summarize:

String	**Name**	***Meaning***
&	**ERO**	Opens a named entity reference.
;	**REFC**	Closes a reference.
<!	**MDO**	Opens a markup declaration.
>	**MDC**	Closes a markup declaration.
<?	**PIO**	Opens a processing instruction.
>	**PIC**	Closes a processing instruction.

B.3.3 Record Boundaries

Not every text processing system breaks its storage entities into records. In those that do, the record boundaries are represented by function characters known as the "record start" (***RS***) and "record end" (***RE***). The record boundaries could be used as general delimiter characters in a variant concrete syntax, or as a special form of delimiter that serves as a "short entity reference". If a record boundary is not a delimiter, its treatment depends on where it occurs.

B.3.3.1 Record Boundaries in Data

In attribute values (discussed later) and in the data content of an element, record starts are ignored. The record ends, though, are treated as part of the data because they might be significant to a processor of the document. A formatter, for example, normally interprets a record end as a space.

However, record ends are ignored when they are caused by markup. That is:

— Record ends are ignored at the start or end of the content. For example,

```
<p>
Short paragraph data.
</p>
```

is the same as

```
<p>Short paragraph data.</p>
```

— Record ends are ignored after a record that contains only processing instructions or markup declarations. As a result,

```
<p>
Starting data
<?instruction 1>
<?instruction 2>
ending data.
</p>
```

and

```
<p>Starting data
<?instruction 1><?instruction 2>ending data.</p>
```

are equivalent. In other words, as far as the flow of data characters is concerned, declarations and processing instructions are simply ignored.

B.3.3.2 Record Boundaries in Markup

In tags or declarations, both record starts and record ends are treated as spaces. They serve as internal separators within the markup (as do horizontal tabs, incidentally).

The meaning of record boundaries within processing instructions depends on the processing system.

B.4 Document Structure

SGML tags serve two purposes:

a) They show the structural relationships among the elements of the document.

b) They identify each element's generic identifier (GI) and attributes.

The rules for specifying structure and attributes are defined by SGML for all documents. (Some rules have already been introduced; the others will be discussed later.) The *particular* elements and attributes allowed in a document, though, are defined by that document's type definition.

B.4.1 Document Type Definitions

A generic identifier (GI) identifies an element as a member of a class, or "type". A *document type definition* is a set of markup declarations that apply to all documents of a particular type.

The three most important kinds of declaration that can occur in a document type definition are:

a) An *element declaration*, which defines the GIs that can occur in each element, and in what order.

b) An *attribute definition list declaration*, which defines the attributes that can be specified for an element, and their possible values.

c) An *entity declaration*, which defines the entities that can be referred to in documents of this type. For example, entity references can simplify the keying of frequently used lengthy phrases:

```
<!ENTITY SGML "Standard Generalized Markup Language">
```

To avoid repetitive keying, the document type definition is usually stored as a separate entity. It is then incorporated in each document by means of a *document type declaration* that identifies the document type and serves as a reference to the external entity.

B.4.2 Element Declarations

Elements can occur in a document only according to the rules of the document type definition. For example, unless the definition allows a paragraph inside a figure, it would be an error to put one there. The element declaration is used to define these rules.

B.4.2.1 Content Models

For each element in the document, the application designer specifies two element declaration parameters: the element's GI and a *content model* of its content. The model parameter defines which subelements and character strings can occur in the content.

For example, the declaration for a textbook might look like this:

```
<!ELEMENT   textbook   (front, body, rear) >
```

Here, "textbook" is the GI whose content is being defined, and "(front, body, rear)" is the model that defines it. The example says that a textbook contains the GIs "front", "body", and "rear". (The GIs probably stand for "front matter", "body matter", and "rear matter", but this is of interest only to humans, not to the SGML parser.)

A model is a kind of *group*, which is a collection of connected members, called *tokens*. A group is bounded by parentheses, the "group open" (***grpo***) and "group close" (***grpc***) delimiters. The parentheses are required even when the model group contains but a single token.

The tokens in a model group are GIs. There are also delimiters called *connectors* that put the GIs in order, and other delimiters, called *occurrence indicators*, that show how many times each GI can occur.

B.4.2.2 Connectors and Occurrence Indicators

A connector is used *between* GIs, to show how they are connected. The textbook model uses the "sequence" connector (***seq***), the comma. It means that the elements must follow one another in the document in the same sequence as their GIs occur in the model.

An occurrence indicator is used *after* the GI to which it applies. There are none in the textbook example: each element must therefore occur once and only once. To make the front matter and rear matter optional, the question mark, which is the optional occurrence indicator (***opt***), would be used.

```
<!ELEMENT   textbook   (front?, body, rear?) >
```

So far only the top level of a textbook has been defined. The type definition must also have structure declarations for "front", "body", and "rear", then for the elements contained in "front", "body", and "rear", and so on down to the data characters.

For the purposes of a simple example, the body can be assumed to be just a sequence of many paragraphs. That is, the only GI allowed in the body is "p" (for paragraph), but it can occur many times. To indicate multiple paragraphs, the GI is followed by a plus sign, which is the "required and repeatable" occurrence indicator (***plus***).

```
<!ELEMENT body (p+) >
```

The plus sign indicates that there must be at least one paragraph in the body. If, for some reason, it were desirable to allow a body to have no paragraphs at all, the added symbol would be an asterisk, which is the "optional and repeatable" occurrence indicator (***rep***).

```
<!ELEMENT body (p*) >
```

Suppose a textbook could have examples in the body, as well as paragraphs. If the GI for an example were "xmp", "many paragraphs or examples" would be indicated like this:

```
<!ELEMENT body (p | xmp)+ >
```

The vertical bar is the "or" connector (***or***). The expression "p | xmp" means "either a paragraph or an example".

Although a model group contains tokens, it is itself a single token to which an occurrence indicator can apply. The grouping is needed in the example because the occurrence indicators ("?", "+", and "*") have higher precedence than the connectors ("," and "|").

Therefore, it would not be effective to say

```
<!ELEMENT body (p+ | xmp+) >
```

because it would mean "either many paragraphs or many examples" rather than the desired "many intermixed paragraphs or examples".

There is one more kind of connector to consider. Suppose the front matter of the textbook had a title page that contained a title, an author's name, and a publisher, but in *any* order. The ***seq*** connector cannot be used because that requires a specific order. Nor can the ***or*** connector, because that chooses only one of the three elements.

Instead, the ampersand ("&"), which is the "and" connector (***and***) should be used. It indicates that all of the GIs in the model group must occur, but in any order.

```
<!ELEMENT titlepage (title & author & publisher) >
```

B.4.2.3 Entity References in Models

Suppose two elements have almost the same content model, but not quite.

```
<!ELEMENT body (p | xmp | h1 | h2 | h3 | h4)+ >
<!ELEMENT rear (p |       h1 | h2 | h3 | h4)+ >
```

Here, the body and rear matter both have paragraphs and headings, but only the body can have examples.

Repetitive parts of markup declaration parameters can be handled with entity references, just like repetitive text in a document. The only difference is that entity references used in declaration parameters ("parameter entity references") begin with a special character, the percent sign ("%"), called the "parameter entity reference open" delimiter (***pero***). The ***pero*** must also be used in the entity declaration to show that the entity being defined is a parameter entity (but as a separate parameter so it won't be misinterpreted as a reference).

```
<!ENTITY % h1to4  "h1 | h2 | h3 | h4"  >
<!ELEMENT  body   (p | xmp | %h1to4;)+ >
<!ELEMENT  rear   (p |       %h1to4;)+ >
```

B.4.2.4 Name Groups

There is another use for groups. The body matter and rear matter might have the same structure declaration.

```
<!ELEMENT body (p | xmp)+ >
<!ELEMENT rear (p | xmp)+ >
```

Keying effort could be saved and the similarity emphasized by letting the declaration apply to a group of elements.

```
<!ELEMENT (body | rear) (p | xmp)+ >
```

B.4.2.5 Data Characters

Up to now, all of the elements discussed contained only other elements. Eventually, though, an application must deal with the actual data, that is, the point where there are no more tags.

Data can be referred to in a model with the reserved name "#PCDATA", which means "zero or more parsed data characters". As it has, in effect, a built-in ***rep*** occurrence indicator, none can be added to it explicitly. The declaration

```
<!ELEMENT p (#PCDATA) >
```

says that a paragraph is a string of zero or more characters (including record ends and spaces).

Incidentally, the reason "#PCDATA" is in upper-case is simply as a reminder that it is defined by the markup parser. In the reference concrete syntax, the markup parser ignores the case of all names, except those of entities.

In most documents, elements that contain data characters can also contain other tagged elements. Such elements normally include short quotations, footnote and figure references, various types of highlighted phrases, and specialized references or citations defined for a particular document type.

For example, the structure of a paragraph might be defined as:

```
<!ENTITY % phrase    "quote | citation | reference" >
<!ELEMENT   p        (#PCDATA | %phrase;)*          >
```

These declarations indicate that a paragraph contains characters mixed with "phrase" elements. There can be none or many of them.

The "#", incidentally, is a delimiter called the "reserved name indicator" (***rni***) that is used whenever a reserved name is specified in a context where a user-created name could also occur. The ***rni*** makes it clear that "#PCDATA" is not a GI.

B.4.2.6 Empty Content

An element can be defined for which the user never enters the content: for example, a figure reference for which a procedure will always generate the content during processing. To show an empty element, a keyword is used to declare the content instead of the usual parenthesized model group:

```
<!ELEMENT figref EMPTY >
```

An element that is declared to be empty cannot have an end-tag. (None is needed, because there is no content to be ended by it.)

B.4.2.7 Non-SGML Data

If the body of the textbook contained photographs, the element declaration could be

```
<!ELEMENT  body  (p | photo)+ >
```

A photo is typically represented by a string of bits that stand for the colors of the various points in the picture. These bit combinations do not have the same meaning as the characters in text and in markup; they have to be interpreted as a unique notation that is not defined by SGML.

As the markup parser does not scan non-SGML data, it must be stored in a separate entity whose name is given by a special attribute (discussed later). The element content will be empty.

```
<!ELEMENT photo EMPTY >
```

B.4.2.8 Summary of Model Delimiters

The following list summarizes the delimiters used in models and their character strings in the reference delimiter set:

— Grouping:

String	Name	*Meaning*
(	GRPO	Opens a group. The expression within the group is treated as a unit for other operations.
)	GRPC	Closes a group.

— Occurrence Indicators:

String	Name	*Meaning*
?	OPT	Optional: can occur 0 or 1 time.
+	PLUS	Required and repeatable: must occur 1 or more times.
*	REP	Optional and repeatable: can occur 0 or more times.

— Connectors:

String	Name	*Meaning*
,	SEQ	All of the connected elements must occur in the document in the same sequence as in the model group.
|	OR	One and only one of the connected elements must occur.
&	AND	All of the connected elements must occur in the document, but in any order.

— Other:

String	Name	*Meaning*
#	RNI	Identifies a reserved name to distinguish it from a user-specified name.

B.5 Attributes

A descriptive tag normally includes the element's generic identifier (GI) and may include attributes as well. The GI is normally a noun; the attributes are nouns or adjectives that describe significant characteristics of the GI. (The use of verbs or formatting parameters, which are procedural rather than descriptive, is strongly discouraged because it defeats the purpose of generalized markup.)

The *particular* attributes allowed for a given element are defined by the type definition, which also determines the range of values that an attribute can have and what the default values are.

B.5.1 Specifying Attributes

A sample tag with two attributes is shown in figure 10.

The attributes follow the GI in the start-tag. Each attribute is specified by giving its *name*, the value indicator delimiter (*vi*), and the attribute *value*. In the example, the attribute named "security" was given the value "Internal Use", and the attribute "sender" was given the value "LTG".

B.5.1.1 Names

Attribute names, such as "security" and "sender" in the example, share certain properties with other kinds of markup language names already encountered, such as entity names and GIs. A name must consist only of characters that have been designated as "name characters", and it must start with one of a subset of the name characters called the "name start" characters.

Normally (that is, in the reference concrete syntax), the name characters are the letters, digits, period, and hyphen, and the name start characters are the letters only. Lower-case letters in a name are normally treated as though they were upper-case, so it makes no difference in which case a name is keyed. In entity names, though, the case is significant.

A name normally has a maximum length of eight characters, but a system could define a variant *quantity set* in which name length and other quantitative characteristics of the language could differ from the reference concrete syntax.

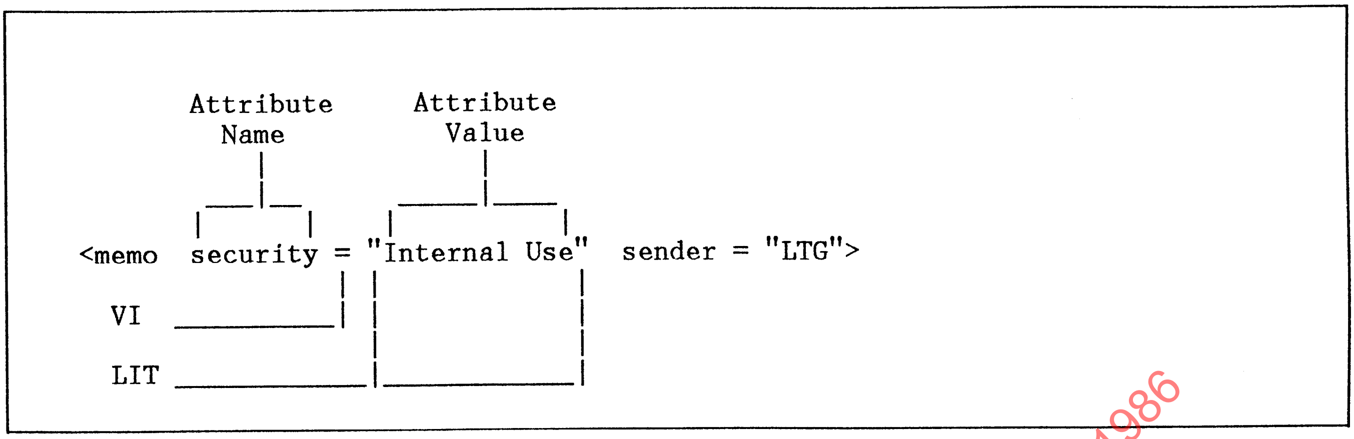

Figure 10 — Start-tag with 2 Attributes

B.5.1.2 Attribute Values

The value of an attribute consists of data characters and entity references, bounded by delimiters called "literal delimiters" (***lit***), which are normally quotation marks ("). Alternative literal delimiters (***lita***), normally apostrophes ('), can also be used, but the two types cannot be paired with one another.

An empty attribute value is indicated by two ***lit*** delimiters in succession:

```
<listing name=Jones phone="555-1234" altphone="">
```

Record ends and tabs in an attribute value are replaced by spaces. Record start characters are ignored (that is, removed).

The following list summarizes the delimiter roles and their string assignments in the reference delimiter set:

String	**Name**	***Meaning***
=	**VI**	Value Indicator
"	**LIT**	Literal Delimiter
'	**LITA**	Literal Delimiter (Alternative)

B.5.2 Declaring Attributes

For every element, the document type definition must contain information that establishes the element's attributes. This is done with an attribute definition list declaration. If there is no attribute definition list for an element, then the element simply has no attributes.

B.5.2.1 Attribute Definition Syntax

The attribute definition list declaration begins with the *associated element type* to which the attribute definition list applies. The declaration can specify a single GI or a group of them, in which case the same attributes apply to all members of the group.

The declaration also includes, for each attribute, an attribute definition consisting of

a) the attribute name,
b) its allowable values (*declared value*), and
c) a default value that will be used if the attribute is not specified in the document and omitted tag markup minimization is in effect (described later).

The attribute list has more parameters than the declarations that have been shown up to now. When dealing with lengthy declarations, it is helpful to add explanatory comments to them. In a markup declaration, comments can occur between any parameters, or a declaration could consist solely of comments.

Comments begin and end with two hyphens, the "comment delimiter" (***com***). One use for comments is to put a heading over the parameters:

```
<!--      ELEMENTS NAME     VALUE         DEFAULT -->
```

The attribute definition list requires that some of the parameters be repeated for each attribute of the element. As declarations can span record boundaries, a tabular form of entry can be used if desired. The following declarations for the memo element in figure 10 are in tabular form:

```
<!--      ELEMENTS          CONTENT                 -->
<!ELEMENT memo              (from, to, subject, body, sig, cc?)>
<!--      ELEMENTS NAME     VALUE         DEFAULT -->
<!ATTLIST memo     status   (final|draft)"final"
                   security CDATA         #REQUIRED
                   version  NUMBER        "01"
                   sender   NAME          #IMPLIED
>
```

The meaning of the attribute definition list is as follows:

status The "status" attribute must have a value of "final" or "draft". As the attribute was not specified in figure 10, the parser will act as though the default value of "final" was entered.

security The "security" attribute has a string of zero or more characters as its value. The keyword "REQUIRED" in the default parameter indicates that the attribute must always be specified in the document.

version The value of the "version" attribute must be a string of 1 or more digits. The default value is the string "01".

sender The "sender" attribute must have a syntactically valid SGML name as a value. The keyword "IMPLIED" indicates that the attribute is optional, and that the value will be supplied by the application if the attribute is not specified in the document. (In figure 10 it was specified as "LTG".)

Note that delimiters can be omitted from the default value when it is composed entirely of name characters. (The default value parameter keywords begin with the ***rni***, so they cannot be confused with application-defined names.) In the above example, the default values "final" and "01" could have been entered without delimiters.

Since the topic of comments was introduced, it should also be noted that an empty markup declaration ("<!>") is also a comment. It can be used to separate parts of the source document with no danger of being misinterpreted as a use of the "short reference" feature (discussed later), as a blank record might be.

B.5.2.2 Complex Attribute Values

An attribute value might logically consist of a number of elements. For example, an illustration of output displayed by a computer could have two colors associated with it: green and black. As an attribute can be specified only once in a tag,

```
<display color=black color=green>
```

could not be entered, as it would be an error.

However, by declaring the value "NAMES" for the "color" attribute, both colors can be specified in a single attribute value:

```
<!ELEMENT display (p+)>
<!ATTLIST display color NAMES "white black">
<!>
<display color="black green">
```

The "NAMES" keyword means "a list of one or more SGML names, separated by spaces, record ends, record starts, or horizontal tab characters". Some other declared value keywords that allow lists are:

NUMBERS a list of one or more numbers.

NMTOKENS a list of one or more name tokens (like names, but they need not start with a name start character: for example, "-abc 123 12.3 123a .123").

NUTOKENS a list of one or more number tokens (like name tokens, but the first character must be a digit: for example, "123 12.3 123a 0.123", but not ".123 456").

The singular forms "NMTOKEN" and "NUTOKEN" can be used when only a single token is permitted. (Note that an ***rni*** is not needed for the declared value parameter keywords, as a user-defined name cannot be specified for that parameter.)

A complex attribute could be avoided altogether in this case by defining two separate attributes:

```
<!ELEMENT display (p+)>
<!ATTLIST display bgcolor NAME "white" fgcolor NAME "black">
<!>
<display bgcolor=black fgcolor=green>
```

B.5.2.3 Name Token Groups

An attribute value can be restricted to a member of a group of unique names or name tokens, called a *name token group*:

```
<!--      ELEMENTS          CONTENT                  -->
<!ELEMENT memo              (from, to, subject, body, sig, cc?)>
<!--      ELEMENTS NAME     VALUE        DEFAULT -->
<!ATTLIST memo     status   (final|draft)"final"
>
```

Given the above declaration, either

```
<memo status="draft">
```

or

```
<memo status="final">
```

can be specified. Any other value for the "status" attribute would be incorrect.

A name token can appear only once in the attribute definition list that applies to a single element type. It is translated to upper-case in the same way as a name.

B.5.2.4 Changing Default Values

If the default value is specified as "CURRENT", the default will automatically become the most recently specified value. This allows an attribute value to be "inherited" by default from the previous element of the same type. (This effect should be kept in mind when adding or deleting tags that specify a current attribute.)

B.6 Entities

Entity references and parameter entity references have figured prominently in some of the examples that have already appeared. Although an entity has a superficial resemblance to a programming language variable, it is actually a portion of the document, and as such, it is a constant. References permit a number of useful techniques:

a) A short name can be used to refer to a lengthy or text string, or to one that cannot be entered conveniently with the available keyboard.

b) Parts of the document that are stored in separate system files can be imbedded.

c) Documents can be exchanged among different systems more easily because references to system-specific objects (such as characters that cannot be keyed directly) can be in the form of entity references that are resolved by the receiving system.

d) The result of a dynamically executed processing instruction (such as an instruction to retrieve the current date) can be imbedded as part of the document.

B.6.1 Entity Reference Syntax

There are two kinds of named entity reference. A general entity reference can be used anywhere in the content of elements and delimited attribute values. Parameter entity references can be used within markup declaration parameters that are delimited with ***lit*** or ***lita*** delimiters. They can also be used to refer to consecutive complete parameters or group tokens, with their intervening separators.

A general entity reference is a name delimited by an "entity reference open" (***ero***), normally an ampersand, and a "reference close" (***refc***), normally a semicolon:

```
  printed at &site; on
```

If an entity reference is followed by a space or record end, the ***refc*** can be omitted:

```
  printed at &site on
```

A parameter entity reference is the same, except that it begins with a "parameter entity reference open" (***pero***), normally a percent sign. (Incidentally, "normally", in the context of delimiter strings, means "in the reference delimiter set".)

The distinction between general and parameter entities is made so that the document preparer can make up entity names without having to know whether the same names were used by the support personnel who created the markup declarations for the document type.

The following list summarizes the delimiters used in entity references and their character strings in the reference concrete syntax:

String	**Name**	***Meaning***
&	**ERO**	Entity Reference Open
%	**PERO**	Parameter Entity Reference Open
;	**REFC**	Reference Close

B.6.2 Declaring Entities

Before an entity can be referred to it must be declared with an entity declaration. There are two main parameters: the *entity name* and the *entity text*. The declaration

```
  <!ENTITY uta "United Typothetae of America">
```

means that a reference to the name "uta" (that is, "&uta;") in the document will be the equivalent of entering the text "United Typothetae of America". The entity text is delimited by ***lit*** (or ***lita***) delimiters (like an attribute value), and is known as a "parameter literal".

An entity does not inherently begin with a record start or end with a record end. If record boundaries are wanted around an entity, they should be put around the entity reference. The source

```
<p>The &uta; is a printing organization.</p>
```

resolves to

```
<p>The United Typothetae of America is a printing organization.</p>
```

while the source

```
<p>Printing organizations:
&uta;
Society of Scientific, Technical, and Medical Publishers
</p>
```

resolves to

```
<p>Printing organizations:
United Typothetae of America
Society of Scientific, Technical, and Medical Publishers
</p>
```

However, if a record end, rather than a ***refc***, is used to terminate an entity reference, the following record is concatenated to the entity.

```
&uta
, Inc.
```

resolves to

```
United Typothetae of America, Inc.
```

B.6.2.1 Processing Instructions

A processing instruction can be stored as an entity. It will be ignored when the entity is created but executed when a reference to the entity occurs.

```
<!ENTITY page PI "newpage; space 3" >
```

The keyword “PI” indicates that the entity will be interpreted as a processing instruction when referenced.

B.6.2.2 Entities with Entity References

A parameter entity is declared by specifying a ***pero*** as the first parameter, ahead of the entity name:

```
<!ENTITY % bullet "o" >
```

Parameter entities can be referenced within a parameter literal:

```
<!ENTITY prefix "%bullet;   " >
```

The reference to “%bullet;” is resolved when the “prefix” entity is declared. It is not resolved on each reference to the “prefix” entity.

B.6.2.3 External Entities

In many text processing systems, there are multiple classes of storage, such as files, library members, macro definitions, and symbols for text strings. Such system dependencies can be kept out of the body of the document by referencing external storage objects as entities:

```
<!ENTITY part2 SYSTEM>
```

If the entity name is not sufficient to enable the system to identify the storage object, additional information (called the "system identifier") can be specified:

```
<!ENTITY part2 SYSTEM "user.sectionX3.textfile" >
```

The system identifier is delimited in the same manner as a parameter literal. The nature and syntax of the system identifier depends on a component of an SGML system called the *entity manager*, whose job it is to convert entity references into real system addresses.

B.6.2.4 Public Entities

An external entity that is known beyond the context of an individual document or system environment is called a "public entity". It is given a "public identifier" by an international, national, or industry standard, or simply by a community of users who wish to share it.

One application of public entities would be shared document type definitions. Another would be shared "entity sets" of entity declarations that support the graphic symbols and terminology of specialized subject areas, such as mathematics or chemistry.

Public entities are declared in a manner similar to other external entities, except that a "public identifier specification" replaces the keyword "SYSTEM":

```
<!ENTITY % ISOgrk1
         PUBLIC "ISO 8879-1986//ENTITIES Greek Letters//EN">
```

The specification consists of the keyword "PUBLIC", the public identifier, which is delimited like a literal, and an optional system identifier (omitted in the example). The public identifier can contain only letters, digits, space, record ends and starts, and a few special characters; they are collectively known as the "minimum data" characters.

B.7 Characters

Each character in a document occupies a position in the document's *character set*. The total number of positions depends on the size of the *code set*; that is, on the number of binary digits ("bits") used to represent each character.

For example, the character set known as ISO 646 International Reference Version (ISO 646 IRV) is a 7-bit set. There are 128 *bit combinations* possible with 7 bits, ranging from the values 0 through 127 in the decimal number base. In 8-bit sets, 256 bit combinations are possible. The position number, or *character number*, is the base-10 integer equivalent of the bit combination that represents the character.

It is also possible to employ "code extension techniques", in which a bit combination can represent more than one character. Use of such techniques with SGML is discussed in clause E.3.

B.7.1 Character Classification

Many character sets have been defined, to accommodate a variety of national alphabets, scientific notations, keyboards, display devices, and processing systems. In each, the sequence of bit combinations is mapped to a different repertoire of digits, letters, and other characters. Any character set large enough to represent the markup characters (name characters, delimiters, and function characters) and the minimum data characters, can be used in an SGML document.

SGML classifies the characters as follows:

function characters The record end and record start, which have already been explained, are function characters, as is the space. The reference concrete syntax adds the horizontal tab (***TAB***), which is used to separate tokens in markup declarations and tags (along with the space, ***RS***, and ***RE*** characters). Function characters can also serve as data characters, and can occur in short reference delimiters (explained later).

name characters These are characters that can be used in a name. They always include the upper-case and lower-case letters A through Z and the digits 0 through 9; the reference concrete syntax adds the period and hyphen. Each name character (other than the digits and upper-case letters) has an associated upper-case form that is used for case substitution in names (except for entity names, in the reference concrete syntax). A subset of the name characters, called the *name start characters*, consists of the lower-case and upper-case letters, plus any other characters that a concrete syntax allows to start a name.

delimiter set These are characters that, in varying contexts, will cause the text to be construed as markup, rather than as data.

non-SGML characters These are characters that a document character set identifies as not occurring in SGML entities, chosen in part from candidates specified by the concrete syntax. The reference concrete syntax, for example, classifies the control characters in this way (except for the few that are used as function characters). Non-SGML characters occur in non-SGML data (such as images) that are in external entities, and in the "envelope" of the file systems, data streams, etc., that contain or transmit the document. A system can also use them for its own purposes, such as padding or delimiting during processing, as there is no possibility of confusing them with markup or data characters.

data characters All other characters, such as punctuation and mathematical symbols that are not used as delimiters, are data characters. (Markup characters can also be data, when they occur in a context in which they are not recognized as markup. Such data is called "parsed character data".)

B.7.2 Character References

It is rarely convenient, or even possible, to enter every character directly:

a) There may be no key for it on the entry device.
b) It may not be displayable.
c) It may be a non-SGML character that cannot occur directly as a markup or data character.
d) It may be a function character that you want treated as data, rather than given its SGML function.

For such situations, a technique called a "character reference" is available to enter the character indirectly. (For visibility, the following illustrations will use a character that could have been keyed, the hyphen.) The two declarations in the following example are equivalent in any character set in which the hyphen is character number 45:

```
<!ENTITY hyphen "-" >
<!ENTITY hyphen "&#45;" >
```

In the entity declarations illustrated so far, the literals contained only *character data* as the entity text. In the above example, the second declaration contains a character reference. The character reference begins with a "character reference open" delimiter (**&#**) and ends with a ***refc*** delimiter, but instead of a name it contains a character number.

A literal can have a mix of character data and character references. The following declarations are also equivalent:

```
<!ENTITY finis "-|-" >
<!ENTITY finis "&#45;|&#45;" >
```

Character references can also be entered directly in attribute values and data content, just as entity references can.

The function characters can also be referenced by named character references: &#RS;, &#RE;, &#SPACE;, and &#TAB;. This form of reference is used when the character's function is wanted; the character number form is used to avoid the function and enter the character as data.

The following list summarizes the delimiters used in character references and their character strings in the reference concrete syntax:

String	**Name**	***Meaning***
&#	**CRO**	Character Reference Open
;	**REFC**	Reference Close

B.7.3 Using Delimiter Characters as Data

The Standard Generalized Markup Language does not compel the use of particular characters as delimiters. Instead, it defines delimiter *roles* as part of an "abstract syntax" and allows character strings to be assigned to them as part of a concrete syntax definition. Although there are many such roles, the user's ability to enter data characters freely is essentially unimpeded because:

— Most of the delimiters occur only within markup declarations or tags; only a few occur in the content of elements.

— The same characters are used for more than one role.

— The delimiter roles that are meaningful in content are contextual; they will only be recognized when followed by an appropriate contextual sequence (or are otherwise enabled). For example, the ***ero*** is only recognized when it is followed by a name start character.

— Most delimiters are multicharacter strings, which reduces the chance of their occurring as data.

— A multicharacter delimiter or a "delimiter-in-context" will not be recognized if an entity starts or ends within it.

Because of the last point, ambiguity can always be avoided by using a reference to enter the data character. Only two entities are needed for most situations in the reference delimiter set, and three more will handle the special cases:

```
<!ENTITY amp    "&" >
<!ENTITY lt     "<" >
```

The entity references "&" and "&lt;" can be used freely in text whenever the ampersand or less-than sign are wanted, as the use of the reference terminates a delimiter-in-context.

```
<!ENTITY rsqb  "]" >
```

The right square bracket entity can be used similarly to avoid recognition of a marked section end (discussed later).

```
<!ENTITY sol CDATA "/" >
```

The solidus is a valid delimiter only when the SHORTTAG feature is used (explained later), and even then it must specifically be enabled for an element. As the delimiter is a single character and requires no contextual sequence, the entity reference alone is not enough to assure that it will be treated as data. The "CDATA" keyword is therefore specified; it causes an entity's text to be treated as character data, even though it may look like a delimiter.

```
<!ENTITY quot  '"' >
```

The "&quot;" (quotation mark) entity is needed only for the rare instance in which both the ***lit*** and ***lita*** delimiter characters occur as data in the same literal. Normally, if a literal contained the ***lit*** character, it would be delimited with ***lita*** delimiters, and vice versa.

B.8 Marked Sections

A *marked section* is a section of a document that is entered as a parameter of a *marked section declaration* in order to label it for a special purpose, such as disabling delimiters within it, or ignoring the section during certain processing runs.

B.8.1 Ignoring a Marked Section

If a document is processed on two different systems, a processing instruction that applies to one will not be understood by the other. The markup parser can be made to ignore one of the instructions if the section of the document that contains it is marked as a section to be ignored:

```
<![ IGNORE [<?instruction for System A>]]>
```

The marked section declaration in the example consists of five parts:

a) the *marked section start*, which consists of the ***mdo*** delimiter followed by the "declaration subset open" delimiter (***dso***), normally a left square bracket;

```
<![
```

b) the *status keywords*, in this case, the single keyword "IGNORE";

```
IGNORE
```

c) another "declaration subset open" delimiter (***dso***) to indicate the start of the marked section content;

d) the content;

```
<?instruction for System A>
```

e) and the *marked section end*, consisting of the "marked section close" delimiter (***msc***), normally two right square brackets (to balance the two ***dso*** delimiters) followed by the ***mdc***.

```
]]>
```

The processing instruction for the other system would be marked as one to be included:

```
<![ INCLUDE [
<?instruction for System B>
]]>
```

Sending the document to system A requires exchanging the status keywords for the two sections. This can be done most easily by using two parameter entities, one for each processing system. One entity would contain the character string "IGNORE" and the other would contain "INCLUDE":

```
<!ENTITY % systema "IGNORE"  >
<!ENTITY % systemb "INCLUDE" >
```

The "IGNORE" keyword would not be used directly in any marked section declaration. Instead, there would be a reference to one of the two system-dependent entities. Given the previous declarations, the instruction for "System A" in the following example will be ignored, while the one for "System B" will be executed:

```
<![%systema;[<?instruction for System A>]]>
<![%systemb;[<?instruction for System B>]]>
```

Every other marked section in the document that refers to "%systema;" or "%systemb;" will be affected similarly.

Now, if the two entity declarations are reversed, as follows,

```
<!ENTITY % systema "INCLUDE" >
<!ENTITY % systemb "IGNORE"  >
```

every marked section in the document that refers to "%systemb;" will be ignored.

Note that even though the section content is ignored, enough parsing is done to recognize nested marked section starts and ends, so the correct end will be used for the section.

B.8.2 Versions of a Single Document

A document could be published in multiple versions whose text differs slightly. SGML allows all versions of such a document to be processed without duplicating the text of the common portions. The version-dependent text is entered in marked sections, using the technique just described, while the common text is not:

```
<![ %v1; [text for version 1]]>
<![ %v2; [text for 2nd version]]>
common text for both versions
<![ %v1; [more text for version 1]]>
<![ %v2; [more text for 2nd version]]>
```

Now, if the following entity declarations are defined:

```
<!ENTITY % v1 "INCLUDE" >
<!ENTITY % v2 "IGNORE"  >
```

version 1 will be processed, as follows:

```
text for version 1
common text for both versions
more text for version 1
```

If the entity declarations are reversed:

```
<!ENTITY % v1 "IGNORE"  >
<!ENTITY % v2 "INCLUDE" >
```

version 2 will be processed, as follows:

```
text for 2nd version
common text for both versions
more text for 2nd version
```

B.8.3 Unparsable Sections

A marked section can be labeled as one that is not to be parsed:

```
<![ CDATA [
<?instruction>
<p>A paragraph with an &entityx;
reference that is not recognized.</p>
<?instruction>
]]>
```

The content of the section is treated as character data, so the two processing instructions and the entity reference will not be recognized or processed as such.

If it is necessary to resolve entity references and character references while ignoring other delimiters, the content can be treated as *replaceable character data* like this:

```
<![ RCDATA [
<?instruction>
<p>A paragraph with an &entityx;
reference that is recognized.</p>
<?instruction>
]]>
```

CDATA and RCDATA marked sections are not nestable, as are IGNORE marked sections. The first marked section end will terminate them.

B.8.4 Temporary Sections

A marked section can be flagged as temporary for easy identification and removal:

```
<![TEMP[<?newpage>]]>
```

Such sections are useful for temporary "fixes" when a processor does not get things quite right.

B.8.5 Keyword Specification

To facilitate keyword entry with entity references, the four status keywords and "TEMP" can be used concurrently, and duplicates are permitted. This allows multiple entity references to be used in a single marked section declaration with no possibility of an error being caused by their resolving to conflicting or duplicate keywords.

The status keywords are prioritized as follows:

IGNORE
CDATA
RCDATA
INCLUDE

If no status keyword is specified, "INCLUDE" is assumed.

B.8.6 Defining a Marked Section as an Entity

A marked section can be defined as an entity so it can be incorporated at many points in the document:

```
<!ENTITY phrase1 MS
"RCDATA[
a repeated phrase with
a <tag> example
" >
```

Given this declaration, the input

```
This is &phrase1; in it.
```

will resolve to

```
This is <![RCDATA[
a repeated phrase with
a <tag> example
]]> in it.
```

The "<tag>" was not recognized as markup because of the "RCDATA" keyword on the marked section declaration.

Note that there was no record start before the marked section nor a record end after it. The following example would have produced them:

```
This is
&phrase1;
in it.
```

A marked section is not parsed or processed as such when the entity is defined. The marked section start and marked section end are added automatically.

B.9 Unique Identifier Attributes

A *unique identifier* ("ID") is an attribute that names an element to distinguish it from all other elements. An SGML markup parser can perform common processing for ID attributes, thereby minimizing the implementation effort for procedures.

The purpose of IDs is to allow one element to refer to another: for example, to allow a figure reference to refer to a figure. Normally, the procedure for the figure would associate some data with the ID (such as the figure number). The figure reference procedure would retrieve the data and print it.

Although the markup parser is normally unaware of the meaning of particular attributes, it can be told when an attribute is a unique identifier:

```
<!--      ELEMENT           CONTENT             -->
<!ELEMENT figure            (figbody, figcap)>
<!--      ELEMENT  NAME     VALUE       DEFAULT -->
<!ATTLIST figure   id       ID          #IMPLIED>
```

Only one ID attribute is allowed for an element.

The value of a unique identifier attribute must be a name that is different from the value of any other ID attribute in the document. As with most names, the upper-case form is used, regardless of how it was entered.

The element that does the referencing must have a "unique identifier reference" attribute, indicated in the declared value by the "IDREF" keyword:

```
<!--      ELEMENT           CONTENT             -->
<!ELEMENT figref            EMPTY>
<!--      ELEMENT  NAME     VALUE       DEFAULT -->
<!ATTLIST figref   refid    IDREF       #IMPLIED>
```

The value of an ID reference attribute must be a name that is the same as that of some element's ID attribute. As usual, the upper-case form of the name is employed.

B.10 Content Reference Attributes

Some documents are formatted and printed as separate chapters. Separate formatting prevents automatic figure references across chapter boundaries, although they would still be possible within a chapter. An element definition could support the two types of figure reference by specifying that the content could sometimes be empty (the intra-chapter case), and that at other times it will be entered explicitly (the cross-chapter case).

The condition for emptiness is the specification of an attribute that is designated in the attribute definition as a "content reference" attribute. (Everything else about the attribute behaves in its usual manner.) The designation is made by entering the keyword "#CONREF" as the default value:

```
<!--      ELEMENTS        CONTENT                    -->
<!ELEMENT figref          (fignum, #PCDATA)>
<!--      ELEMENTS  NAME  VALUE              DEFAULT -->
<!ATTLIST figref    refid IDREF              #CONREF>
```

The keyword means that the attribute is a content reference attribute, and also that it is an impliable attribute (the same as if the default value were "#IMPLIED").

In the following example, the first “figref” has empty content, while the second has both a “fignum” and a character string:

```
Here is text with a generated figure reference to
<figref refid=figdavis>and a user-entered
reference to <figref><fignum>A-1</fignum>in the
Appendix</figref> as well.
```

The first “figref” did not (and could not) have an end-tag because the element was identified as empty by the explicit content reference.

An element that just happens to be empty because its content model is optional and no content is entered is not treated in this way. Such an incidentally empty element must have an end-tag (unless markup minimization is used, as explained later), because it is impossible to tell from the start-tag whether an instance of the element is actually empty.

It would be pointless (and is therefore prohibited) to designate a content reference attribute for an element that is declared to be empty.

To summarize (and expand slightly): content reference attributes can be designated for any element type that is not declared empty. When one or more such attributes has an explicit value specified on the start-tag, that instance of the element is considered empty, and no end-tag is permitted for it.

B.11 Content Model Exceptions

The content model specifies the elements that occur at the top level of the content (the subelements). However, there are instances when it may be necessary to refer to elements that are further down. An optional parameter of the element declaration, called the “exceptions” parameter, is used for this purpose.

B.11.1 Included Elements

In an indexed book, entry elements for the index could be interspersed anywhere in the source. During formatting, the procedures would collect, sort, and merge them, along with their page numbers, and print the index.

It is cumbersome to express the relationship of the book to its index entries in the usual way, as the entries would have to be included in almost every model group. Instead, the exceptions parameter can be employed:

```
<!--       ELEMENTS  CONTENT (EXCEPTIONS)? -->
<!ELEMENT textbook  (front?, body, rear?) +(entry)>
```

The plus sign is the ***plus*** delimiter. Here it means that index entries can occur anywhere in a textbook, even among the data characters. This portion of an exceptions parameter is called an “inclusion” group.

(Incidentally, the question mark on the heading comment is just a reminder that the exceptions parameter is optional.)

B.11.2 Excluded Elements

It might be desirable to keep some element from showing up at *any* level of an element being defined. For example, figures could be kept from nesting with the following declaration:

```
<!--       ELEMENTS  CONTENT (EXCEPTIONS)? -->
<!ELEMENT fig       (figbody, figcap?) -(fig)>
<!ELEMENT figbody   (artwork | p+)         >
<!ELEMENT p         (#PCDATA | fig)        >
>
```

The content model clearly does not allow a figure to occur at the top level of another figure, but the figure body could contain a paragraph that could contain a figure. The exceptions parameter with the ***minus*** prefix (hyphen) prevents such occurrences; it is called an “exclusion” group.

A GI can be in an exclusion group only if its token in all affected model groups has an ***opt*** or ***rep*** occurrence indicator, or is in an ***or*** group or an applicable inclusion group. In other words, you cannot exclude anything with an exclusion group that the model group requires to be in the document. It follows from this that you cannot use an exclusion group to change the required or optional status of a token; for example, you cannot exclude all the members of a required ***or*** group, thereby rendering it no longer required.

It is possible to specify both exclusion and inclusion exception groups, in that order.

```
<!--      ELEMENTS  CONTENT (EXCEPTIONS)? -->
<!ELEMENT fig       (figbody, figcap?) -(fig|xmp) +(gloss)>
```

If the same GI is in both an exclusion and an inclusion group, whether in the same content model or in the content models of different open elements, its presence in the exclusion group governs.

B.12 Document Type Declaration

The document type of an SGML document is identified by a "document type declaration", which occurs in the "prolog" of the document, before any data. The element, attribute list, and other declarations that have been discussed, which constitute the document type definition, are grouped in a parameter called the "declaration subset":

```
<!DOCTYPE manual [
  <!ELEMENT manual (front?, body, rear?) +(entry)>
  <!-- Remainder of declarations constituting the
       document type definition go here. -->
]>
```

The left and right square brackets are the "declaration subset open" (***dso***) and "declaration subset close" (***dsc***) delimiters, respectively.

The part of an SGML document that occurs after the prolog, and which contains the data and descriptive markup, is called an "instance" of the document type (or just "document instance").

If, as is usual, a number of documents conform to the same document type definition, a single copy of the definition can be kept in an external entity and referenced from the documents. The entity can simultaneously be declared and referenced by specifying an external identifier on the document type declaration:

```
<!DOCTYPE manual PUBLIC "-//Cave Press//DTD Manual//EN">
```

Where the entire definition is external, as in the above example, no declaration subset is needed. Normally, an external definition and a subset are used together, with the subset containing things like entity declarations that apply only to the one document.

```
<!DOCTYPE manual PUBLIC "-//Stutely Press//DTD Manual//EN" [
  <!ENTITY title "AIBOHPHOBIA: Fear of Palindromes">
  <!-- Remainder of local declarations supplementing
       the document type definition go here. -->
]>
```

The external entity is technically considered part of the subset, as if a reference to it occurred just before the ***dsc***. This allows the declarations in the document to be executed first, which gives them priority over those in the external entity.

B.13 Data Content

The data content of a document is the portion that is not markup. It has two major characteristics:

a) Its *representation* determines whether the markup parser can scan it.

   There are two main classes: character data, in which the bit combinations represent characters, and bit data, in which the bit combinations (singly or collectively) usually represent binary values.

b) Its *notation* determines how the character or bit strings will be interpreted by the procedures.

In a natural language notation, for example, the character string "delta" might be interpreted as an English word, while in a scientific notation it could be interpreted as a single graphic, a Greek letter.

B.13.1 Data Content Representations

The Standard recognizes two data representations: character data that conforms to the Standard, and non-SGML character or bit data that does not.

B.13.1.1 Character Data (PCDATA, CDATA, and RCDATA)

In character data, each bit combination represents a character in the document character set.

Before a character can be considered character data, it normally must be parsed to determine whether it is markup. Such characters are represented in a model group by the keyword "PCDATA".

In a paragraph with the following structure, a character could either be data or part of a "phrase" or "quote" element tag:

```
<!ELEMENT p (#PCDATA | phrase | quote)* >
```

A character could also be part of a markup declaration, processing instruction, entity reference, or other markup that is allowed in the content of an element. Only if it is not a tag or other markup would it be treated as data and passed to a procedure for processing.

A character used for markup is not normally passed to a procedure, but when the optional "DATATAG" feature (discussed later) is used, a character could be both markup *and* data. A space, for example, could serve as the end-tag for a word and still be part of the data of the sentence in which the word occurs.

It is also possible to enter character data directly, without it being parsed for markup in the usual way. An element declaration can declare its content to contain character data by specifying a keyword, rather than a content model:

```
<!ELEMENT formula CDATA >
```

Note that no ***rni*** was required, because a user-defined name cannot be specified for this parameter (except within model group delimiters).

If an element contains declared character data, it cannot contain anything else. The markup parser scans it only to locate an ***etago*** or ***net***; other markup is ignored. Only the correct end-tag (or that of an element in which this element is nested) will be recognized.

A variation of CDATA, called *replaceable character data*, is specified with the keyword "RCDATA". It is like CDATA except that entity references and character references are recognized.

B.13.1.2 Non-SGML Data (NDATA)

Non-SGML data is data that is not parsable in accordance with this Standard. It is either data in an undefined character set, bit data, or some mix of the two. In undefined character set data, the bit combinations represent characters, but not in the document character set. In bit data, the bit combinations, although they can be managed as characters, do not represent a character repertoire in the usual way.

As non-SGML data cannot be scanned by the markup parser, and may contain bit combinations that could, for example, erroneously cause the operating system to terminate the file, it must be stored in an external entity. Its location is made known to the parser by an entity declaration, in the usual manner, except for additional parameters identifying it as non-SGML data:

```
<!ENTITY japan86 SYSTEM NDATA kanji>
```

Non-SGML data can be entered only in general entities whose declaration specifies the NDATA keyword and the name of a data content notation (explained later). Such an entity can be referenced with a general entity reference anywhere in the content that a reference is recognized and a data character could occur.

If it is desired to associate attributes with the non-SGML data, a dedicated element type can be defined for it. An NDATA element must have empty content and an attribute definition for an "ENTITY" ("general entity name") attribute whose value is the name of the external entity.

```
<!ELEMENT japanese EMPTY>
<!ATTLIST japanese file    ENTITY         #REQUIRED
                   subject (poetry|prose) prose
>
```

The markup for a "japanese" element includes the entity name as the value of the "file" attribute:

```
<japanese file="japan86">
```

An external entity attribute can be defined for any element, even one with a content model. The application determines, as with all attributes, how its value relates to the element content for the processing being performed.

B.13.2 Data Content Notations

The difference between one type of non-SGML data and another (or one kind of character data and another, for that matter) lies in the data content notation. The notation is not significant to the markup parser, but it is quite significant to humans and procedures.

An enormous number of data content notations are possible. However, they tend to fall into a few distinct classes.

B.13.2.1 Notations for Character Data

Some common classes of notation for character data are natural languages, scientific notations, and formatted text.

Natural Languages:

When a natural language notation is used, the procedures interpret the characters as implicitly marked text elements. For example, a sequence of characters bounded by interword spaces could be recognized as a "word" element, a sequence of words terminated by suitable punctuation as a "phrase" or "sentence" element, and so on. In such notations, an ***RE*** is usually interpreted as an interword space.

For additional flexibility, a natural language text notation can be supplemented by explicit markup for specialized variants of elements that are normally implicit (for example, quoted phrases, programming keywords, emphasized phrases).

In conventional text processing systems, implicit markup conventions and the trade-offs between explicit and implicit markup are built in. In SGML, the data tag markup minimization feature (discussed later) allows a user to specify the extent to which such elements will be recognized by the markup parser and the extent to which they will be interpreted by the procedures as part of the data content notation.

Natural language notations occur in elements such as paragraphs, list items, and headings. The content is usually *mixed content*, in which data characters are intermixed with subelements.

Scientific Notations:

These notations look like natural language notations, but the words and phrases are meaningful to the application. In a mathematical notation, for example, the phrase "3 over 4" could be interpreted as the fraction "3/4".

Scientific notations occur in elements such as formulas, equations, chemical structures, formal grammars, music, and so on. The element content is usually CDATA or RCDATA, which allows systems that cannot interpret the scientific notation to process the element as if a natural language notation had been used.

Formatted Text:

Formatted text notations are similar to natural language notations, and features of both might be incorporated in a single notation. Their purpose is to identify elements of the "layout structure" produced by a previous formatting application. Formatted text notations include character sequences that identify formatted line endings, spaces and hyphens introduced for alignment purposes, font change codes, superscripts and subscripts, etc.

These notations are found in the same element types as natural language notations (paragraphs, headings, etc.) and, when the optional concurrent document type feature is used (discussed later), in elements of the layout structure of the formatted document. The content is normally mixed content.

B.13.2.2 Notations for Non-SGML Data

In NDATA entities, the bit combinations (singly or collectively) can represent undefined characters, binary values, or a mixture of binary values and characters.

Undefined Characters:

Natural language, scientific, and formatted text notations are found in NDATA just as in CDATA, but in other than the document character set. The data could, for example, employ code extension techniques that assign multiple bit combinations to a single character, while the document character set does not. Such a scheme is used for languages that require more characters than there are bit combinations in the document character set.

Binary Values:

Binary values can be interpreted as grey scale or color values for pixels in an image, digitized audio, musical wave forms, or as other sets of numeric or logical values. The principal application of binary value notations in text processing is in the representation of illustration elements: half-tones, photographs, etc.

Record boundaries are frequently ignored in binary notations, but need not be.

Mixed Binary and Character:

A scientific or formatted text data notation could use a mixture of characters and binary fields. Such notations are treated as unparsable NDATA notations because bit combinations in the binary fields could erroneously be interpreted as markup delimiters.

B.13.2.3 Specifying Data Content Notations

The set of data content notations used in the document must be declared with *notation declarations*. Each declaration specifies the name of a data content notation used in the document, together with descriptive information about it:

```
<!NOTATION eqn    PUBLIC "-//local//NOTATION EQN Formula//EN">
<!NOTATION tex    PUBLIC "-//local//NOTATION TeX Formula//EN">
<!NOTATION lowres SYSTEM "SCAN.MODULE" -- Low resolution scan -->
```

An element's data content notation is specified with a notation attribute. The attribute's declared value is the keyword "NOTATION" followed by an additional parameter, a name group:

```
<!--      ELEMENTS         CONTENT                  -->
<!ELEMENT formula          RCDATA>
<!--      ELEMENTS NAME    VALUE             DEFAULT -->
<!ATTLIST formula  data    NOTATION(eqn|tex) #REQUIRED>
<!>
<formula data="eqn">3 over 4</formula>
```

The name group contains the valid values of the attribute.

B.14 Customizing

Many of the characteristics of the Standard Generalized Markup Language can be tailored to meet special needs.

B.14.1 The SGML Declaration

The tailoring of a document is described in its *SGML declaration*, a markup declaration that appears as the first thing in a document. The SGML declaration is normally provided automatically by the markup parser, but if a document's tailoring differs from what a processing system expects, the SGML declaration must be entered explicitly.

There are two main categories of tailoring: using optional features of SGML and defining an variant concrete syntax.

B.14.1.1 Optional Features

There are eleven optional features that can be used:

SHORTREF	Short entity reference delimiters
CONCUR	Concurrent document type instances
DATATAG	Data tag minimization
OMITTAG	Omitted tag minimization
RANK	Omitted rank suffix minimization
SHORTTAG	Short tag minimization
SUBDOC	Nested subdocuments
FORMAL	Formal public identifiers
SIMPLE	Simple link process
IMPLICIT	Implicit link process
EXPLICIT	Explicit link process

They are described in annex C.

B.14.1.2 Variant Concrete Syntax

The heart of SGML is the "abstract syntax" that defines the manner in which such markup constructs as generic identifiers, attributes, and entity references are used. The delimiter characters, declaration names and keywords, length restrictions, etc., that have been discussed in this Annex constitute a particular mapping of the abstract syntax to a real set of characters and quantities, known as the "reference concrete syntax".

SGML allows a document to employ a variant concrete syntax to meet the needs of system environments, national languages, keyboards, and so on. The characteristics of the concrete syntax are declared on the SGML declaration in the following categories:

a) Delimiter assignments, including short entity reference delimiters.

b) Character use, including identification of function characters and candidates for non-SGML characters.

c) Naming rules, including name character alphabet and case translation.

d) Definition of substitute declaration names, keywords, and other reserved names.

e) Quantitative characteristics, such as the maximum length of names and attribute values.

B.14.2 Impact of Customization

An SGML document that uses the reference concrete syntax and no features (known as a "minimal SGML document") will be interchangeable among all SGML systems. Over time, though, it is expected that other combinations of features and variant concrete syntaxes will come into wide use. Some possibilities are:

a) Documents that are keyed directly by humans for publishing applications will probably use the SHORTREF, SHORTTAG, and OMITTAG features. (Those that use only these features and the reference concrete syntax are known as "basic SGML documents".)

b) Documents that will be used for linguistic analysis or in connection with structured data bases will probably use the DATATAG feature.

c) Documents produced by intelligent word processors will probably use little minimization. Such systems will also employ the concurrent document type feature so that unformatted "logical structures" and formatted "layout structures" can be represented simultaneously.

d) User organizations will define feature menus to meet their special requirements.

B.15 Conformance

A document that complies with this International Standard in every respect is known as a "conforming SGML document". A system that can process such a document is known as a "conforming SGML system". This International Standard sets no requirements on the architecture, the method of implementation, or the handling of markup errors, employed by conforming systems.

Annex C

Additional Concepts

(This annex does not form an integral part of this International Standard.)

This annex introduces the optional features that can be used in an SGML document. There are three categories: markup minimization, link type, and other features.

— Markup minimization features

These features allow markup to be minimized by shortening or omitting tags, or shortening entity references. Markup minimization features do not affect the document type definition, so a minimized document can be sent to a system that does not support these features by first restoring the omitted markup.

The features are SHORTTAG, OMITTAG, SHORTREF, DATATAG, and RANK.

— Link type features

These features allow the use of "link process definitions", which specify how a source document should be processed to produce a result document of another type (such as a formatted document).

Link processes are specified in *link type declarations*, which are independent of document type declarations or any other markup. However, they must be removed before sending a document to a system that does not support these features.

The link type features are SIMPLE, IMPLICIT, and EXPLICIT, which refer to the degree of control that can be exercised in specifying the result document.

— Other features

These features allow elements and entities to be redefined for different parts of the document, and public identifiers to be interpreted for automatic processing. They do affect the document type definition, so a document using these features may require modification before sending it to a system that does not support them.

The features are CONCUR, SUBDOC, and FORMAL.

Use of the optional features is indicated on the feature use parameter of the SGML declaration (except for SHORTREF, whose use is indicated by the concrete syntax, and by its own "short reference mapping" declarations).

C.1 Markup Minimization Features

The markup minimization features are:

SHORTTAG means short tags with omitted delimiters, attribute specifications, or generic identifiers may be used.
OMITTAG means some tags may be omitted altogether.
SHORTREF means short reference delimiters may be used instead of complete entity references.
DATATAG means data characters may serve simultaneously as tags.
RANK means element ranks may be omitted from tags.

C.1.1 SHORTTAG: Tags With Omitted Markup

A short tag is one from which part or all of the usual markup has been omitted.

C.1.1.1 Unclosed Short Tags

The ***tagc*** can be omitted from a tag that is immediately followed by another one.

For example, this markup:

```
<chapter><p>A short chapter.</p></chapter>
```

can be abbreviated to this:

```
<chapter<p>A short chapter.</p</chapter>
```

by omitting the ***tagc*** from the "chapter" start-tag and the "p" end-tag.

C.1.1.2 Empty Tags

An empty short tag is one in which no GI or attributes are specified; the tag consists solely of its delimiters. The parser assumes, for an empty end-tag, that the GI is the same as that of the most recent open element. For example, the following would be equivalent with short tag minimization:

```
This is a <q>quoted</q> word.
This is a <q>quoted</> word.
```

For empty start-tags, when the omitted tag feature is not also enabled, the parser uses the GI of the most recently ended element and the default values of the attributes. Using both empty start- and end-tags, a list could be marked up as follows:

```
<!ELEMENT -- CONTENT                EXCEPTIONS? --
1  list      (item+)
2  item      (p | list)*
>
<list>
<item>This is the first item (what else ?)</>
<>This is the second item.</>
<>This is the third and last item.</>
</list>
```

(See C.1.2.6 for use of the short tag feature when the omitted tag feature is enabled.)

Markup can be reduced further by using the single character "null end tag" (***net***) delimiter (normally "/") that is enabled by the short tag feature. A ***net*** is interpreted as an empty end-tag only for an element in which it was also used as (that is, in place of) the ***tagc*** delimiter.

```
<p>This paragraph has
a <q/quotation/ in it and
a solidus (/) that is data.</p>
```

The following list summarizes the delimiters used with the short tag feature and their character strings in the reference delimiter set:

String	**Name**	***Meaning***
/	**NET**	Null end-tag.

C.1.1.3 Attribute Minimization

All or part of an attribute specification list can be omitted under the following circumstances:

Value Delimiters:

The delimiters can be omitted from the specification of an attribute value if the value is limited to name characters.

```
<standard security=public>
```

Note that entity references are not permitted in such attribute values.

Defaulting:

If an attribute was declared with an actual default value, or the keywords "#IMPLIED" or "#CONREF", the complete specification for it can be omitted. The attribute will be treated as though the default value had been specified for it. (This rule also applies to current attributes after they have once been specified.)

Names:

An attribute's name and ***vi*** delimiter can be omitted if its declared value included a *name group* or *name token group*.

This form of minimization is useful when the attribute values imply the attribute name. A memo, for example, might have a "status" attribute whose value was either "draft" or "final".

```
<!--       ELEMENTS            CONTENT                   -->
<!ELEMENT  memo                (from, to, subject, body, sig, cc?)>
<!--       ELEMENTS  NAME      VALUE          DEFAULT  -->
<!ATTLIST  memo      status    (final|draft)  final    >
```

The usual markup for the attribute:

```
<memo status="draft">
```

would be cumbersome in this instance, since "draft" implies "status". With SHORTTAG minimization, however, either

```
<memo status="draft">
```

or

```
<memo status=draft>
```

or

```
<memo draft>
```

could be entered in the document.

Omitting attribute names tends to make the document markup more ambiguous. This effect can be ameliorated if the groups are kept small and the name tokens chosen are descriptive adjectives that imply the attribute name (for example, "NEW | REVISED", "SECRET | INTERNAL | PUBLIC"). In the following example, the "compact" attribute is better from this standpoint than is "emphasis":

```
<!--       ELEMENTS            CONTENT             -->
<!ELEMENT  list                (item*)>
<!--       ELEMENTS  NAME      VALUE      DEFAULT  -->
<!ATTLIST  list      compact   (compact)  #IMPLIED
                     emphasis  (0|1|2|3)  0
>
```

C.1.2 OMITTAG: Tags May be Omitted

A type definition establishes the possible variations in structure among documents of its type. An individual document, therefore, need not contain as much markup.

C.1.2.1 Tag Omission Concepts

Assume that a class of magazine article documents, with a GI of "article", has the following type definition:

```
<!--      ELEMENTS  CONTENT                >
<!ELEMENT article   (title, body)          >
<!ELEMENT title     (#PCDATA)              >
<!ELEMENT body      (p*)                   >
<!ELEMENT p         (#PCDATA | list)*      >
<!ELEMENT list      (item+)                >
<!ELEMENT item      (#PCDATA, (p | list)*) >
```

The full markup for an instance of an article might look something like this:

```
<article>
<title>The Cat</title>
<body>
<p>A cat can:
<list>
<item>jump</item>
<item>meow</item>
</list>
</p>
<p>It has 9 lives.
</p>
</body>
</article>
```

Note that the type definition says that a paragraph cannot immediately contain another paragraph; it can contain only text characters and lists. Similarly, a list item cannot immediately contain another list item (even though it could contain another list). As a result, the markup can be minimized by omitting many of the end-tags:

```
<article>
<title>The Cat</title>
<body>
<p>A cat can:
<list>
<item>jump
<item>meow
</list>
<p>It has 9 lives.
</article>
```

It is possible to omit the item end-tag: the occurrence of another item indicates the end of the previous one (because an item cannot contain an item). The paragraph end-tag is omissible by the same reasoning (a paragraph cannot contain another paragraph).

Finally, it is logical that the end of an element should also be the end of everything it contains. In this way, the article end-tag ends the body and the last paragraph as well as the article.

(The markup could be minimized still further, as the reader will no doubt realize.)

C.1.2.2 Specifying Minimization

Markup minimization is a good thing, but not if it makes it harder to detect markup errors. In the last example, had the list end-tag been omitted, it would have been implied by the article end-tag, just as the body end-tag was.

```
<article>
<title>The Cat</title>
<body>
<p>A cat can:
<list>
<item>jump
<item>meow
<p>It has 9 lives.
</article>
```

This would not have been the author's intention, though, as the last paragraph would erroneously have been made part of the last item in the list.

To prevent such misinterpretations, there are two parameters on the element declaration that specify the omitted tag minimization that is allowed for an element. When "OMITTAG YES" is specified on the SGML declaration, the omitted tag feature is enabled, and the two minimization parameters are then required in all element declarations.

C.1.2.3 End-tag Omission: Intruding Start-tag

An end-tag is omissible when its element's content is followed by the start-tag of an element that cannot occur within it. This was the case for the paragraph and item end-tags in the article example. The element declarations would look like this:

```
<!--       ELEMENTS  MIN  CONTENT               >
<!ELEMENT p          - O  (#PCDATA | list)*     >
<!ELEMENT list       - -  (item+)               >
<!ELEMENT item       - O  (#PCDATA, (p | list)*) >
```

and have the following meaning:

a) The "MIN" heading identifies a pair of parameters for start-tag and end-tag minimization. Both parameters must be specified.

b) The "O" indicates that omitted tag minimization is allowed for the end-tag of the "p" and "item" elements.

c) The hyphen is the ***minus*** delimiter; it indicates that no minimization is allowed for the start-tags.

Recall that an end-tag must always be omitted if an element has empty content, because anything that follows the start-tag must then be part of the containing element. Although this rule has nothing to do with markup minimization, it is helpful to mark the "O" as a reminder that no end-tags will be found in the document.

```
<!ELEMENT figref - O EMPTY>
```

C.1.2.4 End-tag Omission: End-tag of Containing Element

A contained element's end-tag is omissible when it is followed by the end-tag of an element that contains it.

```
<!--       ELEMENTS  MIN  CONTENT               >
<!ELEMENT list       - -  (item+)               >
<!ELEMENT item       - O  (#PCDATA)             >
```

The above declaration allows the end-tag of the third item to be omitted from the following list, because it is implied by the list end-tag:

```
<list>
<item>This is the first item (what else ?)</item>
<item>This is the second item.</item>
<item>This is the third and last item.
</list>
```

C.1.2.5 Start-tag Omission: Contextually Required Element

A start-tag is omissible when the element type is contextually required and any other element types that could occur are contextually optional.

Combined with the other minimization discussed previously, the element declaration would be:

```
<!--       ELEMENTS  MIN  CONTENT                >
<!ELEMENT  list      - -  (item+)                >
<!ELEMENT  item      0 0  (#PCDATA)              >
```

and the list could be marked up like this:

```
<list>
This is the first item (what else ?)
<item>This is the second item.
<item>This is the third and last item.
</list>
```

The start-tag was omissible for the first item because it was required; it could not be omitted for the subsequent items because they were optional.

Even when an element is contextually required, its start-tag cannot be omitted if the element type has required attributes or a declared content, or if the instance of the element is empty.

C.1.2.6 Combination with Short Tag Minimization

When short tag and omitted tag minimization are both enabled, empty start- and end-tags are treated uniformly: they are given the GI of the most recent open element. The two forms of minimization can be used together, as in the following example:

```
<!--       ELEMENTS  MIN  CONTENT                >
<!ELEMENT  p         - 0  (#PCDATA | list)*      >
<!ELEMENT  list      - -  (item+)                >
<!ELEMENT  item      0 0  (#PCDATA, (p | list)*) >
<list>
<item>This is the first item (what else ?)
<>This is the second item.
<>This is the third and last item.
</list>
```

Note that it was necessary to say "</list>" instead of "</>" at the end, because the latter would have meant "item end-tag", rather than "list end-tag". Alternatively, "</></>" could have been entered to mean "end the item, then end the list".

Now consider what happens when the markup is minimized further by removing the first item start-tag:

```
<list>
This is the first item (what else ?)
<>This is the second item.
<>This is the third and last item.
</list>
```

The identical results are achieved because when the first empty tag was encountered, "item" and not "list" was the open element, even though "item" was implied by "O" minimization rather than entered explicitly.

C.1.2.7 Markup Minimization Considerations

Short tag minimization requires the user (and the markup parser) to know the current location in the document's structure, and to understand attribute list declarations. Omitted tag minimization, though, also requires knowledge of the element declarations.

This difference should be considered in deciding what kind of minimization the user can be expected to handle properly, as omitting a tag could result in a document that, while free of SGML errors, is not what the user intended.

C.1.3 SHORTREF: Short Reference Delimiters May Replace Complete Entity References

Short references are single characters or short strings that can replace complete delimited entity references. They can be used to emulate common typewriter keyboarding and to simplify the entry of elements with repetitive structures.

C.1.3.1 Typewriter Keyboarding: Generalized WYSIWYG

Some word processors offer text entry operators an interface similar to that of the typewriters on which many operators were trained. On typewriters, each function key produces an immediate formatting result as it is struck: a carriage return starts a new line, a tab key generates horizontal space, and so on. Word processors that emulate this characteristic of typewriters are sometimes known as "WYSIWYG" systems—for "what you see is what you get".

The characters produced by function keys are specific processing instructions. Like all processing instructions, they limit a document to a single formatting style, executable only on machines that understand the instructions.

SGML, though, can offer the benefit of familiar typewriter keyboarding while still maintaining the generality of the document. With short references, typewriter function key characters can be interpreted as descriptive markup. For example, the declarations

```
<!ENTITY   ptag    STARTTAG      "p"  >
<!SHORTREF wysiwyg "&#TAB;"      ptag
                   "&#RS;&#RE;"  ptag >
```

map both the tab and the line feed, carriage return sequence (empty record) to the start-tag for a paragraph element. The actual formatting for a paragraph depends, as always, on the application procedure, just as if the start-tag had been entered explicitly. But the entry device still produces the tabbing or extra line called for by the function characters, thereby giving the user the immediate visual feedback that is such an important aspect of WYSIWYG systems.

The markup declarations have effectively created "generalized WYSIWYG", in which typewriter function characters are interpreted as generalized markup while retaining their visual effect. Moreover, the user can freely intermix generalized WYSIWYG with explicit tags, using whichever is most convenient for a particular part of the document. This ability can be particularly helpful for complex elements, such as multipage tables with running headings, where typewriter functions do not offer a convenient and generally accepted entry convention.

C.1.3.2 Typewriter Keyboarding Example: Defining a Short Reference Map

With short references, a user can create SGML documents with common word processing entry conventions, rather than conscious entry of markup. A large number of character strings are defined as short reference delimiters in the reference concrete syntax. Those containing "invisible" function characters and the quotation mark are particularly useful for supporting common keyboarding; they are:

String	*Description*
&#TAB;	Horizontal tab
&#RE;	Record end
&#RS;	Record start
&#RS;B	Leading blanks (record start, one or more spaces and/or tabs)
&#RS;&#RE;	Empty record (record start, record end)
&#RS;B&#RE;	Blank record (record start, one or more spaces and/or tabs, record end)
B&#RE;	Trailing blanks (one or more spaces and/or tabs, record end)
&#SPACE;	Space
BB	Two or more blanks (two or more spaces and/or tabs)
"	Quotation mark

Each short reference delimiter can be associated with an entity name in a table called a "short reference map". A delimiter that is not "mapped" to an entity is treated as data. Entity definition is done in the usual manner, with an entity declaration. The mapping is done with a "short reference mapping declaration". The following example defines an empty record as a reference to the paragraph start-tag:

```
<!ENTITY   ptag "<p>">
<!SHORTREF map1 "&#RS;&#RE;" ptag>
```

The "SHORTREF" declaration defines a map named "map1". The map contains only one explicit mapping: the empty record short reference is mapped to the entity "ptag". Whenever this map is current (explained later), an empty record will be replaced by the entity reference. For example,

```
Last paragraph text.

Next paragraph text.
```

will be interpreted as

```
Last paragraph text.
&ptag;
Next paragraph text.
```

which will instantaneously, upon resolution of the entity reference, be reinterpreted as

```
Last paragraph text.
<p>
Next paragraph text.
```

As no other short reference strings were mapped, they will be treated as data while this map is current.

C.1.3.3 Typewriter Keyboarding Example: Activating a Short Reference Map

Normally, a map is made current by associating its name with an element type in a "short reference use" declaration. The following example causes "map1" to become the current map whenever a "chapter" begins:

```
<!USEMAP map1 chapter>
```

Whenever a chapter begins, map1 will become current and will remain current except within any nested subelements that are themselves associated with maps.

A short reference map need not be associated with every element type. An element that has none will simply use whichever map is current at its start. In the following example, the quote and chapter elements have maps, but the paragraph element does not:

```
<!ENTITY   ptag     "<p>"         -- paragraph start-tag -- >
<!ENTITY   qtag     "<quote>" >
<!ENTITY   qendtag  "</quote>" >
<!SHORTREF chapmap  "&#RS;&#RE;"  ptag
                    '"'           qtag >
<!SHORTREF qmap     '"'           qendtag>
<!USEMAP   chapmap  chapter>
<!USEMAP   qmap     q>
<!ELEMENT  chapter  (p*)          >
<!ELEMENT  p        (q|#PCDATA)*  >
<!ELEMENT  q        (#PCDATA)     >
```

When a paragraph element begins, "chapmap" will remain current; it maps the quotation mark to the "qtag" entity, which contains the quote start-tag. Once the quote element begins, "qmap" becomes the current map, and the quotation mark will be replaced by a reference to the "qendtag" entity, which contains the quote end-tag. At the end of the quote, the paragraph element resumes, and with it the "chapmap" short reference map.

The markup

```
<chapter<p>Here is "a quotation" in the text.</p>
```

is now the equivalent of

```
<chapter<p>Here is <quote>a quotation</quote> in the text.</p>
```

The declarations allow the quotation mark to serve as the start-tag and end-tag of quotation elements, rather than being treated as data. Formatting procedures could therefore use opening and closing curved quotation marks to distinguish the start and end of the quotation, which would not otherwise be possible with normal typing conventions.

Incidentally, when declaring an entity whose replacement text is a tag, the following forms of entity declaration can be used:

```
<!ENTITY qtag    STARTTAG "quote" >
<!ENTITY qendtag ENDTAG   "quote" >
```

The "STARTTAG" and "ENDTAG" keywords allow the tag delimiters to be omitted. They also allow a system to optimize the handling of the entity because it knows it will most likely be used as a tag rather than as data.

Short entity references can be used only in elements whose content was defined by a model (that is, not in CDATA or RCDATA). They cannot be used in attribute values or declaration parameters.

C.1.3.4 Tabular Matter Example

Many printable characters are defined as short reference delimiters in the reference concrete syntax (see figure 4). These characters can be used as substitutes for the more lengthy general entity references, which allows a more concise and visual style of markup when appropriate, as in tables:

```
<!ENTITY   row      "<row><col>" >
<!ENTITY   col      "</col><col>" >
<!ENTITY   endrow   "</col></row>" >
<!SHORTREF tablemap "(" row
                    "|" col
                    ")" endrow >
<!USEMAP   tablemap table>
<!ELEMENT  table    (row*)>
<!ATTLIST  table    columns NUMBER #REQUIRED>
<!ELEMENT  row      (col+)>
<!ELEMENT  col      (#PCDATA)>
<!>
<table columns=3>
(row1,col1|row1,col2|row1,col3)
(row2,col1|row2,col2|row2,col3)
(row3,col1|row3,col2|row3,col3)
(row4,col1|row4,col2|row4,col3)
</table>
<!>
```

The example allows the parentheses and the vertical bar (|) to be used as entity references. (As short references are only recognized in content, there is no danger of the vertical bar being construed as an ***or*** connector.) The text will resolve to the following:

```
<table columns=3>
<row><col>row1,col1</col><col>row1,col2</col><col>row1,col3</col></row>
<row><col>row2,col1</col><col>row2,col2</col><col>row2,col3</col></row>
<row><col>row3,col1</col><col>row3,col2</col><col>row3,col3</col></row>
<row><col>row4,col1</col><col>row4,col2</col><col>row4,col3</col></row>
</table>
```

The "tablemap" is current only during the table so that the short reference delimiters can be used as data in the rest of the document.

C.1.3.5 Special Requirements

Although short reference maps are normally associated with specific element types, it is possible to use a short reference use declaration to select one arbitrarily:

```
<figure>
Opening text of figure (uses normal figure map).
<!USEMAP graphics -- Enable character graphics short refs -->
Remaining text of figure (uses "graphics" map).
</figure>
```

The map named in the declaration (here "graphics") replaces the current map for the most recently started element (here "figure"), just as if it had been named on the "figure" element declaration. However, it applies only to this instance of a figure, not to any other. If the normal figure map is required later in this figure, it must be activated with a short reference use declaration, but for later figures it will become current automatically.

It is possible to cancel all mappings by using the reserved map name "#EMPTY" instead of a normal map name. The "empty" map, which causes all short reference delimiters to be treated as data (or separators), will be current under the same circumstances as a normal map would have been.

Short references, though powerful, are not adequate for interpreting arbitrary existing documents as if they had been marked up with SGML. They are intended to let a user supplement normal SGML markup with familiar word processing entry conventions. In conjunction with the other markup minimization techniques, short references make it possible to eliminate most conscious markup, even without an "intelligent" word processor.

C.1.4 DATATAG: Data May Also be a Tag

The data tag minimization feature allows a content model to define data characters that will be interpreted as tags while remaining part of the data content.

The techniques of markup minimization discussed so far go a long way toward creating text that appears almost free of markup. The following list, for example, has no visible explicit markup within it:

```
<!ENTITY   itemtag   STARTTAG jobitem --JOBITEM start-tag-->
<!SHORTREF listmap   "&#RS;"  itemtag >
<!USEMAP   listmap   joblist>
<!--       ELEMENTS  MIN      CONTENT  -->
<!ELEMENT  joblist   - -      (jobitem+)>
<!ELEMENT  jobitem   - O      (#PCDATA)>
<joblist>
Sharon Adler,         Vice Chairman
Larry Beck,           Secretary
Anders Berglund,      Publisher
Aaron Bigman,         Past Member
Jim Cox,              Past Member
Bill Davis,           Chairman
Joe Gangemi,          Member
Charles Goldfarb,     Editor
Mike Goodfellow,      Consultant
Randy Groves,         Member
Charles Lightfoot,    Past Member
Sperling Martin,      Past Member
Bettie McDavid Mason, Consultant
Jim Mason,            Member
Lynn Price,           Theoretician
Stanley Rice,         Pioneer
Norm Scharpf,         Observer
Craig Smith,          Theoretician
Joan Smith,           Publicist
Ed Smura,             Member
Bill Tunnicliffe,     Member
Kit von Suck,         Member
</joblist>
```

The end-tag for each "jobitem" but the last is omissible because it is followed by a jobitem start-tag. The last is implied by the end of the list. The start-tags are present explicitly, but not visible, because they are in entities referenced by a non-printing short reference delimiter (the record start). The list thus appears completely marked up with only two tags consciously entered by the user, but there is more that could be done.

The element declaration states that a "jobitem" is merely a sequence of characters. A reader, however, sees it as containing separate "name" and "job" elements because the comma and spaces identify the job title as clearly as a start-tag would have. Such a situation, in which the conventions used in the data can unambiguously identify elements within it, is known as *data tag minimization*.

The SGML data tag feature allows these conventions to be expressed in an element declaration through the use of a *data tag pattern*:

```
<!--       ELEMENTS  MIN   CONTENT  -->
<!ELEMENT  jobitem   - O   ([name, ", ", " "], job)>
<!ELEMENT  name      O O   (#PCDATA)>
<!ELEMENT  job       O O   (#PCDATA)>
```

The declaration states (among other things) that:

a) The start-tag for a "name" is omissible because it must be the first thing in a "jobitem".

b) The "name" end-tag is omissible when it occurs in a "jobitem", because it is implied by the comma and spaces of the data tag pattern.

c) The "job" start-tag is omissible because a "job" element is not allowed in a "name".

d) The "job" end-tag is omissible because it is implied by the end of the "jobitem".

The complete declaration for the "joblist" would appear as follows:

```
<!ENTITY   itemtag  STARTTAG jobitem --JOBITEM start-tag-->
<!SHORTREF listmap  "&#RS;"  itemtag >
<!USEMAP   listmap  joblist>
<!--       ELEMENTS MIN      CONTENT  -->
<!ELEMENT  jobitem  - O      ([name, ", ", " "], job)>
<!ELEMENT  name     O O      (#PCDATA)>
<!ELEMENT  job      O O      (#PCDATA)>
```

Characters are normally either data or markup, but not both. The essence of data tag minimization is that characters are both data and a tag at the same time. A data tag is an end-tag that conforms to a *data tag pattern* that follows the data tag's element in a *data tag group* in its containing element's model. In the above example, the data tag group is

```
[name, ", ", " "]
```

The group is a ***seq*** group, bracketed by the "data tag group open" (***dtgo***) and "data tag group close" (***dtgc***) delimiters (normally "[" and "]"), instead of the usual group delimiters, to indicate its special nature. There are three tokens in the group:

a) The GI of the minimizable element:

```
name
```

b) The *data tag template*, which in this case is a single literal, but could also be an ***or*** group made up of literals:

```
", "
```

The literal means "a comma followed by a space".

c) A *data tag padding template* (which is optional):

```
" "
```

It means that zero or more spaces could follow the required comma and space to constitute the data tag.

Thus, any string consisting of a comma and one or more spaces that occurs after a "name" element begins in a "jobitem" will be construed as the "name" end-tag. It will also, however, be considered part of the data of the "jobitem".

Note the differences between a data tag and a short reference whose entity is an end-tag:

a) The short reference string is markup, not data.

b) The short reference is recognized in any element whose map has it mapped to an entity. The data tag is recognized only when its element is open and the containing element is that in whose model the data tag group occurred.

c) The short reference is a constant string; the data tag can be any of a number of strings that conform to the data tag pattern.

Data tag minimization is useful for applications that analyze text. The following example uses data tags to identify sentences and words within a paragraph:

```
<!ENTITY % stop   '( ".&#RE;" | ".  " | ".)&#RE;" | ".)  " |
                     "?&#RE;" | "?  " | "?)&#RE;" | "?)  " |
                     "!&#RE;" | "!  " | "!)&#RE;" | "!)  " )' >
<!ENTITY % pause  '( " "   | "&#RE;"   | ", "  | ",&#RE;"  |
                     "; "  | ";&#RE;"  | ": "  | ":&#RE;"  |
                     ") "  | ")&#RE;"  | ",) " | ",)&#RE;" |
                     ";) " | ";)&#RE;" | ":) " | ":)&#RE;" |
                     "), " | "),&#RE;" | "); " | ");&#RE;" |
                     "): " | "):&#RE;"  )' >
<!ELEMENT p          - O   ([sentence, %stop;]+)>
<!ELEMENT sentence   O O   ([word, %pause;, " "]+)>
<!ELEMENT word       O O   (#PCDATA)>
<p>The first sentence ends here.
The second sentence ends
here.
This is the third sentence.  The
fourth sentence ends, not here!, but here!
</p>
```

In the example, a word end-tag is a "pause" character string followed by zero or more spaces. All words but the last in each sentence have a data tag that conforms to this pattern.

The sentence end-tag is a "stop" character string that ends with either a record end or two spaces. When the tag is recognized, an omitted end-tag for the last word is implied by the usual "end of containing element" rule.

Care must be taken with text entry when using data tag minimization. In the following example, an abbreviation will erroneously be treated as a sentence end-tag:

```
I wonder whether Mrs. G.
will read this.
```

The following list summarizes the delimiters enabled by the data tag feature and their character strings in the reference delimiter set:

Char	**Name**	***Meaning***
[	**DTGO**	Data tag group open.
]	**DTGC**	Data tag group close.

C.1.5 RANK: Ranks May be Omitted from Tags

The rank of an element is its level of nesting. In many document types, rank is only implied by the beginning and ending of nested elements, such as lists, and is never specified explicitly in the document markup.

Some markup designers, though, prefer to use explicit rank designations for some elements, such as headings and paragraphs (for example, p1, p2). Such elements tend to have declarations like:

```
<!--        ELEMENTS MIN CONTENT   -->
<!ELEMENT p1         - O (#PCDATA, p2*)>
<!ELEMENT p2         - O (#PCDATA, p3*)>
<!ELEMENT p3         - O (#PCDATA, p4*)>
<!ELEMENT p4         - O (#PCDATA)>
```

and document markup like:

```
<p1>Text of 1st level paragraph.
<p1>Another 1st level paragraph.
<p2>Nested 2nd level paragraph.
<p2>Another 2nd level paragraph.
<p1>Back to 1st level.
```

The SGML RANK feature offers a more convenient way of specifying rank explicitly. An element can be designated a *ranked element* by dividing its GI into a *rank stem* and a *rank suffix*, which must be a *number*.

```
<!--      ELEMENTS MIN CONTENT   -->
<!ELEMENT p 1       - O (#PCDATA, p2*)>
<!ELEMENT p 2       - O (#PCDATA, p3*)>
<!ELEMENT p 3       - O (#PCDATA, p4*)>
<!ELEMENT p 4       - O (#PCDATA)>
```

When the rank feature is used, a rank stem can be entered in a tag instead of the complete GI. The complete GI will be derived from the stem and the last rank suffix specified for an element with that stem.

```
<p1>Text of 1st level paragraph.
<p>Another 1st level paragraph.
<p2>Nested 2nd level paragraph.
<p>Another 2nd level paragraph.
<p1>Back to 1st level.
```

A group of element types can share the same rank if they have identical content models. Such a group is called a *ranked group*. For example, if a document had normal paragraphs, numbered paragraphs, and bulleted paragraphs, each might contain any of the three at the next level down. The declaration

```
<!--      ELEMENTS   MIN CONTENT   -->
<!ELEMENT (p|n|b) 1 - O (#PCDATA, (p2|n2|b2)*)>
<!ELEMENT (p|n|b) 2 - O (#PCDATA, (p3|n3|b3)*)>
<!ELEMENT (p|n|b) 3 - O (#PCDATA, (p4|n4|b4)*)>
<!ELEMENT (p|n|b) 4 - O (#PCDATA)>
```

lets a document contain:

```
<p1>Text of 1st level paragraph.
<n>Numbered 1st level paragraph.
<p2>Nested 2nd level paragraph.
<b>Bulleted 2nd level paragraph.
<p1>Back to 1st level.
```

C.2 LINK Features: SIMPLE, IMPLICIT, and EXPLICIT

The discussion thus far has been confined to markup for a single document type: the logical, or abstract, structure of a source document that is to be processed. However, the result of processing a document is also a document, albeit one that might have a radically different document type definition.

For example, a document that conforms to a one-dimensional source document type with a logical structure composed of chapters, sections, and paragraphs, will, after formatting, also conform to a two-dimensional result document type with a "layout" structure whose elements are pages, columns, text blocks, and lines.

The two document types will coincide at certain points, certainly at the highest (document) level and possibly at others. A chapter in the logical structure, for instance, might correspond to a "page set" in the layout structure.

SGML supports multiple document types in two different ways:

a) Link process definitions: document type definitions can be linked to specify how a document can be transformed from one type (the "source") to another (the "result"); for example, how the markup that would describe the data in terms of the result layout structure should be generated from the source logical markup.

b) Concurrent document instances: the markup for instances of multiple document types can exist concurrently in a single document.

C.2.1 Link Process Definitions

Link set declarations can be used in applications such as formatting to specify how logical elements in the source (for example, paragraphs or list items) should be transformed into layout elements in the result (for example, text blocks).

For example, if "para" and "item" elements were defined in the source, and a "block" element were defined in the result with an "indent" attribute, then

```
<!LINK docset para block [ indent=3 ]
              item block [ indent=5 ]
>
```

would cause a paragraph to be formatted as a text block with an indent of 3. List items would also be formatted as text blocks, but with an indention of 5. The source text:

```
<item>Text of list item.</item>
```

would become the result text:

```
<block indent=5>Text of list item.</block>
```

For all other cases (assuming SHORTTAG minimization), all other attributes of the block would have their default values, as defined in the block's element declaration.

The LINK features are described in detail in clause 12.

C.3 Other Features

The remaining features are:

CONCUR means that instances of the specified number of document types (1 or more) may occur concurrently with the base document type.

SUBDOC means the specified number of open subdocument entities (1 or more) may be nested in the SGML document.

FORMAL means that public identifiers are to be interpreted formally.

C.3.1 CONCUR: Document Instances May Occur Concurrently

It is sometimes useful to maintain information about a source and a result document type simultaneously in the same document, as in "what you see is what you get" (WYSIWYG) word processors. There, the user appears to interact with the formatted output, but the editorial changes are actually made in the source, which is then reformatted for display.

There are also cases in which even more than two such "views" of the document can be useful, such as maintaining multiple formatted results for instant display on both a CRT and a printer while still having the logical document type available for other applications.

The concurrent document instance feature supports multiple concurrent structural views in addition to the abstract view. It allows the user to associate element, entity, and notation declarations with particular document type names, via multiple document type declarations.

The document type names are then prefixed to tags and entity references, thus permitting multiple alternative sets of start-tags, end-tags, and entities to be used in a document in addition to the base set. Other markup and data can be associated with an instance of a particular document type by means of marked sections whose "INCLUDE" or "IGNORE" status is set by a qualified keyword associated with that type.

In the previous example, if the layout document type were called "layout", the concurrent markup for the formatted list item would be (without markup minimization):

```
<item>
<(layout)block indent=5>Text of list item.
</item>
</(layout)block>
```

C.3.2 SUBDOC: Nested Subdocument Entities May Occur

A document in SGML is made up of one or more entities. The one in which the document begins is called an "SGML document entity"; it is the one that contains the SGML declaration that identifies the features and concrete syntax used throughout the document. The SGML document entity also contains one or more document type definitions, and is marked up to conform to one or more of those definitions.

An entity that conforms to SGML markup is called an "SGML entity". An SGML entity can contain references to "non-SGML data entities" that contain only data that cannot be parsed, or to other SGML entities that don't contain their own document type definitions, called "SGML text entities".

When the subdocument feature is used, an SGML entity can also contain references to "SGML subdocument entities", which do contain their own document type definitions. An SGML subdocument entity must conform to the SGML declaration of the SGML document entity, but in other respects it establishes its own environment. The document types and entity declarations of the entity referencing a subdocument are suspended while the subdocument entity is open, and restored when it ends. Current rank begins anew in the subdocument, as do unique identifiers, so there can be no ID references between the SGML document and its subdocuments.

Subdocuments are referenced in the same manner as non-SGML data entities. A general entity reference can be used anywhere in content that it can be recognized, and where a data character could occur (that is, in mixed content or in replaceable character data). Alternatively, an element can be declared to have a general entity name attribute that locates the subdocument entity:

```
<!ENTITY  art1         SYSTEM SUBDOC>
<!ELEMENT article - -  EMPTY>
<!ATTLIST article      file ENTITY #REQUIRED>
<!>
<p>This topic is treated in the next article.</p>
<article file=art1>
```

Note that the element type need not be the same as the document type of the subdocument entity. The latter is specified by the document type declaration within it.

This feature allows separately created documents of various types to be incorporated into a single document, such as an anthology.

C.3.3 FORMAL: Public Identifiers are Formal

When this feature is used, public identifiers have a formal structure with a number of components:

a) An owner identifier, which can be an ISO publication number, an identifier that conforms to a registration standard to guarantee uniqueness, or a private ("unregistered") identifier.

b) A public text class, which identifies the kind of text being registered: SGML document, entity set, document type definition, and so on.

c) A public text identifier which names the registered text.

d) A designation of the natural language used, in the form of a standard two-character name.

e) An optional version, which identifies the output devices supported for device-dependent public text that is available in versions for more than one device. A system can automatically substitute the best version for the device in use during a given process.

Annex D

Public Text

(This annex does not form an integral part of this International Standard.)

SGML specifies how markup such as generic identifiers, attributes, and entity references is recognized, but no *specific* GIs or other names are part of the language. The vocabulary is made up by users to meet their needs, and is defined in the document type and link process definitions.

Substantial benefit can be derived from individual use of SGML (as was the case with earlier generic coding and generalized markup language designs). However, trade organizations, technical societies, and similar groups desiring to interchange documents could benefit further by sharing document type definitions and other markup constructs.

To this end, SGML includes a syntax for referencing text with public identifiers. This annex describes some applications for public text, and defines some public entity sets for immediate use.

D.1 Element Sets

Sets of element declarations can be created for elements that normally do not exist as independent documents, but which can occur in a variety of document types.

D.1.1 Common Element Types

There are certain elements that have well-recognized common forms that can occur in a variety of documents. Some involve specialized data content notations as well (see below).

Some examples from the United States, with sources of existing specifications, are:

a) Legal citations (Harvard Law Review)
b) Mathematical formulas (Association of American Publishers)
c) Chemical formulas (American Chemical Society)
d) Bibliography entries (Library of Congress)
e) Name and address (direct mail organizations, directory publishers, telephone companies)

D.1.2 Pro Forma Element Types

There are a number of element configurations that occur in a variety of documents and elements, but usually with differences in the GIs, or other variations. Public *pro forma* descriptions of such configurations would serve as guides for constructing specific descriptions that a user might require.

Examples of such elements are:

a) Paragraph (for typical formatting applications)
b) Paragraph (for detailed syntactic or other analysis)
c) Tables
d) Nested ranked paragraph structures
e) Lists

D.2 Data Content Notations

There are commonly used data content notations that could be adapted for use with SGML and given public identifiers.

For example, the EQN mathematical notation could be adapted as follows:

a) The commands that indicate the start and end of an EQN statement are not needed, as the tags for a formula element would serve that function.

b) The remainder of EQN could be used unmodified as the data content notation (possibly supplemented by descriptive markup for some of the mathematical elements).

```
<p>The formula,
<q><formula notation=EQN>E equals m
times c squared</formula></q>
should provide speedy enlightenment.</p>
```

D.3 Variant Concrete Syntaxes

The *shunned character identification* and *function character identification* parameters of the reference concrete syntax were chosen to maximize the chance of successful interchange with the widest possible set of SGML systems. Unlike dedicated word processors, text processing applications frequently reside in operating system environments over which they have no control, and which do not necessarily conform to ISO standards governing system architecture and communications. In such cases, a bit combination that SGML would treat as data might be interpreted as a control character by the operating system, resulting in abnormal behavior. To prevent such errors, the reference concrete syntax prohibits direct occurrence in the document of any bit combinations that might be construed as controls, except for four universally recognized function characters.

For some national languages, however, additional function characters are needed so that code extension techniques can be used to allow keyboards and displays to respond to changes in the character repertoire as text is entered or revised. This sub-clause defines two public variant concrete syntaxes that allow function characters for code extension to occur in SGML entities in a manner that prevents them, or the additional graphic characters, from mistakenly being interpreted as markup.

D.3.1 Multicode Concrete Syntaxes

The multicode basic concrete syntax is described by the SGML declaration *concrete syntax* parameter shown in figure 11. Its *public identifier* is:

```
"ISO 8879-1986//SYNTAX Multicode Basic//EN"
```

The multicode core concrete syntax is the same as the multicode basic concrete syntax, except that "NONE" is specified for the "SHORTREF" parameter. Its *public identifier* is:

```
"ISO 8879-1986//SYNTAX Multicode Core//EN"
```

These syntaxes allow markup to be recognized only in the G0 set because a shift to G1, G2, or G3 begins with a function character that suppresses markup recognition.

The LS0 function restores markup recognition while shifting back to G0. It must be entered after an escape sequence occurs to allow further markup recognition.

NOTE — Techniques for device-independent code extension that allow mixed use of ISO 2022 and non-ISO 2022 devices are discussed in clause E.3.

D.4 Entity Sets

Tens of thousands of graphic characters are used in the publishing of text, of which relatively few have been incorporated into standard coded character sets. Even where standard coded representations exist, however, there may be situations in which they cannot be keyboarded conveniently, or in which it is not possible to display the desired visual depiction of the characters.

To help overcome these barriers to successful interchange of SGML documents, this sub-clause defines character entity sets for some of the widely-used special graphic characters. The entity repertoires are based on applicable published and proposed International Standards for coded character sets, and current industry and professional society practice.

```
         SYNTAX

SHUNCHAR CONTROLS 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
            18 19 20 21 22 23 24 25 26 27 28 29 30 31 127 255
BASESET  "ISO 646-1983//CHARSET
          International Reference Version (IRV)//ESC 2/5 4/0"
DESCSET    0  14  0
          14   1 "LS0 in ISO 2022"
          15   1 "LS1 in ISO 2022"
          16 112 16
         128  14 UNUSED
         142   1 "SS2 in ISO 2022"
         143   1 "SS3 in ISO 2022"
         144 112 UNUSED
FUNCTION RE                13
         RS                10
         SPACE             32
         TAB      SEPCHAR   9
         -- Functions for graphic repertoire code extension --
         -- Markup is recognized only in the G0 set. --
         ESC      MSOCHAR  27 -- Escape --
         LS0      MSICHAR  15 -- Locking-shift zero (G0 set) --
         LS1      MSOCHAR  14 -- Locking-shift one (G1 set) --
                              -- LS1R, LS2, LS2R, LS3, and
                                 LS3R are ESC sequences --
         SS2      MSSCHAR 142 -- Single-shift two (G2 set) --
         SS3      MSSCHAR 143 -- Single-shift three (G3 set) --
NAMING   LCNMSTRT ""
         UCNMSTRT ""
         LCNMCHAR "-."     -- Lower-case hyphen, period are --
         UCNMCHAR "-."     -- same as upper-case (45 46).   --
         NAMECASE GENERAL  YES
                  ENTITY   NO
DELIM    GENERAL  SGMLREF
         SHORTREF SGMLREF
NAMES    SGMLREF
QUANTITY SGMLREF
```

Figure 11 — Multicode Basic Concrete Syntax

NOTE — Entity repertoires are necessarily larger and more repetitious than character sets, as they deal in general with higher-level constructs. For example, unique entities have been defined for each accented Latin alphabetic character, while a character set might represent such characters as combinations of letters and diacritical mark characters. These public entity sets should therefore not be construed as requirements for new standard coded character sets.

D.4.1 General Considerations

This sub-sub-clause discusses design criteria applicable to the public entity sets included in this annex.

D.4.1.1 Format of Declarations

The entity sets published here are definitional; the entity text simply consists of the entity name in square brackets, and there is a comment describing the symbol, rather than a (possibly) device-dependent coded representation of it:

```
<!ENTITY frac78 SDATA "[frac78]"--=fraction seven-eighths-->
```

If, as in the above example, the comment contains an equals sign, the description is essentially that given the character in ISO 6937, which also contains a visual depiction of the character.

If, as in the following example, the comment includes a name (of any length) preceded by a solidus, the name is an identifier of a visual depiction of the character in *MathSci, an expansion of mathfile*, 26-Apr-85, published by the American Mathematical Society, 201 Charles St., Providence, RI 02940, U.S.A..

```
<!ENTITY frown  SDATA "[frown ]"--/frown R: down curve-->
```

A comment can include an ISO 6937 description, one or more MathSci identifiers, or none or all of them.

NOTE — In the MathSci document, an identifier is preceded by a reverse solidus, rather than a solidus.

A comment can include a single upper-case letter, followed by a colon, as in the previous example. The letter indicates that in conventional mathematical typesetting, the character is treated differently from an ordinary character, as follows:

Letter	*Treated as:*
A	Relation (arrow)
B	Binary operator
C	Closing delimiter
L	Large operator
N	Relation (negated)
O	Opening delimiter
P	Punctuation
R	Relation

D.4.1.2 Corresponding Display Entity Sets

A system will need to provide corresponding display entity sets for the output devices it supports, in which the entity text is replaced by processing instructions or character sequences that will produce the desired visual depiction. The entity name and descriptive comment would, of course, remain the same. For example, the declaration

```
<!ENTITY frac78 SDATA "7/8"--=fraction seven-eighths-->
```

might be used in a display character entity set for output devices that did not support ISO 6937/2, while

```
<!ENTITY frac78 SDATA "&#223;"--=fraction seven-eighths-->
```

might be used in an entity set for 8-bit coded devices that did. For a text formatter driving a photocomposer, a declaration like the following might be used:

```
<!ENTITY frac78 SDATA "?bf pi;&#14;?pf"--=fraction 7/8-->
```

NOTE — All of the entity declarations use the "SDATA" keyword as a reminder that the entity text could be system-specific character data that might require modification for different output devices and applications.

D.4.1.3 Entity Names

The entity names are derived from the English language. They were chosen for maximum mnemonic value, consistent with the logical and systematic use of abbreviations.

NOTE — Translations may be desired for other languages.

The entity names are case-sensitive, so the case of letters within the name can identify the case of the character, indicate the doubling of a line, or be used for some other convention.

The entity names are limited to six characters in length, and employ only letters and numerals, so they can be used with a variety of concrete syntaxes.

NOTE — If shorter names are desired for frequently used entities, they can be defined in the documents where the frequent use occurs.

Some characters have different semantic connotations in different application contexts. Multiple entities were defined for some of them.

NOTE — If a different name would be more expressive in the context of a particular document, the entity can be redefined within the document.

D.4.1.4 Organization of Entity Sets

The entity sets were organized principally to reflect the structure of the ISO 6937 character sets, or to group large numbers of similar characters together. This organization is not likely to be optimal for most applications, which will normally require a mix of entities from a number of sets. Permission is granted to copy all or part of the public entity sets in this sub-clause in any form for use with conforming SGML systems and applications, provided the ISO copyright notice (including the permission-to-copy text) is included in all copies. In particular, entities can be copied from a number of public sets to form a new set, provided the ISO copyright notice is included in the new set.

NOTE — If the same entity name occurs in more than one public set, and both are needed in a document, an entity with a different name should be declared for one of them within the document.

D.4.2 Alphabetic Characters

This group of character entity sets uses a consistent naming scheme in which the character, or a transliteration of it, is followed by an abbreviation for the accent, and/or a designator of a non-Latin alphabet. The character is capitalized in the entity name when the entity represents its capital form.

D.4.2.1 Latin

This entity set consists of Latin alphabetic characters used in Western European languages, other than those in ***UC Letter*** and ***LC Letter***.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOlat1 PUBLIC
       "ISO 8879-1986//ENTITIES Added Latin 1//EN">
     %ISOlat1;
-->
<!ENTITY aacute SDATA "[aacute]"--=small a, acute accent-->
<!ENTITY Aacute SDATA "[Aacute]"--=capital A, acute accent-->
<!ENTITY acirc  SDATA "[acirc ]"--=small a, circumflex accent-->
<!ENTITY Acirc  SDATA "[Acirc ]"--=capital A, circumflex accent-->
<!ENTITY agrave SDATA "[agrave]"--=small a, grave accent-->
<!ENTITY Agrave SDATA "[Agrave]"--=capital A, grave accent-->
<!ENTITY aring  SDATA "[aring ]"--=small a, ring-->
<!ENTITY Aring  SDATA "[Aring ]"--=capital A, ring-->
<!ENTITY atilde SDATA "[atilde]"--=small a, tilde-->
<!ENTITY Atilde SDATA "[Atilde]"--=capital A, tilde-->
<!ENTITY auml   SDATA "[auml  ]"--=small a, dieresis or umlaut mark-->
<!ENTITY Auml   SDATA "[Auml  ]"--=capital A, dieresis or umlaut mark-->
<!ENTITY aelig  SDATA "[aelig ]"--=small ae diphthong (ligature)-->
<!ENTITY AElig  SDATA "[AElig ]"--=capital AE diphthong (ligature)-->
<!ENTITY ccedil SDATA "[ccedil]"--=small c, cedilla-->
<!ENTITY Ccedil SDATA "[Ccedil]"--=capital C, cedilla-->
<!ENTITY eth    SDATA "[eth   ]"--=small eth, Icelandic-->
```

```
<!ENTITY ETH    SDATA "[ETH   ]"--=capital Eth, Icelandic-->
<!ENTITY eacute SDATA "[eacute]"--=small e, acute accent-->
<!ENTITY Eacute SDATA "[Eacute]"--=capital E, acute accent-->
<!ENTITY ecirc  SDATA "[ecirc ]"--=small e, circumflex accent-->
<!ENTITY Ecirc  SDATA "[Ecirc ]"--=capital E, circumflex accent-->
<!ENTITY egrave SDATA "[egrave]"--=small e, grave accent-->
<!ENTITY Egrave SDATA "[Egrave]"--=capital E, grave accent-->
<!ENTITY euml   SDATA "[euml  ]"--=small e, dieresis or umlaut mark-->
<!ENTITY Euml   SDATA "[Euml  ]"--=capital E, dieresis or umlaut mark-->
<!ENTITY iacute SDATA "[iacute]"--=small i, acute accent-->
<!ENTITY Iacute SDATA "[Iacute]"--=capital I, acute accent-->
<!ENTITY icirc  SDATA "[icirc ]"--=small i, circumflex accent-->
<!ENTITY Icirc  SDATA "[Icirc ]"--=capital I, circumflex accent-->
<!ENTITY igrave SDATA "[igrave]"--=small i, grave accent-->
<!ENTITY Igrave SDATA "[Igrave]"--=capital I, grave accent-->
<!ENTITY iuml   SDATA "[iuml  ]"--=small i, dieresis or umlaut mark-->
<!ENTITY Iuml   SDATA "[Iuml  ]"--=capital I, dieresis or umlaut mark-->
<!ENTITY ntilde SDATA "[ntilde]"--=small n, tilde-->
<!ENTITY Ntilde SDATA "[Ntilde]"--=capital N, tilde-->
<!ENTITY oacute SDATA "[oacute]"--=small o, acute accent-->
<!ENTITY Oacute SDATA "[Oacute]"--=capital O, acute accent-->
<!ENTITY ocirc  SDATA "[ocirc ]"--=small o, circumflex accent-->
<!ENTITY Ocirc  SDATA "[Ocirc ]"--=capital O, circumflex accent-->
<!ENTITY ograve SDATA "[ograve]"--=small o, grave accent-->
<!ENTITY Ograve SDATA "[Ograve]"--=capital O, grave accent-->
<!ENTITY oslash SDATA "[oslash]"--=small o, slash-->
<!ENTITY Oslash SDATA "[Oslash]"--=capital O, slash-->
<!ENTITY otilde SDATA "[otilde]"--=small o, tilde-->
<!ENTITY Otilde SDATA "[Otilde]"--=capital O, tilde-->
<!ENTITY ouml   SDATA "[ouml  ]"--=small o, dieresis or umlaut mark-->
<!ENTITY Ouml   SDATA "[Ouml  ]"--=capital O, dieresis or umlaut mark-->
<!ENTITY szlig  SDATA "[szlig ]"--=small sharp s, German (sz ligature)-->
<!ENTITY thorn  SDATA "[thorn ]"--=small thorn, Icelandic-->
<!ENTITY THORN  SDATA "[THORN ]"--=capital THORN, Icelandic-->
<!ENTITY uacute SDATA "[uacute]"--=small u, acute accent-->
<!ENTITY Uacute SDATA "[Uacute]"--=capital U, acute accent-->
<!ENTITY ucirc  SDATA "[ucirc ]"--=small u, circumflex accent-->
<!ENTITY Ucirc  SDATA "[Ucirc ]"--=capital U, circumflex accent-->
<!ENTITY ugrave SDATA "[ugrave]"--=small u, grave accent-->
<!ENTITY Ugrave SDATA "[Ugrave]"--=capital U, grave accent-->
<!ENTITY uuml   SDATA "[uuml  ]"--=small u, dieresis or umlaut mark-->
<!ENTITY Uuml   SDATA "[Uuml  ]"--=capital U, dieresis or umlaut mark-->
<!ENTITY yacute SDATA "[yacute]"--=small y, acute accent-->
<!ENTITY Yacute SDATA "[Yacute]"--=capital Y, acute accent-->
<!ENTITY yuml   SDATA "[yuml  ]"--=small y, dieresis or umlaut mark-->
```

This entity set contains additional Latin alphabetic characters.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOlat2 PUBLIC
       "ISO 8879-1986//ENTITIES Added Latin 2//EN">
     %ISOlat2;
-->
<!ENTITY abreve SDATA "[abreve]"--=small a, breve-->
<!ENTITY Abreve SDATA "[Abreve]"--=capital A, breve-->
<!ENTITY amacr  SDATA "[amacr ]"--=small a, macron-->
<!ENTITY Amacr  SDATA "[Amacr ]"--=capital A, macron-->
```

```
<!ENTITY aogon  SDATA "[aogon ]"--=small a, ogonek-->
<!ENTITY Aogon  SDATA "[Aogon ]"--=capital A, ogonek-->
<!ENTITY cacute SDATA "[cacute]"--=small c, acute accent-->
<!ENTITY Cacute SDATA "[Cacute]"--=capital C, acute accent-->
<!ENTITY ccaron SDATA "[ccaron]"--=small c, caron-->
<!ENTITY Ccaron SDATA "[Ccaron]"--=capital C, caron-->
<!ENTITY ccirc  SDATA "[ccirc ]"--=small c, circumflex accent-->
<!ENTITY Ccirc  SDATA "[Ccirc ]"--=capital C, circumflex accent-->
<!ENTITY cdot   SDATA "[cdot  ]"--=small c, dot above-->
<!ENTITY Cdot   SDATA "[Cdot  ]"--=capital C, dot above-->
<!ENTITY dcaron SDATA "[dcaron]"--=small d, caron-->
<!ENTITY Dcaron SDATA "[Dcaron]"--=capital D, caron-->
<!ENTITY dstrok SDATA "[dstrok]"--=small d, stroke-->
<!ENTITY Dstrok SDATA "[Dstrok]"--=capital D, stroke-->
<!ENTITY ecaron SDATA "[ecaron]"--=small e, caron-->
<!ENTITY Ecaron SDATA "[Ecaron]"--=capital E, caron-->
<!ENTITY edot   SDATA "[edot  ]"--=small e, dot above-->
<!ENTITY Edot   SDATA "[Edot  ]"--=capital E, dot above-->
<!ENTITY emacr  SDATA "[emacr ]"--=small e, macron-->
<!ENTITY Emacr  SDATA "[Emacr ]"--=capital E, macron-->
<!ENTITY eogon  SDATA "[eogon ]"--=small e, ogonek-->
<!ENTITY Eogon  SDATA "[Eogon ]"--=capital E, ogonek-->
<!ENTITY gacute SDATA "[gacute]"--=small g, acute accent-->
<!ENTITY gbreve SDATA "[gbreve]"--=small g, breve-->
<!ENTITY Gbreve SDATA "[Gbreve]"--=capital G, breve-->
<!ENTITY Gcedil SDATA "[Gcedil]"--=capital G, cedilla-->
<!ENTITY gcirc  SDATA "[gcirc ]"--=small g, circumflex accent-->
<!ENTITY Gcirc  SDATA "[Gcirc ]"--=capital G, circumflex accent-->
<!ENTITY gdot   SDATA "[gdot  ]"--=small g, dot above-->
<!ENTITY Gdot   SDATA "[Gdot  ]"--=capital G, dot above-->
<!ENTITY hcirc  SDATA "[hcirc ]"--=small h, circumflex accent-->
<!ENTITY Hcirc  SDATA "[Hcirc ]"--=capital H, circumflex accent-->
<!ENTITY hstrok SDATA "[hstrok]"--=small h, stroke-->
<!ENTITY Hstrok SDATA "[Hstrok]"--=capital H, stroke-->
<!ENTITY Idot   SDATA "[Idot  ]"--=capital I, dot above-->
<!ENTITY Imacr  SDATA "[Imacr ]"--=capital I, macron-->
<!ENTITY imacr  SDATA "[imacr ]"--=small i, macron-->
<!ENTITY ijlig  SDATA "[ijlig ]"--=small ij ligature-->
<!ENTITY IJlig  SDATA "[IJlig ]"--=capital IJ ligature-->
<!ENTITY inodot SDATA "[inodot]"--=small i without dot-->
<!ENTITY iogon  SDATA "[iogon ]"--=small i, ogonek-->
<!ENTITY Iogon  SDATA "[Iogon ]"--=capital I, ogonek-->
<!ENTITY itilde SDATA "[itilde]"--=small i, tilde-->
<!ENTITY Itilde SDATA "[Itilde]"--=capital I, tilde-->
<!ENTITY jcirc  SDATA "[jcirc ]"--=small j, circumflex accent-->
<!ENTITY Jcirc  SDATA "[Jcirc ]"--=capital J, circumflex accent-->
<!ENTITY kcedil SDATA "[kcedil]"--=small k, cedilla-->
<!ENTITY Kcedil SDATA "[Kcedil]"--=capital K, cedilla-->
<!ENTITY kgreen SDATA "[kgreen]"--=small k, Greenlandic-->
<!ENTITY lacute SDATA "[lacute]"--=small l, acute accent-->
<!ENTITY Lacute SDATA "[Lacute]"--=capital L, acute accent-->
<!ENTITY lcaron SDATA "[lcaron]"--=small l, caron-->
<!ENTITY Lcaron SDATA "[Lcaron]"--=capital L, caron-->
<!ENTITY lcedil SDATA "[lcedil]"--=small l, cedilla-->
<!ENTITY Lcedil SDATA "[Lcedil]"--=capital L, cedilla-->
<!ENTITY lmidot SDATA "[lmidot]"--=small l, middle dot-->
<!ENTITY Lmidot SDATA "[Lmidot]"--=capital L, middle dot-->
<!ENTITY lstrok SDATA "[lstrok]"--=small l, stroke-->
<!ENTITY Lstrok SDATA "[Lstrok]"--=capital L, stroke-->
<!ENTITY nacute SDATA "[nacute]"--=small n, acute accent-->
<!ENTITY Nacute SDATA "[Nacute]"--=capital N, acute accent-->
<!ENTITY eng    SDATA "[eng   ]"--=small eng, Lapp-->
```

```
<!ENTITY ENG    SDATA "[ENG   ]"--=capital ENG, Lapp-->
<!ENTITY napos  SDATA "[napos ]"--=small n, apostrophe-->
<!ENTITY ncaron SDATA "[ncaron]"--=small n, caron-->
<!ENTITY Ncaron SDATA "[Ncaron]"--=capital N, caron-->
<!ENTITY ncedil SDATA "[ncedil]"--=small n, cedilla-->
<!ENTITY Ncedil SDATA "[Ncedil]"--=capital N, cedilla-->
<!ENTITY odblac SDATA "[odblac]"--=small o, double acute accent-->
<!ENTITY Odblac SDATA "[Odblac]"--=capital O, double acute accent-->
<!ENTITY Omacr  SDATA "[Omacr ]"--=capital O, macron-->
<!ENTITY omacr  SDATA "[omacr ]"--=small o, macron-->
<!ENTITY oelig  SDATA "[oelig ]"--=small oe ligature-->
<!ENTITY OElig  SDATA "[OElig ]"--=capital OE ligature-->
<!ENTITY racute SDATA "[racute]"--=small r, acute accent-->
<!ENTITY Racute SDATA "[Racute]"--=capital R, acute accent-->
<!ENTITY rcaron SDATA "[rcaron]"--=small r, caron-->
<!ENTITY Rcaron SDATA "[Rcaron]"--=capital R, caron-->
<!ENTITY rcedil SDATA "[rcedil]"--=small r, cedilla-->
<!ENTITY Rcedil SDATA "[Rcedil]"--=capital R, cedilla-->
<!ENTITY sacute SDATA "[sacute]"--=small s, acute accent-->
<!ENTITY Sacute SDATA "[Sacute]"--=capital S, acute accent-->
<!ENTITY scaron SDATA "[scaron]"--=small s, caron-->
<!ENTITY Scaron SDATA "[Scaron]"--=capital S, caron-->
<!ENTITY scedil SDATA "[scedil]"--=small s, cedilla-->
<!ENTITY Scedil SDATA "[Scedil]"--=capital S, cedilla-->
<!ENTITY scirc  SDATA "[scirc ]"--=small s, circumflex accent-->
<!ENTITY Scirc  SDATA "[Scirc ]"--=capital S, circumflex accent-->
<!ENTITY tcaron SDATA "[tcaron]"--=small t, caron-->
<!ENTITY Tcaron SDATA "[Tcaron]"--=capital T, caron-->
<!ENTITY tcedil SDATA "[tcedil]"--=small t, cedilla-->
<!ENTITY Tcedil SDATA "[Tcedil]"--=capital T, cedilla-->
<!ENTITY tstrok SDATA "[tstrok]"--=small t, stroke-->
<!ENTITY Tstrok SDATA "[Tstrok]"--=capital T, stroke-->
<!ENTITY ubreve SDATA "[ubreve]"--=small u, breve-->
<!ENTITY Ubreve SDATA "[Ubreve]"--=capital U, breve-->
<!ENTITY udblac SDATA "[udblac]"--=small u, double acute accent-->
<!ENTITY Udblac SDATA "[Udblac]"--=capital U, double acute accent-->
<!ENTITY umacr  SDATA "[umacr ]"--=small u, macron-->
<!ENTITY Umacr  SDATA "[Umacr ]"--=capital U, macron-->
<!ENTITY uogon  SDATA "[uogon ]"--=small u, ogonek-->
<!ENTITY Uogon  SDATA "[Uogon ]"--=capital U, ogonek-->
<!ENTITY uring  SDATA "[uring ]"--=small u, ring-->
<!ENTITY Uring  SDATA "[Uring ]"--=capital U, ring-->
<!ENTITY utilde SDATA "[utilde]"--=small u, tilde-->
<!ENTITY Utilde SDATA "[Utilde]"--=capital U, tilde-->
<!ENTITY wcirc  SDATA "[wcirc ]"--=small w, circumflex accent-->
<!ENTITY Wcirc  SDATA "[Wcirc ]"--=capital W, circumflex accent-->
<!ENTITY ycirc  SDATA "[ycirc ]"--=small y, circumflex accent-->
<!ENTITY Ycirc  SDATA "[Ycirc ]"--=capital Y, circumflex accent-->
<!ENTITY Yuml   SDATA "[Yuml  ]"--=capital Y, dieresis or umlaut mark-->
<!ENTITY zacute SDATA "[zacute]"--=small z, acute accent-->
<!ENTITY Zacute SDATA "[Zacute]"--=capital Z, acute accent-->
<!ENTITY zcaron SDATA "[zcaron]"--=small z, caron-->
<!ENTITY Zcaron SDATA "[Zcaron]"--=capital Z, caron-->
<!ENTITY zdot   SDATA "[zdot  ]"--=small z, dot above-->
<!ENTITY Zdot   SDATA "[Zdot  ]"--=capital Z, dot above-->
```

D.4.2.2 Greek Alphabetic Characters

This entity set consists of the letters of the Greek alphabet. The entity names reflect their intended use as language characters, rather than as symbols in formulas. (Greek character entities for technical use are defined below.)

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOgrk1 PUBLIC
       "ISO 8879-1986//ENTITIES Greek Letters//EN">
     %ISOgrk1;
-->
<!ENTITY agr    SDATA "[agr   ]"--=small alpha, Greek-->
<!ENTITY Agr    SDATA "[Agr   ]"--=capital Alpha, Greek-->
<!ENTITY bgr    SDATA "[bgr   ]"--=small beta, Greek-->
<!ENTITY Bgr    SDATA "[Bgr   ]"--=capital Beta, Greek-->
<!ENTITY ggr    SDATA "[ggr   ]"--=small gamma, Greek-->
<!ENTITY Ggr    SDATA "[Ggr   ]"--=capital Gamma, Greek-->
<!ENTITY dgr    SDATA "[dgr   ]"--=small delta, Greek-->
<!ENTITY Dgr    SDATA "[Dgr   ]"--=capital Delta, Greek-->
<!ENTITY egr    SDATA "[egr   ]"--=small epsilon, Greek-->
<!ENTITY Egr    SDATA "[Egr   ]"--=capital Epsilon, Greek-->
<!ENTITY zgr    SDATA "[zgr   ]"--=small zeta, Greek-->
<!ENTITY Zgr    SDATA "[Zgr   ]"--=capital Zeta, Greek-->
<!ENTITY eegr   SDATA "[eegr  ]"--=small eta, Greek-->
<!ENTITY EEgr   SDATA "[EEgr  ]"--=capital Eta, Greek-->
<!ENTITY thgr   SDATA "[thgr  ]"--=small theta, Greek-->
<!ENTITY THgr   SDATA "[THgr  ]"--=capital Theta, Greek-->
<!ENTITY igr    SDATA "[igr   ]"--=small iota, Greek-->
<!ENTITY Igr    SDATA "[Igr   ]"--=capital Iota, Greek-->
<!ENTITY kgr    SDATA "[kgr   ]"--=small kappa, Greek-->
<!ENTITY Kgr    SDATA "[Kgr   ]"--=capital Kappa, Greek-->
<!ENTITY lgr    SDATA "[lgr   ]"--=small lambda, Greek-->
<!ENTITY Lgr    SDATA "[Lgr   ]"--=capital Lambda, Greek-->
<!ENTITY mgr    SDATA "[mgr   ]"--=small mu, Greek-->
<!ENTITY Mgr    SDATA "[Mgr   ]"--=capital Mu, Greek-->
<!ENTITY ngr    SDATA "[ngr   ]"--=small nu, Greek-->
<!ENTITY Ngr    SDATA "[Ngr   ]"--=capital Nu, Greek-->
<!ENTITY xgr    SDATA "[xgr   ]"--=small xi, Greek-->
<!ENTITY Xgr    SDATA "[Xgr   ]"--=capital Xi, Greek-->
<!ENTITY ogr    SDATA "[ogr   ]"--=small omicron, Greek-->
<!ENTITY Ogr    SDATA "[Ogr   ]"--=capital Omicron, Greek-->
<!ENTITY pgr    SDATA "[pgr   ]"--=small pi, Greek-->
<!ENTITY Pgr    SDATA "[Pgr   ]"--=capital Pi, Greek-->
<!ENTITY rgr    SDATA "[rgr   ]"--=small rho, Greek-->
<!ENTITY Rgr    SDATA "[Rgr   ]"--=capital Rho, Greek-->
<!ENTITY sgr    SDATA "[sgr   ]"--=small sigma, Greek-->
<!ENTITY Sgr    SDATA "[Sgr   ]"--=capital Sigma, Greek-->
<!ENTITY sfgr   SDATA "[sfgr  ]"--=final small sigma, Greek-->
<!ENTITY tgr    SDATA "[tgr   ]"--=small tau, Greek-->
<!ENTITY Tgr    SDATA "[Tgr   ]"--=capital Tau, Greek-->
<!ENTITY ugr    SDATA "[ugr   ]"--=small upsilon, Greek-->
<!ENTITY Ugr    SDATA "[Ugr   ]"--=capital Upsilon, Greek-->
<!ENTITY phgr   SDATA "[phgr  ]"--=small phi, Greek-->
<!ENTITY PHgr   SDATA "[PHgr  ]"--=capital Phi, Greek-->
<!ENTITY khgr   SDATA "[khgr  ]"--=small chi, Greek-->
<!ENTITY KHgr   SDATA "[KHgr  ]"--=capital Chi, Greek-->
<!ENTITY psgr   SDATA "[psgr  ]"--=small psi, Greek-->
<!ENTITY PSgr   SDATA "[PSgr  ]"--=capital Psi, Greek-->
<!ENTITY ohgr   SDATA "[ohgr  ]"--=small omega, Greek-->
<!ENTITY OHgr   SDATA "[OHgr  ]"--=capital Omega, Greek-->
```

This entity set contains additional characters needed for Monotoniko Greek.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOgrk2 PUBLIC
       "ISO 8879-1986//ENTITIES Monotoniko Greek//EN">
     %ISOgrk2;
-->
<!ENTITY aacgr  SDATA "[aacgr ]"--=small alpha, accent, Greek-->
<!ENTITY Aacgr  SDATA "[Aacgr ]"--=capital Alpha, accent, Greek-->
<!ENTITY eacgr  SDATA "[eacgr ]"--=small epsilon, accent, Greek-->
<!ENTITY Eacgr  SDATA "[Eacgr ]"--=capital Epsilon, accent, Greek-->
<!ENTITY eeacgr SDATA "[eeacgr]"--=small eta, accent, Greek-->
<!ENTITY EEacgr SDATA "[EEacgr]"--=capital Eta, accent, Greek-->
<!ENTITY idigr  SDATA "[idigr ]"--=small iota, dieresis, Greek-->
<!ENTITY Idigr  SDATA "[Idigr ]"--=capital Iota, dieresis, Greek-->
<!ENTITY iacgr  SDATA "[iacgr ]"--=small iota, accent, Greek-->
<!ENTITY Iacgr  SDATA "[Iacgr ]"--=capital Iota, accent, Greek-->
<!ENTITY idiagr SDATA "[idiagr]"--=small iota, dieresis, accent, Greek-->
<!ENTITY oacgr  SDATA "[oacgr ]"--=small omicron, accent, Greek-->
<!ENTITY Oacgr  SDATA "[Oacgr ]"--=capital Omicron, accent, Greek-->
<!ENTITY udigr  SDATA "[udigr ]"--=small upsilon, dieresis, Greek-->
<!ENTITY Udigr  SDATA "[Udigr ]"--=capital Upsilon, dieresis, Greek-->
<!ENTITY uacgr  SDATA "[uacgr ]"--=small upsilon, accent, Greek-->
<!ENTITY Uacgr  SDATA "[Uacgr ]"--=capital Upsilon, accent, Greek-->
<!ENTITY udiagr SDATA "[udiagr]"--=small upsilon, dieresis, accent, Greek-->
<!ENTITY ohacgr SDATA "[ohacgr]"--=small omega, accent, Greek-->
<!ENTITY OHacgr SDATA "[OHacgr]"--=capital Omega, accent, Greek-->
```

D.4.2.3 Cyrillic Alphabetic Characters

This entity set consists of Cyrillic characters used in the Russian language.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOcyr1 PUBLIC
       "ISO 8879-1986//ENTITIES Russian Cyrillic//EN">
     %ISOcyr1;
-->
<!ENTITY acy    SDATA "[acy   ]"--=small a, Cyrillic-->
<!ENTITY Acy    SDATA "[Acy   ]"--=capital A, Cyrillic-->
<!ENTITY bcy    SDATA "[bcy   ]"--=small be, Cyrillic-->
<!ENTITY Bcy    SDATA "[Bcy   ]"--=capital BE, Cyrillic-->
<!ENTITY vcy    SDATA "[vcy   ]"--=small ve, Cyrillic-->
<!ENTITY Vcy    SDATA "[Vcy   ]"--=capital VE, Cyrillic-->
<!ENTITY gcy    SDATA "[gcy   ]"--=small ghe, Cyrillic-->
<!ENTITY Gcy    SDATA "[Gcy   ]"--=capital GHE, Cyrillic-->
<!ENTITY dcy    SDATA "[dcy   ]"--=small de, Cyrillic-->
<!ENTITY Dcy    SDATA "[Dcy   ]"--=capital DE, Cyrillic-->
<!ENTITY iecy   SDATA "[iecy  ]"--=small ie, Cyrillic-->
<!ENTITY IEcy   SDATA "[IEcy  ]"--=capital IE, Cyrillic-->
<!ENTITY iocy   SDATA "[iocy  ]"--=small io, Russian-->
<!ENTITY IOcy   SDATA "[IOcy  ]"--=capital IO, Russian-->
<!ENTITY zhcy   SDATA "[zhcy  ]"--=small zhe, Cyrillic-->
<!ENTITY ZHcy   SDATA "[ZHcy  ]"--=capital ZHE, Cyrillic-->
<!ENTITY zcy    SDATA "[zcy   ]"--=small ze, Cyrillic-->
```

```
<!ENTITY Zcy    SDATA "[Zcy   ]"--=capital ZE, Cyrillic-->
<!ENTITY icy    SDATA "[icy   ]"--=small i, Cyrillic-->
<!ENTITY Icy    SDATA "[Icy   ]"--=capital I, Cyrillic-->
<!ENTITY jcy    SDATA "[jcy   ]"--=small short i, Cyrillic-->
<!ENTITY Jcy    SDATA "[Jcy   ]"--=capital short I, Cyrillic-->
<!ENTITY kcy    SDATA "[kcy   ]"--=small ka, Cyrillic-->
<!ENTITY Kcy    SDATA "[Kcy   ]"--=capital KA, Cyrillic-->
<!ENTITY lcy    SDATA "[lcy   ]"--=small el, Cyrillic-->
<!ENTITY Lcy    SDATA "[Lcy   ]"--=capital EL, Cyrillic-->
<!ENTITY mcy    SDATA "[mcy   ]"--=small em, Cyrillic-->
<!ENTITY Mcy    SDATA "[Mcy   ]"--=capital EM, Cyrillic-->
<!ENTITY ncy    SDATA "[ncy   ]"--=small en, Cyrillic-->
<!ENTITY Ncy    SDATA "[Ncy   ]"--=capital EN, Cyrillic-->
<!ENTITY ocy    SDATA "[ocy   ]"--=small o, Cyrillic-->
<!ENTITY Ocy    SDATA "[Ocy   ]"--=capital O, Cyrillic-->
<!ENTITY pcy    SDATA "[pcy   ]"--=small pe, Cyrillic-->
<!ENTITY Pcy    SDATA "[Pcy   ]"--=capital PE, Cyrillic-->
<!ENTITY rcy    SDATA "[rcy   ]"--=small er, Cyrillic-->
<!ENTITY Rcy    SDATA "[Rcy   ]"--=capital ER, Cyrillic-->
<!ENTITY scy    SDATA "[scy   ]"--=small es, Cyrillic-->
<!ENTITY Scy    SDATA "[Scy   ]"--=capital ES, Cyrillic-->
<!ENTITY tcy    SDATA "[tcy   ]"--=small te, Cyrillic-->
<!ENTITY Tcy    SDATA "[Tcy   ]"--=capital TE, Cyrillic-->
<!ENTITY ucy    SDATA "[ucy   ]"--=small u, Cyrillic-->
<!ENTITY Ucy    SDATA "[Ucy   ]"--=capital U, Cyrillic-->
<!ENTITY fcy    SDATA "[fcy   ]"--=small ef, Cyrillic-->
<!ENTITY Fcy    SDATA "[Fcy   ]"--=capital EF, Cyrillic-->
<!ENTITY khcy   SDATA "[khcy  ]"--=small ha, Cyrillic-->
<!ENTITY KHcy   SDATA "[KHcy  ]"--=capital HA, Cyrillic-->
<!ENTITY tscy   SDATA "[tscy  ]"--=small tse, Cyrillic-->
<!ENTITY TScy   SDATA "[TScy  ]"--=capital TSE, Cyrillic-->
<!ENTITY chcy   SDATA "[chcy  ]"--=small che, Cyrillic-->
<!ENTITY CHcy   SDATA "[CHcy  ]"--=capital CHE, Cyrillic-->
<!ENTITY shcy   SDATA "[shcy  ]"--=small sha, Cyrillic-->
<!ENTITY SHcy   SDATA "[SHcy  ]"--=capital SHA, Cyrillic-->
<!ENTITY shchcy SDATA "[shchcy]"--=small shcha, Cyrillic-->
<!ENTITY SHCHcy SDATA "[SHCHcy]"--=capital SHCHA, Cyrillic-->
<!ENTITY hardcy SDATA "[hardcy]"--=small hard sign, Cyrillic-->
<!ENTITY HARDcy SDATA "[HARDcy]"--=capital HARD sign, Cyrillic-->
<!ENTITY ycy    SDATA "[ycy   ]"--=small yeru, Cyrillic-->
<!ENTITY Ycy    SDATA "[Ycy   ]"--=capital YERU, Cyrillic-->
<!ENTITY softcy SDATA "[softcy]"--=small soft sign, Cyrillic-->
<!ENTITY SOFTcy SDATA "[SOFTcy]"--=capital SOFT sign, Cyrillic-->
<!ENTITY ecy    SDATA "[ecy   ]"--=small e, Cyrillic-->
<!ENTITY Ecy    SDATA "[Ecy   ]"--=capital E, Cyrillic-->
<!ENTITY yucy   SDATA "[yucy  ]"--=small yu, Cyrillic-->
<!ENTITY YUcy   SDATA "[YUcy  ]"--=capital YU, Cyrillic-->
<!ENTITY yacy   SDATA "[yacy  ]"--=small ya, Cyrillic-->
<!ENTITY YAcy   SDATA "[YAcy  ]"--=capital YA, Cyrillic-->
<!ENTITY numero SDATA "[numero]"--=numero sign-->
```

This entity set consists of Cyrillic characters that are not used in the Russian language.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOcyr2 PUBLIC
       "ISO 8879-1986//ENTITIES Non-Russian Cyrillic//EN">
     %ISOcyr2;
```

```
-->
<!ENTITY djcy   SDATA "[djcy  ]"--=small dje, Serbian-->
<!ENTITY DJcy   SDATA "[DJcy  ]"--=capital DJE, Serbian-->
<!ENTITY gjcy   SDATA "[gjcy  ]"--=small gje, Macedonian-->
<!ENTITY GJcy   SDATA "[GJcy  ]"--=capital GJE Macedonian-->
<!ENTITY jukcy  SDATA "[jukcy ]"--=small je, Ukrainian-->
<!ENTITY Jukcy  SDATA "[Jukcy ]"--=capital JE, Ukrainian-->
<!ENTITY dscy   SDATA "[dscy  ]"--=small dse, Macedonian-->
<!ENTITY DScy   SDATA "[DScy  ]"--=capital DSE, Macedonian-->
<!ENTITY iukcy  SDATA "[iukcy ]"--=small i, Ukrainian-->
<!ENTITY Iukcy  SDATA "[Iukcy ]"--=capital I, Ukrainian-->
<!ENTITY yicy   SDATA "[yicy  ]"--=small yi, Ukrainian-->
<!ENTITY YIcy   SDATA "[YIcy  ]"--=capital YI, Ukrainian-->
<!ENTITY jsercy SDATA "[jsercy]"--=small je, Serbian-->
<!ENTITY Jsercy SDATA "[Jsercy]"--=capital JE, Serbian-->
<!ENTITY ljcy   SDATA "[ljcy  ]"--=small lje, Serbian-->
<!ENTITY LJcy   SDATA "[LJcy  ]"--=capital LJE, Serbian-->
<!ENTITY njcy   SDATA "[njcy  ]"--=small nje, Serbian-->
<!ENTITY NJcy   SDATA "[NJcy  ]"--=capital NJE, Serbian-->
<!ENTITY tshcy  SDATA "[tshcy ]"--=small tshe, Serbian-->
<!ENTITY TSHcy  SDATA "[TSHcy ]"--=capital TSHE, Serbian-->
<!ENTITY kjcy   SDATA "[kjcy  ]"--=small kje Macedonian-->
<!ENTITY KJcy   SDATA "[KJcy  ]"--=capital KJE, Macedonian-->
<!ENTITY ubrcy  SDATA "[ubrcy ]"--=small u, Byelorussian-->
<!ENTITY Ubrcy  SDATA "[Ubrcy ]"--=capital U, Byelorussian-->
<!ENTITY dzcy   SDATA "[dzcy  ]"--=small dze, Serbian-->
<!ENTITY DZcy   SDATA "[DZcy  ]"--=capital dze, Serbian-->
```

D.4.3 General Use

D.4.3.1 Numeric and Special Graphic Characters

This set includes, among others, minimum data characters and reference concrete syntax markup characters. Such characters are normally directly keyable, but when they are assigned to delimiter roles, an entity reference may be needed to enter them as data.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOnum PUBLIC
       "ISO 8879-1986//ENTITIES Numeric and Special Graphic//EN">
     %ISOnum;
-->
<!ENTITY half   SDATA "[half  ]"--=fraction one-half-->
<!ENTITY frac12 SDATA "[frac12]"--=fraction one-half-->
<!ENTITY frac14 SDATA "[frac14]"--=fraction one-quarter-->
<!ENTITY frac34 SDATA "[frac34]"--=fraction three-quarters-->
<!ENTITY frac18 SDATA "[frac18]"--=fraction one-eighth-->
<!ENTITY frac38 SDATA "[frac38]"--=fraction three-eighths-->
<!ENTITY frac58 SDATA "[frac58]"--=fraction five-eighths-->
<!ENTITY frac78 SDATA "[frac78]"--=fraction seven-eighths-->

<!ENTITY sup1   SDATA "[sup1  ]"--=superscript one-->
<!ENTITY sup2   SDATA "[sup2  ]"--=superscript two-->
<!ENTITY sup3   SDATA "[sup3  ]"--=superscript three-->

<!ENTITY plus   SDATA "[plus  ]"--=plus sign B:-- >
<!ENTITY plusmn SDATA "[plusmn]"--/pm B: =plus-or-minus sign-->
<!ENTITY lt     SDATA "[lt    ]"--=less-than sign R:-->
```

```
<!ENTITY equals SDATA "[equals]"--=equals sign R:-->
<!ENTITY gt     SDATA "[gt    ]"--=greater-than sign R:-->
<!ENTITY divide SDATA "[divide]"--/div B: =divide sign-->
<!ENTITY times  SDATA "[times ]"--/times B: =multiply sign-->

<!ENTITY curren SDATA "[curren]"--=general currency sign-->
<!ENTITY pound  SDATA "[pound ]"--=pound sign-->
<!ENTITY dollar SDATA "[dollar]"--=dollar sign-->
<!ENTITY cent   SDATA "[cent  ]"--=cent sign-->
<!ENTITY yen    SDATA "[yen   ]"--/yen =yen sign-->

<!ENTITY num    SDATA "[num   ]"--=number sign-->
<!ENTITY percnt SDATA "[percnt]"--=percent sign-->
<!ENTITY amp    SDATA "[amp   ]"--=ampersand-->
<!ENTITY ast    SDATA "[ast   ]"--/ast B: =asterisk-->
<!ENTITY commat SDATA "[commat]"--=commercial at-->
<!ENTITY lsqb   SDATA "[lsqb  ]"--/lbrack O: =left square bracket-->
<!ENTITY bsol   SDATA "[bsol  ]"--/backslash =reverse solidus-->
<!ENTITY rsqb   SDATA "[rsqb  ]"--/rbrack C: =right square bracket-->
<!ENTITY lcub   SDATA "[lcub  ]"--/lbrace O: =left curly bracket-->
<!ENTITY horbar SDATA "[horbar]"--=horizontal bar-->
<!ENTITY verbar SDATA "[verbar]"--/vert =vertical bar-->
<!ENTITY rcub   SDATA "[rcub  ]"--/rbrace C: =right curly bracket-->
<!ENTITY micro  SDATA "[micro ]"--=micro sign-->
<!ENTITY ohm    SDATA "[ohm   ]"--=ohm sign-->
<!ENTITY deg    SDATA "[deg   ]"--=degree sign-->
<!ENTITY ordm   SDATA "[ordm  ]"--=ordinal indicator, masculine-->
<!ENTITY ordf   SDATA "[ordf  ]"--=ordinal indicator, feminine-->
<!ENTITY sect   SDATA "[sect  ]"--=section sign-->
<!ENTITY para   SDATA "[para  ]"--=pilcrow (paragraph sign)-->
<!ENTITY middot SDATA "[middot]"--/centerdot B: =middle dot-->
<!ENTITY larr   SDATA "[larr  ]"--/leftarrow /gets A: =leftward arrow-->
<!ENTITY rarr   SDATA "[rarr  ]"--/rightarrow /to A: =rightward arrow-->
<!ENTITY uarr   SDATA "[uarr  ]"--/uparrow A: =upward arrow-->
<!ENTITY darr   SDATA "[darr  ]"--/downarrow A: =downward arrow-->
<!ENTITY copy   SDATA "[copy  ]"--=copyright sign-->
<!ENTITY reg    SDATA "[reg   ]"--/circledR =registered sign-->
<!ENTITY trade  SDATA "[trade ]"--=trade mark sign-->
<!ENTITY brvbar SDATA "[brvbar]"--=broken (vertical) bar-->
<!ENTITY not    SDATA "[not   ]"--/neg /lnot =not sign-->
<!ENTITY sung   SDATA "[sung  ]"--=music note (sung text sign)-->

<!ENTITY excl   SDATA "[excl  ]"--=exclamation mark-->
<!ENTITY iexcl  SDATA "[iexcl ]"--=inverted exclamation mark-->
<!ENTITY quot   SDATA "[quot  ]"--=quotation mark-->
<!ENTITY apos   SDATA "[apos  ]"--=apostrophe-->
<!ENTITY lpar   SDATA "[lpar  ]"--O: =left parenthesis-->
<!ENTITY rpar   SDATA "[rpar  ]"--C: =right parenthesis-->
<!ENTITY comma  SDATA "[comma ]"--P: =comma-->
<!ENTITY lowbar SDATA "[lowbar]"--=low line-->
<!ENTITY hyphen SDATA "[hyphen]"--=hyphen-->
<!ENTITY period SDATA "[period]"--=full stop, period-->
<!ENTITY sol    SDATA "[sol   ]"--=solidus-->
<!ENTITY colon  SDATA "[colon ]"--/colon P:-->
<!ENTITY semi   SDATA "[semi  ]"--=semicolon P:-->
<!ENTITY quest  SDATA "[quest ]"--=question mark-->
<!ENTITY iquest SDATA "[iquest]"--=inverted question mark-->
<!ENTITY laquo  SDATA "[laquo ]"--=angle quotation mark, left-->
<!ENTITY raquo  SDATA "[raquo ]"--=angle quotation mark, right-->
<!ENTITY lsquo  SDATA "[lsquo ]"--=single quotation mark, left-->
<!ENTITY rsquo  SDATA "[rsquo ]"--=single quotation mark, right-->
<!ENTITY ldquo  SDATA "[ldquo ]"--=double quotation mark, left-->
```

```
<!ENTITY rdquo  SDATA "[rdquo ]"--=double quotation mark, right-->
<!ENTITY nbsp   SDATA "[nbsp  ]"--=no break (required) space-->
<!ENTITY shy    SDATA "[shy   ]"--=soft hyphen-->
```

D.4.3.2 Diacritical Mark Characters

These entities are considered to represent independent characters.

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOdia PUBLIC
       "ISO 8879-1986//ENTITIES Diacritical Marks//EN">
     %ISOdia;
-->
<!ENTITY acute  SDATA "[acute ]"--=acute accent-->
<!ENTITY breve  SDATA "[breve ]"--=breve-->
<!ENTITY caron  SDATA "[caron ]"--=caron-->
<!ENTITY cedil  SDATA "[cedil ]"--=cedilla-->
<!ENTITY circ   SDATA "[circ  ]"--=circumflex accent-->
<!ENTITY dblac  SDATA "[dblac ]"--=double acute accent-->
<!ENTITY die    SDATA "[die   ]"--=dieresis-->
<!ENTITY dot    SDATA "[dot   ]"--=dot above-->
<!ENTITY grave  SDATA "[grave ]"--=grave accent-->
<!ENTITY macr   SDATA "[macr  ]"--=macron-->
<!ENTITY ogon   SDATA "[ogon  ]"--=ogonek-->
<!ENTITY ring   SDATA "[ring  ]"--=ring-->
<!ENTITY tilde  SDATA "[tilde ]"--=tilde-->
<!ENTITY uml    SDATA "[uml   ]"--=umlaut mark-->
```

D.4.3.3 Publishing Characters

```
<!-- (C) International Organization for Standardization 1986
     Permission to copy in any form is granted for use with
     conforming SGML systems and applications as defined in
     ISO 8879, provided this notice is included in all copies.
-->
<!-- Character entity set. Typical invocation:
     <!ENTITY % ISOpub PUBLIC
       "ISO 8879-1986//ENTITIES Publishing//EN">
     %ISOpub;
-->
<!ENTITY emsp   SDATA "[emsp  ]"--=em space-->
<!ENTITY ensp   SDATA "[ensp  ]"--=en space (1/2-em)-->
<!ENTITY emsp13 SDATA "[emsp3 ]"--=1/3-em space-->
<!ENTITY emsp14 SDATA "[emsp4 ]"--=1/4-em space-->
<!ENTITY numsp  SDATA "[numsp ]"--=digit space (width of a number)-->
<!ENTITY puncsp SDATA "[puncsp]"--=punctuation space (width of comma)-->
<!ENTITY thinsp SDATA "[thinsp]"--=thin space (1/6-em)-->
<!ENTITY hairsp SDATA "[hairsp]"--=hair space-->
<!ENTITY mdash  SDATA "[mdash ]"--=em dash-->
<!ENTITY ndash  SDATA "[ndash ]"--=en dash-->
<!ENTITY dash   SDATA "[dash  ]"--=hyphen (true graphic)-->
<!ENTITY blank  SDATA "[blank ]"--=significant blank symbol-->
<!ENTITY hellip SDATA "[hellip]"--=ellipsis (horizontal)-->
<!ENTITY nldr   SDATA "[nldr  ]"--=double baseline dot (en leader)-->
<!ENTITY frac13 SDATA "[frac13]"--=fraction one-third-->
<!ENTITY frac23 SDATA "[frac23]"--=fraction two-thirds-->
<!ENTITY frac15 SDATA "[frac15]"--=fraction one-fifth-->
```

```
<!ENTITY frac25 SDATA "[frac25]"--=fraction two-fifths-->
<!ENTITY frac35 SDATA "[frac35]"--=fraction three-fifths-->
<!ENTITY frac45 SDATA "[frac45]"--=fraction four-fifths-->
<!ENTITY frac16 SDATA "[frac16]"--=fraction one-sixth-->
<!ENTITY frac56 SDATA "[frac56]"--=fraction five-sixths-->
<!ENTITY incare SDATA "[incare]"--=in-care-of symbol-->
<!ENTITY block  SDATA "[block ]"--=full block-->
<!ENTITY uhblk  SDATA "[uhblk ]"--=upper half block-->
<!ENTITY lhblk  SDATA "[lhblk ]"--=lower half block-->
<!ENTITY blk14  SDATA "[blk14 ]"--=25% shaded block-->
<!ENTITY blk12  SDATA "[blk12 ]"--=50% shaded block-->
<!ENTITY blk34  SDATA "[blk34 ]"--=75% shaded block-->
<!ENTITY marker SDATA "[marker]"--=histogram marker-->
<!ENTITY cir    SDATA "[cir   ]"--/circ B: =circle, open-->
<!ENTITY squ    SDATA "[squ   ]"--=square, open-->
<!ENTITY rect   SDATA "[rect  ]"--=rectangle, open-->
<!ENTITY utri   SDATA "[utri  ]"--/triangle =up triangle, open-->
<!ENTITY dtri   SDATA "[dtri  ]"--/triangledown =down triangle, open-->
<!ENTITY star   SDATA "[star  ]"--=star, open-->
<!ENTITY bull   SDATA "[bull  ]"--/bullet B: =round bullet, filled-->
<!ENTITY squf   SDATA "[squf  ]"--/blacksquare =sq bullet, filled-->
<!ENTITY utrif  SDATA "[utrif ]"--/blacktriangle =up tri, filled-->
<!ENTITY dtrif  SDATA "[dtrif ]"--/blacktriangledown =dn tri, filled-->
<!ENTITY ltrif  SDATA "[ltrif ]"--/blacktriangleleft R: =l tri, filled-->
<!ENTITY rtrif  SDATA "[rtrif ]"--/blacktriangleright R: =r tri, filled-->
<!ENTITY clubs  SDATA "[clubs ]"--/clubsuit =club suit symbol-->
<!ENTITY diams  SDATA "[diams ]"--/diamondsuit =diamond suit symbol-->
<!ENTITY hearts SDATA "[hearts]"--/heartsuit =heart suit symbol-->
<!ENTITY spades SDATA "[spades]"--/spadesuit =spades suit symbol-->
<!ENTITY malt   SDATA "[malt  ]"--/maltese =maltese cross-->
<!ENTITY dagger SDATA "[dagger]"--/dagger B: =dagger-->
<!ENTITY Dagger SDATA "[Dagger]"--/ddagger B: =double dagger-->
<!ENTITY check  SDATA "[check ]"--/checkmark =tick, check mark-->
<!ENTITY cross  SDATA "[ballot]"--=ballot cross-->
<!ENTITY sharp  SDATA "[sharp ]"--/sharp =musical sharp-->
<!ENTITY flat   SDATA "[flat  ]"--/flat =musical flat-->
<!ENTITY male   SDATA "[male  ]"--=male symbol-->
<!ENTITY female SDATA "[female]"--=female symbol-->
<!ENTITY phone  SDATA "[phone ]"--=telephone symbol-->
<!ENTITY telrec SDATA "[telrec]"--=telephone recorder symbol-->
<!ENTITY copysr SDATA "[copysr]"--=sound recording copyright sign-->
<!ENTITY caret  SDATA "[caret ]"--=caret (insertion mark)-->
<!ENTITY lsquor SDATA "[lsquor]"--=rising single quote, left (low)-->
<!ENTITY ldquor SDATA "[ldquor]"--=rising dbl quote, left (low)-->

<!ENTITY fflig  SDATA "[fflig ]"--small ff ligature-->
<!ENTITY filig  SDATA "[filig ]"--small fi ligature-->
<!ENTITY fjlig  SDATA "[fjlig ]"--small fj ligature-->
<!ENTITY ffilig SDATA "[ffilig]"--small ffi ligature-->
<!ENTITY ffllig SDATA "[ffllig]"--small ffl ligature-->
<!ENTITY fllig  SDATA "[fllig ]"--small fl ligature-->

<!ENTITY mldr   SDATA "[mldr  ]"--em leader-->
<!ENTITY rdquor SDATA "[rdquor]"--rising dbl quote, right (high)-->
<!ENTITY rsquor SDATA "[rsquor]"--rising single quote, right (high)-->
<!ENTITY vellip SDATA "[vellip]"--vertical ellipsis-->

<!ENTITY hybull SDATA "[hybull]"--rectangle, filled (hyphen bullet)-->
<!ENTITY loz    SDATA "[loz   ]"--/lozenge - lozenge or total mark-->
<!ENTITY lozf   SDATA "[lozf  ]"--/blacklozenge - lozenge, filled-->
<!ENTITY ltri   SDATA "[ltri  ]"--/triangleleft B: l triangle, open-->
<!ENTITY rtri   SDATA "[rtri  ]"--/triangleright B: r triangle, open-->
```