INTERNATIONAL STANDARD

ISO/IEC 10179

First edition
1996-04-01

Information technology — Processing languages — Document Style Semantics and Specification Language (DSSSL)

Technologies de l'information — Langages de traitement — Sémantique de présentation de documents et langage de spécifications (DSSSL)

Reference number
ISO/IEC 10179:1996(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 10179 was prepared by Joint Technical Committee ISO/IEC JTC1, *Information technology*.

Annex A of this International Standard is for information only.

Introduction

This International Standard defines the Document Style Semantics and Specification Language (DSSSL) used to specify the formatting and transformation of SGML documents. The initial focus of DSSSL is on formatting for both paper and electronic media and on the transformation of SGML documents marked up according to different DTDs. DSSSL may be used with any SGML documents without requiring modifications or constraining the document type definitions.

The main objective of this International Standard is to provide a language for expressing formatting and other document processing specifications in a formal and rigorous manner so that these specifications may be processed by a broad range of formatters, either natively or using a translation mechanism.

The DSSSL style language allows users to specify the types of formatting to be applied to various objects during composition, layout, and pagination. The DSSSL transformation language allows users to specify the transformation of documents from one application of SGML markup into another.

DSSSL is designed for specifications that apply to a class of documents. These specifications are applicable to all possible SGML documents for an SGML application as well as to a particular SGML document.

The DSSSL specification languages are declarative. They are not intended to be complete programming languages, although they contain constructs normally associated with such languages. DSSSL specifications can be unambiguously parsed and interpreted by heterogeneous systems. In addition, DSSSL specifications may be used by existing formatting systems through the use of 'front-end' DSSSL processors and translators. DSSSL has no bias toward batch or interactive formatting systems and does not prescribe any pre-defined formatting algorithms.

The standardization of formatting semantics is provided in DSSSL through a set of basic structures known as flow objects and an associated set of formatting characteristics that are applied to those objects. DSSSL provides mechanisms for defining and extending the semantic constructs so that DSSSL application designers can construct DSSSL applications best suited to their application environments.

0.1 Background

The concepts behind DSSSL are associated with the development of generic coding and specifically with SGML, the Standard Generalized Markup Language (ISO 8879).

Historically, electronic manuscripts contained control codes or macro calls that caused the document to be formatted in a particular way ('specific coding'). In contrast, generic coding, which began in the late 1960s, uses descriptive tags (for example, 'heading' rather than 'Space 3 lines; 14 point Bodoni'). Central to the concept of generic coding is the separation of the information content of documents from the format or appearance of the content. The generic coding concept gained prominence in the early 1970s and came to fruition with the development of SGML.

While SGML provides the language for modeling classes of documents, it does not prescribe any particular model or pre-defined tag set. A set of rules (consisting primarily of a DTD and its supporting documentation) that applies SGML to a class of documents is known as an SGML application.

SGML standardizes the representation of the document structure, leaving it to users to develop their own techniques for interfacing with formatters and other processors, such as general purpose translators. DSSSL is designed to support this second class of applications by providing a standardized architecture for formatting and other processing specifications, allowing users to interchange such specifications within a standardized framework.

A DSSSL specification is normally external to the SGML document to which it applies, and thus multiple specifications may be applied to a given SGML document to yield various presentations of the same data.

SGML provides the ability to distinguish between the intrinsic content and structure of a document, on the one hand, and the specifications for processing it on the other. With DSSSL, formatting and other processing specifications may be interchanged in conjunction with SGML documents to provide the standardized specification of document display while preserving the essential distinction between content and format.

Information technology — Processing languages — Document Style Semantics and Specification Language (DSSSL)

1 Scope

This International Standard is designed to specify the processing of valid SGML documents.

DSSSL defines the semantics, syntax, and processing model of two languages for the specification of document processing:

a) The transformation language for transforming SGML documents marked up in accordance with one or more DTDs into other SGML documents marked up in accordance with other DTDs. The specification of this transformation process is fully defined by this International Standard.

b) The style language, where the result is achieved by applying a set of formatting characteristics to portions of the data, and the specification is, therefore, as precise as the application requires, leaving some formatting decisions, such as line-end and column-end decisions, to the composition and layout process.

The DSSSL style language is intended to be used in a wide variety of environments with typographic requirements ranging from simple single-column layouts to complex multiple-column layouts. This International Standard does not standardize a formatter nor does it standardize composition or other processing algorithms. Rather, it provides the means whereby an implementation may externalize 'style characteristics' and other techniques for associating style information with an SGML document.

DSSSL provides a mechanism for specifying the use of 'external processes' to manipulate data. The nature of these processes is outside the scope of DSSSL, but may include typical data management functions, such as sorting and indexing; typical composition functions, such as hyphenation algorithms; and graphics or multimedia processes for non-SGML data.

Documents that have already been formatted or do not contain any hierarchical structural information or generic markup are not within the field of application of this International Standard.

DSSSL expresses specifications to be performed by some processor that accepts an input document and produces an output document. DSSSL is independent of the type of formatter, formatting system, or other transformation processor.

DSSSL includes

a) Constructs that provide access to, and control of, all possible marked-up information in an SGML document, as well as mechanisms for string processing to allow for the manipulation of non-marked up data. This is provided by the Standard Document Query Language (SDQL) component of DSSSL.

NOTE 1 String processing is necessary so that no special 'markers' need be embedded in the source document to indicate presentational changes. The display of a dropped or raised capital letter in a larger point size at the beginning of a line or paragraph is an example of a case where string processing may be used to isolate the first character or group of characters in order to achieve a desired presentational effect.

b) Provisions for specifying the relationship between one or more SGML documents as input to a transformation process and zero or more resulting SGML documents as the output of the process.

c) Provisions for specifying the relationships between the SGML document(s), as expressed in the source Document Type Definition(s), and the result of the formatting process. The output of the formatting process may be an ISO/IEC 10180 Standard Page Description Language (SPDL) document or it may be a document in some other, possibly proprietary, form.

d) Provisions for describing the typographic style and layout of a document.

e) Definitions of a machine-processable syntax for the representation of a DSSSL specification and its various components.

f) Provisions for creating new DSSSL characteristics and their associated values, as well as new flow object classes. These are declared in the declarations for the style language portion of the DSSSL specification.

This International Standard is intended for use in a wide variety of SGML application environments, including both electronic publishing and conventional printing.

2 Conformance

DSSSL includes two independent languages, the transformation language and the style language, which specify processing of an SGML document. A DSSSL specification contains a number of process specifications, each of which uses either the style language or the transformation language. A process specification that uses the style language is a style-specification. A process specification that uses the transformation language is a transformation-specification.

If a style-specification complies with all the provisions of this International Standard, it is a conforming DSSSL style-specification. If a transformation-specification complies with all the provisions of this International Standard, it is a conforming DSSSL transformation-specification.

In both the style language and transformation language, some facilities are optional. Each optional facility is associated with a named *feature*. A process specification that makes use of an

optional facility shall enable the feature with which it is associated using the `features` element type form.

A conforming DSSSL system shall support the style language, the transformation language, or both the style language and the transformation language.

The documentation for a conforming DSSSL system shall state whether it supports the transformation language or the style language or both and, for each language that the system supports, shall state which features of the language it supports.

A conforming DSSSL system that supports the style language shall be able to process any conforming SGML document using any conforming DSSSL style-specification that enables only features of the style language that the DSSSL system is documented to support.

A conforming DSSSL system that supports the transformation language shall be able to process any conforming SGML document using any conforming DSSSL transformation-specification that enables only features of the transformation language that the DSSSL system is documented to support.

3 Normative References

The following standards contain provisions which, through reference in this text, consititute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 639:1988, *Code for the representation of names of languages.*

ISO 3166:1993, *Codes for the representation of names of countries.*

ISO/IEC 6429:1992, *Information technology — Control functions for coded character sets.*

ISO 8601:1988, *Data elements and interchange formats — Information exchange — Representation of dates and times.*

ISO 8879:1986, *Information processing — Text and office systems — Standard Generalized Markup Language (SGML).*

ISO/IEC 9070:1991, *Information technology — SGML support facilities — Registration procedures for public text owner identifiers.*

ISO/IEC 9541-1:1992, *Information technology — Font information interchange — Part 1: Architecture.*

ISO/IEC 9541-2:1992, *Information technology — Font information interchange — Part 2: Interchange Format.*

ISO/IEC 9945-2:1993, *Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities.*

ISO/IEC 10180:1995, *Information technology — Processing languages — Standard Page Description Language (SPDL).*

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

ISO/IEC 10744:1992, *Information technology — Hypermedia/Time-based Structuring Language (HyTime).*

4 Definitions

For the purposes of this International Standard, the definitions given in ISO 8879 and the following definitions apply.

4.1 area
A rectangular box with a fixed width and height produced by the formatting of a flow object. An area can be imaged on a presentation medium to produce a set of marks.

4.2 association
A triple consisting of a query-expression, a transform-expression, and a priority-expression. The priority-expression defaults to 0. Associations are used to control the transformation process.

4.3 atomic flow object
A flow object that has no ports.

4.4 auxiliary grove
A grove created by parsing nodes in another grove.

4.5 characteristic
A named parameter of a flow object.

4.6 complete grove
The grove that would be built using a grove plan that selected all the classes and properties from the property set.

4.7 component name
A name defined in a property set with three variants: a reference concrete syntax name, an application name, and a full name.

4.8 creation origin
The node relative to which the position of a node in a result grove is specified.

4.9 descendants
The union of the subtrees of the children of a node.

4.10 enumerator
A possible value of an enumeration data type.

4.11 flow object
A specification of a task to be performed by the formatter. A flow object has a class, which specifies the kind of task, and characteristics which further parameterize the task.

4.12 formatting process
The process partially specified by the style language.

4.13 grove
A set of nodes connected into a graph by their nodal properties. A grove is built using a grove plan.

4.14 grove plan
A set of classes and properties selected from a property set.

4.15 grove root
The unique node in a grove that has no origin.

4.16 intrinsic property
A property that is automatically part of a property set, without being defined in the property set.

4.17 line-progression-direction
A direction associated with inline areas. The line-progression-direction is perpendicular to the inline-progression-direction of the inlined area.

4.18 nodal property
A property whose value is a node or list of nodes. Nodal properties are categorized by their property set as subnode, irefnode, or urefnode.

4.19 node
An ordered set of property assignments. A node is a member of a grove, and belongs to a class defined in the grove plan used to build its grove.

4.20 origin
For a node *x*, the node that exhibits for a subnode property a value that includes *x*. Every node in a grove other than the grove root has a unique origin.

4.21 origin-to-subnode relationship
The subnode property of the origin of a node that includes the node in its value.

4.22 port
A point on a flow object in a flow object tree to which an ordered list of flow objects can be attached. A port is either the principal port of the flow object or it is named.

4.23 primitive data type
A data type that has no super type. The primitive data type of a data type is the data type itself, if the data type has no super type, and otherwise the primitive data type of the super type of the data type.

4.24 property assignment
The assignment of a property value to a property name.

4.25 property set
A set of classes and properties with associated definitions.

4.26 process specification
The combination of the specification in a process specification element and the specifications in any other process specification elements that the process specification element is declared to use.

4.27 process specification element
An instance of a transformation-specification or style-specification element type form.

4.28 process specification part
A section of the process specification coming from a single process specification element. Any process specification elements referred to using the `use` attribute are separate parts. A part of a process specification takes precedence over any later parts of the process specification.

4.29 siblings (of a node)
The other nodes in the grove that occur in the value of the origin-to-subnode relationship property of the origin of the node.

4.30 sosofo
A specification of a sequence of flow objects.

4.31 source grove
The grove parsed to create an auxiliary grove.

4.32 spread
Consecutive back/front pair of pages in a page-sequence.

4.33 stream
An ordered list of flow objects attached to a port of a flow object.

4.34 subgrove
The union of a node and the values of the subnode properties of the node.

4.35 subtree
A node together with the subtrees of its children.

4.36 synchronization set
A set of flow objects in different streams whose relative positioning is constrained.

4.37 transformation process
The process specified by the transformation language. It transforms one or more SGML documents into zero or more other SGML documents.

4.38 tree
The subtree of a node that has no parent.

4.39 verification grove
The grove that would be built by parsing the SGML document or subdocument generated from the result grove using a grove plan that included all classes and properties of the SGML property set.

4.40 zone
One of four named subdivisions of a column. The four zones are: top-float, body-text, bottom-float, and footnote. The positioning of an area to be placed in a column-set area container can be controlled by labeling it with the name of a zone.

5 Notation and Conventions

5.1 Syntax Productions

In this International Standard, formal syntax is described in a manner similar to ISO 8879 with the following exceptions.

A sequence of expressions indicates that the expressions shall occur in the order shown. The , operator is not used.

The occurrence indicators ?, +, and * have higher precedence than sequencing, which in turn has higher precedence than the connectors | and &. For example,

*a b | c d**

is equivalent to

(a b) | (c (d))*

A syntactic-literal is indicated by a monospaced typeface as shown.

`syntactic-literal`

In a syntax production, double square brackets ([[ ]]) can be used to surround an **or** group. The meaning of this is similar to an **and** group. However, if any of the members of the **or** group have a * or + occurrence indicator, then they can occur the number of times indicated but intermixed with other members of the group. For example,

[[*a** | *b+* | *c* | *d*?]]

means a sequence containing only *a*'s, *b*'s, *c*'s, and *d*'s in which any number of *a*'s occur, one or more *b*'s, exactly one *c*, and at most one *d*.

5.2 Procedure Prototypes

Each procedure is defined by a procedure prototype:

```
(foo a b)
```

This indicates that the identifier `foo` is bound in the top-level environment to a procedure that has two arguments.

If the name of an argument is also the name of a type, then that argument shall be of the named type. The following naming conventions for arguments also imply type restrictions:

— *obj*: any object

— *list*: list

— *q*: quantity

— *x*: real number

— *y*: real number

— *n*: integer

— *k*: exact non-negative integer

If the procedure also accepts keyword arguments, the prototype is of the form:

```
(foo a b #!key key1: key2:)
```

This indicates that the procedure in addition accepts two keyword arguments. The names of the keyword arguments indicate the keywords that are used to specify them and do not constrain the type.

6 DSSSL Overview

A key feature of generalized markup is that the formatting and other processing information associated with the document is separate from the generic tags embedded in it.

In any generalized markup scheme, there is a method for associating processing specifications with the SGML markup. This method of association allows the information to be attached to specific instances of elements as well as to general classes of element types. The primary goal of

DSSSL is to provide a standardized framework and methods for associating processing information with the markup of SGML documents or portions of documents.

DSSSL is intended for use with documents structured as a hierarchy of elements. For the purpose of describing in detail the concepts of DSSSL in the subsequent clauses of this International Standard, SGML terminology is used.

DSSSL enables formatting and other processing specifications to be associated with these elements to produce a formatted document for presentation. For example, a designer may wish to specify that all chapters begin on a new recto page and that all tables begin with a page-wide rule to be positioned only at the top or bottom of the page. During the DSSSL transformation process, formatting information may be added to the result of the transformation. This information may be represented as SGML attributes. These, in turn, may be used by the style language to create formatting characteristics with specific values.

6.1 Areas of Standardization

DSSSL provides four distinct areas of standardization:

a) A language and processing model for transforming one or more SGML documents into zero or more other SGML documents.

This is called the transformation language. This transformation is controlled by the transformation-specification. A transformation-specification contains a list of associations. An association contains up to three parts: the query-expressions, the transform-expressions, and the optional priority-expressions. Functionally, this specification allows the user to specify the creation of new structures, the replication of existing structures, and the reordering and regrouping of existing structures.

b) A language for specifying the application of formatting characteristics onto an SGML document.

The process that applies formatting and other formatting-related processing characteristics to an SGML document is called the *formatting process*. This process is controlled by the style-specification. A style-specification contains a sequence of construction rules. There are several kinds of construction rules. For more details, refer to 12.4.1.

NOTE 2 It is important to note that for the DSSSL style language and the associated formatting process, DSSSL does not standardize the process itself, but merely standardizes the form and semantics of the style language controlling a portion of the process. The remaining formatting functions, such as line-breaking, column-breaking, page-breaking, and other aspects of whitespace distribution, are not standardized and are under control of the formatter.

c) A query language, Standard Document Query Language, used for identifying portions of an SGML document.

SDQL is part of both the DSSSL transformation language and the DSSSL style language. It is used for navigating through the hierarchical structure of the SGML document, identifying

the relevant pieces of the SGML markup and content on which processing is to be performed. SDQL adds additional data types to the DSSSL expression language. In addition to the full query language, this International Standard defines a subset called the *core query language*. For more information on the core query language, see 10.2.4. For a complete discussion of the full SDQL, see clause 10.

d) An expression language.

The DSSSL expression language is used in SDQL, the DSSSL transformation language, and the DSSSL style language. It is used to create and manipulate objects. In addition to the full expression language, this International Standard defines a subset called the *core expression language*. See 8.6. The DSSSL expression language is based on the Scheme Programming Language as defined in the IEEE Scheme standard, R^4RS. DSSSL uses only the functional, side-effect free subset of Scheme. See clause 8 for a complete discussion of the DSSSL expression language.

6.2 Conceptual Model

The DSSSL conceptual model has two distinct processes: (1) a transformation process and (2) a formatting process. The two processes may be used in conjunction with each other, or each may be used alone.

An illustration of the DSSSL conceptual model is shown in Figure 1.

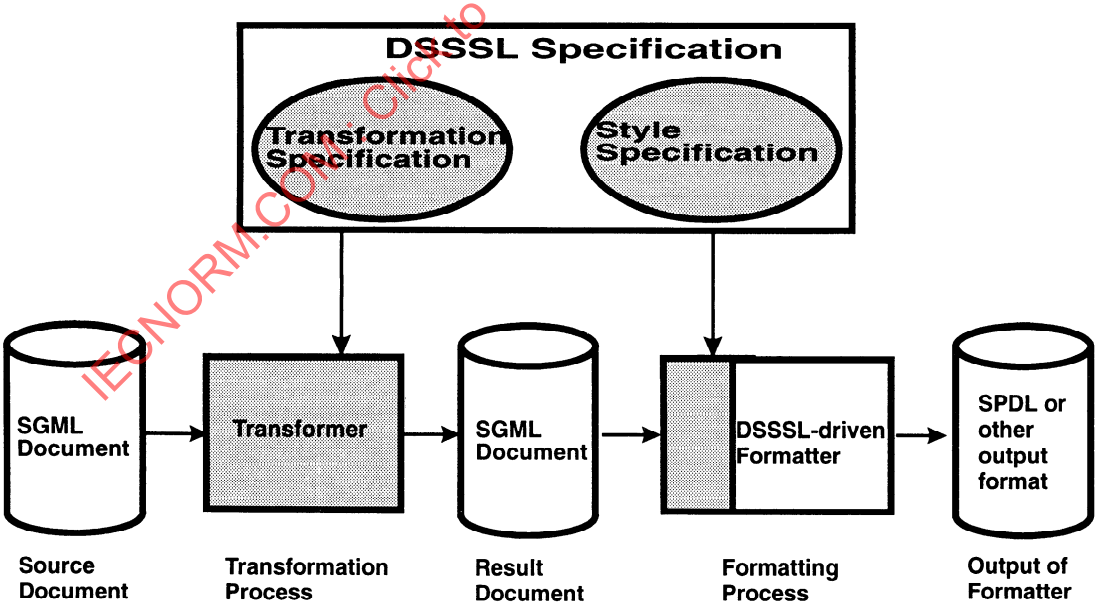

Figure 1 – DSSSL Conceptual Model

The shaded areas indicate the parts of the processing model that are standardized by DSSSL.

6.3 DSSSL Languages

Each of the DSSSL processes is controlled by the appropriate DSSSL language. The transformation language controls the transformation process. Likewise, the style language controls aspects of the formatting process.

6.3.1 The Transformation Language

The transformation process transforms an SGML document into another SGML document under the control of the transformation-specification. The SGML document that is the result of this transformation process may then be used as input to the formatting process.

In the transformation process, a user identifies portions of the SGML document that are to be mapped or transformed. For each node matching the specified portions of SGML content and structure, the transformation is accomplished according to the specification describing the new structures to be created.

All operations performed in this transformation process are independent of the later formatting process. Operations during the transformation process may include the following:

— Combining structures

SGML structures may be reordered and regrouped to create totally new structures. For example, footnotes that are inline with footnote references according to the source DTD may be collected to place the footnotes at the end of each chapter when the document is formatted.

— Creating new elements with user-specifiable relationships to other elements

New structures or attributes may be created. For example, special formatting descriptions such as the need for a 3-point rule, expressed as an SGML attribute, may be associated with every fifth row in a table to provide visual impact.

— Associating new descriptions with particular sequences of content

A sequence of elements in the source document may trigger the association of different formatting characteristics. For example, a paragraph following a warning may be required to be presented differently from all other paragraphs.

— Associating new descriptions with particular components of content

An association may be used to attach special formatting to particular strings of text that may not be specially tagged in the source document, as, for example, in the replacement of the character string 'ISO' with the ISO logo.

DSSSL allows formatting information to be associated with, and dependent on, any combination of the above. Both the content and structure of the SGML document can be modified.

The transformation language can be used to facilitate the formatting process as indicated in the examples above, or it can be used to enhance or modify documents created in accordance with a DTD that has changed over time. It may also be used to transform documents using a public DTD into a proprietary or 'in-house' DTD.

The importance and use of the transformation language will vary depending on the SGML application, the DSSSL application, the capabilities of the formatter, and the implementation. Many formatting applications may require no transformation process at all.

6.3.1.1 Components of the Transformation Process

The component processes are:

a) Grove Building Processor

An SGML document is input to this process. The SGML document or subdocument is parsed and is represented by a collection of nodes called a grove. A grove is similar to an element tree, but may include other subtrees, for example, a subtree of attribute values. Relationships in a grove are expressed in terms of properties. For a complete description of the grove and SGML property definitions, see clause 9.

b) Transformer

The input to the transformation process includes the SGML document as created during the grove building step and the transformation-specification.

The transformation-specification consists of a collection of associations. Each association specifies the transformation of like objects in the source document into objects in the result grove. Key to this transformation is that not only can each object be mapped to an explicit location in the result grove, but it can also be mapped to a location using the result of transforming some other source object as a reference point.

The output of the transformation process is the result grove. The transformation process may operate on multiple SGML documents as input to the process, and likewise may transform them into multiple SGML documents. For a complete description of the transformation process, see clause 11.

c) SGML Generator

The transformation process produces a grove that must be converted to an SGML document for interchange, validation, and input to the formatting process. The SGML generator is used for this purpose. The output of the SGML generator shall be a valid SGML document. For a complete description of the SGML generator, see 11.4.

The model of the transformation process is illustrated in the Figure 2. Note that the shaded areas indicate the components of the DSSSL specification standardized by this International Standard.

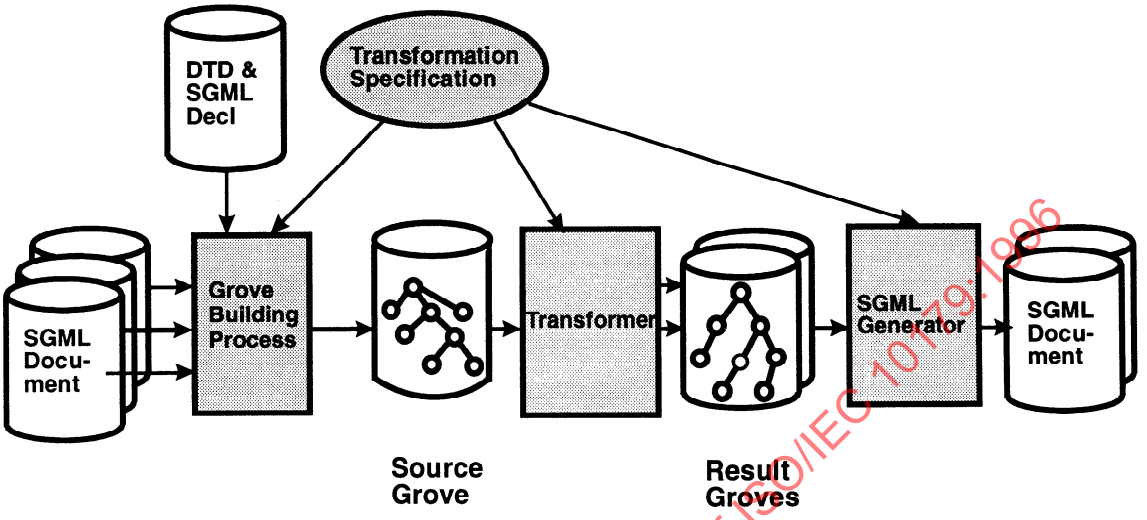

Figure 2 – The Transformation Process

6.3.1.2 Model for Coded Characters, Characters, and Glyph Identifiers

There are three distinct components of this model:

— the coded characters in the SGML source document,

— the characters in the grove,

— the glyph identifiers of the final result document.

The characters in the SGML source document are typically encoded in accordance with a particular character encoding standard, such as ISO 8859-1 ('Latin 1'). The SGML declaration contains a specification of the character set either in the form of a description or in terms of codepoints in one or more particular, normally standardized or at least registered, coded character sets. It is, however, permitted to refer to a private coded character set as well as giving just a description as a minimum literal of the coded character.

There are many character coding schemes. Some of these use non-spacing characters together with a base character to represent a character with a diacritic. SGML also permits the use of entity references to represent 'non-keyable' characters. For example, a lower case e with acute accent may be represented, in the same document, as

— a single character,

— a non-spacing diacritic and e (2 characters),

— an e and combining diacritic (2 characters),

— the entity reference é.

This variation may cause problems in searching using regular expressions.

In DSSSL, the input characters are 'normalized' into a sequence of characters that each represents a specific 'meaning' regardless of how it was originally encoded — as a single character, as multiple characters in a particular character set, or as an entity reference. Each DSSSL specification defines a single character repertoire. The character repertoire shall include all characters used in the DSSSL specification, in the source groves, and in the flow object tree; therefore, only these characters may be used. The declaration of each character also includes a set of properties that may be significant in the formatting process, for example, that the character represents a 'word space'.

The DSSSL specification, which may have been encoded using a different coded character set than the source document, is also translated into a sequence of characters belonging to the same repertoire as the characters used in the DSSSL trees. All comparisons, such as matching an element name, are performed by comparing these characters rather than using the coded characters of the original SGML document.

A sequence of characters in the input grove may be manipulated by a transformation process into another sequence under the control of a character-to-character map. This technique is typically used when parts of the source document contain transliterated text.

The characters in the input grove to the formatter are transformed into glyph identifiers during the formatting process. The transformation is controlled by character-to-glyph and ligature-to-glyph maps in which one or more characters are mapped into one or more glyph identifiers. The map to be used is not fixed for a document, but is expressed as a formatting characteristic that may be specified for an area or for a portion of the input grove. Ligatures are specified by mapping more than one character to a single glyph.

Additional properties specify the font to be used. This information, together with the glyph identifier, selects an actual shape to be used in rendering. Hyphenation points are determined based on the characters, but width calculations are based on the metrics of the actual rendering shapes (i.e., based on the glyphs).

6.3.2 The Style Language

The term 'formatting' when used in this International Standard means any combination of the following:

— the process that applies presentation styles to source document content and determines its position on the presentation medium,

— the selection and reordering of content in the result document with respect to its position in the input document,

— the inclusion of material not explicitly present in the input document, such as the generation of new material,

— the exclusion of material from the input document in the result document.

DSSSL defines the visual appearance of a formatted document in terms of formatting characteristics attached to an intermediate tree called the *flow object tree*. DSSSL allows enough flexibility in the specification so that it is not tied to a set of composition or formatting algorithms, i.e., line-breaking, page-breaking, or whitespace distribution algorithms, used by any particular formatting system. These aspects of the layout process are specific to individual implementations. In this International Standard, line-breaking and page-breaking rules may be expressed in terms of constraints and other formatting characteristics that govern the formatting process. The output of the formatter, undefined in this International Standard, is a formatted document suitable for printing or imaging.

The formatting process uses the style-specification, which may include construction rules, page-model definitions, column-set-model definitions, and other general and application-defined declarations and definitions.

6.3.2.1 Components of the Formatting Process

The conceptual processes that constitute the formatting process are as follows:

a) Build grove from SGML document.

b) Apply construction rules to the objects in the source grove to create the flow object tree.

c) Define page and column geometry by characteristics on the page-sequence flow object and column-set sequence flow objects referring to page-models and column-set-models, respectively.

d) Compose and lay out the content based on the rules specified by the semantics of the flow object classes and the values of the characteristics associated with those objects. Each flow object (an instance of a flow object class) is formatted to produce a sequence of areas having explicit dimensions and positioned by a parent in the flow object tree.

6.3.2.2 Grove Building

The formatting process uses the same grove building step as the transformation process to convert the SGML document into a grove of hierarchically structured objects. For more information, see clause 9.

6.3.2.3 Flow Object Tree

The grove is then further processed, using the construction rules, to create a flow object tree consisting of flow objects with the appropriate formatting and page-layout characteristics. For the formal definition of the construction rules, see 12.4.1. Each flow object (except an atomic flow object) has one or more sequences of flow object children. Each sequence of flow object children is attached to a point of a flow object called a *port*. The port is either the principal port of the flow object, or it may be named.

A flow object class defines a set of formatting characteristics that apply to some category of flow objects. Each flow object class also defines a set of port names. The class of a child flow object shall be compatible with the class and port name of the port to which it is attached. The flow objects attached to any particular port are ordered, but there is no order defined between flow objects attached to different ports of the same flow object.

The process of creating the flow object tree includes the following steps:

a) Formatting characteristics are associated with each flow object.

b) Nodes representing data characters from the grove are converted to character flow objects. Each character flow object has characteristics governing glyph selection and style parameters such as font family, font weight, etc.

In constructing the flow object tree, SDQL may be used to identify portions of the SGML document that have specific formatting characteristics as well as those that can be treated together for purposes of flowing onto the same column or page. The content that is flowed together is placed as a sequence of flow objects in a port of the parent in the flow tree.

NOTE 3 For example, if a document consists of several normal paragraphs and some footnote paragraphs, the footnote paragraphs can be grouped as the content of a port of the parent flow object that represents the footnote. Similarly, the normal paragraphs can be grouped in a port of a flow object representing a sequence of columns.

6.3.2.4 Flow Object Classes

The flow object classes and the characteristics that apply to them define the formatting appearance and behavior of the contents of the document.

The following flow object classes are provided in this International Standard:

Sequence flow object class

Display-group flow object class

Simple-page-sequence flow object class

Page-sequence flow object class

Column-set-sequence flow object class

Paragraph flow object class

Paragraph-break flow object class

Line-field flow object class

Sideline flow object class

Anchor flow object class

Character flow object class

Leader flow object class

Embedded-text flow object class

Rule flow object class

External-graphic flow object class

Included-container-area flow object class

Score flow object class

Box flow object class

Side-by-side flow object class

Glyph-annotation flow object class

Alignment-point flow object class

Aligned-column flow object class

Multi-line-inline-note flow object class

Emphasizing-mark flow object class

Flow object classes for mathematical formulae

Flow object classes for tables

Flow object classes for online display

In addition, DSSSL applications may define their own set of flow object classes as well as their own set of characteristics that may apply to these or to DSSSL-defined flow object classes.

6.3.2.5 Areas

The result of formatting a flow object is a sequence of areas. An area is a rectangular box with a fixed width and height. There are two types of areas: inline areas that are parts of lines and display areas that are not directly parts of lines.

Both types of areas are positioned by a process of filling. The exact nature of the filling process is different for each of these types of areas. See 12.3 for more information on the filling of areas.

A display area is positioned by being filled into an area container. The size of an area container may grow in the filling-direction, but is fixed in the other direction.

6.3.2.6 Page and Column Geometry

Page layout in DSSSL is specified by page-model characteristics on the page-sequence flow object and column-set-model characteristics on the column-set sequence flow object.

The page-sequence flow object is formatted to produce a sequence of page areas. A page-model is the specification of the possible structure and positioning of the area hierarchy of the page, including the height and width of the page and the specification of page-regions. Page-regions are area containers with fixed dimensions into which formatted content is placed as specified by the page-region-flow-map. The page-region-flow-map provides the connection between the port name and a page-region. Each of the page-regions may have a header and a footer specification. For complete information on the page-sequence flow object and the associated page models, see 12.6.4 and 12.6.4.1.

The column-set-sequence flow object is formatted to produce a sequence of column-set areas. A column-set area contains a set of parallel columns. The structure and positioning of each column-set area is controlled by the column-set-model to which it conforms. A column-set-model specifies the possible hierarchy of areas for each column-set. Column-sets may be nested. The column-set area is divided geometrically in a direction parallel to the filling direction into a number of columns. Associated with each column-set may be zones that constrain the placement of areas relative to other areas in the filling-direction. The allowed zones are: top-float, body-text, bottom-float, and footnote.

The column-set-model specifies the possible structure and positioning of the area hierarchy of the column-set through the column-subset specification, the filling-direction specification, width and height specifications, etc. The column-subset specification includes a column-subset-flow-map that indicates the ports from which the contents are flowed into the specified zone. The column-set-model also supports spanning. For complete information on the column-set sequence flow object, see 12.6.5; for complete information on the column-set-model, see 12.6.5.1.

6.3.2.7 Expression Language

The formatting process uses the core expression language defined in 8.6 or, as an optional feature, the full expression language as described in 8.

Figure 3 illustrates the model of the formatting process.

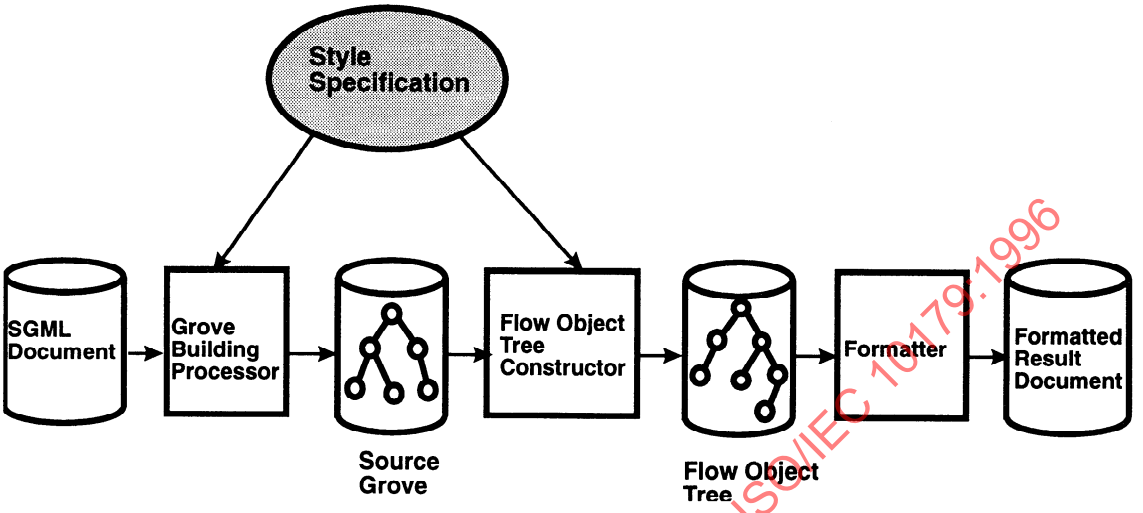

Figure 3 – Formatting Process

6.3.2.8 Model for Coded Characters, Characters, and Glyph Identifiers

The formatting process uses the model for coded characters, characters, and glyph identifiers described in 6.3.1.2.

7 DSSSL Specifications

A DSSSL specification is an SGML document conforming to the DSSSL document architecture. The DSSSL document architecture is a document architecture conforming to the Architectural Form Definition Requirements of ISO/IEC 10744.

An SGML document can declare its conformance to the DSSSL document architecture by including a token `ArcBase` in the APPINFO parameter of its SGML declaration and the following declarations in its DTD:

```
<?ArcBase DSSSL>
<!NOTATION DSSSL    PUBLIC "ISO/IEC 10179:1996//NOTATION
                    DSSSL Architecture Definition Document//EN"
                  -- A document architecture conforming to the
                    Architectural Form Definition Requirements of
                    ISO/IEC 10744.      --
>
<!ATTLIST #NOTATION DSSSL
                              -- Support attributes for all architectures --
                    ArcFormA -- Attribute name: architectural form --
                              NAME      #FIXED DSSSL
                    ArcNamrA -- Attribute name: attribute renamer --
                              NAME      #FIXED DNames
                    ArcBridA -- Attribute name: bridge functions --
```

```
                    NAME      #FIXED DBrid
                ArcDocF   -- Architectural form name: document element --
                          CDATA     #FIXED dsssl-specification
                ArcVer    -- Architecture version identifier --
                          CDATA     #FIXED "ISO/IEC 10179:1996"
>
<!ENTITY DSSSL SYSTEM CDATA DSSSL>
```

7.1 DSSSL Document Architecture

The DSSSL document architecture is defined by the following meta-DTD.

```
<!-- DSSSL Document Architecture -->

<!ENTITY % declarations
 "features | baseset-encoding | literal-described-char | add-name-chars
  | add-separator-chars | standard-chars | other-chars
  | combine-char | map-sdata-entity | char-repertoire | sgml-grove-plan"
>

<!element dsssl-specification - O
          ((%declarations;)*,
           (style-specification | transformation-specification
            | external-specification)+)>
<!attlist dsssl-specification
          dsssl NAME dsssl-specification
          version CDATA #FIXED "ISO/IEC 10179:1996"
>

<!element transformation-specification - O
          ((%declarations;)*, transformation-specification-body*)>
<!attlist transformation-specification
          dsssl NAME transformation-specification
          id ID #IMPLIED
          desc CDATA #IMPLIED
          -- human readable description of specification --

          partial (partial | complete) complete
          -- is the specification complete is or is it just a fragment
             to be used in other specifications? --

          use
          -- reftype(transformation-specification|external-specification) --
              IDREFS #IMPLIED -- Default: none --

          entities
          -- entities available to be specified as DTD for validation
             of result document --
              ENTITIES #IMPLIED -- Default: none --
>

<!element style-specification - O
          ((%declarations;)*, style-specification-body*)>
```

```
<!attlist style-specification
          dsssl NAME style-specification
          id ID #IMPLIED
          desc CDATA #IMPLIED
          -- human readable description of specification --

          partial (partial | complete) complete
          -- is the specification complete is or is it just a fragment
             to be used in other specifications? --

          use   -- reftype(style-specification|external-specification) --
             IDREFS #IMPLIED -- Default: none --
>

<!-- Assign a local ID to a specification in another document. -->
<!element external-specification - O EMPTY>
<!attlist external-specification
          dsssl NAME external-specification
          id ID #REQUIRED
          document -- document containing spec --
                  ENTITY #REQUIRED
          specid -- id of spec in document --
                  NAME #IMPLIED -- Default: first spec in document --
>

<!-- Declares features used by specification. -->

<!element features - O (#PCDATA)
  -- lextype(featurename*) -->
<!attlist features
          dsssl NAME features
>

<!-- Map character numbers in a base character set to character names;
not needed when system knows a character set, and all characters
in character set have universal code. -->
<!element baseset-encoding - O (#PCDATA)
  -- lextype((number, charname)*) -->
<!attlist baseset-encoding
          dsssl NAME baseset-encoding
          name CDATA #REQUIRED -- public identifier of baseset --
>

<!-- Map a character described in the SGML declaration with a minimum literal
to a character name. -->
<!element literal-described-char - O (#PCDATA)
  -- lextype(charname) -->
<!attlist literal-described-char
          dsssl NAME literal-described-char
          desc CDATA #REQUIRED -- the literal description --
>

<!-- Declare additional characters allowed in name within DSSSL notation. -->
<!element add-name-chars - O (#PCDATA)
  -- lextype(charname*) -->
<!attlist add-name-chars
```

```
          dsssl NAME add-name-chars
>

<!-- Declare additional characters allowed as separators within
     DSSSL notation. -->
<!element add-separator-chars - O (#PCDATA)
  -- lextype(charname*) -->
<!attlist add-separator-chars
          dsssl NAME add-separator-chars
>

<!-- Define characters associating names with universal codes. -->

<!element standard-chars - O (#PCDATA)
  -- lextype((charname, number))*) -->
<!attlist standard-chars
          dsssl NAME standard-chars
>

<!-- Define characters with no universal codes. -->

<!element other-chars - O (#PCDATA)
  -- lextype(charname*) -->
<!attlist other-chars
          dsssl NAME other-chars
>

<!-- Map an SDATA entity onto a character. -->

<!element map-sdata-entity - O (#PCDATA)
  -- lextype(charname) -->
<!attlist map-sdata-entity
         dsssl NAME map-sdata-entity
         name CDATA #IMPLIED -- Default: mapping uses replacement text only --
         text CDATA #IMPLIED -- Default: mapping uses name only --
>

<!-- Declare character combining. -->

<!element combine-char - O (#PCDATA)
  -- lextype(charname, charname, charname+) -->
<!attlist combine-char
          dsssl NAME combine-char
>

<!-- Declare a character repertoire. -->
<!element char-repertoire - O EMPTY>
<!attlist char-repertoire
          dsssl NAME char-repertoire
          name -- public identifier for repertoire --
               CDATA #REQUIRED
>
<!-- Declare the grove plan for the SGML property set. -->
<!element sgml-grove-plan - O EMPTY>
<!attlist sgml-grove-plan
          dsssl NAME sgml-grove-plan
```

```
          modadd -- names of modules to be added to default grove plan --
               NAMES #IMPLIED -- Default: none added --
>

<!element style-specification-body - - CDATA
  -- content uses notation of DSSSL style language -->
<!attlist style-specification-body
          dsssl NAME style-specification-body
          content ENTITY #CONREF -- Default: syntactic content --
>

<!element transformation-specification-body - - CDATA
  -- content uses notation of DSSSL transformation language -->
<!attlist transformation-specification-body
          dsssl NAME transformation-specification-body
          content ENTITY #CONREF -- Default: syntactic content --
>
```

The element type form `dsssl-specification` is a container for one or more process specification element type forms. Declaration elements in a `dsssl-specification` element apply to all the process specification elements in the `dsssl-specification` element.

There are two types of process specification element type forms. The element type form `transformation-specification` specifies a transformation process. The element type form `style-specification` specifies a formatting process. Instances of these element type forms are called process specification elements. Each process specification element may be self-contained, or it may make use of other process specification elements of the same type. Process specification elements are identified by an SGML unique identifier. A process specification element in one SGML document may use a process specification element in another SGML document by using the `external-specification` element type form to assign a local unique identifier to the process specification element in the other document. The combination of a process specification element with the process specification elements that it uses is a *process specification*.

A user specifies processing of an SGML document by identifying a process specification element. The manner in which these elements are identified is system-dependent.

NOTE 4 A system may identify a process specification element with a system identifier for the document and an optional unique identifier for the element within the document, with the first process specification element in a document being used if no unique identifier is specified.

Each process specification element may contain elements, called body elements, whose content specifies processing in a process-specific notation. For a `transformation-specification`, this notation is the DSSSL transformation language; for a `style-specification`, this notation is the DSSSL style language. In addition, each process specification element may contain declaration elements that contain information needed to parse these notations.

The process specification described by a sequence of process specification elements is considered as a sequence of parts, where each part consists of declarations expressed using

element type forms, and a specification in the process-specific notation, called the body of the part. The parts from a sequence of process specification elements consist of the sequence of parts from the first process specification element, followed by the sequence of parts from the next process specification element, and so on. The sequence of parts from a single process specification element consists of a part constructed from the content of the process specification element followed by the sequence of parts from the sequence of process specification elements that it uses. The declarations in the first part comprise the declarations contained in the process specification element together with those contained in the `dsssl-specification` element that contains the process specification element. The body of the first part consists of the concatenation of the body elements contained in the process specification element.

A process specification shall be processed by first processing the declarations of all of the parts, and then processing the bodies of all of the parts in order. Within a single part, there shall not be conflicting declarations; when two declarations in different parts conflict, the declaration in the earlier part shall take precedence. Similarly, within the body of a single part, there shall not be conflicting specifications, but when two specifications in the bodies of different parts conflict, the specification in the earlier part shall take precedence.

The declarations of a process specification shall specify how each bit combination occurring in the bodies of the parts of the specification and in all the SGML input documents are to be converted to characters. Declarations may occur in any order. In particular, character names may be used before they are declared.

Every character name used either in declarations or in body elements shall be declared using either a `standard-chars` element type form, an `other-chars` element type form, or a `char-repertoire` element type form.

All declaration element type forms other than the `char-repertoire`, `features`, and `sgml-grove-plan` element type forms require the `charset` feature.

7.1.1 Features

The `features` element type form declares the features used by a specification. A process specification shall declare all the features that it uses.

The content of the element shall be a list of feature names.

This declaration is cumulative.

7.1.2 SGML Grove Plan

The `sgml-grove-plan` element type form names additional modules that should be included in the grove plan for the SGML property set. The `modadd` attribute specifies the modules to be added. The following modules are included automatically:

— baseabs

— prlgabs0

— instabs

For the transformation language, the prlgabs1 module is also included automatically.

This declaration is cumulative.

7.1.3 Character Repertoire

The `char-repertoire` element type form declares that the specification uses the character repertoire whose public identifier is given by the `name` attribute.

A `char-repertoire` element is equivalent to a sequence of instances of the element type forms `baseset-encoding`, `literal-described-char`, `add-name-chars`, `add-separator-chars`, `standard-chars`, `other-chars`, and `map-sdata-entity`, and of *character-property-declaration* and *added-char-properties-declaration* language forms.

7.1.4 Standard Characters

The `standard-chars` element type form declares the names of characters in the character repertoire which correspond to characters defined in ISO/IEC 10646-1 or ISO/IEC 6429. A character in ISO/IEC 10646-1 or ISO/IEC 6429 is identified by its code in the corresponding character set, called its *universal code*.

The content of the element shall be a list of pairs of character names and numbers expressed in decimal. It declares that each character name corresponds to the character with the universal code specified by the following number.

A process specification shall declare character names for each of the following character numbers in ISO/IEC 10646-1: 32 (space), 34 (quotation mark), 35 (number sign), 39 (apostrophe), 40 (left parenthesis), 41 (right parenthesis), 42 (asterisk), 43 (plus sign), 45 (hyphen-minus), 46 (full stop), 47 (solidus), 48 to 57 (digit zero to digit nine), 58 (colon), 59 (semicolon), 60 (less-than sign), 61 (equals sign), 62 (greater-than sign), 63 (question mark), 65 to 90 (Latin capital letter A to Latin capital letter Z), 92 (reverse solidus), and 97 to 122 (Latin small letter a to Latin small letter z). It shall also declare character names for each of the following character numbers in ISO/IEC 6429: 10 (line feed), and 13 (carriage return).

It shall be an error for a single character name to occur more than once in the `standard-chars` elements in a single part. The declaration for a character name in one part in the `standard-chars` element type form takes precedence over any declaration for that character name in any later parts.

A system may inherently know for a base character set identified by a public identifier with an ISO owner identifier how bit combinations in that character set correspond to universal codes. Thus, if a base character set has a formal public identifier that includes an ISO owner identifier, and, for each character used by the document character set from that base character set, exactly one character name is declared using the `standard-chars` element type form, then no `baseset-encoding` element type form is required for that base character set.

7.1.5 Other Characters

The `other-chars` element type form declares the names of characters in the character repertoire which do not correspond to characters defined in ISO/IEC 10646-1 or ISO/IEC 6429.

The content of the element shall consist of a list of character names.

EXAMPLE 1

```
<other-chars>
logoSGML runic-f runic-u
</other-chars>
```

These declarations are cumulative.

7.1.6 Baseset Encoding

The `baseset-encoding` element type form specifies how bit combinations in an SGML document whose meaning was declared in the SGML declaration to be that of a character number in a base character set are to be converted to characters.

The content of a `baseset-encoding` element shall consist of a list of pairs of corresponding character numbers, specified in decimal, and character names. It specifies the character names corresponding to character numbers in the character set whose public identifier is given by the `name` characteristic.

Conflicts between `baseset-encoding` elements are resolved separately for each character number. There can be multiple `baseset-encoding` elements for the same base character set, but it shall be an error to have two specifications for the same character number in the same base character set in a single part.

EXAMPLE 2

```
<baseset-encoding name="Character set for the Viking age runic script">
31 runic-f
32 runic-u
</baseset-encoding>
```

7.1.7 Literal Described Character

The `literal-described-char` element type form specifies that bit combinations in an SGML document whose meaning was declared in the SGML declaration using a minimum literal equal to the value of the `desc` attribute are to be converted to the character whose name is specified in the content of the element.

EXAMPLE 3

```
<literal-described-char desc="SGML User's Group logo">
logoSGML
</literal-described-char>
```

7.1.8 Sdata Entity Mapping

The `map-sdata-entity` element type form declares that a reference to an internal SDATA entity whose name is equal to the value of the `name` attribute and/or whose replacement text is equal to the value of the `text` attribute represents the character whose name is given in the content of the element. The content of the element shall be a single character name.

If the grove plan includes the `entity-name` property for the `sdata` node class, then an SDATA entity shall be mapped by first searching for a mapping for its name and then, if no mapping is found, searching for a mapping for its text.

EXAMPLE 4

```
<map-sdata-entity name="Alpha" text="[Alpha]">greekA</map-sdata-entity>
<map-sdata-entity name="V.Beta" text="[V.Beta]">greekB</map-sdata-entity>
```

7.1.9 Separator Characters

The `add-separator-chars` element type form declares characters as ***separator-character***s allowed in whitespace in the DSSSL transformation and style languages.

These declarations are cumulative.

7.1.10 Name Characters

The `add-name-chars` element type form declares additional characters as ***added-name-character***s allowed in identifiers in the DSSSL transformation and style languages.

These declarations are cumulative.

7.1.11 Character Combination

The `combine-char` element type form contains a list of three or more character names. It declares that a sequence of characters comprising the second and following characters shall be replaced by the first character. Use of this element type form requires the `combine-char` feature.

7.2 Public Identifiers

Within this International Standard, public identifiers shall conform to the canonical string form of a public identifier defined in ISO/IEC 9070.

7.3 Lexical Conventions

7.3.1 Case Sensitivity

Upper- and lower-case forms of a letter are always distinguished.

NOTE 5 Traditionally Lisp systems are case-insensitive.

7.3.2 Identifiers

[1] identifier = *initial* (*subsequent* final*)? | *peculiar-identifier*

[2] initial = *letter* | *special-initial* | ***added-name-character***

[3] letter = a | b | c |...| z | A | B | C |...| Z

[4] special-initial = *special* | :

[5] special = ! | $ | % | & | * | / | < | = | > | ? | ~ | _ | ^

[6] subsequent = *initial* | *digit* | *special-subsequent*

[7] special-subsequent = . | + | -

[8] final = *letter* | *special* | ***added-name-character*** | *digit* | *special-subsequent*

[9] peculiar-identifier = + | - | . . .

Most identifiers allowed by other programming languages are also acceptable in DSSSL. In addition to letters and digits, identifiers may contain the characters `$%&*/:<=>?~_^+-.` and any characters declared as ***added-name-character***s by the `add-name-chars` or `char-repertoire` element type forms. An identifier shall not begin with a character that can begin a number; however, `+`, `-`, and `...` are identifiers. An identifier shall not end with `:` (unless the entire identifier is `:`).

NOTE 6 `...` are three period characters and not a single ellipsis character.

7.3.3 Tokens, Whitespace, and Comments

[10] token = *identifier* | *keyword* | *boolean* | *number* | *character* | *string* | *named-constant* | *glyph-identifier* | (|) | ' | . | ` | , | ,@

[11] delimiter = *whitespace* | (|) | " | ;

[12] whitespace = space | record-start | record-end | tab | form-feed | ***separator-character***

[13] comment = ; ***any-character-except-record-end****

[14] atmosphere = *whitespace* | *comment*

[15] intertoken-space = *atmosphere**

Whitespace characters are spaces, record starts, record ends, and ***separator-character***s. Whitespace is used for improved readability and, as necessary, to separate tokens from each

other, a token being an indivisible lexical unit such as an identifier or number, but is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

A semicolon (;) indicates the start of a comment. The comment continues to the end of the record on which the semicolon appears. Comments are invisible, but the record end is visible as whitespace. This prevents a comment from appearing in the middle of an identifier or number.

intertoken-space may occur on either side of any token, but not within a token.

Tokens which require implicit termination (identifiers, numbers, characters, dot, and #! constants) may be terminated by any *delimiter*, but not necessarily by anything else.

8 Expression Language

The expression language is inspired by the Scheme Programming Language defined in the IEEE Scheme standard, R^4RS. The following specification is based on this definition.

The expression language differs from Scheme in a number of ways:

— The expression language uses only the functional, side-effect free subset of Scheme. Features of Scheme that are not useful in the absence of side-effects have been removed (for example, begin).

— The vector data type is not provided.

— A character object is uniquely identified by its name rather than its code.

— Dependencies in Scheme on the ASCII character set have been removed.

— The number data type is a subtype of a more general quantity data type that adds the concept of dimension to a number.

— Continuations are not provided.

— Some optional features of R^4RS are not provided.

— The `gcd` and `lcm` procedures are not provided.

— Keyword arguments are provided.

In addition, DSSSL specifies certain choices that the definition of Scheme leaves open to implementations.

A subset of the expression language, called the *core expression language*, is defined in 8.6.

8.1 Overview of the Expression Language

Following Algol, the expression language is statically scoped. Each use of a variable is associated with a lexically apparent binding of that variable.

The expression language has latent as opposed to manifest types. Types are associated with values (also called objects) rather than with variables. (Some authors refer to languages with latent types as weakly typed or dynamically typed languages.) Other languages with latent types are other dialects of Lisp, APL, and Snobol. Languages with manifest types (sometimes referred to as strongly typed or statically typed languages) include Algol 60, Pascal, and C.

All objects created in the course of a computation, including procedures, have unlimited extent. No expression language object is ever destroyed. The reason that implementations do not (usually!) run out of storage is that they are permitted to reclaim the storage occupied by an object if they can prove that the object cannot possibly matter to any future computation. Other languages in which most objects have unlimited extent include other dialects of Lisp and APL.

Implementations are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus, with a tail-recursive implementation, iteration may be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as syntactic sugar.

Procedures are objects in their own right. Procedures may be created dynamically, stored in data structures, returned as results of procedures, and so on. Other languages with these properties include Common Lisp and ML.

Arguments to procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not. ML, C, and APL are three other languages that always pass arguments by value. This is distinct from the lazy-evaluation semantics of Haskell, or the call-by-name semantics of Algol 60, where an argument expression is not evaluated unless its value is needed by the procedure.

The expression language, like most dialects of Lisp, employs a fully parenthesized prefix notation for expressions and (other) data; the grammar of the expression language generates a sublanguage of the language used for data.

8.2 Basic Concepts

8.2.1 Variables and Regions

Any identifier that is not a *syntactic-keyword* may be used as a variable. A variable may name a value. A variable that does so is said to be *bound* to the value. The set of all visible bindings in effect at some point is known as the *environment* in effect at that point. The value to which a variable is bound is called the variable's value.

Certain expression types are used to bind variables to new values. The most fundamental of these *binding constructs* is the lambda expression, because all other binding constructs can be explained in terms of lambda expressions. The other binding constructs are `let`, `let*`, and `letrec` expressions.

Like Algol and Pascal, and unlike most other dialects of Lisp except for Common Lisp, the expression language is a statically scoped language with block structure. To each place where a variable is bound in an expression there corresponds a *region* of the expression text within which the binding is effective. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every reference to, or assignment of, a variable refers to the binding of the variable that established the innermost of the regions containing the use. If there is no binding of the variable whose region contains the use, then the use refers to the binding for the variable in the top-level environment, if any; if there is no binding for the identifier, it is said to be *unbound*.

8.2.2 True and False

Any expression language value may be used as a boolean value for the purpose of a conditional test. All values count as true in such a test except for #f. This International Standard uses the word 'true' to refer to any value that counts as true, and the word 'false' to refer to #f.

8.2.3 External Representations

An important concept in the expression language (and Lisp) is that of the *external representation* of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters '`28`', and an external representation of a list consisting of the integers 8 and 13 is the sequence of characters '`(8 13)`'.

The external representation of an object is not necessarily unique. The list in the previous paragraph also has the representations '`( 08 13 )`' and '`(8 . (13 . ()))`'.

Many objects have external representations, but some, such as procedures, do not.

An external representation may be written in an expression to obtain the corresponding object.

External representations may also be used for communicating between processes defined in this International Standard.

The syntax of external representations of various kinds of objects accompanies the description of the primitives for manipulating the objects.

8.2.4 Disjointness of Types

No object satisfies more than one of the following predicates:

```
boolean?
pair?
```

```
symbol?
keyword?
quantity?
char?
string?
procedure?
```

These predicates define the types *boolean*, *pair*, *symbol*, *keyword*, *quantity*, *char* (or *character*), *string*, and *procedure*.

8.3 Expressions

An expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.

[16] expression = *primitive-expression* | *derived-expression*

Expression types are categorized as *primitive* or *derived*. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive but can instead be explained in terms of the primitive constructs. They are redundant in the strict sense of the word, but they capture common patterns of usage, and are, therefore, provided as convenient abbreviations.

8.3.1 Primitive Expression Types

[17] primitive-expression = *variable-reference* | *literal* | *procedure-call* | *lambda-expression* | *conditional* .

8.3.1.1 Variable Reference

[18] variable-reference = *variable*

An expression consisting of a variable is a variable reference. The value of the variable reference is the value to which the variable is bound. It shall be an error to reference an unbound variable.

EXAMPLE 5

```
(define x 28)
x    ⇒ 28
```

[19] variable = *identifier*

[20] syntactic-keyword = *expression-keyword* | `else` | `=>` | `define`

[21] expression-keyword = `quote` | `lambda` | `if` | `cond` | `and` | `or` | `case` | `let` | `let*` | `letrec` | `quasiquote` | `unquote` | `unquote-splicing`

Any identifier that is not a *syntactic-keyword* may be used as a variable. DSSSL languages may reserve identifiers as *syntactic-keywords* in addition to those listed above.

8.3.1.2 Literals

[22] literal = *quotation* | *self-evaluating*

[23] quotation = `'` *datum* | `(quote` *datum*`)`

`(quote` *datum*`)` evaluates to *datum*.

[24] datum = *simple-datum* | *list*

[25] simple-datum = *boolean* | *number* | *character* | *string* | *symbol* | *keyword* | *named-constant* | *glyph-identifier*

datum may be any external representation of an expression language object. This notation is used to include literal constants in expressions. A *glyph-identifier* is allowed only within a *style-language-body*.

EXAMPLE 6

```
(quote a)                    ⇒ a
(quote (+ 1 2))              ⇒ (+ 1 2)
```

`(quote` *datum*`)` may be abbreviated as `'` *datum*. The two notations are equivalent in all respects.

EXAMPLE 7

```
'a                  ⇒ a
'()                 ⇒ ()
'(+ 1 2)            ⇒ (+ 1 2)
'(quote a)          ⇒ (quote a)
''a                 ⇒ (quote a)
```

[26] self-evaluating = *boolean* | *number* | *character* | *string* | *keyword* | *named-constant* | *glyph-identifier*

Boolean constants, numerical constants, character constants, string constants, keywords,named constants, and glyph identifiers evaluate 'to themselves'; they need not be quoted.

EXAMPLE 8

```
'"abc"      ⇒ "abc"
"abc"       ⇒ "abc"
'145932     ⇒ 145932
145932      ⇒ 145932
'#t         ⇒ #t
#t          ⇒ #t
abc:        ⇒ abc:
'abc:       ⇒ abc:
```

8.3.1.3 Procedure Call

[27] procedure-call = (*operator operand**)

[28] operator = *expression*

[29] operand = *expression*

A procedure call is written by simply enclosing in parentheses expressions for the procedure to be called and the arguments to be passed to it. The operator and operand expressions are evaluated, and the resulting procedure is passed the resulting arguments.

EXAMPLE 9

```
(+ 3 4)                    ⇒  7
((if #f + *) 3 4)          ⇒ 12
```

If more than one of the operator or operand expressions signals an error, it is system-dependent which of the errors will be reported to the user.

A number of procedures are available as the values of variables in the initial environment; for example, the addition and multiplication procedures in the above examples are the values of the variables + and *. New procedures are created by evaluating lambda expressions.

Procedure calls are also called *combinations*.

NOTE 7 In contrast to other dialects of Lisp, the operator expression and the operand expressions are always evaluated with the same evaluation rules.

8.3.1.4 Lambda Expression

[30] lambda-expression = (lambda (*formal-argument-list*) *body*)

A lambda expression evaluates to a procedure. The environment in effect when the lambda expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual arguments, the environment in which the lambda expression was evaluated shall be extended by binding the variables in the formal argument list to the corresponding actual argument values, and the body of the lambda expression shall be evaluated in the extended environment. The result of the body shall be returned as the result of the procedure call.

EXAMPLE 10

```
(lambda (x) (+ x x))         ⇒ a procedure
((lambda (x) (+ x x)) 4)     ⇒ 8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)      ⇒ 3

(define add4
```

```
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                    ⇒ 10
```

[31] formal-argument-list = *required-formal-argument** (#!optional *optional-formal-argument**)? (#!rest *rest-formal-argument*)? (#!key *keyword-formal-argument**)?

[32] required-formal-argument = *variable*

[33] optional-formal-argument = *variable* | ((*variable initializer*))

[34] rest-formal-argument = *variable*

[35] keyword-formal-argument = *variable* | ((*variable initializer*))

[36] initializer = *expression*

When the procedure is applied to a list of actual arguments, the formal and actual arguments are processed from left to right as follows:

a) *Variable*s in *required-formal-argument*s are bound to successive actual arguments starting with the first actual argument. It shall be an error if there are fewer actual arguments than *required-formal-argument*s.

b) Next *variable*s in *optional-formal-argument*s are bound to remaining actual arguments. If there are fewer remaining actual arguments than *optional-formal-argument*s, then the variables are bound to the result of evaluating *initializer*, if one was specified, and otherwise to #f. The *initializer* is evaluated in an environment in which all previous formal arguments have been bound.

c) If there is a *rest-formal-argument*, then it is bound to a list of all remaining actual arguments. These remaining actual arguments are also eligible to be bound to *keyword-formal-argument*s. If there is no *rest-formal-argument* and there are no *keyword-formal-argument*s, then it shall be an error if there are any remaining actual arguments.

d) If #!key was specified in the *formal-argument-list*, there shall be an even number of remaining actual arguments. These are interpreted as a series of pairs, where the first member of each pair is a keyword specifying the argument name, and the second is the corresponding value. It shall be an error if the first member of a pair is not a keyword. It shall be an error if the argument name is not the same as a variable in a *keyword-formal-argument*, unless there is a *rest-formal-argument*. If the same argument name occurs more than once in the list of actual arguments, then the first value is used. If there is no actual argument for a particular *keyword-formal-argument*, then the variable is bound to the result of evaluating *initializer* if one was specified, and otherwise to #f. The *initializer* is evaluated in an environment in which all previous formal arguments have been bound.

NOTE 8 Use of #!key in a *formal-argument-list* in the transformation language or style language requires the keyword feature.

It shall be an error for a *variable* to appear more than once in a *formal-argument-list*.

EXAMPLE 11

```
((lambda x x) 3 4 5 6)                ⇒ (3 4 5 6)
((lambda (x y #!rest z) z)
 3 4 5 6)                             ⇒ (5 6)
((lambda (x y #!optional z #!rest r #!key i (j 1)) (list x y z i: i j: j))
 3 4 5 i: 6 i: 7)                     ⇒ (3 4 5 i: 6 j: 1)
```

8.3.1.5 Conditional Expression

[37] conditional = `(if` *test consequent alternate*`)`

[38] test = *expression*

[39] consequent = *expression*

[40] alternate = *expression*

A *conditional* is evaluated as follows: first, *test* is evaluated. If it yields a true value, then *consequent* is evaluated and its value is returned. Otherwise, *alternate* is evaluated and its value is returned.

EXAMPLE 12

```
(if (> 3 2) 'yes 'no)    ⇒ yes
(if (> 2 3) 'yes 'no)    ⇒ no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))             ⇒ 1
```

8.3.2 Derived Expression Types

[41] derived-expression = *cond-expression* | *case-expression* | *and-expression* | *or-expression* | *binding-expression* | *named-let* | *quasiquotation*

8.3.2.1 Cond-expression

[42] cond-expression = `(cond` *cond-clause*+`)` | `(cond` *cond-clause** `(else` *expression*`))`

[43] cond-clause = (*test expression*) | (*test*) | (*test* => *recipient*)

[44] recipient = *expression*

A *cond-expression* is evaluated by evaluating the *test* expressions of each successive *cond-clause* in order until one of them evaluates to a true value. When a *test* evaluates to a true value, then the result of evaluating the *expression* in the *cond-clause* is returned as the result of the entire `cond` expression. If the selected *cond-clause* contains only the *test* and no *expression*, then the value of the *test* is returned as the result. If the *cond-clause* contains a *recipient*, then *recipient* is evaluated. Its value shall be a procedure of one argument; this procedure is then invoked on the

value of the *test*. If all *test*s evaluate to false values, and there is no else clause, then an error is signaled; if there is an else clause, then the result of evaluating its *expression* is returned.

EXAMPLE 13

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))    ⇒ greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))      ⇒ equal
```

8.3.2.2 Case-expression

[45] case-expression = `(case` *key case-clause+*`)` | `(case` *key case-clause** `(else` *expression*`) )`

[46] key = *expression*

[47] case-clause = `(` (*datum**) *expression*`)`

All the *datum*s shall be distinct. A *case-expression* is evaluated as follows. *key* is evaluated and its result is compared against each *datum*. If the result of evaluating *key* is equal (in the sense of `equal?`) to a *datum*, then the result of evaluating the *expression* in the corresponding *case-clause* is returned as the result of the *case-expression*. If the result of evaluating *key* is different from every *datum*, and if there is an else clause, then the result of evaluating its expression is the result of the *case-expression*; otherwise, an error is signaled.

EXAMPLE 14

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite))      ⇒ composite
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant))             ⇒ consonant
```

8.3.2.3 And-expression

[48] and-expression = `(and` *test**`)`

The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

EXAMPLE 15

```
(and (= 2 2) (> 2 1))          ⇒ #t
(and (= 2 2) (< 2 1))          ⇒ #f
```

```
(and 1 2 'c '(f g))          ⇒ (f g)
(and)                        ⇒ #t
```

8.3.2.4 Or-expression

[49] or-expression = `(or` *test*`*)`

The *test* expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.

EXAMPLE 16

```
(or (= 2 2) (> 2 1))      ⇒ #t
(or (= 2 2) (< 2 1))      ⇒ #t
(or #f #f #f)             ⇒ #f
```

8.3.2.5 Binding expressions

[50] binding-expression = *let-expression* | *let*-expression* | *letrec-expression*

The three binding constructs `let`, `let*`, and `letrec` give the expression language a block structure, like Algol 60. The syntax of the three constructs is identical, but they differ in the regions they establish for their variable bindings. In a `let` expression, the initial values are computed before any of the variables become bound; in a `let*` expression, the bindings and evaluations are performed sequentially; while in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions.

[51] let-expression = `(let` *bindings body*`)`

[52] bindings = `(`*binding-spec*`*)`

[53] binding-spec = `(`*variable init*`)`

[54] init = *expression*

It shall be an error for a *variable* to appear more than once in any *bindings*. The *inits* are evaluated in the current environment, the *variables* are bound to the results, and the result of evaluating *body* in the extended environment is returned. Each binding of a *variable* has *body* as its region.

EXAMPLE 17

```
(let ((x 2) (y 3))
  (* x y))                ⇒ 6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))             ⇒ 35
```

See also *named-let*.

[55] let*-expression = `(let*` *bindings body*`)`

A *let*-expression* is similar to a *let-expression*, but the bindings are performed sequentially from left to right, and the region of a binding indicated by a *binding-spec* is that part of the *let*-expression* to the right of the *binding-spec*. Thus, the second binding is done in an environment in which the first binding is visible, and so on.

EXAMPLE 18

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))                    ⇒ 70
```

[56] letrec-expression = `(letrec` *bindings body*`)`

Each *variable* in a *binding-spec* is bound to the result of evaluating the corresponding *init*, and the result of evaluating *body* in the extended environment is returned. The *init*s are evaluated in the extended environment. Each binding of a *variable* in a *binding-spec* has the entire *letrec-expression* as its region, making it possible to define mutually recursive procedures. It shall be an error if the evaluation of an *init* references the value of any of the *variable*s. In the most common uses of `letrec`, all the *init*s are lambda expressions, and this restriction is satisfied automatically.

EXAMPLE 19

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
    ⇒ #t
```


8.3.2.6 Named-let

[57] named-let = `(let` *variable* `(`*binding-spec*`*)` *body*`)`

Named `let` has the same syntax and semantics as ordinary `let` except that *variable* is bound within *body* to a procedure whose formal arguments are the bound variables and whose body is *body*. Thus, the execution of *body* may be repeated by invoking the procedure named by *variable*.

EXAMPLE 20

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))
  ⇒  ((6 1 3) (-5 -2))
```

8.3.2.7 Quasiquotation

The following grammar for quasiquote expressions is not context-free. It is presented as a recipe for generating an infinite number of production rules. Imagine a copy of the following rules for D = 1, 2, 3, ... D keeps track of the nesting depth.

[58] quasiquotation = *quasiquotation_1*

[59] template_0 = *expression*

[60] quasiquotation_D = `*template_D* | (`quasiquote` *template_D*)

[61] template_D = *simple-datum* | *list-template_D* | *unquotation_D*

[62] list-template_D = (*template-or-splice_D**) | (*template-or-splice_D*+ . *template_D*) | '*template_D* | *quasiquotation_D+1*

[63] unquotation_D = ,*template_D-1* | (`unquote` *template_D-1*)

[64] template-or-splice_D = *template_D* | *splicing-unquotation_D*

[65] splicing-unquotation_D = ,@*template_D-1* | (`unquote-splicing` *template_D-1*)

In *quasiquotations*, a *list-template_D* may sometimes be confused with either an *unquotation_D* or a *splicing-unquotation_D*. The interpretation as an *unquotation_D* or *splicing-unquotation_D* takes precedence.

'Backquote' or 'quasiquote' expressions are useful for constructing a list structure when most but not all of the desired structure is known in advance. If no commas appear within the `template`, the result of evaluating `` `template `` is equivalent to the result of evaluating `'template`. If a comma appears within the `template`, however, the expression following the comma is evaluated ('unquoted'), and its result is inserted into the structure instead of the comma and the expression. If a comma appears followed immediately by an at-sign (`@`), then the following expression shall evaluate to a list; the opening and closing parentheses of the list are then

'stripped away' and the elements of the list are inserted in place of the comma at-sign expression sequence.

EXAMPLE 21

```
`(list ,(+ 1 2) 4)  ⇒  (list 3 4)
(let ((name 'a)) `(list ,name ',name))
          ⇒  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
          ⇒  (a 3 4 5 6 b)
`((foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
          ⇒  ((foo 7) . cons)
```

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation and decreases by one inside each unquotation.

EXAMPLE 22

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ⇒  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
          ⇒  (a `(b ,x ,'y d) e)
```

The notations `*template_D* and (`quasiquote` *template_D*) are identical in all respects. ,*expression* is identical to (`unquote` *expression*), and ,@*expression* is identical to (`unquote-splicing` *expression*).

EXAMPLE 23

```
(quasiquote (list (unquote (+ 1 2)) 4))  ⇒  (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
  ⇒  `(list ,(+ 1 2) 4)  i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

Unpredictable behavior may result if any of the symbols `quasiquote`, `unquote`, or `unquote-splicing` appear in positions within a `template` other than as described above.

8.4 Definitions

[66] definition = *variable-definition* | *procedure-definition*

Definitions may take two possible forms.

[67] variable-definition = (`define` *variable expression*)

This syntax is primitive.

[68] procedure-definition = (`define` (*variable formal-argument-list*) *body*)

This form is equivalent to

(define *variable*
  (lambda (*variable formal-argument-list*) *body*)).

A definition that does not occur within an expression is known as a *top-level definition.*

A top-level definition

(define *variable expression*)

evaluates *expression* in the top-level environment and binds *variable* to the result in the top-level environment.

EXAMPLE 24

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                          ⇒  6
(define first car)
(first '(1 2))                    ⇒  1
```

A single variable shall not be defined by more than one top-level definition in any process specification part. A top-level definition of a variable in a process specification part is ignored if that variable has been defined at the top level in a previous process specification part. See 7.1.

The *expression* in a top-level definition shall not be evaluated until all top-level variables that would be referenced by evaluating the *expression* have been defined.

NOTE 9 This constraint does not prevent the definition of mutually recursive procedures, because evaluating a lambda expression does not reference variables that occur free within it.

It shall be an error if it is impossible to evaluate all the *expression*s occurring in top-level definitions in such a way that this constraint is not violated.

The built-in definition of a variable may be replaced by a top-level definition. The replacement definition shall be used for all references to that variable, even those that occur in process specification parts preceding the part that contains the first top-level definition.

NOTE 10 This rule is not easy to implement, but it allows built-in procedures to be added in future versions of this International Standard without changing the meaning of any conforming DSSSL specifications.

[69] body = *definition* expression*

Definitions may also occur at the beginning of a *body*. These are known as internal definitions. The variable defined by an internal definition is local to the *body*. The region of the binding is the entire *body*. For example,

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))                  ⇒  45
```

A *body* containing internal definitions may always be converted into a completely equivalent `letrec` expression. For example, the `let` expression in the previous example is equivalent to

```
(let ((x 5))
  (letrec ((foo (lambda (y) (bar x y)))
           (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

Just as for the equivalent `letrec` expression, it shall be possible to evaluate each *expression* of every internal definition in a *body* without referring to the value of any *variable* being defined.

8.5 Standard Procedures

This section describes the expression language's built-in procedures. The initial (or 'top-level') environment starts out with a number of variables bound to useful values, most of which are primitive procedures that manipulate data. For example, the variable `abs` is bound to a procedure of one argument that computes the absolute value of a number, and the variable + is bound to a procedure that computes sums.

It shall be an error for a procedure to be passed an argument of a type that it is not specified to handle.

8.5.1 Booleans

[70] boolean = `#t` | `#f`

The standard boolean objects for true and false are written as #t and #f. What really matters, though, are the objects that the conditional expressions (`if`, `cond`, `and`, `or`) treat as true or false. The phrase 'a true value' (or sometimes just 'true') means any object treated as true by the conditional expressions, and the phrase 'a false value' (or 'false') means any object treated as false by the conditional expressions.

Of all the standard values, only #f counts as false in conditional expressions. Except for #f, all standard values, including #t, pairs, the empty list, symbols, numbers, strings, and procedures, count as true.

NOTE 11 Programmers accustomed to other dialects of Lisp should be aware that the expression language distinguishes both #f and the empty list from the symbol `nil`.

Boolean constants evaluate to themselves, so they don't need to be quoted in expressions.

EXAMPLE 25

```
#t          ⇒  #t
#f          ⇒  #f
'#f         ⇒  #f
```

8.5.1.1 Negation

`(not` *obj*`)`

`not` returns #t if *obj* is false, and returns #f otherwise.

EXAMPLE 26

```
(not #t)          ⇒  #f
(not 3)           ⇒  #f
(not (list 3))    ⇒  #f
(not #f)          ⇒  #t
(not '())         ⇒  #f
(not (list))      ⇒  #f
(not 'nil)        ⇒  #f
```

8.5.1.2 Boolean Type Predicate

`(boolean?` *obj*`)`

`boolean?` returns #t if *obj* is either #t or #f and returns #f otherwise.

EXAMPLE 27

```
(boolean? #f)          ⇒  #t
(boolean? 0)           ⇒  #f
(boolean? '())         ⇒  #f
```

8.5.2 Equivalence

`(equal?` obj_1 obj_2`)`

The `equal?` procedure defines an equivalence relation on objects. It returns #t if obj_1 and obj_2 should be regarded as the same object, and otherwise returns #f. For objects that have external representations, two objects shall be the same if their external representations are the same. If each of obj_1 and obj_2 is of type boolean, symbol, char, pair, quantity, or string, then the `equal?` procedure shall return #t if and only if:

— obj_1 and obj_2 are both #t or both #f.

— obj_1 and obj_2 are both symbols and

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
    ⇒  #t
```

— obj_1 and obj_2 are both numbers, are numerically equal in the sense of =, and are either both exact or both inexact.

— obj_1 and obj_2 are both strings and are the same string according to the `string=?` procedure.

— obj_1 and obj_2 are both characters and are the same character according to the `char=?` procedure.

— obj_1 and obj_2 are both the empty list.

— obj_1 and obj_2 are both pairs and the car of obj_1 is `equal?` to the car of obj_2 and the cdr of obj_1 is `equal?` to the cdr of obj_2.

If one of obj_1 and obj_2 is a procedure and the other is not, then `equal?` shall return #f. If obj_1 and obj_2 are both procedures then `equal?` shall return #f if obj_1 and obj_2 would return a different value for some arguments, and otherwise shall return either #t or #f.

NOTE 12 In other words equality for procedures is not well defined.

8.5.3 Pairs and Lists

A *pair* (sometimes called a *dotted pair*) is a record structure with two fields called the car and cdr fields (for historical reasons). Pairs are created by the procedure `cons`. The car and cdr fields are accessed by the procedures `car` and `cdr`. Pairs are used primarily to represent lists. A list may be defined recursively as either the empty list or a pair whose cdr is a list. More precisely, the set of lists is defined as the smallest set *X* such that:

— The empty list is in *X*.

— If *list* is in *X*, then any pair whose cdr field contains *list* is also in *X*.

The objects in the car fields of successive pairs of a list are the elements of the list. For example, a two-element list is a pair whose car is the first element and whose cdr is a pair whose car is the second element and whose cdr is the empty list. The length of a list is the number of elements, which is the same as the number of pairs.

The empty list is a special object of its own type (it is not a pair); it has no elements and its length is zero.

NOTE 13 The above definitions imply that all lists have finite length and are terminated by the empty list.

[71] list = (*datum**) | (*datum*+ . *datum*) | *abbreviation*

The most general notation (external representation) for pairs is the 'dotted' notation (c_1 . c_2) where c_1 is the value of the car field and c_2 is the value of the cdr field. For example `(4 . 5)` is a pair whose car is 4 and whose cdr is 5. Note that `(4 . 5)` is the external representation of a pair, not an expression that evaluates to a pair.

A more streamlined notation may be used for lists: the elements of the list are simply enclosed in parentheses and separated by spaces. The empty list is written `()`. For example,

```
(a b c d e)
```

and

```
(a . (b . (c . (d . (e . ()))))))
```

are equivalent notations for a list of symbols.

A chain of pairs not ending in the empty list is called an *improper list*. Note that an improper list is not a list. The list and dotted notations may be combined to represent improper lists:

```
(a b c . d)
```

is equivalent to

```
(a . (b . (c . d)))
```

Whether a given pair is a list depends upon what is stored in the cdr field.

[72] abbreviation = *abbrev-prefix datum*

[73] abbrev-prefix = ' | ` | , | ,@

Within literal expressions, the forms '*datum*, `*datum*, ,*datum*, and ,@*datum* denote the two-element list whose first element are the symbols `quote`, `quasiquote`, `unquote`, and `unquote-splicing`, respectively. The second element in each case is *datum*. This convention is supported so that arbitrary expressions and portions of the specification may be represented as lists. That is, according to the DSSSL expression language grammar, every *expression* is also a *datum*, and a *transformation-language-body* is a sequence of *datums*.

8.5.3.1 Pair Type Predicate

`(pair?` *obj*`)`

Returns #t if *obj* is a pair, and otherwise returns #f.

EXAMPLE 28

```
(pair? '(a . b))          ⇒  #t
(pair? '(a b c))          ⇒  #t
(pair? '())               ⇒  #f
```

8.5.3.2 Pair Construction Procedure

`(cons` obj_1 obj_2`)`

Returns a pair whose car is obj_1 and whose cdr is obj_2.

EXAMPLE 29

```
(cons 'a '())             ⇒  (a)
(cons '(a) '(b c d))      ⇒  ((a) b c d)
(cons "a" '(b c))         ⇒  ("a" b c)
(cons 'a 3)               ⇒  (a . 3)
(cons '(a b) 'c)          ⇒  ((a b) . c)
```

8.5.3.3 car Procedure

`(car` *pair*`)`

Returns the contents of the car field of *pair*. Note that it shall be an error to take the car of the empty list.

EXAMPLE 30

```
(car '(a b c))          ⇒  a
(car '((a) b c d))      ⇒  (a)
(car '(1 . 2))          ⇒  1
(car '())               ⇒  error
```

8.5.3.4 cdr Procedure

(cdr *pair*)

Returns the contents of the cdr field of *pair*. Note that it shall be an error to take the cdr of the empty list.

EXAMPLE 31

```
(cdr '((a) b c d))      ⇒  (b c d)
(cdr '(1 . 2))          ⇒  2
(cdr '())               ⇒  error
```

8.5.3.5 c...r Procedures

(caar *pair*)
(cadr *pair*)
(cdar *pair*)
(cddr *pair*)
(caaar *pair*)
(caadr *pair*)
(cadar *pair*)
(caddr *pair*)
(cdaar *pair*)
(cdadr *pair*)
(cddar *pair*)
(cdddr *pair*)
(caaaar *pair*)
(caaadr *pair*)
(caadar *pair*)
(caaddr *pair*)
(cadaar *pair*)
(cadadr *pair*)
(caddar *pair*)
(cadddr *pair*)
(cdaaar *pair*)
(cdaadr *pair*)
(cdadar *pair*)
(cdaddr *pair*)

`(cddaar` *pair*`)`
`(cdddr` *pair*`)`
`(cddddr` *pair*`)`
`(cdddddr` *pair*`)`

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

Arbitrary compositions, up to four deep, are provided. There are twenty-eight of these procedures in all.

8.5.3.6 Empty List Type Predicate

`(null?` *obj*`)`

Returns #t if *obj* is the empty list, and otherwise returns #f.

8.5.3.7 List Type Predicate

`(list?` *obj*`)`

Returns #t if *obj* is a list, and otherwise returns #f. By definition, all lists have finite length and are terminated by the empty list.

EXAMPLE 32

```
(list? '(a b c))      ⇒  #t
(list? '())           ⇒  #t
(list? '(a . b))      ⇒  #f
```

8.5.3.8 List Construction

`(list` *obj* ...`)`

Returns a list of its arguments.

EXAMPLE 33

```
(list 'a (+ 3 4) 'c)          ⇒  (a 7 c)
(list)                        ⇒  ()
```

8.5.3.9 List Length

`(length` *list*`)`

Returns the length of *list*.

EXAMPLE 34

```
(length '(a b c))             ⇒  3
(length '(a (b) (c d e)))     ⇒  3
(length '())                  ⇒  0
```

8.5.3.10 Lists Appendance

`(append` *`list`* `…)`

Returns a list consisting of the elements of the first *`list`* followed by the elements of the other *`list`*s.

EXAMPLE 35

```
(append '(x) '(y))              ⇒  (x y)
(append '(a) '(b c d))          ⇒  (a b c d)
(append '(a (b)) '((c)))        ⇒  (a (b) (c))
```

The last argument may actually be any object; an improper list results if the last argument is not a proper list.

EXAMPLE 36

```
(append '(a b) '(c . d))        ⇒  (a b c . d)
(append '() 'a)                 ⇒  a
```

8.5.3.11 List Reversal

`(reverse` *`list`*`)`

Returns a list consisting of the elements of *`list`* in reverse order.

EXAMPLE 37

```
(reverse '(a b c))              ⇒  (c b a)
(reverse '(a (b c) d (e (f))))  ⇒  ((e (f)) d (b c) a)
```

8.5.3.12 Sublist Extraction

`(list-tail` *`list k`*`)`

Returns the sublist of *`list`* obtained by omitting the first *k* elements. `List-tail` could be defined by

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

8.5.3.13 List Access

`(list-ref` *`list k`*`)`

Returns the *k*th element of *`list`*. (This is the same as the car of `(list-tail` *`list k`*`)`.)

EXAMPLE 38

```
(list-ref '(a b c d) 2)                    ⇒  c
(list-ref '(a b c d)
          (inexact->exact (round 1.8)))    ⇒  c
```

8.5.3.14 List Membership

`(member` *obj list*`)`

Returns the first sublist of *list* whose car is `equal?` to *obj*, where the sublists of *list* are the non-empty lists returned by `(list-tail` *list k*`)` for *k* less than the length of *list*. If *obj* does not occur in *list*, then #f (not the empty list) is returned.

EXAMPLE 39

```
(member 'a '(a b c))              ⇒  (a b c)
(member 'b '(a b c))              ⇒  (b c)
(member 'a '(b c d))              ⇒  #f
```

8.5.3.15 Association Lists

`(assoc` *obj alist*`)`

alist (for 'association list') shall be a list of pairs. This procedure finds the first pair in *alist* whose car field is `equal?` to *obj* and returns that pair. If no pair in *alist* has *obj* as its car, then #f (not the empty list) is returned.

EXAMPLE 40

```
(define e '((a 1) (b 2) (c 3)))
(assoc 'a e)     ⇒  (a 1)
(assoc 'b e)     ⇒  (b 2)
(assoc 'd e)     ⇒  #f
```

NOTE 14 Although they are ordinarily used as predicates, `member` and `assoc` do not have question marks in their names because they return useful values rather than just #t or #f.

8.5.4 Symbols

Symbols are objects whose usefulness rests on the fact that two symbols are identical (in the sense of `equal?`) if and only if their names are spelled the same way. This is exactly the property needed to represent identifiers, so most implementations of Lisp dialects use them internally for that purpose. Symbols are useful for many other applications; for instance, they may be used the way enumerated values are used in Pascal. Typically, two symbols may be compared for equality in constant time, no matter how long their names.

[74] symbol = *identifier*

The rules for writing a symbol are exactly the same as the rules for writing an identifier.

8.5.4.1 Symbol Type Predicate

`(symbol? obj)`

Returns #t if *obj* is a symbol, and otherwise returns #f.

EXAMPLE 41

```
(symbol? 'foo)          ⇒  #t
(symbol? (car '(a b)))  ⇒  #t
(symbol? "bar")         ⇒  #f
(symbol? 'nil)          ⇒  #t
(symbol? '())           ⇒  #f
(symbol? #f)            ⇒  #f
```

8.5.4.2 Symbol to String Conversion

`(symbol->string symbol)`

Returns the name of *symbol* as a string.

EXAMPLE 42

```
(symbol->string 'flying-fish)        ⇒  "flying-fish"
(symbol->string
   (string->symbol "Malvina"))       ⇒  "Malvina"
```

8.5.4.3 String to Symbol Conversion

`(string->symbol string)`

Returns the symbol whose name is *string*. This procedure may create symbols with names containing special characters, but it is usually a bad idea to create such symbols because they have no external representation. See `symbol->string`.

EXAMPLE 43

```
(equal? 'mISSISSIppi 'mississippi)               ⇒  #f
(equal? 'bitBlt (string->symbol "bitBlt"))       ⇒  #t
(equal? 'JollyWog
        (string->symbol
          (symbol->string 'JollyWog)))           ⇒  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D."))  ⇒  #t
```

8.5.5 Keywords

Keywords are similar to symbols. The main difference is that keywords are self-evaluating and therefore do not need to be quoted in expressions. They are used mainly for specifying keyword arguments.

[75] keyword = *identifier*:

A keyword is a single token; therefore, no whitespace is allowed between the *identifier* and the `:`. The `:` is not considered part of the name of the keyword.

8.5.5.1 Keyword Type Predicate

`(keyword?` *obj*`)`

Returns #t if *obj* is a keyword, and otherwise returns #f.

8.5.5.2 Keyword to String Conversion

`(keyword->string` *keyword*`)`

Returns the name of *keyword* as a string.

EXAMPLE 44

```
(keyword->string Argentina:) ⇒ "Argentina"
```

8.5.5.3 String to Keyword Conversion

`(string->keyword` *string*`)`

Returns the keyword whose name is *string*.

EXAMPLE 45

```
(string->keyword "foobar") ⇒ foobar:
```

8.5.6 Named Constants

[76] named-constant = `#!optional` | `#!rest` | `#!key`

Named-constants are used in *formal-argument-lists*. They are self-evaluating. The named objects have their own unique (unnamed) type that is distinct from the type of any other object.

8.5.7 Quantities and Numbers

8.5.7.1 Numerical Types

The expression language provides a quantity data type which represents lengths and quantities derived from lengths, such as areas and volumes. The SI meter is used as the base unit for representing quantities. The name of this unit is `m`. Any quantity may be represented as the product of a number and the base unit raised to the power of an integer. The dimension of a quantity is the power to which the base unit is raised when the quantity is so represented. A quantity with dimension 0 is *dimensionless*.

It is convenient to be able to express quantities not only in terms of the base unit but also in terms of other derived units.

[77] unit-declaration = `(define-unit` *unit-name expression*`)`

expression shall evaluate to a quantity. A *unit-declaration* declares the derived quantity *unit-name* to be equivalent to this quantity. In this context, *unit-name* is a separate token.

Derived units for centimeters, millimeters, inches, picas, and points, corresponding to the following declarations, are pre-defined.

```
(define-unit cm 0.01m)
(define-unit mm 0.001m)
(define-unit in 0.0254m)
(define-unit pt 0.0003527778m)
(define-unit pica 0.004233333m)
```

The number data type is considered to be a subtype of quantity that represents dimensionless quantities. The expression language provides two types of number: reals and integers. Integers are considered to be a subtype of reals, and reals are a subtype of numbers. For example, the integer 3 is also considered to be a real number, which, in turn, is considered to be a (dimensionless) quantity. The types quantity, number, real, and integer are defined by the predicates `quantity?`, `number?`, `real?`, and `integer?`.

Angle (or more precisely, plane angle) is considered to be a dimensionless quantity (the ratio of two lengths). The integer 1 is equivalent to 1 radian. It is recommended that `rad` be declared as the name of a derived unit equal to the dimensionless quantity 1.

8.5.7.2 Exactness

It is necessary to distinguish between quantities that are represented exactly and those that may not be. For example, indexes into data structures shall be known exactly. In order to catch uses of inexact quantities where exact quantities are required, the expression language explicitly distinguishes exact from inexact quantities. This distinction is orthogonal to the dimension of type.

Quantities are either exact or inexact. A quantity is exact if it was written as an exact constant or was derived from exact quantities using only exact operations. A quantity is inexact if it was written as an inexact constant, if it was derived using inexact ingredients, or if it was derived using inexact operations. Thus, inexactness is a contagious property of a quantity.

If two implementations produce exact results for a computation that did not involve inexact intermediate results, the two ultimate results shall be mathematically equivalent. This is generally not true of computations involving inexact quantities since approximate methods such as floating point arithmetic may be used, but implementations should make the result as close as practical to the mathematically ideal result.

Rational operations such as + should always produce exact results when given exact arguments. If the operation is unable to produce an exact result, then it may either report the violation of an implementation restriction, or it may silently coerce its result to an inexact value.

With the exception of `inexact->exact`, the operations described in this section shall generally return inexact results when given any inexact arguments. An operation may, however, return an exact result if it can prove that the value of the result is unaffected by the inexactness of

its arguments. For example, multiplication of any quantity by an exact zero may produce an exact zero result, even if the other argument is inexact.

8.5.7.3 Implementation Restrictions

Implementations may also support only a limited range of numbers of any type, subject to the requirements of this section. The supported range for exact numbers of any type may be different from the supported range for inexact numbers of that type. For example, an implementation that uses floating point numbers to represent all its inexact real numbers may support a practically unbounded range of exact integers while limiting the range of inexact reals (and, therefore, the range of inexact integers) to the dynamic range of the floating point format. All implementations are required to support exact integers between -2147483647 and 2147483647.

An implementation shall support exact integers throughout the range of numbers that may be used for indexes of lists and strings or that may result from computing the length of a list or string. The `length` and `string-length` procedures shall return an exact integer, and it shall be an error to use anything but an exact integer as an index. Furthermore, any integer constant within the index range, if expressed by an exact integer syntax, shall indeed be read as an exact integer, regardless of any implementation restrictions that may apply outside this range. Finally, the procedures listed below shall always return an exact integer result provided all their arguments are exact integers and the mathematically expected result is representable as an exact integer within the implementation:

```
+             -             *
quotient      remainder     modulo
max           min           abs
floor         ceiling       truncate
round         expt
```

If one of these procedures is unable to deliver an exact result when given exact arguments, then it may either report a violation of an implementation restriction or it may silently coerce its result to an inexact number. Such a coercion may cause an error later.

An implementation may use floating point and other approximate representation strategies for inexact numbers.

This International Standard recommends, but does not require, that the IEEE 32-bit and 64-bit floating point standards be followed by implementations that use floating point representations, and that implementations using other representations should match or exceed the precision achievable using these floating point standards.

In particular, implementations that use floating point representations shall follow these rules. A floating point result shall be represented with at least as much precision as is used to express any of the inexact arguments to that operation. It is desirable (but not required) for potentially inexact operations such as `sqrt`, when applied to exact arguments, to produce exact answers whenever possible (for example the square root of an exact 4 ought to be an exact 2). If, however, an exact quantity is operated upon so as to produce an inexact result (as by `sqrt`), and if the result is represented as a floating point number, then the most precise floating point format

available shall be used; but if the result is represented in some other way, then the representation shall have at least as much precision as the most precise floating point format available.

If an implementation encounters an exact numerical constant that it cannot represent as an exact quantity, then it may either report a violation of an implementation restriction, or it may silently represent the constant by an inexact quantity.

8.5.7.4 Syntax of Numerical Constants

[78] number = *num-2* | *num-8* | *num-10* | *num-16*

[79] num-2 = `#b` *sign*? *digit-2*+

[80] num-8 = `#o` *sign*? *digit-8*+

[81] num-16 = `#x` *sign*? *digit-16*+

[82] num-10 = `#d`? *sign*? *decimal exponent*? *unit*?

[83] decimal = *digit-10*+ | . *digit-10*+ | *digit-10*+ . *digit-10**

[84] exponent = *exponent-marker sign*? *digit*+

[85] exponent-marker = `e`

[86] unit = *unit-name* (*sign*? *digit-10*+)?

[87] unit-name = *letter*+

[88] sign = `+` | `-`

[89] digit-2 = `0` | `1`

[90] digit-8 = `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

[91] digit-10 = *digit*

[92] digit-16 = *digit-10* | `a` | `b` | `c` | `d` | `e` | `f`

[93] digit = `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

A quantity may be written in binary, octal, decimal, or hexadecimal by the use of a radix prefix. The radix prefixes are `#b` (binary), `#o` (octal), `#d` (decimal), and `#x` (hexadecimal). With no radix prefix, a quantity is assumed to be expressed in decimal.

A numerical constant is inexact if it contains a decimal point, an *exponent* or a *unit*; otherwise, it is exact.

NOTE 15 The examples used in this section assume that any numerical constant written using an exact notation is indeed represented as an exact quantity. Some examples also assume that certain numerical constants written using an inexact notation may be represented without loss of accuracy; the inexact constants were chosen so that this is likely to be true in implementations that use floating point numbers to represent inexact quantities.

A numerical constant may have a *unit* suffix. Each *unit-name* shall be the name of the base unit or shall be declared by a *unit-declaration*. A *unit-name* shall not be an *exponent-marker*. If no number follows the *unit-name*, the constant is multiplied by the quantity associated with the unit. If a number with no *sign* or a *sign* of + follows the *unit-name*, the constant is multiplied by the quantity associated with the number name raised to the power of the following number. If a number with a *sign* of – follows the *unit-name*, the constant is divided by the quantity associated with the *unit-name* raised to the power of the absolute value of the following number.

8.5.7.5 Number Type Predicates

`(quantity?` *obj*`)`
`(number?` *obj*`)`
`(real?` *obj*`)`
`(integer?` *obj*`)`

These type predicates may be applied to any kind of argument, including non-quantities. They return #t if the object is of the named type, and otherwise they return #f. In general, if a type predicate is true of a quantity, then all higher type predicates are also true of that quantity. Consequently, if a type predicate is false for a quantity, then all lower type predicates are also false for that quantity.

If *x* is an inexact real number, then `(integer?` *x*`)` is true if and only if `(=` *x* `(round` *x*`))`.

EXAMPLE 46

```
(real? 3)          ⇒  #t
(integer? 3.0)     ⇒  #t
```

NOTE 16 The behavior of these type predicates on inexact quantities is unreliable, since any inaccuracy may affect the result.

8.5.7.6 Exactness Predicates

`(exact?` *q*`)`
`(inexact?` *q*`)`

These numerical predicates provide tests for the exactness of a quantity. For any quantity, precisely one of these predicates is true.

8.5.7.7 Comparison Predicates

`(=` q_1 q_2 q_3 ...`)`
`(<` q_1 q_2 q_3 ...`)`
`(>` q_1 q_2 q_3 ...`)`
`(<=` q_1 q_2 q_3 ...`)`

`(>= q1 q2 q3 ...)`

These procedures return #t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

These predicates are required to be transitive.

The dimensions of all the arguments shall be identical.

NOTE 17 While it is not an error to compare inexact quantities using these predicates, the results may be unreliable because a small inaccuracy may affect the result; this is especially true of = and `zero?`.

8.5.7.8 Numerical Property Predicates

```
(zero? q)
(positive? q)
(negative? q)
(odd? n)
(even? n)
```

These predicates test a quantity for a particular property, returning #t or #f. See note above.

8.5.7.9 Maximum and Minimum

```
(max q1 q2 ...)
(min q1 q2 ...)
```

These procedures return the maximum or minimum of their arguments. The dimensions of all the arguments shall be identical; the dimension of the result shall be the same as the dimension of the arguments.

EXAMPLE 47

```
(max 3 4)          ⇒  4     ; exact
(max 3.9 4)        ⇒  4.0   ; inexact
```

NOTE 18 If any argument is inexact, then the result shall also be inexact (unless the procedure can prove that the inaccuracy is not large enough to affect the result, which is possible only in unusual implementations). If `min` or `max` is used to compare quantities of mixed exactness, and the numerical value of the result cannot be represented as an inexact quantity without loss of accuracy, then the procedure may report a violation of an implementation restriction.

8.5.7.10 Addition

`(+ q1 ...)`

Returns the sum of its arguments, which shall all have the same dimension. The result shall have the same dimension as the arguments.

EXAMPLE 48

```
(+ 3 4)                    ⇒  7
(+ 3)                      ⇒  3
(+)                        ⇒  0
```

8.5.7.11 Multiplication

(* q_1 ...)

Returns the product of its arguments. The dimension of the result shall be the sum of the dimensions of the arguments.

EXAMPLE 49

```
(* 4)                      ⇒  4
(*)                        ⇒  1
```

8.5.7.12 Subtraction

(- q_1 q_2)
(- q)
(- q_1 q_2 ...)

With two or more arguments, returns the difference of its arguments, associating to the left; with one argument, returns the negation of its argument. The dimensions of all the arguments shall be identical. The dimension of the result shall be the same as the dimension of the arguments.

EXAMPLE 50

```
(- 3 4)                    ⇒  -1
(- 3 4 5)                  ⇒  -6
(- 3)                      ⇒  -3
```

8.5.7.13 Division

(/ q_1 q_2)
(/ q)
(/ q_1 q_2 ...)

With two or more arguments, returns the quotient of its arguments, associating to the left; with one argument, returns 1 divided by the argument. The dimension of the result shall be the difference of the dimensions of each of the arguments.

EXAMPLE 51

```
(/ 3 4 5)                  ⇒  3/20
(/ 3)                      ⇒  1/3
```

8.5.7.14 Absolute Value

(abs q)

Returns the magnitude of its argument.

EXAMPLE 52

```
(abs -7)                    ⇒  7
```

8.5.7.15 Number-theoretic Division

(quotient n_1 n_2)
(remainder n_1 n_2)
(modulo n_1 n_2)

These procedures implement number-theoretic (integer) division: For positive integers n_1 and n_2, if n_3 and n_4 are integers such that $n_1 = n_2 n_3 + n_4$ and $0 \leq n_4 < n_2$, then the following is true.

```
(quotient n1 n2)        ⇒  n3
(remainder n1 n2)       ⇒  n4
(modulo n1 n2)          ⇒  n4
```

For integers n_1 and n_2 with n_2 not equal to 0,

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                            ⇒  #t
```

provided all numbers involved in that computation are exact. The value returned by `quotient` always has the sign of the product of its arguments. `remainder` and `modulo` differ on negative arguments — the `remainder` is either zero or has the sign of the dividend, whereas the `modulo` always has the sign of the divisor:

EXAMPLE 53

```
(modulo 13 4)           ⇒  1
(remainder 13 4)        ⇒  1

(modulo -13 4)          ⇒  3
(remainder -13 4)       ⇒  -1

(modulo 13 -4)          ⇒  -3
(remainder 13 -4)       ⇒  1

(modulo -13 -4)         ⇒  -1
(remainder -13 -4)      ⇒  -1

(remainder -13 -4.0)    ⇒  -1.0   ; inexact
```

8.5.7.16 Real to Integer Conversion

(floor x)
(ceiling x)
(truncate x)
(round x)

These procedures return integers.

`floor` returns the largest integer not larger than *x*. `ceiling` returns the smallest integer not smaller than *x*. `truncate` returns the integer closest to *x* whose absolute value is not larger than the absolute value of *x*. `round` returns the closest integer to *x*, rounding to even when *x* is halfway between two integers.

NOTES

19 `round` rounds to even for consistency with the default rounding mode specified by the IEEE floating point standard.

20 If the argument to one of these procedures is inexact, then the result shall also be inexact. If an exact value is needed, the result should be passed to the `inexact->exact` procedure.

EXAMPLE 54

```
(floor -4.3)          ⇒  -5.0
(ceiling -4.3)        ⇒  -4.0
(truncate -4.3)       ⇒  -4.0
(round -4.3)          ⇒  -4.0

(floor 3.5)           ⇒  3.0
(ceiling 3.5)         ⇒  4.0
(truncate 3.5)        ⇒  3.0
(round 3.5)           ⇒  4.0   ; inexact

(round 7)             ⇒  7
```

8.5.7.17 e^n and Natural Logarithm

```
(exp x)
(log x)
```

Returns *e* raised to the power of *x*. `log` computes the natural logarithm of *x* (not the base-ten logarithm). If x is zero or negative, an error shall be signaled.

8.5.7.18 Trigonometric Functions

```
(sin x)
(cos x)
(tan x)
```

`sin`, `cos`, and `tan` return the sine, cosine, and tangent of their arguments, respectively. The result shall be a number.

8.5.7.19 Inverse Trigonometric Functions

```
(asin x)
(acos x)
(atan x)
(atan q1 q2)
```

`asin`, `acos`, and `atan` return the arcsine, arccosine, and arctangent of their arguments, respectively. The result shall be a number. The two-argument variant of `atan` returns the angle of the complex number whose real part is the numerical value of q_2 and whose imaginary part is the numerical value of q_1; the dimensions of q_1 and q_2 shall be identical.

`asin` returns a value in the range $-\pi/2$ to $\pi/2$. `acos` returns a value in the range 0 to π. `atan` returns a value in the range $-\pi/2$ to $\pi/2$.

8.5.7.20 Square Root

`(sqrt` *q*`)`

Returns the square root of *q*. The dimension of *q* shall be even. The dimension of the result shall be half the dimension of *q*. If *q* is negative, an error is signaled.

8.5.7.21 Exponentiation

`(expt` x_1 x_2`)`

Returns x_1 raised to the power x_2. `(expt` x_1 `0)` is defined to be equal to 1.

8.5.7.22 Exactness Conversion

`(exact->inexact` *q*`)`
`(inexact->exact` *q*`)`

`Exact->inexact` returns an inexact representation of *q*. The value returned is the inexact quantity that is numerically closest to the argument. If an exact argument has no reasonably close inexact equivalent, then a violation of an implementation restriction may be reported.

`Inexact->exact` returns an exact representation of *q*. The value returned is the exact quantity that is numerically closest to the argument. If an inexact argument has no reasonably close exact equivalent, then a violation of an implementation restriction may be reported.

These procedures implement the natural one-to-one correspondence between exact and inexact integers throughout an implementation-dependent range.

8.5.7.23 Quantity to Number Conversion

`(quantity->number` *q*`)`

Returns the number of the quantity *q*.

8.5.7.24 Number to String Conversion

`(number->string` *number*`)`
`(number->string` *number radix*`)`

Radix shall be an exact integer, either 2, 8, 10, or 16. If omitted, *radix* defaults to 10. The procedure `number->string` takes a number and a radix and returns as a string an external representation of the given number in the given radix such that

```
(let ((number number)
      (radix radix))
  (equal? number
          (string->number (number->string number
                                          radix)
                          radix)))
```

is true. It shall be an error if no possible result makes this expression true.

If *number* is inexact, the radix is 10, and the above expression may be satisfied by a result that contains a decimal point, then the result contains a decimal point and is expressed using the minimum number of digits (exclusive of exponent and trailing zeroes) needed to make the above expression true; otherwise, the format of the result is unspecified.

The result returned by `number->string` never contains an explicit radix prefix.

NOTE 21 If *number* is an inexact number represented using floating-point numbers, and the radix is 10, then the above expression is normally satisfied by a result containing a decimal point. The unspecified case allows for infinities, NaNs, and non-floating-point representations.

`(format-number` *n string*`)`

Returns a string representation of *n*. *string* specifies the format to use as follows:

— `1` means use 0, 1, 2 ...

— `01` means use 00, 01, 02, ... 10, 11 ... 100, 101 ... and similarly for any number of leading zeros;

— `a` means use 0, a, b, c, ... z, aa, ab, ...

— `A` means use 0, A, B, C, ... Z, AA, AB, ...

— `i` means use 0, i, ii, iii, iv, v, vi, vii, viii, ix, x, ...

— `I` means use 0, I, II, III, IV, V, VI, VII, VIII, IX, X, ...

`(format-number-list` *list* obj_1 obj_2`)`

Returns a string representation of *list*, where *list* is a list of integers. obj_1 specifies the format to use for each number. It shall be either a single string specifying the format to use for all numbers in the same manner as `format-number` or a list of strings with the same number of members as *list* specifying the format to use for each string in the same manner as `format-number`. obj_2 is either a single string or a list of strings specifying the separator to be used between the strings representing each number; it shall contain either a single string or a list of strings with one fewer members than *list*.

8.5.7.25 String to Number Conversion

`(string->number` *`string`*`)`
`(string->number` *`string radix`*`)`

Returns a number of the maximally precise representation expressed by the given *`string`*. *`radix`* shall be an exact integer, either 2, 8, 10, or 16. If supplied, *`radix`* is a default radix that may be overridden by an explicit radix prefix in *`string`* (e.g., `"#o177"`). If *`radix`* is not supplied, then the default radix is 10. If *`string`* is not a syntactically valid notation for a number, then `string->number` returns #f.

EXAMPLE 55

```
(string->number "100")          ⇒  100
(string->number "100" 16)       ⇒  256
(string->number "1e2")          ⇒  100.0
```

8.5.8 Characters

The character object represents a character.

[94] character = `#\` any-character | `#\` *character-name*

[95] character-name = *letter* (*letter* | *digit* | - | .)+

Characters are written using the notation `#\`character or `#\`*character-name*. For example:

— `#\a`: lower-case letter 'a'

— `#\A`: upper-case letter 'A'

— `#\(`: left parenthesis

— `#\ `: the space character

— `#\space`: the preferred way to write a space

If the *character* in `#\`any-character is alphabetic, then the character following any-character shall be a delimiter character such as a space or parenthesis. This rule resolves the ambiguous case where, for example, the sequence of characters '`#\ space`' could be taken to be either a representation of the space character or a representation of the character '`#\ s`' followed by a representation of the symbol '`pace`.'

The *character-name* shall be the name of a character declared in the character repertoire declaration.

Characters written in the `#\` notation are self-evaluating. That is, they do not have to be quoted in expressions.

8.5.8.1 Character Properties

Every character has a set of named properties. Each property has a default value.

[96] character-property-declaration = `(declare-char-property` *identifier expression*`)`

This declares *identifier* to be a character property with the default value equal to the value of *expression*.

[97] added-char-properties-declaration = `(add-char-properties` *keyword-argument-list character+*`)`

[98] keyword-argument-list = (*keyword expression*)*

The *added-char-properties-declaration* adds properties to each of the *characters*. The *keyword-argument-list* specifies the properties to be added. The *keyword* specifies the property name, and the *expression* specifies the property value. Each property either shall be a property that is pre-defined in this International Standard or it shall be explicitly declared using a *character-property-declaration*.

The following character property is pre-defined:

— `numeric-equiv:` is an integer giving the numeric equivalent of the character or #f. The default value is #f.

Additional properties are pre-defined for the style language.

8.5.8.2 Language-dependent Operations

Certain operations on characters such as case-conversion and collation are dependent on the natural language for which the characters are being used. The language data type describes how language-dependent operations should be performed. Expressions may be evaluated with respect to a current language. It shall be an error to call procedures which use the current language if there is no current language.

Some of the procedures that operate on characters ignore the difference between upper case and lower case. The procedures that ignore case have '`-ci`' (for 'case-insensitive') embedded in their names. These procedures always behave as if they converted their arguments to upper case. These procedures all use the current language. See 8.5.8.5 for these procedures.

`(language?` *obj*`)`

Returns #t if *obj* is of type language, and otherwise returns #f.

`(current-language)`

At any point in a computation there may be a current language. `current-language` returns the current language if there is one, and otherwise returns #f.

[99] default-language-declaration = `(declare-default-language` *expression*`)`

A *default-language-declaration* declares the current language which is used initially in the evaluation of an expression. The *expression* shall evaluate to a language object.

`(with-language` *language proc*`)`

The `with-language` procedure calls *proc*, which shall be a procedure of no arguments, with *language* as the current language.

8.5.8.2.1 Language Definition

[100] language-definition = `(define-language` *variable* [[*collation-specification*? | *toupper-specification*? | *tolower-specification*?]]`)`

A *language-definition* defines *variable* to be an object of type language.

8.5.8.2.1.1 Collation

[101] collation-specification = `(collate` [[*multi-collating-element-specification** | *collating-symbol-specification**]] *order-specification* `)`

A *collation-specification* determines the relative order of strings.

NOTE 22 The syntax of the collation-specification is based on ISO 9945-2, which contains examples that may assist the reader.

[102] multi-collating-element-specification = `(element` *multi-collating-element string*`)`

[103] multi-collating-element = *identifier*

When two strings are compared, each string is divided up into collating elements. Each collating element is either a single character or a sequence of consecutive characters that is to be treated as a single unit. A *multi-collating-element-specification* declares that the sequence of characters in the string is to be treated as a collating element. Within the *order-specification*, this collating element is identified by the *multi-collating-element*. Identifiers declared as *multi-collating-element*s shall be distinct from those used as *weight-identifier*s.

[104] collating-symbol-specification = `(symbol` *weight-identifier*`)`

[105] weight-identifier = *identifier*

A *collating-symbol-specification* declares that *weight-identifier* is a symbolic identifier for a weight, which may be used within the *order-specification*.

[106] order-specification = `(order` *sort-rules collation-entry**`)`

[107] sort-rules = `(` *level-sort-rules*+ `)`

Each order specification defines a number of different comparison levels. If two strings compare equal at the first level, they are compared at the second level. If they also compare equal at the second level, they are compared at the third level. This process is repeated until there are no more levels or until the strings compare unequal. The number of levels in the order specification is determined by the number of *level-sort-rules*.

[108] level-sort-rules = *sort-keyword* | ((*sort-keyword*+))

[109] sort-keyword = `forward` | `backward` | `position`

The *level-sort-rules* determine for each level how the strings are to be compared. At a given level, each `collating-element` in the strings to be compared is assigned zero or more weights. This results in an ordered list of weights for each string.

The `backward` and `forward` *sort-keyword*s determine the comparison direction for the level. If the `backward` *sort-keyword* is specified, then comparison proceeds from the last weight to the first; otherwise, it proceeds from the first weight to the last.

If the `position` *sort-keyword* is specified, then the position of the collating element corresponding to each weight is considered when comparing weights. When comparing two weights with different positions, the weight with the earlier position (in the comparison direction) shall collate first.

A single *level-sort-rules* shall not contain both `forward` and `backward`.

[110] collation-entry = ((*collating-element level-weight**)) | *weight-identifier* | *collating-element*

Each collation entry is associated with a weight determined by its position in the *order-specification*. The first collation entry is associated with the lowest weight, the second with the next lowest weight, and so on.

When a *collation-entry* is a *weight-identifier*, then the effect of the *collation-entry* is to associate the *weight-identifier* with the weight with which the *collation-entry* is associated.

A *collation-entry* that contains a *collating-element* serves two purposes. First, it assigns weights for each level to the *collating-element*. Second, it makes *collating-element* stand for the weight associated with the *collation-entry* when the *collating-element* is used in a *weight*.

If a *level-weight* is not specified for some level, then the single weight associated with the *collation-entry* shall be assigned. For example, a *collation-entry* of `#\a` is equivalent to a *collation-entry* of `(#\a #\a)`.

[111] collating-element = *character* | *multi-collating-element* | `#t`

When #t is used as a collating-element, then the specified weights are assigned to all collating elements to which no weight has been explicitly assigned by a *collation-entry*.

[112] level-weight = *weight* | *weight-list*

[113] weight-list = (*weight**)

The *level-weight* specifies the weights to be assigned for a particular level.

[114] weight = *weight-identifier* | *multi-collating-element* | *character* | *string*

Specifying a string is equivalent to specifying a list of the characters it contains.

8.5.8.2.1.2 Case Conversion

[115] toupper-specification = (`toupper` *case-conversion-list*)

[116] tolower-specification = (`tolower` *case-conversion-list*)

[117] case-conversion-list = ((*character character*))*

In the *case-conversion-list*, the upper-case or lower-case equivalent of the first character in each pair is the second character in that pair according as the *case-conversion-list* occurs in a *toupper-specification* or a *tolower-specification*.

8.5.8.3 Character Type Predicate

`(char?` *obj*`)`

Returns #t if *obj* is a character, and otherwise returns #f.

8.5.8.4 Character Comparison Predicates

`(char=?` $char_1$ $char_2$`)`
`(char<?` $char_1$ $char_2$`)`
`(char>?` $char_1$ $char_2$`)`
`(char<=?` $char_1$ $char_2$`)`
`(char>=?` $char_1$ $char_2$`)`

These procedures impose a total ordering on the set of characters. All the procedures other than `char=?` use the current language.

8.5.8.5 Case-insensitive Character Predicates

`(char-ci=?` $char_1$ $char_2$`)`
`(char-ci<?` $char_1$ $char_2$`)`
`(char-ci>?` $char_1$ $char_2$`)`
`(char-ci<=?` $char_1$ $char_2$`)`
`(char-ci>=?` $char_1$ $char_2$`)`

These procedures are similar to `char=?` etc., but they treat upper-case and lower-case letters as the same. All these procedures use the current language. For example, `(char-ci=? #\A #\a)` returns #t.

8.5.8.6 Character Case Conversion

`(char-upcase` *`char`*`)`
`(char-downcase` *`char`*`)`

The procedures return the upper- or lower-case equivalent of *`char`* as defined by the current language. If *`char`* has no upper- or lower-case equivalent, *`char`* is returned.

8.5.8.7 Character Properties

`(char-property` *`symbol char`*`)`
`(char-property` *`symbol char obj`*`)`

Returns the value of the property *`symbol`* of *`char`*. If *`symbol`* is not a character property, an error is signaled. If *`char`* does not have a property *`symbol`*, then *`obj`* is returned, or if *`obj`* was not specified, the default value of the property is returned.

8.5.9 Strings

Strings are sequences of characters.

[118] string = " *string-element** "

[119] string-element = ***any-character-other-than-"-or-***\ | \" | \\ | *character-name*; ?

Strings are written as sequences of characters enclosed within doublequotes (`"`). A doublequote may be written inside a string by escaping it with a backslash (\), as in

```
"The word \"recursion\" has many meanings."
```

A backslash may be written inside a string by escaping it with another backslash. Any character may be written inside a string by writing its name after a backslash. The name shall be followed by a semi-colon, unless there are no following characters in the string, or the following character is not a *subsequent*. The name used here is the same as the name used in `#\` syntax for characters.

A string constant may continue from one record to the next and shall contain the characters that separate the two records in the entity.

The *length* of a string is the number of characters that it contains. This number is a non-negative integer that is fixed when the string is created. The *valid indexes* of a string are the exact non-negative integers less than the length of the string. The first character of a string has index 0, the second has index 1, and so on.

In phrases such as 'the characters of *`string`* beginning with index *`start`* and ending with index *`end`*,' it is understood that the index *`start`* is inclusive and the index *`end`* is exclusive.

Thus, if *start* and *end* are the same index, a null substring is referred to, and if *start* is zero and *end* is the length of *string*, then the entire string is referred to.

Some of the procedures that operate on strings ignore the difference between upper and lower case by converting the strings to upper case before performing the operation. The versions that ignore case have '-ci' (for 'case-insensitive') embedded in their names.

8.5.9.1 String Type Predicate

(string? *obj*)

Returns #t if *obj* is a string, and otherwise returns #f.

8.5.9.2 String Construction

(string *char* ...)

Returns a string composed of the arguments.

8.5.9.3 String Length

(string-length *string*)

Returns the number of characters in the given *string*.

8.5.9.4 String Access

(string-ref *string k*)

k shall be a valid index of *string*. string-ref returns character *k* of *string* using zero-origin indexing.

8.5.9.5 String Equivalence

(string=? $string_1$ $string_2$)
(string-ci=? $string_1$ $string_2$)

Return #t if the two strings are the same length and contain the same characters in the same positions, and otherwise return #f. string-ci=? treats upper- and lower-case letters as though they were the same character, but string=? treats upper- and lower-case letters as distinct characters. string-ci=? uses the current language.

(string-equiv? $string_1$ $string_2$ *k*)

Returns #t if the two strings compare the same at the first *k* comparison levels of the collation specification of the current language, and otherwise returns #f. *k* shall be strictly positive.

8.5.9.6 String Comparison

(string<? $string_1$ $string_2$)

(string>? <*string*$_1$ *string*$_2$)
(string<=? *string*$_1$ *string*$_2$)
(string>=? *string*$_1$ *string*$_2$)
(string-ci<? *string*$_1$ *string*$_2$)
(string-ci>? *string*$_1$ *string*$_2$)
(string-ci<=? *string*$_1$ *string*$_2$)
(string-ci>=? *string*$_1$ *string*$_2$)

These procedures are the lexicographic extensions to strings of the corresponding orderings on characters. For example, string<? is the lexicographic ordering on strings induced by the ordering char<? on characters. If two strings differ in length but are the same up to the length of the shorter string, the shorter string is considered to be lexicographically less than the longer string. These procedures use the current language.

8.5.9.7 Substring Extraction

(substring *string start end*)

Returns a string formed from the characters of *string* beginning with index *start* (inclusive) and ending with index *end* (exclusive).

8.5.9.8 String Appendance

(string-append *string* …)

Returns a string formed by the concatenation of the given strings.

8.5.9.9 Conversion between Strings and Lists

(string->list *string*)
(list->string *chars*)

string->list returns a list of the characters that make up the given string. list->string returns a string formed from the characters in the list *chars*. string->list and list->string are inverses so far as equal? is concerned.

8.5.10 Procedures

8.5.10.1 Procedure Type Predicate

(procedure? *obj*)

Returns #t if *obj* is a procedure, and otherwise returns #f.

EXAMPLE 56

```
(procedure? car)              ⇒  #t
(procedure? 'car)             ⇒  #f
(procedure? (lambda (x) (* x x)))
                              ⇒  #t
```

```
(procedure? '(lambda (x) (* x x)))
                                ⇒  #f
```

8.5.10.2 Procedure Application

(apply *proc args*)
(apply *proc* arg_1 … *args*)

Proc shall be a procedure and *args* shall be a list. The first (essential) form calls *proc* with the elements of *args* as the actual arguments. The second form is a generalization of the first that calls *proc* with the elements of the list (append (list arg_1 …) *args*) as the actual arguments.

EXAMPLE 57

```
(apply + (list 3 4))                ⇒  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)            ⇒  30
```

8.5.10.3 Mapping Procedures over Lists

(map proc $list_1$ $list_2$ …)

The *list*s shall be lists, and *proc* shall be a procedure taking as many arguments as there are *list*s. If more than one *list* is given, then they shall all be the same length. map applies *proc* element-wise to the elements of the *list*s and returns a list of the results, in order from left to right.

EXAMPLE 58

```
(map cadr '((a b) (d e) (g h)))     ⇒  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))                  ⇒  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6))           ⇒  (5 7 9)
```

8.5.10.4 External Procedures

(external-procedure *string*)

Returns a procedure object which when called shall execute the external procedure with public identifier *string*. If the system is unable to find the external procedure, then #f is returned. The arguments passed to the procedure object shall be passed to the external procedure. If the number or type of arguments do not match those expected by the external procedure, then an error may be signaled. The result of the external procedure shall be returned as the result of the call of the procedure object.

External procedures should be side-effect free, and implementations are free to assume that they are. They should be used to retrieve information from the system rather than to change the state of the system.

8.5.11 Date and Time

```
(time)
(time->string k)
(time->string k boolean)
```

`time` returns the number of seconds since 1970-01-01 00:00:00 GMT as an integer.

`time->string` converts an integer representation as returned by `time` of the time and date into a string in the format of ISO 8601.

If the *`boolean`* argument is present and true, then the string representation shall be in GMT; otherwise the string shall be in local time.

```
(time<? string1 string2)
(time>? string1 string2)
(time<=? string1 string2)
(time>=? string1 string2)
```

These procedures impose a total ordering on the set of strings that represent dates and times in ISO 8601 format. It shall be an error if any argument does not represent a date or time in ISO 8601 format.

8.5.12 Error Signaling

```
(error string)
```

`error` signals an error. The *`string`* argument describes the error. The action a system takes when an error is signaled is system-dependent. In particular, the manner in which the error is reported to the user is system-dependent. It should, however, use *`string`* in its report and describe the context in which the error occurred. No value is returned from `error`.

8.6 Core Expression Language

This clause defines a subset of the expression language called the *core expression language*. In the core expression language, only those expressions and definitions allowed by the productions in this clause are permitted, and only those procedures with prototypes in this clause are available. Any expression or definition that is valid in the core expression language has the same meaning that it does in the full expression language.

8.6.1 Syntax

[120] expression = *primitive-expression* | *derived-expression*

[121] primitive-expression = *variable-reference* | *literal* | *procedure-call* | *conditional*

[122] variable-reference = *variable*

[123] variable = *identifier*

[124] literal = *quotation* | *self-evaluating*

[125] quotation = '*datum* | (`quote` *datum*)

[126] datum = *simple-datum* | *list*

[127] simple-datum = *boolean* | *number* | *character* | *string* | *symbol* | *keyword* | *glyph-identifier*

[128] list = (*datum**) | '*datum*

[129] self-evaluating = *boolean* | *number* | *character* | *string* | *keyword* | *glyph-identifier*

[130] procedure-call = (*operator operand**)

[131] operator = *expression*

[132] operand = *expression*

[133] conditional = (`if` *test consequent alternate*)

[134] test = *expression*

[135] consequent = *expression*

[136] alternate = *expression*

[137] derived-expression = *cond-expression* | *case-expression* | *and-expression* | *or-expression*

[138] cond-expression = (`cond` *cond-clause*+) | (`cond` *cond-clause** (`else` *expression*))

[139] cond-clause = (*test expression*)

[140] case-expression = (`case` *key case-clause*+) | (`case` *key case-clause** (`else` *expression*))

[141] key = *expression*

[142] case-clause = ((*datum**) *expression*)

[143] and-expression = (`and` *test**)

[144] or-expression = (`or` *test**)

[145] definition = (`define` *variable expression*)

8.6.2 Procedures

(`not` *obj*)
(`boolean?` *obj*)
(`equal?` obj_1 obj_2)
(`null?` *obj*)
(`list?` *obj*)
(`list` *obj* …)
(`length` *list*)
(`append` *list* …)
(`reverse` *list*)
(`list-tail` *list k*)
(`list-ref` *list k*)
(`member` *obj list*)
(`symbol?` *obj*)
(`keyword?` *obj*)
(`quantity?` *obj*)
(`number?` *obj*)
(`real?` *obj*)
(`integer?` *obj*)
(`=` q_1 q_2 q_3 …)
(`<` q_1 q_2 q_3 …)
(`>` q_1 q_2 q_3 …)
(`<=` q_1 q_2 q_3 …)
(`>=` q_1 q_2 q_3 …)
(`max` q_1 q_2 …)
(`min` q_1 q_2 …)
(`+` q_1 …)
(`*` q_1 …)
(`-` q_1 q_2)
(`-` q)
(`/` q_1 q_2)
(`/` q)
(`abs` q)
(`quotient` n_1 n_2)
(`remainder` n_1 n_2)
(`modulo` n_1 n_2)
(`floor` *x*)
(`ceiling` *x*)
(`truncate` *x*)
(`round` *x*)
(`sqrt` *q*)
(`number->string` *number*)

```
(number->string number radix)
(string->number string)
(string->number string radix)
(char? obj)
(char=? char1 char2)
(char-property symbol char)
(char-property symbol char obj)
(string? obj)
(string char …)
(string-length string)
(string-ref string k)
(string=? string1 string2)
(substring string start end)
(string-append string …)
(procedure? obj)
(apply proc args)
(external-procedure string)
(time)
(time->string k)
(time->string k boolean)
(error string)
```

9 Groves

A grove is a set of nodes constructed according to a grove plan. Every node in the grove belongs to a named class in the grove plan. A node is a set of property assignments, each consisting of a property name and a property value.

A grove plan defines a set of classes and, for each class, an ordered set of properties.

For each property assignment of a node, there is a unique corresponding property of the node's class whose name is the same as the name part of the property assignment. This is referred to as the property of the property assignment. The value part of a property assignment is referred to as a value of the property of the property assignment. A node is said to exhibit a value *v* for a property *p* if there is a property assignment of the node whose property is *p* and whose value part is *v*. The properties for which the node exhibits a value are referred to as the properties of the node.

The ordering of the properties of a class determines for nodes of that class the ordering of the corresponding property assignments.

Every property value has a data type. The definition of a property declares a certain data type to be possible for values of the property. This data type is referred to as the *declared data type* of the property.

In addition to simple abstract data types such as boolean or string, there are three special data types called the *nodal* data types, whose values are nodes or lists of nodes. These are described in 9.3.3.

The definition of a property may also allow that property to have a *null* value in certain circumstances, instead of a value having the declared data type. This null value is the unique object of the null data type. The null data type can never be used as a declared data type.

9.1 Nodal Properties

A property of a class may be a *subnode* property. The declared data type of a subnode property shall be nodal. When a node exhibits a value for a subnode property, all the nodes in the value of the property are in the same grove as the node exhibiting the value. The values of subnode properties of nodes in the grove can be viewed as connecting all the nodes in the grove into a single tree with labeled branches. More precisely,

— in any grove there is a unique node called the *grove root* that does not occur in the value of any subnode property of a node.

— for every node n, other than the grove root, there is a unique node o and there is a unique property p such that both

  – p is a subnode property, and

  – o exhibits a value for p that is or includes n.

  o is called the *origin* of n and p is called the *origin-to-subnode relationship* of n.

— for every node n, other than the grove root, there exists a sequence of nodes $m_1, m_2, \ldots m_k$ such that m_1 is the grove root, m_k is n, and, for each $1 \le i \le k - 1$, m_i is the origin of m_{i+1}.

This tree is referred to as the subnode tree. It is often useful for applications to deal with certain subtrees of the subnode tree in which all the children of a node occur as part of the value of a single property of the node. For this purpose, one property of the class can be distinguished as the *children property* for the class. This is done indirectly by making one property the *content property* for the class. If the data type of this property is nodal, then this is the children property, otherwise the primitive data type of the data type shall be char or string and the property is the *data property* of the node. The term *children* as applied to a node refers to the nodes occurring as the value of the children property. The data of a node that has a children property is the data of each of its children separated by the value of the *data separator property*, if any, of the class. A node has a *parent* if its origin has a children property which includes that node in its value; if a node does have a parent, its parent will be the same as its origin. The term *tree* without qualification refers to the tree formed using these parent/children relationships. The *ancestors* of a node comprise the parent of the node, if any, together with the ancestors of the parent of the node. The *tree root* of a node, x, is x if x has no ancestors or otherwise is the node that is an ancestor of x and that has no ancestors. The *siblings* of a node are an empty set for the grove root

and are otherwise the nodes in the value of the origin-to-subnode relationship property of the node's origin other than the node itself.

NOTE 23 A node can have siblings even if it does not have a parent because its origin-to-subnode relationship property need not be the children property of its origin.

The *subtree* of a node is the node together with the subtrees of its children. The *descendants* of a node are the subtrees of children of the node. A total ordering called *tree order* is defined on the set of nodes in the subtree of any node: this ordering corresponds to a pre-order traversal of the subtree in which a node is visited before its children.

There are two possibilities for properties with a declared data type that is nodal but which are not subnode properties:

— The property may be an *irefnode* (internal reference) property; for such a property the nodes in the value are in the same grove as the node that exhibits the value. The subnode and irefnode properties connect all the nodes in a grove into a single directed graph. The names of the properties may be considered as labeling the arcs of the graph.

— The property may be a *urefnode* (unrestricted reference) property; for such a property the nodes in the value may be in different groves from the node that exhibits the value. Thus, the subnode, irefnode, and urefnode properties connect the nodes in multiple groves together into a graph. The set of groves thus connected is called a *hypergrove*.

9.2 Grove Plans

A grove plan specifies a selection of classes and properties from a property set. A property set is defined by a property set definition expressed in SGML as described in 9.3.

For any source for the grove, the property set determines the *complete grove* that would be built using a grove plan that selected all the classes and properties from the property set.

NOTE 24 The complete grove contains all the information that the parser is capable of making available about the source of the grove. For any particular application, much of this information may be irrelevant. The grove plan provides a way for an application to get a grove that contains just the information it requires.

The grove to be constructed from the grove plan shall be the same as a grove obtained by modifying the complete grove in the following manner:

— To *mark the subgrove* of a node, first mark the node itself; then for each subnode property, if the property is included in the grove plan, mark the subgrove of each node in the value whose class is included in the grove plan. The nodes to be included in the grove are determined by marking the subgrove of the grove root. Only nodes thereby marked will be included in the constructed grove.

— A node in the constructed grove only exhibits values for those properties that are specified to be included in the grove.

— If a node in the complete grove exhibits a value for an intrinsic property whose semantics are that it is the name of a non-intrinsic property exhibited by the node, then if the non-intrinsic property is not included in the grove plan, the node in the constructed grove shall exhibit a null value for the intrinsic property.

— If a node in the complete grove exhibits a value for an intrinsic property whose semantics are that it is a list of names of non-intrinsic properties exhibited by the node, then the node in the constructed grove shall exhibit a value for the intrinsic property that is obtained from the value in the complete grove by removing the names of any of the non-intrinsic properties not included in the grove plan.

— If a node in the complete grove exhibits a value for an irefnode property that has a declared value of node, but the value of the property is not marked for inclusion in the constructed grove, then the node shall exhibit a null value for that property in the constructed grove.

— If a node in the complete grove exhibits a value for an irefnode property that has a declared value of nodelist or nmndlist, then the value in the constructed grove is obtained by removing from the value exhibited for the property in the complete grove all nodes that are not marked for inclusion in the constructed grove.

9.3 Property Set Definition

Property set definitions are described fully in the Property Set Definition Requirements of ISO/IEC 10744. This clause presents a simplified version that includes only those details necessary for the understanding of this International Standard.

The top-level element is a `propset` element. The `psn` and `fullnm` attributes specify a short SGML name and a long descriptive name. At various places within the property set, the following elements are allowed:

— `desc` contains a description of the object that is being defined by the element in which it occurs.

— `note` contains notes about the object being defined.

9.3.1 Common Attributes

9.3.1.1 Component Names

The name of a class, property, or enumerator is not a simple string but a triple of strings, each appropriate for use as a name in a different context:

— The reference concrete syntax (RCS) name is appropriate for use in a context where a valid name in the SGML reference concrete syntax is required.

— The application name specifies a name that is appropriate for use as an identifier in a programming or scripting language. An application name can include multiple words

separated by spaces; the name must be transformed to be a valid identifier in the language in which it is to be used, using the normal conventions of that language for multi-word identifiers. For example, the application name 'processing instruction', when bound to a programming language, might become 'ProcessingInstruction', 'processing-instruction', or 'PROCESSING_INSTRUCTION', depending on the language.

— The full name is an unabbreviated name appropriate for use in documentation.

A three-part name of this kind is called a *component name*.

These three names are specified by attributes as follows:

— `rcsnm` specifies the RCS name of the property.

— `appnm` specifies the application name of the property; this defaults to the RCS name.

— `fullnm` specifies the full name of the property; this defaults to the application name.

9.3.1.2 Specification Documents

Various elements occurring in a property set define components by referencing them in a specification document. These elements use the following common attributes:

— `sd` specifies the specification document; this defaults to `SGML`. Formally, the value is the name of a notation. Other allowed values are `GenFac` for the General Facilities of ISO/IEC 10744 and `DSSSL`.

— `clause` specifies the applicable clause of the specification document; for SGML this uses the notation of ISO/IEC 13673.

9.3.2 Modules

A property set definition is divided into named modules each described by a `psmodule` element. The attributes have the following meaning:

— `rcsnm` gives the RCS name of the module.

— `fullnm` gives the full name.

— `dependon` lists the names of the modules on which this module depends.

— `required` specifies whether the module is required, that is, shall be included in every grove plan. A value of `required` means that it is required; a value of `nrequire` means that it is not. The default is `nrequire`.

Including a module in a grove plan is equivalent to including in the grove plan:

— all the classes and properties defined within the module,

— any modules on which the module depends, and, recursively, any modules on which they depend.

In addition to modules defined in property sets, there are a number of intrinsic modules defined in this International Standard that are automatically part of every property set. Properties defined in intrinsic modules are called *intrinsic properties*. Intrinsic modules are treated as occurring before all non-intrinsic modules.

9.3.3 Data Type Definition

Every data type is defined by a `datadef` element. The attributes have the following meaning:

— `rcsnm` gives the RCS name of the data type.

NOTE 25 There is no application name for a data type, because when the property set is used in a programming or scripting language, each abstract data type has to be explicitly bound to one of the data types provided by the language.

— `fullnm` gives the full name of the data type.

— `nodal` specifies whether the data type is nodal; the allowed values are `nodal` or `nonnodal`; the default is `nonnodal`.

— `listof` allows formal specification of the semantics of a data type in the case where the data type is an ordered list or array of some other data type; that other data type is specified as the value of the attribute.

— `super` allows for the formal specification of a subtyping hierarchy among defined data types; the value of the attribute is a list of the names of the super types.

The *primitive data type* of a data type is the data type itself if the data type has no super type, and otherwise is the primitive data type of the super type.

Some data types are defined in the following intrinsic module:

```
<psmodule rcsnm=intrdt fullnm="intrinsic data types" required>
<datadef rcsnm=node nodal>
<desc>
A single node.

<datadef rcsnm=nodelist listof=node nodal>
<desc>
An ordered list of zero or more nodes.

<datadef rcsnm=nmndlist fullnm="named node list" super=nodelist nodal>
<desc>
This is a node list in which each node is uniquely identified within
the node-list by a name, which is the value of one of its properties.
A named node list identifies, for each class of node that occurs in
it, a property of that class, which has data type string, whose value
serves as the name of nodes of that class within that named node list.
```

```
In addition, a named node-list also identifies, for each class of node
that occurs in it, a normalization rule to be applied to a
string before it is compared against the name of a node of that class
in the process of name space addressing.

<datadef rcsnm=enum fullnm=enumeration>
<desc>
This is used for a data type that represents one of an enumerated set
of values, called enumerators.  The possible enumerators are
defined in each context in which the enum data type is used.

<datadef rcsnm=char fullnm=character>

<datadef rcsnm=string listof=char>

<datadef rcsnm=integer>

<datadef rcsnm=intlist fullnm="integer list" listof=integer>

<datadef rcsnm=strlist fullnm="string list" listof=string>

<datadef rcsnm=compname fullnm="component name">
<desc>
A component name, that is, a name with three variants, an RCS name,
an application name, and a full name.

<datadef rcsnm=cnmlist fullnm="component name list" listof=compname>

</psmodule>
```

9.3.4 Class Definition

A class is defined by a `classdef` element. In addition to the component name attributes and specification document attributes, the following attributes are allowed:

— `conprop` identifies the content property of the class, if any.

— `dsepprop` identifies the data separator property of the class, if any. A class can have a data separator property only if it has a children property (i.e., a nodal content property).

— `mayadd` identifies a category of classes that is used in the definition of the verification mapping in the transformation language. See 11.4.1. Only the value `mayadd` is allowed for this attribute. The attribute name can be omitted for this attribute.

9.3.5 Property Definition

A property is defined by a `propdef` element. In addition to the component name attributes and specification document attributes, the following attributes are allowed:

— `cn` specifies the class to which this property belongs. When a `propdef` element occurs within a `classdef` element, the property belongs to that class. Otherwise, the `cn` attribute

shall be specified, specifying the class name. A value of `#all` means that it belongs to all classes of node; a value of `#grove` means that it belongs to the node at the root of the grove.

— `datatype` specifies the RCS name of the data type, as defined by a datadef element.

— `ac` specifies the classes allowed in the value of the property; this applies only if the data type is nodal. The default is that any class is allowed in the value.

— `acnmprop` applies when the data type is `nmndlist` and specifies for each of the classes allowed in the property value the name of the property that serves as the name of a node of that class in the named node list. There shall be one property name for each class in `ac`.

— `strnorm` specifies a string normalization rule applicable to the value. It applies when the data type is a string, is a list of strings, or has a super type that is a string. The default is for no normalization to be applied. Each string normalization rule shall be defined by a `normdef` element.

NOTE 26 The upper-case substitution that SGML performs on general names when the reference concrete syntax is used is an example of a string normalization rule.

— `noderel` specifies whether the property is a subnode, irefnode, or urefnode property; this applies only if the data type is nodal. The attribute name is usually omitted for this attribute.

— `vrfytype` categorizes the property as either derived, optional, or other for purposes of defining the verification mapping in the transformation language. See 11.4.1. The default is other. A property set shall not allow a node in a complete grove to exhibit an empty value for a property that has a declared data type of nodelist or nmmdlist and a vrfytype of optional.

NOTE 27 This does not prohibit a node from exhibiting a null value for such a property.

— `strlex` gives a lexical type. The value is a lexical type defined by a `lexdef` element. The lexical type of a property is not used in this International Standard. The semantics of lexical types are defined in ISO/IEC 10744.

A `propdef` can have subelements of the following types in addition to `desc` and `note` elements:

— `when` specifies a condition that shall be satisfied for a node to exhibit a value with the declared data type. If this condition is not satisfied, the node shall exhibit a null value for this property.

— `enumdef` defines the possible enumerators when the data type is enum. It has only the component name attributes.

9.3.6 Normalization Rule Definition

A string normalization rule is defined by a `normdef` element. It has an `rcsnm` attribute and the specification document attributes.

9.4 Intrinsic Properties

The following module defines the intrinsic properties of all nodes:

```
<psmodule rcsnm=intrbase fullnm="intrinsic base" required>
<propdef rcsnm=classnm appnm="class name" datatype=compname>
<desc>
The name of the node's class.
<propdef cn="#all" rcsnm=grovroot appnm="grove root" datatype=node irefnode>

<propdef cn="#all" rcsnm=subpns appnm="subnode property names"
datatype=cnmlist>
<desc>
The names of all the subnode properties exhibited by the node.

<propdef cn="#all" rcsnm=allpns appnm="all property names" datatype=cnmlist>
<desc>
The names of all the properties exhibited by the node.

<propdef cn="#all" rcsnm=childpn appnm="children property name"
datatype=compname>
<desc>
The name of the children property.
<when>
The class has a children property.

<propdef cn="#all" rcsnm=datapn appnm="data property name" datatype=compname>
<when>
The class has a data property.

<propdef cn="#all" rcsnm=dseppn appnm="data sep property name"
fullnm="data separator property name" datatype=compname>
<when>
The class has a data separator property.

<propdef cn="#all" rcsnm=parent datatype=node irefnode>
<when>
The node has a parent.

<propdef cn="#all" rcsnm=treeroot appnm="tree root" datatype=node irefnode>
<note>
The value of this property for a node shall be the node itself
if the node has no parent.
</note>

<propdef cn="#all" rcsnm=origin datatype=node irefnode>
<when>
The node is not the grove root.

<propdef cn="#all" rcsnm=otsrelpn appnm="origin-to-subnode rel property name"
fullnm="origin-to-subnode relationship property name" datatype=compname>
<when>
The node is not the grove root.
</psmodule>

<psmodule rcsnm=intrhy fullnm="intrinsic hytime">
```

```
<datadef rcsnm=grovepos appnm="grove position" strlex=GROVEPOS>
<desc>
A list each of whose members is either (a) an integer, (b) a pair
consisting of a component name and a string, (c) a pair consisting of
a component name and an integer, or (d) a component name

<propdef cn="#all" rcsnm=grovepos appnm="grove position" sd=GenFac
datatype=grovepos>
<desc>
The position of a node in a grove.

<propdef cn="#all" rcsnm=treepos appnm="tree position" sd=GenFac
datatype=intlist
strlex="marker+">
<desc>
The position of a node in its tree in treeloc format.

<propdef cn="#all" rcsnm=pathpos appnm="path position" sd=GenFac
datatype=intlist
strlex="(marker,marker)+">
<desc>
The position of a node in its tree in pathloc format.
</psmodule>

<propdef cn="#grove" rcsnm=ptreert appnm="principal tree root" sd=GenFac
datatype=node
irefnode>
```

9.5 Auxiliary Groves

It is sometimes convenient to group nodes in a grove in an application-dependent manner. This is done by using nodes in the grove as the source for a further parse, called an *auxiliary parse*. A grove created by an *auxiliary parse* is called an *auxiliary grove*. The grove parsed to create the auxiliary grove is called the *source grove* of the auxiliary grove. Each node in an auxiliary grove has an intrinsic urefnode property, `source`, that points to those nodes in the source grove from which it was derived.

```
<propdef cn="#all" rcsnm=source datatype=nodelist urefnode sd=DSSSL>
```

9.6 SGML Property Set

The property set for SGML is:

```
<!-- SGML Property Set -->
<!doctype propset public "ISO/IEC 10744:1993/DTD Property Set//EN"
"sgmlprop.dtd">
<propset psn="sgmlprop" fullnm="SGML Property Set">
<desc>
Defines the classes and properties to be used in the construction of
groves from the parsing of SGML documents.

Classes and properties are classified as follows:
  o Abstract or SGML document string (SDS)
```

```
  o SGML declaration, document prolog, or document instance
  o Required only for support of datatag, rank, shortref, link, subdoc,
    formal.

ESIS corresponds roughly to the combination of baseabs (base abstract),
prlgabs0, and instabs (instance abstract).
</desc>
<!--Note: Clause numbering conforms to the rules specified in Clause 6.3
          of ISO/IEC 13673, which defines how the components of
          ISO/IEC 8879 should be identified within conformance tests.
          The first number/letter represents the clause number (letters
          can be treated as hexadecimal in this document), the second
          number identifies the sub-clause, the third the
          sub-sub-clause, and the fourth the
          sub-sub-sub-clause (if any) with the final number/letter
          identifying the paragraph number. (Productions,
          notes and items in a list are counted as separate paragraphs.)
          Where figures are referred to, the clause, sub-clause, and
          sub-sub-clause numbers are replaced by FIG and the
          sub-sub-sub-clause number is replaced by the figure number.
          As an extension to ISO/IEC 13673, subclauses in clause 4
          are referred to using numbers of the form 4xxxy where xxx
          is the decimal subclause number and y is the paragraph number
          as normal.
-->

<!-- Base abstract classes and properties -->

<psmodule rcsnm=baseabs fullnm="base abstract">

<classdef rcsnm=sgmldoc appnm="sgml document" clause="62001">
<desc>
The parsed SGML document or subdocument.  The root of the grove.

<propdef subnode rcsnm=sgmlcsts appnm="sgml constants" datatype=node
ac=sgmlcsts clause="41170 41180">

<propdef rcsnm=appinfo appnm="application info"
fullnm="application information" datatype=string strlex=mindata
clause="d6001">
<desc>
Application information provided by the SGML declaration.
<when>
A literal was specified as the value of the APPINFO parameter
of the SGML declaration applicable to the document/subdocument.

<propdef subnode rcsnm=prolog datatype=nodelist
ac="doctpdcl lktpdcl comdcl pi ssep" cn=sgmldoc clause="71001">

<propdef subnode rcsnm=epilog datatype=nodelist ac="comdcl pi ssep"
cn=sgmldoc clause="71002">
<desc>
Other prolog following the document instance.

<classdef rcsnm=sgmlcsts appnm="sgml constants" clause="b6004 c2101">
<desc>
```

```
A holding pen for selected nodes intrinsic to all SGML documents,
which may be needed as irefnodes elsewhere.
<note>
This has no properties unless the srabs (shortref abstract)
or linkabs (link abstract) modules are included.

<classdef rcsnm=attasgn appnm="attribute assignment"
conprop=value dsepprop=tokensep clause="79002">
<desc>
An attribute assignment, whether specified or defaulted.
<note>
In the base module because of data attributes.

<propdef subnode rcsnm=value datatype=nodelist
ac="attvaltk datachar sdata intignch entstart entend" clause="79401">
<note>
If the attribute value is tokenized, the children are of type attvaltk;
otherwise, they are of the other allowed types.
<when>
The attribute is not an impliable attribute for which there is no
attribute specification.

<propdef rcsnm=name datatype=string strlex=name strnorm=general
clause="93001">

<propdef rcsnm=implied datatype=boolean clause="b3407">
<desc>
True if and only if the attribute is an impliable attribute
for which there is no attribute specification.

<propdef rcsnm=tokensep appnm="token sep" fullnm="token separator"
datatype=char clause="79400">
<desc>
The separator between the tokens of the value. Always equal
to the SPACE character in the concrete syntax.
<when>
The node has two or more children of class attvaltk.

<classdef rcsnm=attvaltk appnm="attribute value token" conprop=token
clause="79305">

<propdef rcsnm=token datatype=string strlex=nmtoken clause="93003">

<classdef rcsnm=datachar appnm="data char" fullnm="data character"
conprop=char clause="92002">

<propdef rcsnm=char fullnm=character datatype=char clause="92003">
<desc>
The character returned by the parser to the application.

<classdef rcsnm=sdata
fullnm="internal specific character data entity reference result"
conprop=char clause="92101">

<propdef rcsnm=sysdata appnm="system data" datatype=string clause="43041">
<note>
```

```
The replacement text of a specific character data entity is treated
as system data when referenced.

<propdef rcsnm=char fullnm=character datatype=char sd=DSSSL>
<desc>
The character associated with the SDATA entity by the map-sdata-entity
architectural form.
<when>
A character has been associated with the SDATA entity by the
map-sdata-entity architectural form.

<classdef rcsnm=pi fullnm="processing instruction" clause="80000">
<desc>
Processing instruction.

<propdef rcsnm=sysdata appnm="system data" datatype=string clause="80002">

</psmodule>

<!-- Prolog-related abstract classes and properties, level 0 -->

<psmodule rcsnm=prlgabs0 fullnm="prolog abstract level 0" dependon=baseabs>

<propdef irefnode rcsnm=govdt appnm="governing doctype" datatype=node
ac=doctype
cn=sgmldoc clause="71004">
<desc>
The document type that governs the parse.  When there are more than one
"active" document types specified, each active document type gives rise
to a separate parse, which, in turn, creates a separate sgmldoc grove.

<propdef subnode rcsnm=dtlts appnm="doctypes and linktypes"
fullnm="document types and link types"
datatype=nmndlist ac="doctype linktype" acnmprop="name name" cn=sgmldoc
clause="71001">
<desc>
The document types and link types declared in the prolog, in declaration
order.

<classdef rcsnm=doctype appnm="document type" clause="b1000">
<desc>
The abstraction of a document type declaration.
<note>
It includes entities declared in that declaration's DTD,
entities treated as being declared therein because they
occur in a link type for which that DTD is the source DTD,
and entities declared in the base declaration which may be
referenced when this document type is active.

<propdef rcsnm=name datatype=string strlex=name strnorm=general clause="b1002">
<desc>
The name associated with the DTD by the document type declaration;
necessarily also the name of the type of the outermost element.

<propdef rcsnm=govrning appnm=governing datatype=boolean clause="71005">
<desc>
```

```
True if either this was the active document type or there was
no active document type and this is the base document type.
<note>
The "governing" document type governs the parsing process.
If more than one document type is specified as "active",
each active document type gives rise to a separate parse,
for which it is the governing document type, and thereby
produces a separate grove.

<propdef subnode rcsnm=genents appnm="general entities" datatype=nmndlist
ac=entity acnmprop=name clause="b1004">
<desc>
The general entities of the document or subdocument declared in the DTD.
<note>
Includes entities not explicitly declared, as discussed above
in the description of this class.
<note>
If the DTD provides a default declaration for undeclared
general entity names, there is no entry in the list
corresponding to this declaration, nor any entry for any
such undeclared name.  (But such entities are in the
entities property of the sgmldoc class.)  See dfltent following.

<propdef subnode rcsnm=nots appnm=notations datatype=nmndlist ac=notation
acnmprop=name clause="b1005">

<classdef rcsnm=entity clause="60000">

<propdef rcsnm=name datatype=string strlex=name strnorm=entity clause="93001">

<propdef rcsnm=enttype appnm="entity type" datatype=enum clause="a5502">

<enumdef rcsnm=text fullnm="SGML text">
<enumdef rcsnm=cdata>
<enumdef rcsnm=sdata>
<enumdef rcsnm=ndata>
<enumdef rcsnm=subdoc appnm=subdocument>
<enumdef rcsnm=pi>

<propdef rcsnm=text fullnm="replacement text" datatype=string clause="92101">
<when>
The entity is an internal entity.

<propdef subnode rcsnm=extid appnm="external id" fullnm="external identifier"
datatype=node ac=extid clause="a1601">
<when>
The entity is an external entity.

<propdef subnode rcsnm=atts appnm=attributes
datatype=nmndlist ac=attasgn acnmprop=name clause="b4120">
<desc>
A list of data attribute assignments, one for each declared attribute of
the entity in the order in which they were declared in the attribute
definition list declaration.
<when>
The entity is an external data entity.
```

```
<propdef rcsnm=notname appnm="notation name" datatype=string strlex=name
strnorm=general clause="79408">
<when>
The entity is an external data entity.

<propdef irefnode rcsnm=notation datatype=node ac=notation clause="b4001">
<when>
The entity is an external data entity.

<classdef rcsnm=notation fullnm="data content notation" clause="b4000">

<propdef rcsnm=name datatype=string strlex=name strnorm=general clause="79441">

<propdef subnode rcsnm=extid appnm="external id" fullnm="external identifier"
datatype=node ac=extid clause="a1601">

<classdef rcsnm=extid appnm="external id" fullnm="external identifier"
clause="a1600">

<propdef rcsnm=pubid appnm="public id" fullnm="public identifier"
datatype=string strlex=mindata clause="a1602">
<when>
The external identifier contained an explicit public identifier.

<propdef rcsnm=sysid appnm="system id" fullnm="system identifier"
datatype=string clause="a1603">
<when>
The external identifier contained an explicit system identifier.

<propdef optional rcsnm=gensysid appnm="generated system id"
fullnm="generated system identifier"
datatype=string>
<desc>
The system identifier generated by the system from the external
identifier and other information available to the system.
<when>
The external identifier is not the external identifier of
the default entity.
</psmodule>

<!-- Document instance related abstract classes and properties -->

<psmodule rcsnm=instabs fullnm="instance abstract" dependon=baseabs>

<propdef subnode rcsnm=docelem appnm="document element" datatype=node
ac=element cn=sgmldoc clause="72003">
<desc>
The document element for the governing document type.

<propdef irefnode rcsnm=elements datatype=nmndlist ac=element acnmprop=id
cn=sgmldoc clause="73001">
<desc>
All the elements in the document which have unique identifiers in the
order in which they are detected by the parser: parents occur
before children; siblings occur in left-to-right order.
```

```
<propdef irefnode rcsnm=entities datatype=nmndlist ac=entity acnmprop=name
cn=sgmldoc clause="94410">
<desc>
The explicitly declared general entities from the governing document
type, followed by the defaulted entities.
<note>
This includes both internal and external entities. It does not
include unnamed entities.

<propdef subnode rcsnm=dfltents appnm="defaulted entities" datatype=nmndlist
ac=entity acnmprop=name cn=sgmldoc clause="94412">
<desc>
An entity for each entity name in the document that referenced
the default entity in the governing document type.

<!-- Attribute value token -->

<propdef irefnode rcsnm=entity datatype=node ac=entity cn=attvaltk
clause="79401">
<when>
Declared value of attribute is ENTITY or ENTITIES.

<propdef irefnode rcsnm=notation datatype=node ac=notation cn=attvaltk
clause="79408">
<when>
Declared value of attribute is NOTATION.

<propdef irefnode rcsnm=referent datatype=node ac=element cn=attvaltk
clause="79403">
<when>
Declared value is IDREF or IDREFS.

<classdef rcsnm=element conprop=content clause="73000">

<propdef rcsnm=gi fullnm="generic identifier" datatype=string strlex=name
strnorm=general clause="78001">
<desc>
Generic identifier (element type name) of element.

<propdef derived rcsnm=id fullnm="unique identifier" datatype=string
strlex=name strnorm=general clause="79403">
<when>
A unique identifier was specified for the element.

<propdef subnode rcsnm=atts appnm=attributes
datatype=nmndlist ac=attasgn acnmprop=name clause="79301">
<desc>
A list of attribute assignments, one for each declared attribute
of the element in the order in which they were declared in the
attribute definition list declaration.

<propdef subnode rcsnm=content datatype=nodelist
ac="datachar sdata element extdata subdoc pi msignch ignrs ignre repos
    usemap uselink entstart entend ssep comdcl msstart msend ignmrkup"
clause="76001">
```

```
<classdef rcsnm=extdata appnm="external data"
fullnm="reference to external data" clause="a5500">
<desc>
The result of referencing an external data entity.

<propdef rcsnm=entname appnm="entity name" datatype=string strlex=name
strnorm=entity clause="a5101">

<propdef irefnode rcsnm=entity datatype=node ac=entity clause="94410">

</psmodule>

<!-- Base SDS classes and properties -->

<psmodule rcsnm=basesds0 fullnm="base SGML document string level 0"
dependon=baseabs>

<!-- Sdata -->

<propdef optional rcsnm=entname appnm="entity name" datatype=string
strlex=name strnorm=entity cn=sdata clause="a5101">

<propdef irefnode rcsnm=entity datatype=node ac=entity cn=sdata
clause="94410">

<!-- Processing instruction -->

<propdef rcsnm=entname appnm="entity name" datatype=string strlex=name
strnorm=entity cn=pi clause="a5101">
<when>
The processing instruction resulted from referencing a PI entity.

<propdef irefnode rcsnm=entity datatype=node ac=entity cn=pi
clause="94410">
<when>
The processing instruction resulted from referencing a PI entity.

<!-- Entity -->

<propdef rcsnm=dflted appnm=defaulted datatype=boolean cn=entity
clause="94412">
<desc>
True if this was created because of a reference to the default entity.

</psmodule>

<psmodule rcsnm=basesds1 fullnm="base SGML document string level 1"
dependon=basesds0>

<propdef subnode optional rcsnm=entref appnm="entity ref"
fullnm="entity reference" datatype=nodelist
ac="gendelm name ssep entstart entend refendre shortref" cn=pi
clause="94401">
<desc>
The markup of the entity reference.
```

```
<note>
ssep, entstart, and entend may occur only in a name group in a named
entity reference.
<when>
The processing instruction resulted from referencing a PI entity with
a named entity reference or a short reference.

<propdef subnode optional rcsnm=open appnm="open delim"
fullnm="open delimiter" datatype=node ac=gendelm cn=pi clause="80001">
<when>
The processing instruction did not result from referencing a PI entity.

<propdef subnode optional rcsnm=close appnm="close delim"
fullnm="close delimiter" datatype=node ac=gendelm cn=pi clause="80001">
<when>
The processing instruction did not result from referencing a PI entity.

<!-- Attribute -->

<propdef irefnode rcsnm=attspec appnm="attribute spec" fullnm="attribute
specification"
datatype=nodelist ac="name ssep gendelm literal attvalue" cn=attasgn
clause="79002">
<when>
The attribute was specified rather than defaulted or implied.

<propdef irefnode rcsnm=attvalsp appnm="attribute value spec"
fullnm="attribute value specification" datatype=node
ac="attvalue literal" cn=attasgn clause="79301">
<when>
The attribute is not implied.

<!-- Data character -->

<propdef rcsnm=intrplch appnm="interp replaced char"
fullnm="interpretation replaced character" datatype=char cn=datachar
clause="a1704">
<desc>
The character that was replaced.
<note>
When a sequence of RE and/or SPACE characters in a minimum literal
is replaced by a single SPACE character, then the first
character is represented by a datachar possibly with an intrplch
property, and the other characters are represented by an intignch.
<when>
The data character replaced another character
when a literal was interpreted: a SPACE character that replaced a
RE or SEPCHAR in an attribute value literal or an RE in a minimum
literal.

<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" cn=datachar clause="95001">
<when>
The data character was the replacement of a named character reference.
```

```
<propdef subnode optional rcsnm=numcref appnm="numeric char ref"
fullnm="numeric character reference" datatype=nodelist
ac="gendelm name crefcnum refendre" cn=datachar clause="95001">
<when>
The data character was the replacement of a numeric character reference.

<!-- Specific character data -->

<propdef subnode optional rcsnm=markup datatype=nodelist
ac="gendelm name ssep entstart entend refendre shortref" cn=sdata
clause="94401">
<note>
ssep, entstart, and entend can occur only in a name group in a named
entity reference.

<classdef rcsnm=ssep appnm="s sep" fullnm="s separator" mayadd
clause="62100">

<propdef rcsnm=char fullnm=character datatype=char clause="92003">

<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" clause="95001">
<when>
The character was the replacement of a named character reference.

<classdef rcsnm=comment clause="a3002">

<propdef subnode optional rcsnm=open appnm="open delim"
fullnm="open delimiter" datatype=node ac=gendelm clause="a3002">

<propdef rcsnm=chars fullnm=characters datatype=string clause="92101">
<desc>
The characters in the comment (excluding the com delimiters).

<propdef subnode optional rcsnm=close appnm="close delim"
fullnm="close delimiter" datatype=node ac=gendelm clause="a3002">

<classdef rcsnm=comdcl appnm="comment decl" fullnm="comment declaration"
conprop=markup mayadd clause="a3001">

<propdef subnode rcsnm=markup datatype=nodelist ac="comment ssep"
clause="a3001">

<classdef rcsnm=ignmrkup appnm="ignored markup" conprop=markup
clause="77002 94405 c3007">
<desc>
Ignored markup.  Either a start-tag or end-tag that is ignored because
it contains a document type specification that contains a name group
none of the names in which is the name of an active document type, or
a general or parameter entity reference that is ignored because it
contains a name group none of the names in which is the name of an
active document or link type, or a link set use declaration that is
ignored because its link type name is not an active link type,
or a general entity reference in an attribute value literal in
a start-tag that is itself ignored markup, or an entity declaration
```

```
that is ignored because the entity was already declared.

<propdef subnode rcsnm=markup datatype=nodelist
ac="gendelm name ssep attvalue literal entstart entend refendre"
clause="74001 75001 94401 c3001">

<classdef rcsnm=entstart appnm="entity start" conprop=markup>
<desc>
The start of the replacement text of an entity.
<note>
The end shall be marked by an entend node. This is the result of an
entity reference that was replaced by the parser.

<propdef subnode optional rcsnm=markup datatype=nodelist
ac="gendelm name ssep entstart entend refendre shortref">
<desc>
The markup of the entity reference.

<propdef optional rcsnm=entname appnm="entity name" datatype=string
strlex=name strnorm=entity>

<propdef irefnode rcsnm=entity datatype=node ac=entity clause="a5201">

<classdef rcsnm=entend appnm="entity end" clause="94500">
<desc>
The end of an entity reference that was replaced by the parser.

<classdef rcsnm=msignch appnm="marked section ignored char"
fullnm="marked section ignored character" clause="a4204">
<desc>
A character that has been ignored within a marked section.

<propdef rcsnm=char fullnm=character datatype=char clause="92101">

<classdef rcsnm=intignch appnm="interp ignored char"
fullnm="interpretation ignored char" clause="79303 a1704">
<desc>
A character in a literal that was ignored when the literal was
interpreted: an RS in an attribute value literal or in a minimum literal,
an RE or SPACE character in a minimum literal that immediately
followed another RE or SPACE character in a minimum literal,
or an RE or SPACE character that was the first or last character
in a minimum literal.

<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" clause="95001">
<when>
The character was the replacement of a named character reference.

<propdef rcsnm=char fullnm=character datatype=char clause="92101">

<classdef rcsnm=gendelm appnm="general delim" fullnm="general delimiter"
clause="FIG30">
<desc>
A general delimiter.
```

```
<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" clause="95001">
<note>
This may happen only for a delimiter that is the first child
of its parent or the value of a close delimiter property.
<when>
The first character of the delimiter was entered with a named
character reference.

<propdef rcsnm=role datatype=string strnorm=rcsgener clause="96001 FIG30">
<desc>
The name of the delimiter role.

<propdef optional rcsnm=origdelm appnm="original delim"
fullnm="original delimiter" datatype=string clause="92102 FIG22">
<desc>
The delimiter as originally entered before any upper-case substitution.

<classdef rcsnm=name clause="93001">
<desc>
A name within markup.
<note>
Names in attribute values are represented by nodes of type attvaltk
rather than name.

<propdef rcsnm=origname appnm="original name" datatype=string clause="93005">
<desc>
The characters of the name as originally entered before
any upper-case substitution.

<classdef rcsnm=rname appnm="reserved name" clause="d4701">
<desc>
A token in markup that is recognized as a reserved name.

<propdef rcsnm=refname appnm="ref name" fullnm="reference name"
datatype=string strnorm=rcsgener clause="d4704">
<desc>
The reference reserved name.

<propdef optional rcsnm=origname appnm="original name" datatype=string
clause="93005">
<desc>
The reserved name as originally entered before any upper-case
substitution.

<classdef rcsnm=literal conprop=value clause="a1201 79302 a1701 a1603">
<desc>
A parameter literal, attribute value literal, minimum literal, or
system identifier.

<propdef subnode optional rcsnm=open appnm="open delim"
fullnm="open delimiter" datatype=node ac=gendelm clause="96100 FIG30">

<propdef subnode rcsnm=value datatype=nodelist
```

```
ac="entstart entend datachar sdata intignch"
clause="a1202 91001 a1702 80002">
<desc>
Interpreted value of literal.
<note>
If the literal is an attribute value literal for a tokenized value,
the value of the literal represents the attribute value before
tokenization but after interpretation.

<propdef subnode optional rcsnm=close appnm="close delim"
fullnm="close delimiter" datatype=node ac=gendelm clause="96100 FIG30">

<classdef rcsnm=number clause="93002">
<desc>
A number in markup that is not a character number in
a character reference.
<note>
Numbers in attribute values are represented by nodes of type attvaltk
rather than number.

<propdef rcsnm=digits datatype=string strlex=number clause="93002">

<classdef rcsnm=crefcnum appnm="char ref char number"
fullnm="character reference character number" clause="95001">
<desc>
A character number occurring in a character reference.
<note>
The numeric value of the number is determined by the char property of
the datachar node.

<propdef optional rcsnm=ndigits appnm="n digits" fullnm="number of digits"
datatype=integer clause="95003 93002">
<desc>
The number of digits used to specify the value.

<classdef rcsnm=refendre appnm="ref end re" fullnm="reference end RE"
clause="94502">
<desc>
An RE in markup that is used as a reference end.

<classdef rcsnm=attvalue appnm="attribute value" clause="79400">
<desc>
An attribute value specification that is an attribute value
rather than an attribute value literal.
<note>
Do not confuse this with the attasgn class.

<propdef rcsnm=value datatype=string clause="93005">
<desc>
The value before any upper-case substitution.

<classdef rcsnm=nmtoken appnm="name token" clause="93003">
<desc>
A name token in markup.
<note>
This is used only for name tokens in name token groups in
```

```
declared values. Name tokens in attribute values are represented by
nodes of type attvaltk rather than nmtoken.

<propdef rcsnm=origname appnm="original name token" datatype=string
clause="93005">
<desc>
The characters of the name token as originally entered before
any upper-case substitution.

<classdef rcsnm=msstart appnm="marked section start"
fullnm="marked section declaration start" conprop=markup clause="a4002">
<desc>
The part of a marked section declaration preceding the marked section.

<propdef subnode optional rcsnm=markup datatype=nodelist
ac="gendelm rname ssep entstart entend comment ignmrkup" clause="a4002">
<note>
First child will be gendelm for mdo, last will be gendelm for
dso.

<propdef rcsnm=status datatype=enum clause="a4201">
<desc>
Effective status of marked section.

<enumdef rcsnm=ignore>
<enumdef rcsnm=cdata>
<enumdef rcsnm=rcdata>
<enumdef rcsnm=include>
<enumdef rcsnm=temp>

<classdef rcsnm=msend appnm="marked section end" conprop=markup
clause="a4003">

<propdef subnode optional rcsnm=markup datatype=nodelist ac=gendelm
clause="FIG3e FIG3h">
<note>
Will be a gendelm for the msc and a gendelm for the mdc.

</psmodule>

<!-- SGML Declaration-related abstract classes and properties -->

<psmodule rcsnm=sdclabs fullnm="sgml declaration abstract" dependon=baseabs>

<propdef rcsnm=sgmlver appnm="sgml version" datatype=string strlex=mindata
cn=sgmldoc clause="d0002">
<desc>
The minimum literal specified as the first parameter of the SGML
declaration applicable to this document or subdocument.

<propdef subnode rcsnm=docchset appnm="document char set"
fullnm="document character set" datatype=node ac=charset cn=sgmldoc
clause="d1001">

<propdef subnode rcsnm=capset appnm="capacity set" datatype=node
ac=capset cn=sgmldoc clause="d2001">
```

```
<propdef rcsnm=synscope appnm="syntax scope"
fullnm="concrete syntax scope" datatype=enum cn=sgmldoc clause="d3002">

<enumdef rcsnm=instance>
<enumdef rcsnm=document>

<propdef subnode rcsnm=dclsyn appnm="decl syntax"
fullnm="declared concrete syntax" datatype=node ac=syntax cn=sgmldoc
clause="d4001">

<propdef subnode rcsnm=refsyn appnm="ref syntax"
fullnm="reference concrete syntax" datatype=node ac=syntax cn=sgmldoc
clause="d4002 e0001 FIG70">
<desc>
The reference concrete syntax used for the SGML declaration and,
if the concrete syntax scope is INSTANCE, the prolog.
<note>
Not a property of sgmlcsts because it depends on the document character
set.

<propdef irefnode rcsnm=prosyn appnm="prolog syntax"
fullnm="prolog concrete syntax" datatype=node ac=syntax cn=sgmldoc
clause="d4001">
<desc>
The concrete syntax for the prolog.

<propdef subnode rcsnm=features fullnm="feature use" datatype=node
ac=features cn=sgmldoc clause="d5001">

<classdef rcsnm=charset appnm="char set" fullnm="character set"
conprop=chdescs clause="d1000">

<propdef subnode rcsnm=chdescs appnm="char descs"
fullnm="character descriptions" datatype=nodelist ac=chardesc
clause="d1101">

<classdef rcsnm=chardesc appnm="char desc" fullnm="character description"
clause="d1122">

<propdef rcsnm=descnum appnm="desc set number"
fullnm="described set character number" datatype=integer clause="d1123">

<propdef rcsnm=nchars appnm="n chars" fullnm="number of characters"
datatype=integer clause="d1125">

<propdef rcsnm=basenum appnm="base set number"
fullnm="base set character number" datatype=integer clause="d1124">
<when>
Character description included a base set character number.

<propdef rcsnm=baseset appnm="base char set" fullnm="base character set"
datatype=string strlex=mindata clause="d1111">
<desc>
The public identifier of the base character set.
<when>
```

```
Character description included a base set character number.

<propdef rcsnm=desclit appnm="desc literal"
fullnm="description literal" datatype=string strlex=mindata
clause="a1701">
<when>
Character description not entered as base set number.

<classdef rcsnm=syntax fullnm="concrete syntax" clause="d4000">
<note>
This represents a concrete syntax bound to this document's document
character set. Characters are characters in the document character set
not in the syntax reference character set.

<propdef rcsnm=shunctrl appnm="shunchar controls" datatype=boolean
clause="d4204">
<desc>
True if SHUNCHAR included CONTROLS.

<propdef rcsnm=shunchar fullnm="shunned character numbers"
datatype=intlist clause="d4201">

<propdef subnode rcsnm=synchset appnm="syntax ref char set"
fullnm="syntax-reference character set" datatype=node ac=charset
clause="d4301">

<propdef rcsnm=re fullnm="record end" datatype=char clause="d4401">

<propdef rcsnm=rs fullnm="record start" datatype=char clause="d4401">

<propdef rcsnm=space datatype=char clause="d4401">

<propdef subnode rcsnm=addfuns appnm="added function chars"
fullnm="added function characters" datatype=nmndlist ac=addfun
acnmprop=name clause="d4401">

<propdef rcsnm=lcnmstrt datatype=string clause="d4503">

<propdef rcsnm=ucnmstrt datatype=string clause="d4504">

<propdef rcsnm=lcnmchar datatype=string clause="d4505">

<propdef rcsnm=ucnmchar datatype=string clause="d4506">

<propdef rcsnm=substgen appnm="subst general names"
fullnm="substitute general names" datatype=boolean clause="d4507">
<desc>
True if GENERAL YES is specified in NAMECASE.

<propdef rcsnm=substent appnm="subst entity names"
fullnm="substitute entity names" datatype=boolean clause="d4507">
<desc>
True if ENTITY YES is specified in NAMECASE.

<propdef subnode rcsnm=gdasns appnm="general delim assocs"
fullnm="general delimiter role associations"
```

```
datatype=nmndlist ac=dlmrlas acnmprop=role clause="d4611">
<desc>
There is a term for every general delimiter role whether or not
it is changed from that prescribed by the reference concrete syntax.
The terms occur in alphabetical order of their (abstract-syntax)
role names.

<propdef rcsnm=srdelms appnm="shortref delims"
fullnm="short reference delimiters" datatype=strlist clause="d4621">

<propdef subnode rcsnm=slitasns appnm="syntax literal assocs"
fullnm="syntax literal associations" datatype=nmndlist ac=synlitas
acnmprop=refname clause="d4701">
<desc>
The syntax literal/reserved name associations specified by the concrete
syntax. There is a term for every reserved name whether or not
it is changed from that prescribed by the reference concrete syntax.
The terms occur in alphabetical order of the syntactic literals.

<propdef rcsnm=attcnt datatype=integer clause="FIG41">
<propdef rcsnm=attsplen datatype=integer clause="FIG42">
<propdef rcsnm=bseqlen datatype=integer clause="FIG43">
<propdef rcsnm=dtaglen datatype=integer clause="FIG44">
<propdef rcsnm=dtemplen datatype=integer clause="FIG45">
<propdef rcsnm=entlvl datatype=integer clause="FIG46">
<propdef rcsnm=grpcnt datatype=integer clause="FIG47">
<propdef rcsnm=grpgtcnt datatype=integer clause="FIG48">
<propdef rcsnm=grplvl datatype=integer clause="FIG49">
<propdef rcsnm=litlen datatype=integer clause="FIG4a">
<propdef rcsnm=namelen datatype=integer clause="FIG4b">
<propdef rcsnm=normsep datatype=integer clause="FIG4c">
<propdef rcsnm=pilen datatype=integer clause="FIG4d">
<propdef rcsnm=taglen datatype=integer clause="FIG4e">
<propdef rcsnm=taglvl datatype=integer clause="FIG4f">

<classdef rcsnm=addfun appnm="added function char"
fullnm="added function character" clause="d4400">

<propdef rcsnm=name datatype=string strlex=name strnorm=general
clause="d4402">

<propdef rcsnm=class fullnm="function class" datatype=enum clause="d4403">
<enumdef rcsnm=funchar>
<enumdef rcsnm=msichar>
<enumdef rcsnm=msochar>
<enumdef rcsnm=msschar>
<enumdef rcsnm=sepchar>

<propdef rcsnm=char fullnm=character datatype=char clause="95003">
<desc>
Character assigned to function.

<classdef rcsnm=dlmrlas appnm="delim role assoc"
fullnm="delimiter role association" clause="d4610">
<desc>
The association, made by a concrete syntax, of a character string with
```

```
an abstract-syntax delimiter role.

<propdef rcsnm=role datatype=string strnorm=rcsgener clause="d4612">
<desc>
The name of the role.

<propdef rcsnm=delm appnm=delim fullnm=delimiter datatype=string
strnorm=general clause="d4611">
<desc>
The string to be used in the document.

<classdef rcsnm=synlitas appnm="syntactic literal assoc"
fullnm="syntactic literal association" clause="d4700">
<desc>
The association, made by a concrete syntax, of a reserved name with
an abstract-syntax syntactic literal.

<propdef rcsnm=synlit appnm="syntactic literal"
datatype=string strnorm=rcsgener clause="d4702">
<desc>
The syntactic literal.  (More precisely, the name which when enclosed in
double quotation marks becomes the syntactic literal.)

<propdef rcsnm=resname appnm="reserved name" datatype=string strlex=name
strnorm=general clause="d4702">
<desc>
The reserved name to be used in the document.
<note>
In the reference concrete syntax, the syntactic literal is
identical to the reserved name.

<classdef rcsnm=capset appnm="capacity set" clause="d2000">

<propdef rcsnm=totalcap datatype=integer clause="FIG51">
<propdef rcsnm=entcap datatype=integer clause="FIG52">
<propdef rcsnm=entchcap datatype=integer clause="FIG53">
<propdef rcsnm=elemcap datatype=integer clause="FIG54">
<propdef rcsnm=grpcap datatype=integer clause="FIG55">
<propdef rcsnm=exgrpcap datatype=integer clause="FIG56">
<propdef rcsnm=exnmcap datatype=integer clause="FIG57">
<propdef rcsnm=attcap datatype=integer clause="FIG58">
<propdef rcsnm=attchcap datatype=integer clause="FIG59">
<propdef rcsnm=avgrpcap datatype=integer clause="FIG5a">
<propdef rcsnm=notcap datatype=integer clause="FIG5b">
<propdef rcsnm=notchcap datatype=integer clause="FIG5c">
<propdef rcsnm=idcap datatype=integer clause="FIG5d">
<propdef rcsnm=idrefcap datatype=integer clause="FIG5e">
<propdef rcsnm=mapcap datatype=integer clause="FIG5f">
<propdef rcsnm=lksetcap datatype=integer clause="FIG5g">
<propdef rcsnm=lknmcap datatype=integer clause="FIG5h">

<classdef rcsnm=features fullnm="feature use" clause="d5000">

<propdef rcsnm=datatag datatype=boolean clause="d5101">
<desc>
True if DATATAG is YES.
```

```
<propdef rcsnm=omittag datatype=boolean clause="d5101">
<desc>
True if OMITTAG is YES.

<propdef rcsnm=rank datatype=boolean clause="d5101">
<desc>
True if RANK is YES.

<propdef rcsnm=shorttag datatype=boolean clause="d5101">
<desc>
True if SHORTTAG is YES.

<propdef rcsnm=simple datatype=integer clause="d5201">
<desc>
0 if SIMPLE is NO.

<propdef rcsnm=implicit datatype=boolean clause="d5201">
<desc>
True if IMPLICIT is YES.

<propdef rcsnm=explicit datatype=integer clause="d5201">
<desc>
0 if EXPLICIT is NO.

<propdef rcsnm=concur datatype=integer clause="d5301">
<desc>
0 if CONCUR is NO.

<propdef rcsnm=subdoc datatype=integer clause="d5301">
<desc>
0 if SUBDOC is NO.

<propdef rcsnm=formal datatype=boolean clause="d5301">
<desc>
True if FORMAL is YES.

</psmodule>

<!-- SGML Declaration-related SGML document string classes and properties -->

<psmodule rcsnm=sdclsds fullnm="SGML declaration SGML document string"
dependon=basesds1>

<propdef subnode optional rcsnm=sgmldcl appnm="sgml decl"
fullnm="SGML declaration" datatype=node ac=sgmldcl cn=sgmldoc
clause="d0001">
<when>
SGML declaration was explicitly present.

<propdef rcsnm=sdcltype appnm="sgml decl type"
fullnm="SGML declaration type" datatype=enum cn=sgmldoc clause="62300">

<enumdef rcsnm=explicit>
<desc>
The SGML declaration was explicitly specified.
```

```
<enumdef rcsnm=implied>
<desc>
The SGML declaration was implied.

<enumdef rcsnm=inherit>
<desc>
The SGML declaration comes from the SGML document of which
this is a subdocument.

<classdef rcsnm=sgmldcl appnm="sgml decl" fullnm="SGML declaration"
conprop=markup clause="d0000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="ssep comment name number rname literal gendelm" clause="d0001">
<note>
Also includes any s separators before the SGML declaration;
last child is gendelm for mdc delimiter.

</psmodule>

<!-- Prolog-related abstract classes and properties, level 1 -->

<psmodule rcsnm=prlgabs1 fullnm="prolog abstract level 1"
dependon=prlgabs0>

<propdef subnode rcsnm=attdefs appnm="attribute defs"
fullnm="attribute definitions" datatype=nmndlist ac=attdef acnmprop=name
cn=notation clause="b3002">

<propdef irefnode rcsnm=attdef appnm="attribute def"
fullnm="attribute definition" datatype=node ac=attdef cn=attasgn
clause="b3003">

<propdef irefnode rcsnm=elemtype appnm="element type" datatype=node ac=elemtype
cn=element clause="b2101">

<propdef subnode rcsnm=dfltent appnm="default entity" datatype=node ac=dfltent
clause="a5105" cn=doctype>
<when>
The DTD declared a default for undeclared entity names.  (Each such
undeclared name is associated with an entity using this node as
a pattern, but in certain cases, the system may not select the
same entity for each name.)

<propdef subnode rcsnm=elemtps appnm="element types" datatype=nmndlist
ac="elemtype rankstem" acnmprop="gi rankstem" cn=doctype clause="b2101" >
<desc>
Generic identifiers or rank stems used to name elements.

<propdef subnode rcsnm=parments appnm="parameter entities"
datatype=nmndlist ac=entity acnmprop=name cn=doctype
clause="b1004" >
<note>
Includes entities not explicitly declared, as discussed above in
the description of this class.
```

```
<classdef rcsnm=elemtype appnm="element type"
fullnm="element type definition" clause="b2000">

<propdef rcsnm=gi fullnm="generic identifier" datatype=string
strlex=name strnorm=general clause="78002">

<propdef rcsnm=omitstrt appnm="omit start tag" datatype=boolean
clause="b2202">
<desc>
True if start-tag minimization was "O".
<when>
Element type declaration specified omitted tag minimization.

<propdef rcsnm=omitend appnm="omit end tag" datatype=boolean
clause="b2203">
<desc>
True if end-tag minimization was "O".
<when>
Element type declaration specified omitted tag minimization.

<propdef rcsnm=contype appnm="content type" datatype=enum clause="b2300">

<enumdef rcsnm=cdata>
<desc>
Declared content of CDATA.

<enumdef rcsnm=rcdata>
<desc>
Declared content of RCDATA.

<enumdef rcsnm=empty>
<desc>
Declared content of EMPTY.

<enumdef rcsnm=any>
<desc>
Content model of ANY.

<enumdef rcsnm=modelgrp appnm="model group">
<desc>
Content model that is a model group.

<propdef subnode rcsnm=modelgrp appnm="model group" datatype=node
ac=modelgrp clause="b2402">
<when>
Element type declaration includes content model that has a model group.

<propdef rcsnm=excls appnm=exclusions datatype=strlist clause="b2521">
<when>
Contype is any or modelgrp.

<propdef rcsnm=incls appnm=inclusions datatype=strlist clause="b2511">
<when>
Contype is any or modelgrp.
```

```
<propdef subnode rcsnm=attdefs appnm="attribute defs"
fullnm="attribute definitions" datatype=nmndlist ac=attdef acnmprop=name
clause="b3003">

<classdef rcsnm=modelgrp appnm="model group" conprop=tokens
clause="b2402">
<desc>
A model group or a data tag group.
<note>
A data tag group is represented by a model group node with connector
equal to seq whose first token is an elemtk and whose second token
is a pcdatatk.

<propdef rcsnm=connect appnm=connector datatype=enum clause="b2410">
<desc>
Connector used within model group.

<enumdef rcsnm=and>
<enumdef rcsnm=or>
<enumdef rcsnm=seq>

<propdef rcsnm=occur appnm="occur indicator" fullnm="occurrence indicator"
datatype=enum clause="b2420">
<when>
Model group has an occurrence indicator.

<enumdef rcsnm=opt>
<enumdef rcsnm=plus>
<enumdef rcsnm=rep>

<propdef subnode rcsnm=tokens appnm="content tokens" datatype=nodelist
ac="modelgrp pcdatatk elemtk" clause="b2403">

<classdef rcsnm=pcdatatk appnm="pcdata token" clause="b2404">

<classdef rcsnm=elemtk appnm="element token" clause="b2405">

<propdef rcsnm=gi fullnm="generic identifier" datatype=string
strlex=name strnorm=general clause="b2405">

<propdef rcsnm=occur appnm="occur indicator" fullnm="occurrence indicator"
datatype=enum clause="b2405">
<when>
Element token has an occurrence indicator.

<enumdef rcsnm=opt>
<enumdef rcsnm=plus>
<enumdef rcsnm=rep>

<classdef rcsnm=attdef appnm="attribute def" fullnm="attribute definition"
conprop=dfltval clause="b3003">

<propdef rcsnm=name datatype=string strlex=name strnorm=general
clause="b3201">

<propdef rcsnm=dcltype appnm="decl value type"
```

```
fullnm="declared value prescription type" datatype=enum clause="b3301">

<enumdef rcsnm=cdata>
<enumdef rcsnm=entity>
<enumdef rcsnm=entities>
<enumdef rcsnm=id>
<enumdef rcsnm=idref>
<enumdef rcsnm=idrefs>
<enumdef rcsnm=name>
<enumdef rcsnm=names>
<enumdef rcsnm=nmtoken>
<enumdef rcsnm=nmtokens>
<enumdef rcsnm=number>
<enumdef rcsnm=numbers>
<enumdef rcsnm=nutoken>
<enumdef rcsnm=nutokens>
<enumdef rcsnm=notation>
<enumdef rcsnm=nmtkgrp appnm="name token group">
<desc>
The declared value was a name token group.

<propdef rcsnm=tokens datatype=strlist clause="b3301">
<desc>
A list of strings specifying the allowed tokens.
<when>
Declared value is a name token group or a notation.

<propdef rcsnm=dflttype appnm="default value type" datatype=enum
clause="b3401">

<enumdef rcsnm=value>
<desc>
The default value was an attribute value specification without #FIXED.

<enumdef rcsnm=fixed>
<enumdef rcsnm=required>
<enumdef rcsnm=current>
<enumdef rcsnm=conref>
<enumdef rcsnm=implied>

<propdef subnode rcsnm=dfltval appnm="default value" datatype=nodelist
ac="attvaltk datachar sdata intignch entstart entend" clause="b3409">
<when>
The default value includes an attribute value specification.

<propdef irefnode rcsnm=curgrp appnm="current group" datatype=nodelist
ac=attdef clause="b3001">
<desc>
All the attdef nodes that represent the same attribute definition
and which will therefore share the same current value.
<note>
There will be as many members as there were associated element types
in the attribute definition list declaration
that declared this attribute definition.
<when>
The default value type is CURRENT.
```

```
<propdef rcsnm=curattix appnm="current attribute index" datatype=integer
clause="b3001">
<desc>
The number of preceding attribute definitions in the document type
declaration with a default value type of CURRENT.
<note>
All the attdef nodes in the value of the curgrp property of an attdef
node will exhibit the same value for the curattix property.
Two attdef nodes will share the same current value just in case they
exhibit the same value for the curattix property.
<when>
The default value type is CURRENT.

<classdef rcsnm=dfltent appnm="default entity">

<propdef rcsnm=enttype appnm="entity type" datatype=enum clause="a5502">

<enumdef rcsnm=text fullnm="SGML text">
<enumdef rcsnm=cdata>
<enumdef rcsnm=sdata>
<enumdef rcsnm=ndata>
<enumdef rcsnm=subdoc appnm=subdocument>
<enumdef rcsnm=pi>

<propdef rcsnm=text datatype=string fullnm="replacement text"
clause="92101">
<when>
The default entity declaration declares an internal entity.

<propdef subnode rcsnm=extid appnm="external id"
fullnm="external identifier" datatype=node ac=extid clause="a1601">
<when>
The default entity declaration declares an external entity.

<propdef subnode rcsnm=atts appnm=attributes
datatype=nmndlist ac=attasgn acnmprop=name clause="b4120">
<desc>
A list of data attribute assignments, one for each declared attribute of the
entity in the order in which they were declared in the attribute
definition list declaration.
<when>
The default entity declaration declares an external entity.

<propdef rcsnm=notname appnm="notation name" datatype=string strlex=name
strnorm=general clause="79408">
<when>
The default entity declaration declares an external entity.

<propdef irefnode rcsnm=notation datatype=node ac=notation clause="b4001">
<when>
The default entity declaration declares an external entity.

</psmodule>

<!-- Prolog-related SDS classes and properties -->
```

```
<psmodule rcsnm=prlgsds fullnm="prolog SGML document string"
dependon=basesds1>

<propdef irefnode rcsnm=entdcl appnm="entity decl"
fullnm="entity declaration" datatype=node ac=entdcl cn=entity
clause="a5001">

<propdef irefnode rcsnm=entdcl appnm="entity decl"
fullnm="entity declaration" datatype=node ac=entdcl cn=dfltent
clause="a5001">

<propdef irefnode rcsnm=notdcl appnm="notation decl"
fullnm="notation declaration" datatype=node ac=notdcl cn=notation
clause="b4001">

<propdef irefnode rcsnm=attdldcl appnm="attribute def list decl"
fullnm="attribute definition list declaration" datatype=node ac=attdldcl
cn=notation clause="b4111">
<when>
The notation has an associated ATTLIST.

<propdef irefnode rcsnm=eltpdcl appnm="element type decl"
fullnm="element type declaration" datatype=node ac=eltpdcl cn=elemtype
clause="b2001">

<propdef irefnode rcsnm=attdldcl appnm="attribute def list decl"
fullnm="attribute definition list declaration"
datatype=node ac=attdldcl cn=elemtype clause="b3001">
<when>
The element type has an associated ATTLIST declaration.

<propdef irefnode rcsnm=doctpdcl fullnm="document type declaration"
datatype=node ac=doctpdcl cn=doctype clause="b1001">

<propdef irefnode rcsnm=attvalsp appnm="attribute value spec"
fullnm="attribute value specification"
datatype=node ac="attvalue literal" cn=attdef clause="79002">
<when>
Default value includes attribute value specification.

<classdef rcsnm=doctpdcl fullnm="document type declaration" mayadd
clause="b1000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="ssep comment name rname literal msstart msend msignch entstart entend
    comdcl pi eltpdcl entdcl notdcl attdldcl usemap srmapdcl"
    clause="b1001">
<note>
First child is gendelm for mdo delimiter; last is gendelm
for mdc delimiter. If there is an external entity, its entend node
will appear immediately before the gendelm for the dsc delimiter,
if there is one, and otherwise immediately before the gendelm node
for the mdc delimiter.

<propdef irefnode rcsnm=doctype appnm="document type" datatype=node
```

```
ac=doctype clause="b1008">

<propdef subnode rcsnm=entity datatype=node ac=entity clause="b1008">
<when>
Document type declaration includes external identifier.

<classdef rcsnm=attdldcl appnm="attribute def list decl"
fullnm="attribute definition list declaration" mayadd clause="b3000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="ssep comment entstart entend gendelm name nmtoken attvalue literal"
clause="b3001">

<propdef irefnode rcsnm=asseltps appnm="assoc element types"
fullnm="associated element types" datatype=nodelist ac=elemtype
clause="b3001">
<desc>
The element types to which the attribute definition list is applicable,
ordered as their names occur in the attribute definition
list declaration. This does not include undefined element types.

<propdef irefnode rcsnm=assnots appnm="assoc notations"
fullnm="associated notations" datatype=nodelist ac=notation clause="b3001">

<classdef rcsnm=eltpdcl appnm="element type decl"
fullnm="element type declaration" mayadd clause="b2000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="ssep comment entstart entend gendelm name number" clause="b2001">

<propdef irefnode rcsnm=elemtype appnm="element type"
fullnm="element type" datatype=node ac=elemtype clause="b2101">

<classdef rcsnm=entdcl appnm="entity decl" fullnm="entity declaration"
mayadd clause="a5000">
<desc>
An entity declaration that is not ignored.

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment gendelm name rname literal attvalue"
clause="a5001">

<propdef subnode rcsnm=entity datatype=node ac=entity clause="a5201">
<desc>
The entity declared by the entity declaration.

<classdef rcsnm=notdcl appnm="notation decl"
fullnm="notation declaration" mayadd clause="b4000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment literal name rname" clause="b4001">

<propdef irefnode rcsnm=notation datatype=node ac=notation clause="b4001">
<desc>
The declared notation.
```

```
</psmodule>

<!-- Document instance-related SDS classes and properties -->

<psmodule rcsnm=instsds0 fullnm="instance SGML document string level 0">

<propdef derived rcsnm=included datatype=boolean cn=element>
<desc>
True if and only if the element was an included subelement.

<propdef derived rcsnm=momitend appnm="must omit end tag" datatype=boolean
cn=element clause="b2209">
<desc>
True if and only if the end tag for the element had to be omitted
because the element had a declared content of empty or
an explicit content reference.

</psmodule>

<psmodule rcsnm=instsds1 fullnm="instance SGML document string level 1"
dependon="instsds0 basesds1">

<!-- Element -->

<propdef subnode optional rcsnm=starttag appnm="start tag" datatype=nodelist
ac="gendelm name ssep entstart entend literal attvalue" cn=element
clause="74001">
<note>
First child is gendelm for stago.
Nodes of type entstart and entend can occur only
in the document type specification.
<when>
A start-tag was specified for the element.

<propdef subnode optional rcsnm=endtag appnm="end tag" datatype=nodelist
ac="gendelm name ssep entstart entend ignmrkup" cn=element clause="75001">
<note>
First child is gendelm for etago or net. Nodes of type entstart,
entend, and ignmrkup can occur only in the document type specification.
<when>
An end-tag (not a data tag) was specified for the element.

<!-- Data character -->

<propdef rcsnm=movedre appnm="moved re" datatype=boolean cn=datachar
clause="7610a">
<desc>
True if and only if this character is an RE that was deemed to occur
at a point other than that at which it in fact occurred.
<note>
A node of type repos will indicate the position at which
it in fact occurred.

<propdef irefnode rcsnm=repos appnm="re position" datatype=node cn=datachar
ac=repos clause="7610a">
<desc>
```

```
The position at which this RE character in fact occurred.
<when>
This character is an RE that was deemed to occur at a point other
than that at which it in fact occurred.

<propdef subnode optional rcsnm=markup datatype=nodelist
ac="gendelm name ssep entstart entend refendre shortref" cn=extdata
clause="94401 94402">
<desc>
The markup of the entity reference.
<note>
ssep, entstart, and entend can occur only in a name group in a named
entity reference.

<classdef rcsnm=ignrs appnm="ignored rs" clause="76101">
<desc>
An RS that was ignored because of the rules in 7.6.1 of ISO 8879.

<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" clause="95001">
<when>
The character was the replacement of a named character reference.

<classdef rcsnm=ignre appnm="ignored re" clause="76100">
<desc>
An RE in content that was ignored because of the rules in 7.6.1 of ISO
8879.
<note>
This occurs at the point where the RE originally occurred rather
than at the point it was determined that the RE should be ignored.

<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" clause="95001">
<when>
The character was the replacement of a named character reference.

<classdef rcsnm=repos appnm="re position" clause="7610a">
<desc>
The original position of an RE that was deemed by the rules of clause
7.6.1 of ISO 8879 to occur at some point other than that at which it
in fact occurred.
<note>
For each node of type repos, there will be a node of type datachar
with a property movedre that is true.

<propdef irefnode rcsnm=re appnm="record end" datatype=node ac=datachar
clause="7610a">
<desc>
The character for which this is the repos.

</psmodule>

<!-- Datatag-related abstract classes and properties -->
<psmodule rcsnm=dtgabs fullnm="datatag abstract" dependon=baseabs>
```

```
<propdef derived rcsnm=datatag datatype=boolean cn=element clause="73201">
<desc>
True if and only if a data tag served as the end tag of the element.
<note>
The data characters comprising the data tag will follow the element in
the content of the containing element.
   .
<propdef rcsnm=dtgtemps appnm="data tag templates" datatype=strlist
cn=elemtype clause="b2444">
<when>
The model group was a data tag group.

<propdef rcsnm=dtgptemp appnm="data tag padding template" datatype=string
cn=elemtype clause="b2445">
<when>
The model group was a data tag group whose data tag pattern included a
data tag padding template.

</psmodule>

<!-- Rank-related abstract classes and properties -->
<psmodule rcsnm=rankabs fullnm="rank abstract" dependon=prlgabs1>

<propdef derived rcsnm=ranksuff appnm="rank suffix" datatype=string
cn=elemtype clause="b2114">
<when>
The element type in the element type declaration included a rank suffix.

<propdef rcsnm=rankstem appnm="rank stem" datatype=string cn=elemtype
clause="b2113">
<when>
The element type in the element type declaration used a ranked element
or ranked group.

<propdef rcsnm=rankgrp appnm="rank group" datatype=strlist cn=elemtype
clause="b2112">
<desc>
The rank stems in the ranked group.
<when>
The element type declaration included a ranked group.

<classdef rcsnm=rankstem appnm="rank stem" clause="b2113">

<propdef rcsnm=stem datatype=string strlex=name strnorm=general
clause="b2113">
<desc>
Name of rank stem.

<propdef irefnode rcsnm=elemtps appnm="element types"
datatype=nodelist ac=elemtype clause="b2112">
<desc>
The element types for which this is a rank stem.

</psmodule>
```

```
<!-- Shortref-related abstract classes and properties -->
<psmodule rcsnm=srabs fullnm="shortref abstract" dependon=prlgabs0>

<propdef subnode rcsnm=emptymap appnm="empty short ref map"
fullnm="empty short reference map" datatype=node ac=srmap cn=sgmlcsts
clause="b6004">
<desc>
The empty short reference map.

<propdef subnode rcsnm=srmaps appnm="short ref maps"
fullnm="short reference maps" datatype=nmndlist ac=srmap acnmprop=name
cn=doctype clause="b1006">
<note>
Does not include #EMPTY map.

<propdef rcsnm=srmapnm appnm="short ref map name"
fullnm="short reference map name" datatype=string strlex=rniname
strnorm=general cn=elemtype clause="b6004">
<when>
The element type has an associated short reference map.

<propdef irefnode rcsnm=srmap appnm="short ref map"
fullnm="short reference map" datatype=node ac=srmap cn=elemtype
clause="b6101">
<when>
The element type has an associated short reference map.

<classdef rcsnm=srmap appnm="short ref map" fullnm="short reference map"
clause="b5000">

<propdef rcsnm=name datatype=string strlex=name strnorm=general clause="b5002">
<when>
Map is not the implicitly declared #EMPTY map.

<propdef subnode rcsnm=map datatype=nmndlist ac=srassoc acnmprop=shortref
clause="b5004">

<classdef rcsnm=srassoc appnm="short ref assoc"
fullnm="short reference association" clause="b5004">

<propdef rcsnm=shortref appnm="short ref"
fullnm="short reference delimiter" datatype=string strnorm=general
clause="b5004">

<propdef rcsnm=entname appnm="entity name" datatype=string strlex=name
strnorm=entity clause="b5004">

<propdef irefnode rcsnm=entity datatype=node ac=entity clause="b5001">

</psmodule>

<!-- Shortref-related SDS classes and properties -->
<psmodule rcsnm=srsds fullnm="shortref SGML document string"
dependon=basesds1>

<classdef rcsnm=usemap appnm="short ref use decl"
```

```
fullnm="short reference use declaration" conprop=markup clause="b6000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment gendelm name rname ignmrkup"
clause="b6001">
<note>
First child is gendelm for mdo delimiter; last is gendelm for mdc
delimiter.

<propdef irefnode rcsnm=asseltps appnm="assoc element types"
fullnm="associated element types" datatype=nodelist ac=elemtype
clause="a1501">
<note>
SGML specifies that this does not include element types which had
already been associated with a map.
<when>
The short reference use declaration includes an associated element
type.

<propdef irefnode rcsnm=srmap datatype=node ac=srmap clause="b6002">

<classdef rcsnm=shortref appnm="short ref"
fullnm="short reference delimiter" clause="e4620">

<propdef rcsnm=origdelm appnm="original delim"
fullnm="original delimiter" datatype=string clause="96601">
<desc>
The short reference delimiter as originally entered.

<propdef subnode optional rcsnm=namecref appnm="named char ref"
fullnm="named character reference" datatype=nodelist
ac="gendelm name refendre" clause="95001">
<when>
The first character of the delimiter was entered with a named
character reference.

<classdef rcsnm=srmapdcl appnm="short ref map decl"
fullnm="short reference mapping declaration" mayadd clause="b5000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment gendelm name rname literal"
clause="b5001">
<note>
First child is gendelm for mdo delimiter; last is gendelm for mdc
delimiter.

<propdef irefnode rcsnm=map datatype=node ac=srmap clause="b5001">

</psmodule>

<!-- Link-related abstract classes and properties -->
<psmodule rcsnm=linkabs fullnm="link abstract" dependon=prlgabs0>

<propdef subnode rcsnm=emptylks appnm="empty link set" datatype=node ac=linkset
cn=sgmlcsts clause="c3004">
<desc>
```

```
Empty link set used to disable current link set.

<propdef subnode optional rcsnm=simplelk appnm="simple link info"
fullnm="simple link information" datatype=nmndlist ac=simplelk
acnmprop=linkset cn=element clause="c1431">
<when>
Element is the document element and there are active simple link
processes.

<propdef irefnode rcsnm=linkatts appnm="link attributes"
datatype=nmndlist ac=attasgn acnmprop=name cn=element clause="c1402">
<desc>
A list of attribute assignments, one for each declared link attribute
of the element.
<note>
The origin of the link attributes will be the link rule.

<propdef derived rcsnm=rsltgi appnm="result gi"
fullnm="result element generic identifier" datatype=string strlex=name
strnorm=general cn=element clause="c2202">
<when>
There is an applicable link rule which is an explicit link rule whose
result element is not implied.

<propdef irefnode rcsnm=rsltelem appnm="result element type"
datatype=node ac=elemtype cn=element clause="c2202">
<when>
There is an applicable link rule which is an explicit link rule whose
result element is not implied.

<propdef irefnode rcsnm=rsltatts appnm="result attributes"
datatype=nmndlist ac=attasgn acnmprop=name cn=element clause="c2203">
<note>
The origin of the attributes will be the link rule.
<when>
There is an applicable link rule which is an explicit link rule whose
result element is not implied.

<propdef irefnode rcsnm=lksetinf appnm="link set info"
fullnm="link set information" datatype=nodelist ac=linkrule cn=element
clause="c2205">
<desc>
Link rules in the current link set whose source element type is implied.
<when>
There is an active explicit link process.

<propdef irefnode rcsnm=lksetinf appnm="link set info"
fullnm="link set information" datatype=nodelist ac=linkrule cn=datachar>
<desc>
Link rules in the current link set whose source element type is implied.
<when>
There is an active explicit link process and the character occurs
in content.

<classdef rcsnm=simplelk appnm="simple link info"
fullnm="simple link information" clause="c1430">
```

```
<propdef rcsnm=linktype appnm="link type" datatype=string strlex=name
strnorm=general clause="c1001">
<desc>
The link type name of the simple link process.

<propdef subnode rcsnm=atts appnm=attributes
datatype=nmndlist ac=attasgn acnmprop=name clause="c1402">

<classdef rcsnm=linktype appnm="link type">

<propdef rcsnm=name datatype=string strlex=name strnorm=general
clause="c1002">

<propdef rcsnm=active datatype=boolean>
<desc>
True if and only if link type is active.

<propdef rcsnm=ltkind appnm="link type kind"
fullnm="kind of link type" datatype=enum clause="c1001">
<enumdef rcsnm=simple>
<enumdef rcsnm=implicit>
<enumdef rcsnm=explicit>

<propdef rcsnm=srcname appnm="source document type name" datatype=string
strlex=name strnorm=general clause="c1302">

<propdef irefnode rcsnm=source appnm="source document type" datatype=node
ac=doctype clause="c1305 c1306">
<note>
For a simple link type, this will always be the base document type.

<propdef rcsnm=rsltname appnm="result document type name" datatype=string
strlex=name strnorm=general clause="c1303">

<propdef irefnode rcsnm=result appnm="result document type" datatype=node
ac=doctype clause="c1306">
<when>
The link type is an explicit link type.

<propdef subnode rcsnm=inilkset appnm="initial link set" datatype=node
ac=linkset clause="c2004">
<when>
The link type is not simple.

<propdef subnode rcsnm=idlkset appnm="id link set" datatype=node ac=linkset
clause="c2300">
<when>
The link type declaration subset includes an ID link set declaration.

<propdef subnode rcsnm=linksets appnm="link sets" datatype=nmndlist
ac=linkset acnmprop=name clause="c1401">
<note>
Does not include #INITIAL or #EMPTY or ID link set.

<classdef rcsnm=linkset appnm="link set" conprop=lkrules clause="c2000">
```

```
<propdef rcsnm=name datatype=string strlex=name strnorm=general
clause="c2003">
<when>
Link set is not #INITIAL nor #EMPTY nor the ID link set.

<propdef subnode rcsnm=lkrules appnm="link rules" datatype=nodelist
ac=linkrule clause="c2002">

<classdef rcsnm=linkrule appnm="link rule" clause="c2002">

<propdef rcsnm=assgis appnm="assoc gis"
fullnm="associated generic identifiers" datatype=strlist strlex=name
clause="c2101">
<desc>
The names of the associated element types.
<when>
The link rule is not an explicit link rule whose source element type
is implied.

<propdef irefnode rcsnm=asseltps appnm="assoc element types"
fullnm="associated element types" datatype=nodelist ac=elemtype
clause="c2101">
<when>
The link rule is not an explicit link rule whose source element type
is implied.

<propdef rcsnm=id fullnm="unique identifier" datatype=string strlex=name
strnorm=general clause="c2301">
<when>
Link rule occurs in ID link set declaration.

<propdef irefnode rcsnm=uselink datatype=node ac=linkset clause="c2104">
<when>
The link rule includes a USELINK parameter.

<propdef rcsnm=uselknm appnm="uselink name" datatype=string strlex=rniname
strnorm=general clause="c2104">
<desc>
The link set named by the USELINK parameter.
<when>
The link rule includes a USELINK parameter.

<propdef derived rcsnm=postlkrs appnm="postlink restore" datatype=boolean
clause="c2101">
<desc>
True if the link rule includes a POSTLINK parameter of #RESTORE.

<propdef irefnode rcsnm=postlkst appnm="postlink set" datatype=node
ac=linkset clause="c2101">
<when>
The link set specification did not specify #RESTORE.

<propdef rcsnm=postlknm datatype=string strlex=rniname strnorm=general
clause="c2101">
<desc>
```

```
The token specified for the link set specification following POSTLINK.
<when>
The link rule includes a POSTLINK parameter.

<propdef subnode rcsnm=linkatts appnm="link attributes"
datatype=nmndlist ac=attasgn acnmprop=name clause="c2102">
<when>
The link rule is not an explicit link rule whose source element type
is implied.

<propdef rcsnm=rsltgi appnm="result gi"
fullnm="result element generic identifier" datatype=string strlex=name
strnorm=general clause="c2202">
<when>
The link rule is an explicit link rule whose result element type is
not implied.

<propdef irefnode rcsnm=rsltelem appnm="result element type" datatype=node
ac=elemtype clause="c2202">
<when>
The link rule is an explicit link rule whose result element type is
not implied.

<propdef subnode rcsnm=rsltatts appnm="result attributes"
datatype=nmndlist ac=attasgn acnmprop=name clause="c2203">
<when>
The link rule is an explicit link rule whose result element type is
not implied.

</psmodule>

<!-- Link-related SDS classes and properties -->
<psmodule rcsnm=linksds fullnm="link SGML document string"
dependon=basesds1>

<propdef irefnode rcsnm=lksetdcl appnm="link set decl"
fullnm="link set declaration" datatype=node ac="lksetdcl idlkdcl"
cn=linkset clause="c2001">
<when>
Link set is not #EMPTY.

<propdef irefnode rcsnm=lktpdcl appnm="link type decl"
fullnm="link type declaration" datatype=node ac=lktpdcl cn=linktype
clause="c1001">

<classdef rcsnm=lktpdcl appnm="link type decl" fullnm="link type declaration"
mayadd clause="c1000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="ssep comment name rname literal msstart msignch msend
    entstart entend pi comdcl entdcl attdldcl lksetdcl idlkdcl"
    clause="c1001">

<propdef irefnode rcsnm=linktype appnm="link type" datatype=node
ac=linktype>
```

```
<propdef subnode rcsnm=entity datatype=node ac=entity clause="c1004">
<when>
Link type definition includes external identifier.

<classdef rcsnm=lksetdcl appnm="link set decl" fullnm="link set declaration"
mayadd clause="c2000">

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment gendelm name rname literal attvalue"
clause="c2001">

<propdef irefnode rcsnm=linkset appnm="link set" datatype=node
ac=linkset clause="c2001">

<classdef rcsnm=idlkdcl appnm="id link set decl"
fullnm="ID link set declaration" mayadd clause="c2300">

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment gendelm name rname literal attvalue"
clause="c2301">

<propdef irefnode rcsnm=linkset appnm="link set" datatype=node ac=linkset
clause="c2301">

<classdef rcsnm=uselink appnm="link set use decl"
fullnm="link set use declaration" conprop=markup clause="c3000">
<desc>
A link set use declaration that is not ignored.

<propdef subnode rcsnm=markup datatype=nodelist
ac="entstart entend ssep comment gendelm name rname ignmrkup"
clause="c3001">
<note>
First child is gendelm for mdo delimiter; last is gendelm
for mdc delimiter.

<propdef derived rcsnm=restore datatype=boolean clause="c3002">
<desc>
True if the link set specification specified #RESTORE.

<propdef irefnode rcsnm=linkset datatype=node ac=linkset clause="c3002">
<when>
The link set specification did not specify #RESTORE.

<propdef rcsnm=lksetnm datatype=string strlex=rniname strnorm=general
clause="c3002">
<desc>
The token specified for the link set specification.

<propdef rcsnm=linktpnm appnm="link type name" datatype=string
strlex=name strnorm=general clause="c3001">

<propdef irefnode rcsnm=linktype appnm="link type" datatype=node
ac=linktype clause="c3001">

</psmodule>
```

```
<!-- Subdoc-related abstract classes and properties -->
<psmodule rcsnm=subdcabs fullnm="subdoc abstract" dependon=baseabs>

<classdef rcsnm=subdoc appnm=subdocument fullnm="reference to subdocument">
<desc>
The result of referencing a subdocument entity.

<propdef rcsnm=entname appnm="entity name" datatype=string strlex=name
strnorm=entity clause="a5101">

<propdef irefnode rcsnm=entity datatype=node ac=entity clause="c5501">

</psmodule>

<!-- Subdoc-related SDS classes and properties -->
<psmodule rcsnm=subdcsds fullnm="subdoc SGML document string"
dependon="basesds1 subdabs">

<propdef subnode optional rcsnm=markup datatype=nodelist
ac="gendelm name ssep entstart entend refendre shortref" cn=subdoc
clause="94401">
<desc>
The markup of the entity reference.
<note>
ssep, entstart, and entend can occur only in a name group in a named
entity reference.

</psmodule>

<!-- Formal public identifier-related abstract classes and properties -->
<psmodule rcsnm=fpiabs fullnm="formal public identifier abstract"
dependon=baseabs>

<propdef subnode optional rcsnm=fpi appnm="formal public id"
fullnm="formal public identifier" datatype=node ac=fpi cn=extid
clause="a2001">
<when>
FORMAL YES was specified in the SGML declaration.

<classdef rcsnm=fpi appnm="formal public id" fullnm="formal public identifier"
clause="a2000">
<note>
The string which is the value of each of the string-valued properties
provided by this class is the minimum data specified as such in the
governing productions, without any accompanying "//", "-//", "+//"
or s characters.

<propdef rcsnm=ownertp appnm="owner type" datatype=enum clause="a2100">
<desc>
Type of owner identifier.

<enumdef rcsnm=iso>
<enumdef rcsnm=regist appnm=registered>
<enumdef rcsnm=unregist appnm=unregistered>
```

```
<propdef rcsnm=ownerid appnm="owner id" fullnm="owner identifier"
datatype=string strlex=mindata clause="a2100">

<propdef rcsnm=textclas appnm="text class" fullnm="public text class"
datatype=enum clause="a2210">
<enumdef rcsnm=capacity>
<enumdef rcsnm=charset>
<enumdef rcsnm=document>
<enumdef rcsnm=dtd>
<enumdef rcsnm=elements>
<enumdef rcsnm=entities>
<enumdef rcsnm=lpd>
<enumdef rcsnm=nonsgml>
<enumdef rcsnm=notation>
<enumdef rcsnm=shortref>
<enumdef rcsnm=subdoc>
<enumdef rcsnm=syntax>
<enumdef rcsnm=text>

<propdef rcsnm=unavail appnm=unavailable datatype=boolean clause="a2202">
<desc>
True if and only if unavailable text indicator was specified.

<propdef rcsnm=textdesc appnm="text description"
fullnm="public text description" datatype=string strlex=mindata clause="a2221">

<propdef rcsnm=textlang appnm="text language"
fullnm="public text language" datatype=string clause="a2231">
<when>
The text identifier included a public text language.

<propdef rcsnm=textdseq appnm="text designating sequence"
fullnm="public text designating sequence" datatype=string clause="a2241">
<when>
The text identifier included a public text designating sequence.

<propdef rcsnm=textdver appnm="text display version"
fullnm="public text display version" datatype=string clause="a2251">
<when>
The text identifier included a public text display version
(that is, there was a // following the public text language
or public text designating sequence).

</psmodule>

                <!-- String Normalization Rules -->
<normdef rcsnm=general sd=SGML clause="d4506">
<desc>
Declared concrete syntax general namecase substitution.
<normdef rcsnm=entity  sd=SGML clause="d4506">
<desc>
Declared concrete syntax entity namecase substitution.
<normdef rcsnm=rcsgener sd=SGML clause="d4506">
<desc>
Reference concrete syntax general namecase substitution.
```

```
<datadef rcsnm=integer lextype=integer>
<datadef rcsnm=boolean lextype=boolean>
<datadef rcsnm=strlist fullnm="string list" listof=string lextype=strlist>
<datadef rcsnm=intlist fullnm="integer list" listof=int lextype=intlist>

                        <!-- Lexical Types -->
<!-- Datatypes -->
<lexdef ltn=boolean   norm model="[01]">
<lexdef ltn=integer  unorm model="'0'|marker">
<lexdef ltn=intlist  norm model="integer+">
<lexdef ltn=literal   spec sd=SGML clause="96107">
<desc>
Delimited literal as in declared concrete syntax. Character reference
can be used to enter delimiter string within literal, as in SGML
documents.
<lexdef ltn=strlist  norm model="literal,(',',literal)*">
<desc>
String list in so-called "comma-delimited ASCII" format supported by
data base and spreadsheet programs. The literals, exclusive of their
delimiters, shall conform to the applicable lexical type of the
individual strings.

<!-- Other lexical types -->
<lexdef ltn=mindata spec sd=SGML clause="a1702">
<desc>Minimum data.
<lexdef ltn=NAME spec sd=SGML clause="93001">
<desc>Name in declared concrete syntax.
<lexdef ltn=NMTOKEN spec sd=SGML clause="93004">
<desc>Name token in declared concrete syntax.
<lexdef ltn=number spec sd=SGML clause="93002">
<desc>Number in declared concrete syntax.
<lexdef ltn=nmchar spec sd=SGML clause="92103">
<desc>Name character in declared concrete syntax.
<lexdef ltn=ATTNAME nmsp provider=element property=atts sd=SGML clause="b3201">
<desc>Name of attribute of an element.
<lexdef ltn=attspecs spec sd=SGML clause="79001">
<desc>Attribute specification list.
<lexdef ltn=ENTITY nmsp provider=sgmldoc property=entities sd=SGML
clause="a5101">
<desc>General entity name.
<lexdef ltn=IDREF nmsp provider=sgmldoc property=elements sd=SGML
clause="79403">
<desc>ID of an element (specified in document).
<lexdef ltn=GI nmsp provider=dtd property=elemtps sd=SGML clause="78001">
<desc>Element type name (if dtd:effective is true).
<lexdef ltn=rniname spec sd=SGML>
<desc>A name optionally preceded by an RNI delimiter.
```


9.7 DSSSL SGML Grove Plan

A DSSSL specification has a single grove plan specified by the `sgml-grove-plan` architectural form in the DSSSL specification. See 7.1.2.

10 Standard Document Query Language

SDQL adds two data types to the expression language, `node-list` and `named-node-list`. It also adds some additional syntax for expressions: in SDQL, in any context in which an *expression* is allowed, a *special-query-expression* is also allowed.

A subset of SDQL called the *core query language* is defined in 10.2.4.

The `node-list` data type represents an ordered list of zero or more nodes in a grove.

NOTES

28 There is no `node` data type. A single node is represented by a `node-list` with a single member.

29 A node-list will typically be implemented in a lazy fashion. In other words, the internal representation of a node-list is not a list of nodes, but a representation of the specification that constructed the node-list. For example, if an application uses the `node-list-count` procedure on a node-list, it would be inefficient to build the node-list, count it, and then discard the node-list; it would be better simply to count how many distinct nodes match the node-list's specification.

A node-list with a single member is referred to as a *singleton* node-list.

The `named-node-list` data type is a subtype of the `node-list` data type that represents a `node-list` each of whose members has a string-valued property that uniquely identifies the node in the node-list.

`nl` is used for an argument that shall be a node-list. *`snl`* is used for an argument that shall be a singleton node-list. *`nnl`* is used for an argument that shall be a named-node-list.

10.1 Primitive Procedures

The procedures in this clause are the primitive procedures, in the sense that all other procedures in SDQL could be defined in terms of the procedures in this clause, but no procedure in this clause is capable of being defined in terms of the other procedures in this clause.

10.1.1 Application Binding

`(current-node)`

Returns a singleton node-list. The semantics of this are defined by the context in which the SDQL expression occurs.

`(current-root)`

Returns a singleton node-list. The semantics of this are defined by the context in which the SDQL expression occurs.

10.1.2 Node Lists

`(node-list? obj)`

Returns #t if *obj* is a node-list, and otherwise returns #f.

`(node-list-empty? nl)`

Returns #t if *nl* is the empty node-list, and otherwise returns #f.

`(node-list-first nl)`

Returns a node-list containing the first member of *nl*, if any, and otherwise returns the empty node-list.

`(node-list-rest nl)`

Returns a node-list containing all members of *nl* except the first, if *nl* has at least one member, and otherwise returns the empty node-list.

`(node-list` nl_1 nl_2 ...`)`

Returns the node-list that results from appending the members of nl_1, nl_2, ... If there are no arguments, returns the empty node-list.

`(node-list=?` nl_1 nl_2`)`

Returns #t if nl_1 and nl_2 are the same node-list, that is, they contain the same members in the same order, and otherwise returns #f.

`(node-list-no-order nl)`

Returns a node-list that has the same members as *nl* but in an unspecified order.

NOTE 30 An implementation may be able to implement `(node-list-no-order q)` more efficiently than *q*.

10.1.3 Named Node Lists

`(named-node-list? obj)`

Returns #t if *obj* is a named-node-list and otherwise returns #f.

`(named-node` *string nnl*`)`

Returns a singleton node-list comprising the node in *nnl* whose name is *string*, if there is such a node, and otherwise returns the empty node-list. *string* is normalized according to the string normalization rule associated with *nnl* before being compared to the names of the members of *nnl*.

`(named-node-list-normalize` *`string nnl symbol`*`)`

Returns *`string`* normalized according to the normalization rule of the named node list *`nnl`* applicable to nodes of class *`symbol`*.

`(named-node-list-names` *`nnl`*`)`

Returns a list of the names of the members of *`nnl`* in the same order as *`nnl`*. The result shall be a list of strings with the same number of members as *`nnl`*.

10.1.4 Error Reporting

`(node-list-error` *`string nl`*`)`

This signals an error in a similar way to the `error` procedure. When an error is signaled with `node-list-error`, the system should report to the user that the error is associated with the nodes in *`nl`*. The manner in which this is done is system-dependent.

10.1.5 Application Name Transformation

In all contexts in SDQL, application names are transformed by replacing each space with a hyphen and adding a question mark (`?`) to the application names of properties whose declared data type is boolean.

10.1.6 Property Values

`(node-property` *`propname snl`* `#!key default: null: rcs?:)`

Returns the value that the node represented by *`snl`* exhibits for the property *`propname`*. If the node does not exhibit the property *`propname`*, then if the `default:` is supplied, it is returned; otherwise, an error is signaled. If the node exhibits a null value for the property, then if `null:` is supplied, it is returned; otherwise, if `default:` is supplied, it is returned; otherwise, an error is signaled.

`propname` shall be a symbol or a string specifying either the application name (transformed as specified in 10.1.5) or the RCS name of the property. *`propname`* is compared against the property name in a case-independent manner.

Property values are represented as expression language objects according to their abstract data type:

— An abstract character is represented by an object of type char.

— An abstract string is represented by an object of type string.

— An abstract boolean is represented by an object of type boolean.

— An abstract integer is represented by an object of type integer.

— An abstract integer list is represented by a list of integers.

— An abstract string list is represented by a list of strings.

— An enumeration is represented by a symbol whose name is equal to the application name of the enumerator (transformed as specified in 10.1.5).

— A component name is represented by a symbol. The name of the symbol shall be the application name (transformed as specified in 10.1.5), unless the `rcs?:` argument is supplied with a true value, in which case the RCS name will be used.

— An abstract component name list is represented by a list of the symbols that represent each component name.

— An abstract node is represented by a singleton node-list.

— An abstract nodelist is represented by an object of type node-list.

— An abstract nmndlist is represented by an object of type named-node-list.

— Null values have no representation in the expression language.

10.1.7 SGML Grove Construction

`(sgml-parse` *`string`* `#!key active: parent:)`

Returns a node-list containing a single node that is the root of a grove built by parsing an SGML document or subdocument using the SGML property set. *`string`* is the system identifier of the SGML document entity or SGML subdocument entity. `active:` is a list of strings specifying the names of the active DTD or LPDs. At most one DTD shall be active. If `parent:` is specified, then the entity to be parsed is an SGML subdocument entity, and the value shall be a singleton node-list in the grove in which the subdocument should be treated as being declared. This uses the default grove plan, which is determined in an application-dependent manner.

10.2 Derived Procedures

For some procedures, a formal definition in the expression language is supplied. These formal definitions do not handle errors. A correct implementation would need first to verify that arguments meet the requirements indicated by the procedure prototypes and the procedure description.

10.2.1 HyTime Support

Use of the facilities in this clause in the style or transformation languages requires the `hytime` feature.

The grovepos abstract data type is represented by a list each of whose members is

— an integer,

— a list containing a symbol and a string, or

— a list containing a symbol and an integer.

`(value-proploc` *`propname snl`* `#!key apropsrc?: default:)`

Returns the value that the member of *`snl`* exhibits for the property named *`propname`*. *`propname`* shall be a symbol or string, interpreted as for the `node-property` procedure. If the member of *`snl`* does not exhibit a value for *`propname`* or exhibits a null value, then if `default:` is supplied, `default:` shall be returned; otherwise, an error shall be signaled. `apropsrc?:`, if true, has the same effect as specifying an `apropsrc` attribute with a value of `apropsrc` for the code `proploc` form in ISO/IEC 10744.

`(list-proploc` *`propname nl`* `#!key apropsrc?: ignore-missing?:)`

Returns a list of objects, one for each member of *`nl`*, where each object is the value that the member of *`nl`* exhibits for *`propname`*. *`propname`* shall be a symbol or string, interpreted as for the `node-property` procedure. If some member of *`nl`* does not exhibit a value for *`propname`* or exhibits a null value, then if `ignore-missing?:` is true, the resulting list shall contain no object for that member; otherwise, an error shall be signaled. `apropsrc?:`, if true, has the same effect as specifying an `apropsrc` attribute with a value of `apropsrc` for the code `proploc` form in ISO/IEC 10744.

`(node-list-proploc` *`propname nl`* `#!key apropsrc?: ignore-missing?:)`

Returns the node-list that results from concatenating the values that each member of *`nl`* exhibits for *`propname`*. *`propname`* shall be a symbol or string, interpreted as for the `node-property` procedure. For the class of each member of *`nl`*, *`propname`* shall be nodal. If some member of *`nl`* does not exhibit a value for *`propname`* or exhibits a null value, then if `ignore-missing?:` is true, the resulting node-list shall contain no nodes for that member; otherwise, an error shall be signaled. `apropsrc?:`, if true, has the same effect as specifying an `apropsrc` attribute with a value of `apropsrc` for the code `proploc` form in ISO/IEC 10744.

`(listloc` *`dimlist nl`* `#!key overrun:)`
`(listloc` *`dimlist list`* `#!key overrun:)`
`(listloc` *`dimlist string`* `#!key overrun:)`

This addresses the members of the second argument in the same manner as the `listloc` architectural form defined in ISO/IEC 10744. Returns a node-list, list, or string according to the type of the second argument. *`dimlist`* is a list of integers. `overrun:` is one of the symbols `error`, `wrap`, `truncate`, or `ignore`. The default is `error`.

`(nameloc` *`nmlist nnl`* `#!key ignore-missing?:)`

Returns a node-list containing one member for each member of *nmlist*, where *nmlist* is a string, symbol, or a list of strings and/or symbols. It shall be an error if any member of *nmlist* does not match the name of some member of *nl*, unless ignore-missing?: is true.

```
(groveloc list nl #!key overrun:)
```

Returns a list of nodes located in the same manner as with the groveloc architectural form of ISO/IEC 10744. *list* is a list in the same format as the representation of the grovepos abstract data type. overrun: is interpreted as with listloc.

```
(treeloc marklist nl #!key overrun: treecom?:)
```

Returns a list of nodes located in the same manner as with the treeloc architectural form of ISO/IEC 10744. *marklist* is list of integers. overrun: is interpreted as with listloc. treecom?:, if true, corresponds to a treecom attribute with a value of treecom.

```
(pathloc dimlist nl #!key overrun: treecom?:)
```

Returns a list of nodes located in the same manner as with the pathloc architectural form of ISO/IEC 10744. *dimlist* is a list of integers. overrun: is interpreted as with listloc. treecom?:, if true, corresponds to a treecom attribute with a value of treecom.

```
(relloc-anc dimlist nl #!key overrun:)
(relloc-esib dimlist nl #!key overrun:)
(relloc-ysib dimlist nl #!key overrun:)
(relloc-des dimlist nl #!key overrun:)
```

Returns a list of nodes located in the same manner as with the relloc architectural form of ISO/IEC 10744. The procedures relloc-anc, relloc-esib, relloc-ysib, and relloc-des correspond to values for the relation attribute of anc, esib, ysib, and des. *dimlist* is a list of integers. overrun: is interpreted as with listloc.

NOTE 31 Relations of parent and children are handled by parent and children procedures.

```
(datatok nl #!key  filter: concat: catsrcsp: catressp: tokensep:
ascp: stop: min: max: nlword: stem?:)
```

Returns a list of nodes located in the same manner as with the datatok architectural form of ISO/IEC 10744.

— filter: is a symbol having one of the values allowed for the filter attribute.

— concat: is one of the symbols catshi, catslo, cattk, catshitk, catslotk, catrhitk, catrlotk, or nconcat interpreted in the same manner as the concat attribute.

— `catsrcsp:`, `catressp:`, `tokensp:`, and `ascp:` are strings interpreted in the same manner as the attributes with the same name.

— `nlword:` is a string specifying an ISO 639 language code.

— `stem?:`, if true, has the same effect as specifying `#STEM` for the `nlword` attribute.

— `stop:` is a list of strings specifying a stop list; the default is the empty list.

— `min:` is an integer specifying the minimum untruncated token length.

— `max:` is an integer specifying the maximum untruncated token length.

`(make-grove` *`string nl`*`)`

`make-grove` constructs a new grove and returns a node-list containing the grove root. *`string`* is the name of a grove plan. *`nl`* is the source text.

`(literal-match` *`string nl`* `#!key level: boundary: min-hits: max-hits:)`
`(hylex-match` *`string nl`* `#!key norm?: level: boundary: min-hits: max-hits:)`

These functions construct a new grove using the Data Tokenizer Property Set containing one tokenized string node for each non-overlapping match found in the data of each member of *`nl`*. A node-list of all tokenized string nodes is returned.

— `boundary:` is one of the symbols `sodeod`, `sodiec`, `isceod`, or `isciec`, which shall be interpreted in the same manner as the `boundary` attribute of the `HyLex` element defined in ISO/IEC 10744.

— `level:` is a number of comparison levels in the collation specification of the current language on which string comparison shall be performed; if `level:` is not specified, strings shall be compared simply by comparing their constituent characters for equality.

— `min-hits:` and `max-hits:` are strictly positive integers specifying the minimum and maximum number of hits: any match whose parent node does not contain a number of hits within the specified range shall be excluded from the list of nodes returned. The default for `min-hits:` is 1. If `max-hits:` is not specified, there shall be no maximum.

— `norm?:` is a boolean specifying whether the lexical model shall be normalized.

`(compare` *`proc list`*`)`

Returns #t if *`proc`* applied to each successive pair of strings returns #t, where *`proc`* is an argument of two strings that returns a boolean. This could be defined by:

```
(define (compare proc l)
  (if (null? l)
      #t
```

```
(let loop ((prev (car l))
           (rest (cdr l)))
  (cond ((null? rest) #t)
        ((proc prev (car rest))
         (loop (car rest) (cdr rest)))
        (else #f)))))
```

`(ordered-may-overlap? nl)`
`(ordered-no-overlap? nl)`

Each node shall be in an auxiliary grove, and the source nodes of all the nodes shall be in a single tree. Returns #t if the source nodes are ordered within that tree, and otherwise returns #f. For `ordered-no-overlap?`, the source nodes are considered to be ordered if, for each argument node, all of its source nodes are before any of the source nodes of the next argument node. For `ordered-may-overlap?`, the source nodes are considered to be ordered if, for each argument node, the first of its source nodes is before the first of the source nodes of the next argument node.

`(span nl symbol)`

Each node shall be in an auxiliary grove, and the source nodes of all the nodes shall be in a single tree. Returns the number of quanta between the first and the last source nodes. *symbol* specifies the quantum. It shall have one of the values allowed for the `filter:` argument of the `datatok` procedure.

10.2.2 List Operations

These procedures are similar to procedures on normal lists.

`(empty-node-list)`

Returns an empty node-list.

`(node-list-reduce nl proc obj)`

If *nl* has no members, returns *obj*, and otherwise returns the result of applying `node-list-reduce` to

— a node-list containing all but the first member of *nl*,

— *proc*, and

— the result of applying *proc* to *obj* and the first member of *nl*.

`node-list-reduce` could be defined as follows:

```
(define (node-list-reduce nl combine init)
  (if (node-list-empty? nl)
      init
      (node-list-reduce (node-list-rest nl)
```

```
                        combine
                        (combine init (node-list-first nl)))))
```

`(node-list-contains? nl snl)`

Returns #t if *nl* contains a node equal to the member of *snl*, and otherwise returns #f. This could be defined as follows:

```
(define (node-list-contains? nl snl)
  (node-list-reduce nl
                    (lambda (result i)
                      (or result
                          (node-list=? snl i)))
                    #f))
```

`(node-list-remove-duplicates nl)`

Returns a node-list which is the same as *nl* except that any member of *nl* which is equal to a preceding member of *nl* is removed. This could be defined as follows:

```
(define (node-list-remove-duplicates nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (if (node-list-contains? result snl)
                          result
                          (node-list result snl)))
                    (empty-node-list)))
```

`(node-list-union #!rest args)`

Returns a node-list containing the union of all the arguments, which shall be node-lists. The result shall contain no duplicates. With no arguments, an empty node-list shall be returned. This could be defined as follows:

```
(define (node-list-union #!rest args)
  (reduce args
          (lambda (nl1 nl2)
            (node-list-reduce nl2
                              (lambda (result snl)
                                (if (node-list-contains? result
                                                         snl)
                                    result
                                    (node-list result snl)))
                              nl1))
          (empty-node-list)))
```

where `reduce` is defined as follows:

```
(define (reduce list combine init)
  (let loop ((result init)
             (list list))
    (if (null? list)
        result
        (loop (combine result (car list))
              (cdr list)))))
```

```
(node-list-intersection #!rest args)
```

Returns a node-list containing the intersection of all the arguments, which shall be node-lists. The result shall contain no duplicates. With no arguments, an empty node-list shall be returned. This could be defined as follows:

```
(define (node-list-intersection #!rest args)
  (if (null? args)
      (empty-node-list)
      (reduce (cdr args)
              (lambda (nl1 nl2)
                (node-list-reduce nl1
                                  (lambda (result snl)
                                    (if (node-list-contains? nl2 snl)
                                        (node-list result snl)
                                        result))
                                  (empty-node-list)))
              (node-list-remove-duplicates (car args)))))
```

```
(node-list-difference #!rest args)
```

Returns a node-list containing the set difference of all the arguments, which shall be node-lists. The set difference is defined to be those members of the first argument that are not members of any of the other arguments. The result shall contain no duplicates. With no arguments, an empty node-list shall be returned. This could be defined as follows:

```
(define (node-list-difference #!rest args)
  (if (null? args)
      (empty-node-list)
      (reduce (cdr args)
              (lambda (nl1 nl2)
                (node-list-reduce nl1
                                  (lambda (result snl)
                                    (if (node-list-contains? nl2 snl)
                                        result
                                        (node-list result snl)))
                                  (empty-node-list)))
              (node-list-remove-duplicates (car args)))))
```

```
(node-list-symmetric-difference #!rest args)
```

Returns a node-list containing the symmetric set difference of all the arguments, which shall be node-lists. The symmetric set difference is defined to be those nodes that occur in exactly one of the arguments. The result shall contain no duplicates. With no arguments, an empty node-list shall be returned. This could be defined as follows:

```
(define (node-list-symmetric-difference #!rest args)
  (if (null? args)
      (empty-node-list)
      (reduce (cdr args)
              (lambda (nl1 nl2)
                (node-list-difference (node-list-union nl1 nl2)
                                      (node-list-intersection nl1 nl2)))
              (node-list-remove-duplicates (car args)))))
```

(node-list-map *proc nl*)

For each member of *nl*, applies *proc* to a singleton node-list containing just that member and appends the resulting node-lists. It shall be an error if *proc* does not return a node-list when applied to any member of *nl*. This could be defined as follows:

```
(define (node-list-map proc nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (node-list (proc snl)
                                 result))
                    (empty-node-list)))
```

(node-list-union-map *proc nl*)

For each member of *nl*, applies *proc* to a singleton node-list containing just that member and returns the union of the resulting node-lists. It shall be an error if *proc* does not return a node-list when applied to any member of *nl*. This could be defined as follows:

```
(define (node-list-union-map proc nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (node-list-union (proc snl)
                                       result))
                    (empty-node-list)))
```

(node-list-some? *proc nl*)

Returns #t if, for some member of *nl*, *proc* does not return #f when applied to a singleton node-list containing just that member, and otherwise returns #f. An implementation is allowed, but not required, to signal an error if, for some member of *nl*, *proc* would signal an error when applied to a singleton node-list containing just that member. This could be defined as follows:

```
(define (node-list-some? proc nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (if (or result (proc snl))
                          #t
                          #f))
                    #f))
```

(node-list-every? *proc nl*)

Returns #t if, for every member of *nl*, *proc* does not return #f when applied to a singleton node-list containing just that member, and otherwise returns #f. An implementation is allowed to signal an error if, for some member of *nl*, *proc* would signal an error when applied to a singleton node-list containing just that member. This could be defined as follows:

```
(define (node-list-every? proc nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (if (and result (proc snl))
                          #t
```

```
                        #f))
                  #t))
```

`(node-list-filter proc nl)`

Returns a node-list containing just those members of `nl` for which `proc` applied to a singleton node-list containing just that member does not return #f. This could be defined as follows:

```
(define (node-list-filter proc nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (if (proc snl)
                          (node-list snl result)
                          result))
                    (empty-node-list)))
```

`(node-list->list nl)`

Returns a list containing, for each member of `nl`, a singleton node-list containing just that member. This could be defined as follows:

```
(define (node-list->list nl)
  (reverse (node-list-reduce nl
                             (lambda (result snl)
                               (cons snl result))
                             '())))
```

`(node-list-length nl)`

Returns the length of `nl`. This could be defined as follows:

```
(define (node-list-length nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (+ result 1))
                    0))
```

`(node-list-reverse nl)`

Returns a node-list containing the members of `nl` in reverse order. This could be defined as follows:

```
(define (node-list-reverse nl)
  (node-list-reduce nl
                    (lambda (result snl)
                      (node-list snl result))
                    (empty-node-list)))
```

`(node-list-ref nl k)`

Returns a node-list containing the *k*th member of `nl` (zero-based), if there is such a member, and otherwise returns the empty node-list. This could be defined as follows:

```
(define (node-list-ref nl i)
  (cond ((< i 0)
         (empty-node-list))
        ((zero? i)
```

```
        (node-list-first nl))
       (else
        (node-list-ref (node-list-rest nl) (- i 1)))))
```

`(node-list-tail nl k)`

Returns the node-list comprising all but the first k members of `nl`. If `nl` has k or fewer members, returns the empty node-list. This could be defined as follows:

```
(define (node-list-tail nl i)
  (cond ((< i 0) (empty-node-list))
        ((zero? i) nl)
        (else
         (node-list-tail (node-list-rest nl) (- i 1)))))
```

`(node-list-head nl k)`

Returns a node-list comprising the first k members of `nl`. If `nl` has k or fewer members, returns `nl`. This could be defined as follows.

```
(define (node-list-head nl i)
  (if (zero? i)
      (empty-node-list)
      (node-list (node-list-first nl)
                 (node-list-head nl (- i 1)))))
```

`(node-list-sublist nl` k_1 k_2`)`

Returns a node-list containing those members of `nl` that are preceded in `nl` by at least k_1 members but fewer than k_2 members. This is equivalent to selecting those members whose zero-based index in `nl` is greater than or equal to k_1 but less than k_2. This could be defined as follows:

```
(define (node-list-sublist nl i j)
  (node-list-head (node-list-tail nl i)
                  (- j i)))
```

`(node-list-count nl)`

Returns the number of distinct members of `nl`. This could be defined as follows:

```
(define (node-list-count nl)
  (node-list-length (node-list-remove-duplicates nl)))
```

`(node-list-last nl)`

Returns a node-list containing the last member of `nl`, if `nl` is not empty, and otherwise returns the empty node-list. This could be defined as follows:

```
(define (node-list-last nl)
  (node-list-ref nl
                 (- (node-list-length nl) 1)))
```

When using `node-list-some?`, `node-list-every?`, `node-list-filter`, and `node-list-union-map`, the first argument is often a lambda expression with a variable. A syntax

that avoids the need to use an explicit lambda expression in this case is provided in this International Standard.

[146] special-query-expression = *there-exists?-expression* | *for-all?-expression* | *select-each-expression* | *union-for-each-expression*

[147] there-exists?-expression = `(there-exists?` *variable expression expression*`)`

An expression

`(there-exists?` *var nl-expr expr*`)`

is equivalent to:

`(node-list-some? (lambda (`*var*`)` *expr*`)` *nl-expr*`)`

Read this as: there exists a *var* in *nl-expr* such that *expr*.

[148] for-all?-expression = `(for-all?` *variable expression expression* `)`

An expression

`(for-all?` *var nl-expr expr*`)`

is equivalent to:

`(node-list-every? (lambda (`*var*`)` *expr*`)` *nl-expr*`)`

Read this as: for all *var* in *nl-expr*, *expr*.

[149] select-each-expression = `(select-each` *variable expression expression*`)`

An expression

`(select-each` *var nl-expr expr*`)`

is equivalent to:

`(node-list-filter (lambda (`*var*`)` *expr*`)` *nl-expr*`)`

Read this as: select each *var* in *nl-expr* such that *expr*.

[150] union-for-each-expression = `(union-for-each` *variable expression expression*`)`

An expression

`(union-for-each` *var nl-expr expr*`)`

is equivalent to:

`(node-list-union-map (lambda (`*var*`)` *expr*`)` *nl-expr*`)`

Read this as: the union of, for each *var* in *nl-expr*, *expr*.

10.2.3 Generic Property Operations

These procedures work with any grove, but use only intrinsic properties.

The result of many of the following procedures is the *mapping* of a function on a node over a node-list, which is defined to be the node-list that results from appending in order the result of applying the function to each member of the node-list.

`(node-list-property` *`propname nl`*`)`

Returns the mapping over *`nl`* of the function on a node that returns the value that the node exhibits for the property *`propname`* or an empty node-list if the node does not exhibit a value or exhibits a null value for *`propname`*. *`propname`* can be specified in any of the ways allowed for the `node-property` procedure. It shall be an error if any node in *`nl`* exhibits a non-null, non-nodal value for *`propname`*. This could be defined as follows:

```
(define (node-list-property prop nl)
  (node-list-map (lambda (snl)
                   (node-property prop snl default: (empty-node-list)))
                 nl))
```

`(origin` *`nl`*`)`

This is equivalent to:

```
(define (origin nl)
  (node-list-property 'origin nl))
```

`(origin-to-subnode-rel` *`snl`*`)`

Returns the value that the member of *`snl`* exhibits for the `origin-to-subnode-rel-property-name` property, or #f if it does not exhibit a value or exhibits a null value. This could be defined as follows:

```
(define (origin-to-subnode-rel snl)
  (node-property 'origin-to-subnode-rel-property-name snl default: #f))
```

`(tree-root` *`nl`*`)`

This is equivalent to:

```
(define (tree-root nl)
  (node-list-property 'tree-root nl))
```

`(grove-root` *`nl`*`)`

This is equivalent to:

```
(define (grove-root nl)
  (node-list-property 'grove-root nl))
```

`(children` *`nl`*`)`

Returns the mapping over *nl* of the function on a node that returns the value of the node's children property, if any, and otherwise the empty node-list. This could be defined as follows:

```
(define (children nl)
  (node-list-map (lambda (snl)
                   (let ((childprop (node-property 'children-property-name
                                                   snl
                                                   default: #f)))
                     (if childprop
                         (node-property childprop
                                        snl
                                        default: (empty-node-list))
                         (empty-node-list))))
                 nl))
```

(data *nl*)

Returns a string containing the concatenation of the data of each member of *nl*. The data of a node is:

— if the node has a data property, the value of its data property converted to a string, if necessary,

— if the child has a children property, the concatenation of the data of each of the children of the node, separated by the value of the data separator property, if it has a non-null value, or

— otherwise, an empty string.

(parent *nl*)

This is equivalent to:

```
(define (parent nl)
  (node-list-property 'parent nl))
```

(source *nl*)

This is equivalent to:

```
(define (source nl)
  (node-list-property 'source nl))
```

(subtree *nl*)

Returns the mapping over *nl* of the function on a node that returns the subtree of a node, where the subtree of a node is defined to be the node-list comprising the node followed by the subtrees of its children. This could be defined as follows:

```
(define (subtree nl)
  (node-list-map (lambda (snl)
                   (node-list snl (subtree (children snl))))
                 nl))
```

(subgrove *nl*)

Returns the mapping over *nl* of the function on a node that returns the subgrove of a node, where the subgrove of a node is defined to be the node-list comprising the node followed by the subgroves of members of the values of each of the node's subnode properties. This could be defined as follows:

```
(define (subgrove nl)
  (node-list-map
   (lambda (snl)
     (node-list snl
                (subgrove
                 (apply node-list
                        (map (lambda (name)
                               (node-property name snl))
                             (node-property 'subnode-property-names
                                            snl))))))
   nl))
```

`(descendants nl)`

Returns the mapping over *nl* of the function on a node that returns the descendants of the node, where the descendants of a node are defined to be the result of appending the subtrees of the children of the node. This could be defined as follows:

```
(define (descendants nl)
  (node-list-map (lambda (snl)
                   (subtree (children snl)))
                 nl))
```

`(ancestors nl)`

Returns the mapping over *nl* of the function on a node that returns the ancestors of the node, where the ancestors of a node are an empty node-list if the node is a tree root, and otherwise are the result of appending the ancestors of the parent of the node and the parent of the node. This could be defined as follows:

```
(define (ancestors nl)
  (node-list-map (lambda (snl)
                   (let loop ((cur (parent snl))
                              (result (empty-node-list)))
                     (if (node-list-empty? cur)
                         result
                         (loop (parent snl)
                               (node-list cur result)))))
                 nl))
```

`(grove-root-path nl)`

Returns the mapping over *nl* of the function on a node that returns the grove root path of the node, where the grove root path of a node is defined to be an empty node-list if the node is the grove root, and otherwise is the result of appending the grove root path of the origin of the node and the origin of the node. This could be defined as follows:

```
(define (grove-root-path nl)
  (node-list-map (lambda (snl)
                   (let loop ((cur (origin snl))
```

```
                    (result (empty-node-list)))
                (if (node-list-empty? cur)
                    result
                    (loop (origin nl)
                          (node-list cur result)))))
            nl))
```

`(rsiblings nl)`

Returns the mapping over `nl` of the function on a node that returns the reflexive siblings of the node, where the reflexive siblings of a node are defined to be the value of the origin-to-subnode relationship property of the node's origin, if the node has an origin, and otherwise the node itself. This could be defined as follows:

```
(define (rsiblings nl)
  (node-list-map (lambda (snl)
                   (let ((rel (origin-to-subnode-rel snl)))
                     (if rel
                         (node-property rel
                                        (origin snl)
                                        default: (empty-node-list))
                         snl)))
                 nl))
```

`(ipreced nl)`

Returns the mapping over `nl` of the function on a node that returns the immediately preceding sibling of the node, if any. This could be defined as follows:

```
(define (ipreced nl)
  (node-list-map (lambda (snl)
                   (let loop ((prev (empty-node-list))
                              (rest (siblings snl)))
                     (cond ((node-list-empty? rest)
                            (empty-node-list))
                           ((node-list=? (node-list-first rest) snl)
                            prev)
                           (else
                            (loop (node-list-first rest)
                                  (node-list-rest rest))))))
                 nl))
```

`(ifollow nl)`

Returns the mapping over `nl` of the function on a node that returns the immediately following sibling of the node, if any. This could be defined as follows:

```
(define (ifollow nl)
  (node-list-map (lambda (snl)
                   (let loop ((rest (siblings snl)))
                     (cond ((node-list-empty? rest)
                            (empty-node-list))
                           ((node-list=? (node-list-first rest) snl)
                            (node-list-first (node-list-rest rest)))
                           (else
```

```
                                (loop (node-list-rest rest))))))
                nl))
```

(preced *nl*)

Returns the mapping over *nl* of the function on a node that returns the preceding siblings of the node, if any. This could be defined as follows:

```
(define (preced nl)
  (node-list-map (lambda (snl)
                   (let loop ((scanned (empty-node-list))
                              (rest (siblings snl)))
                     (cond ((node-list-empty? rest)
                            (empty-node-list))
                           ((node-list=? (node-list-first rest) snl)
                            scanned)
                           (else
                            (loop (node-list scanned
                                             (node-list-first rest))
                                  (node-list-rest rest))))))
                 nl))
```

(follow *nl*)

Returns the mapping over *nl* of the function on a node that returns the following siblings of the node, if any. This could be defined as follows:

```
(define (follow nl)
  (node-list-map (lambda (snl)
                   (let loop ((rest (siblings snl)))
                     (cond ((node-list-empty? rest)
                            (empty-node-list))
                           ((node-list=? (node-list-first rest) snl)
                            (node-list-rest rest))
                           (else
                            (loop (node-list-rest rest))))))
                 nl))
```

(grove-before? snl_1 snl_2)

Returns #t if snl_1 is strictly before snl_2 in grove order. It is an error if snl_1 and snl_2 are not in the same grove. This could be defined as follows:

```
(define (grove-before? snl1 snl2)
  (let ((sorted
         (node-list-intersection (subgrove (grove-root snl1))
                                 (node-list snl1 snl2))))
    (and (= (node-list-length sorted) 2)
         (node-list=? (node-list-first sorted) snl1))))
```

(sort-in-tree-order *nl*)

Returns the members of *nl* sorted in tree order. Any duplicates shall be removed. It is an error if the members of *nl* are not all in the same tree. This could be defined as follows:

```
(define (sort-in-tree-order nl)
  (node-list-intersection (subtree (tree-root nl))
                          nl))
```

`(tree-before? snl1 snl2)`

Returns #t if snl_1 is strictly before snl_2 in tree order. It is an error if snl_1 and snl_2 are not in the same tree. This could be defined as follows:

```
(define (tree-before? snl1 snl2)
  (let ((sorted
         (sort-in-tree-order (node-list snl1 snl2))))
    (and (= (node-list-length sorted) 2)
         (node-list=? (node-list-first sorted) snl1))))
```

`(tree-before nl)`

Returns the mapping over *nl* of the function on a node that returns those nodes in the same tree as the node that are before the node. This could be defined as follows:

```
(define (tree-before nl)
  (node-list-map (lambda (snl)
                   (node-list-filter (lambda (x)
                                       (tree-before? x snl))
                                     (subtree (tree-root snl))))
                 nl))
```

`(property-lookup propname snl if-present if-not-present)`

If *snl* exhibits a non-null value for the property *propname*, `property-lookup` returns the result of applying *if-present* to that value, and otherwise returns the result of calling *if-not-present* without arguments. *propname* can be specified in any of the ways allowed for the `node-property` procedure. This could be defined as follows:

```
(define (property-lookup name snl if-present if-not-present)
  (let ((val (node-property name snl default: #f)))
    (cond (val (if-present val))
          ((node-property name snl default: #t) (if-not-present))
          (else (if-present val)))))
```

`(select-by-class nl sym)`

Returns a node-list comprising members of *nl* that have node class *sym*. *sym* is either the application name (transformed as specified in 10.1.5) or the RCS name of the class.

`(select-by-property nl sym proc)`

Returns a node-list comprising those members of *nl* that have a non-nodal property named *sym* that exhibits a non-null value such that *proc* applied to it returns a true value.

`(select-by-null-property nl sym)`

Returns a node-list comprising members of *nl* for which the property *sym* exhibits a null value.

`(select-by-missing-property nl sym)`

Returns a node-list comprising members of *nl* for which the property *sym* does not exhibit a value.

10.2.4 Core Query Language

This clause defines a subset of SDQL. In addition to the procedures defined in this clause, the `current-node`, `node-list-empty?`, `node-list?`, `parent`, and `node-list-error` procedures are allowed in the subset. This subset is designed so that a node-list never contains more than one node and so that any node that it does contain is always of type element.

In the following procedures, the argument that is of type node-list can be omitted and defaults to `(current-node)`. *osnl* (optional singleton node-list) denotes an argument that shall be a node-list containing zero or one nodes.

10.2.4.1 Navigation

`(ancestor string osnl)`

Returns a node-list containing the nearest ancestor of *osnl* with a gi equal to *string*, or an empty node-list if there is no such ancestor or if *osnl* is empty.

`(gi osnl)`

Returns the value of the `gi` property of the node contained in *osnl* or #f if *osnl* is empty or if *osnl* has no `gi` property or a null `gi` property.

`(first-child-gi osnl)`

Returns the value of the `gi` property of the first child of *osnl* of class `element` or #f if *osnl* is empty or has no such child.

`(id osnl)`

Returns the value of the `id` property of the node contained in *osnl* or #f if *osnl* is empty or if *osnl* has no `id` property or a null `id` property.

10.2.4.2 Counting

`(child-number snl)`

Returns the child number of *snl*. The *child number* of an element is one plus the number of element siblings of the current element that precede in tree order the current element and that have the same generic identifier as the current element.

`(ancestor-child-number string snl)`

Returns the child number of the nearest ancestor of *snl* whose generic identifier is *string*, or #f if there is no such ancestor.

`(hierarchical-number` *list snl*`)`

Returns a list of non-negative integers with the same number of members as *list*. *list* shall be a list of strings. The last member is the child number of the nearest ancestor of *snl* whose generic identifier is equal to the last member of *list*, the next to last member is the child number of the nearest ancestor of that element whose generic identifier is equal to the next to last member, and so on for each member of *list*.

`(hierarchical-number-recursive` *string snl*`)`

Returns a list of non-negative integers. The last member of the list is the child number of the nearest ancestor of the *snl* element whose generic identifier is equal to *string*, the next to last member is the child number of the nearest ancestor of that element whose generic identifier is equal to *string*, and so on for each ancestor of the current element with generic identifier equal to *string*. Note that the length of this list is the nesting level of *string*.

`(element-number` *snl*`)`

Returns the number of elements before or equal to *snl* with the same gi as *snl*.

`(element-number-list` *list snl*`)`

Returns a list of non-negative integers, one for each member of *list*, which shall be a list of strings, where the *i*-th integer is the number of elements that:

— are before or equal to *snl*,

— have a generic identifier equal to the *i*-th member of *list*, and

— if *i* is greater than 1, are after the last element before *snl* whose generic identifier is equal to the *i-1*th member of *list*.

NOTES

32 In effect the counter for each argument is reset at the start of the element referred to by the previous argument.

33 An element is considered to be after its parent.

34 This procedure could be used to number footnotes sequentially within a chapter (by using the last number in the list). It could also be used to number headings in a document whose DTD lacks container elements.

10.2.4.3 Accessing Attribute Values

In the following procedures, attribute values are represented as strings by applying the `data` procedure to the `attribute-assignment` node.

`(attribute-string` *`string osnl`*`)`

Returns a string representation of the attribute with name equal to *`string`* of *`osnl`*, or #f if *`osnl`* has no such attribute, or the attribute is implied, or *`osnl`* is empty.

`(inherited-attribute-string` *`string osnl`*`)`

Returns a string representation of the attribute with name equal to *`string`* of *`osnl`* or of the nearest ancestor of *`osnl`* for which this attribute is present and not implied, or #f if there is no such element or *`osnl`* is empty. For the purpose of this procedure, a node is considered an ancestor of itself.

`(inherited-element-attribute-string` $string_1$ $string_2$ *`osnl`* `)`

Returns a string representation of the attribute with name equal to $string_2$ of the nearest ancestor of *`osnl`* whose generic identifier is equal to $string_1$ and for which this attribute is present and not implied, or #f if there is no such element or *`osnl`* is empty. For the purpose of this procedure, a node is considered an ancestor of itself.

10.2.4.4 Testing Current Location

`(first-sibling?` *`snl`*`)`

Returns #t if *`snl`* has no preceding sibling that is an element with the same generic identifier as itself, and otherwise returns #f.

`(absolute-first-sibling?` *`snl`*`)`

Returns #t if *`snl`* has no preceding sibling that is an element, and otherwise returns #f.

`(last-sibling?` *`snl`*`)`

Returns #t if *`snl`* has no following sibling that is an element with the same generic identifier as itself, and otherwise returns #f.

`(absolute-last-sibling?` *`snl`*`)`

Returns #t if *`snl`* has no following sibling that is an element, and otherwise returns #f.

`(have-ancestor?` *`obj snl`*`)`

`obj` shall be either a string or a list of strings. If *`obj`* is a string, then `have-ancestor?` returns #t if *`snl`* has an ancestor with a generic identifier that matches that string and otherwise returns #f. If *`obj`* is a list of strings, then `have-ancestor?` returns #t if *`snl`* has an ancestor with generic identifier equal to the last member of *`obj`*, which itself has an ancestor with generic identifier equal to the next to last member of *`obj`*, and so on for each member, and otherwise returns #f.

10.2.4.5 Entities and Notations

snl here determines the document in which to find the entity.

`(entity-public-id` *string snl*`)`

Returns the value of the public-id property of the value of the external-id property of the general entity whose name is *string* in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has a null value for the external-id property or the external-id has a null value for the public-id property.

`(entity-system-id` *string snl*`)`

Returns the value of the system-id property of the value of the external-id property of the general entity whose name is *string* in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has a null value for the external-id property or the external-id has a null value for the system-id property.

`(entity-generated-system-id` *string snl*`)`

Returns the value of the generated-system-id property of the value of the external-id property of the general entity whose name is *string* in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has a null value for the external-id property or the external-id has a null value for the generated-system-id property.

`(entity-text` *string snl*`)`

Returns the value of the text property of the general entity whose name is *string* in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has a null value for the text property.

`(entity-notation` *string snl*`)`

Returns the value of the notation-name property of the general entity whose name is *string* in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has a null value for the notation-name property.

`(entity-attribute-string` $string_1$ $string_2$ *snl*`)`

Returns a string representation of the value of the attribute named $string_2$ of the general entity whose name is $string_1$ in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has no such attribute or the attribute is implied.

`(entity-type` *string snl*`)`

Returns the value of the entity-type property of the general entity whose name is *string* in the governing document type of the same grove as *snl*, or #f if there is no such entity or the entity has a null value for the entity-type property.

`(notation-public-id` *`string snl`*`)`

Returns the value of the public-id property of the value of the external-id property of the general notation whose name is *`string`* in the governing document type of the same grove as *`snl`*, or #f if there is no such notation or the external-id has a null value for the public-id property.

`(notation-system-id` *`string snl`*`)`

Returns the value of the system-id property of the value of the external-id property of the general notation whose name is *`string`* in the governing document type of the same grove as *`snl`*, or #f if there is no such notation or the external-id has a null value for the system-id property.

`(notation-generated-system-id` *`string snl`*`)`

Returns the value of the generated-system-id property of the value of the external-id property of the general notation whose name is *`string`* in the governing document type of the same grove as *`snl`*, or #f if there is no such notation or the external-id has a null value for the generated-system-id property.

10.2.4.6 Name Normalization

`(general-name-normalize` *`string snl`*`)`

Returns *`string`* transformed using the general namecase substitution string normalization rule of the grove in which *`snl`* occurs. This could be defined as follows:

```
(define (general-name-normalize string snl)
  (named-node-list-normalize string
                             (node-property 'elements (grove-root snl))
                             'element))
```

`(entity-name-normalize` *`string snl`*`)`

Returns *`string`* transformed using the entity namecase substitution string normalization rule of the grove in which *`snl`* occurs. This could be defined as follows:

```
(define (entity-name-normalize string snl)
  (named-node-list-normalize string
                             (node-property 'entities (grove-root snl))
                             'entity))
```

10.2.5 SGML Property Operations

These procedures make use of particular properties that are defined by the property set for SGML.

`(attributes` *`nl`*`)`

This is equivalent to:

```
(define (attributes nl)
  (node-list-property 'attributes nl))
```

`(attribute` *`string nl`*`)`

Returns the mapping over *`nl`* of the function that returns the member of the value of the `attributes` property whose name is equal to *`string`*. This could be defined as follows:

```
(define (attribute name nl)
  (node-list-map (lambda (snl)
                   (named-node name (attributes snl)))
                 nl))
```

`(element-with-id` *`string snl`*`)`

Returns a singleton node-list returning the element in the same grove as *`snl`* whose unique identifier is *`string`*, if there is such an element, and otherwise returns the empty node-list. *`snl`* defaults to `(current-node)`.

`(referent` *`nl`*`)`

This is equivalent to:

```
(define (referent nl)
  (node-list-property 'referent nl))
```

`(match-element?` *`pattern snl`*`)`

Returns #t if *`snl`* is a node of class element that matches *`pattern`*. *`pattern`* is either a list or a single string or symbol. A string or symbol is equivalent to a list containing just that string or symbol. The list can contain strings or symbols. The element matches the list if the last string or symbol matches the gi of the element, and the next to last matches the gi of the element's parent, and so on. Each string or symbol may optionally be followed by a list containing an even number of strings or symbols, which are interpreted as attribute name and value pairs all of which the element whose gi matches the preceding string or symbol shall have.

For example,

```
(match-element? '(e1 (a1 v1 a2 v2) e2 (a3 v3) e3 e4) n)
```

returns true if

— the gi of `n` is `e4`,

— the gi of `n`'s parent is `e3`,

— the gi of `n`'s grandparent is `e2`,

— `n`'s grandparent has an `a3` attribute with a value equal to `v3`,

— the gi of `n`'s great grandparent is `e1`,

— n's great grandparent has an a2 attribute with a value equal to v2, and

— n's great grandparent has an a1 attribute with a value equal to v1.

sn1 defaults to the node-list returned by the current-node procedure.

When a string or symbol in the pattern is compared against a property value, and the property value was subject to upper-case substitution, upper-case substitution shall also be performed on the string before comparison.

(select-elements *n1 pattern*)

Returns a node-list comprising those members of *n1* that match *pattern* as defined by the match-element? procedure.

(q-element *pattern n1*)
(q-element *pattern*)

Searches in the subgroves whose roots are each members of *n1* for elements matching *pattern*, as defined by the match-element? procedure. *n1* defaults to the node-list returned by current-node.

(q-class *symbol n1*)
(q-class *symbol*)

Searches in the subgroves whose roots are each members of *n1* for nodes whose class is *symbol*. *n1* defaults to the node-list returned by current-node.

(q-sdata *string n1*)
(q-sdata *string*)

Searches in the subgroves whose roots are each members of *n1* for nodes whose class is sdata and the value of whose sysdata property is *string*. *n1* defaults to the node-list returned by current-node.

10.3 Auxiliary Parsing

10.3.1 Word Searching

Use of the facilities in this clause in the style or transformation languages requires the word feature.

(word-parse *n1 string*)
(word-parse *n1*)

This builds a new grove by performing an auxiliary parse using the Data Tokenizer Property Set. *string*, if specified, is the ISO 639 language code of the language which should be assumed for

the purposes of determining what constitutes a word. The algorithm to be used is not specified in this International Standard.

```
<propset psn=datatok fullnm="Data Tokenizer Property Set">
<classdef rcsnm=tokroot appnm="tokenized root" conprop=strings>
<propdef rcsnm=strings datatype=nodelist ac=tokenstr>
<classdef rcsnm=tokenstr appnm="tokenized string" conprop=string>
<propdef rcsnm=string datatype=string>
```

For each member of *nl*, a tokenized string node is created for each word in the data of that member. The root of the auxiliary grove has these tokenized string nodes as children. A node-list of all the tokenized string nodes is returned. If a member, *x*, of *nl* contains another member, *y*, of *nl* as a descendant, then the data of *y* is removed from the data of *x* before *x* is parsed for words.

`(select-tokens` *nl string*`)`

Returns a node-list containing each member of *nl* that is a tokenized-string node with a `string` property equal to *string*.

10.3.2 Node Regular Expressions

Use of the facilities in this clause in the style or transformation languages requires the `regexp` feature.

The regexp type represents a node regular expression. A node regular expression is an object that can be used to perform an auxiliary parse of a grove. This auxiliary parse creates a new grove that contains nodes that group together nodes that correspond to nodes in the original grove. The semantics of a node regular expression define for any node-list *s* and any node-list *t* that is a sublist of *s* whether *t* matches the node regular expression with respect to *s*. This is defined inductively for each of the procedures that construct regexps. *s* is referred to as the search list.

A node-list *s* immediately precedes a node-list *t* with respect to a node-list *x* that contains all the members of both *s* and *t* if

— *s* is empty, or

— *t* is empty, or

- the member of *s* that occurs latest in *x* occurs in *x* before the element of *t* that occurs first in *x*, and
- there is no node in *x* that
  - follows in *x* all those members of *x* that occur in *s*, and
  - precedes in *x* all those members of *x* that occur in *t*.

`(regexp? obj)`

Returns #t if *obj* is a regexp, and otherwise returns #f.

10.3.3 Regexp Constructors

The procedures in this section construct regexp objects that are used by the subparsing procedures.

`(regexp-node proc)`

Returns a regexp that matches a node-list with respect to any search list if the node-list contains exactly one node and *proc* applied to that node-list returns a true value.

`(regexp-seq` $regexp_1$ $regexp_2$ … $regexp_n$`)`

Returns a regexp that matches a node-list with respect to a search list x if the node-list can be split into sublists $s_1, s_2, \ldots, s_n$ such that $regexp_i$ matches s_i with respect to the search list x for $1 \le i \le n$ and such that s_i immediately precedes s_{i+1} with respect to x for $1 \le i \le n-1$.

`(regexp-or` $regexp_1$ $regexp_2$ … $regexp_n$`)`

Returns a regexp that matches a node-list with respect to a search list x if, for some i such that $1 \le i \le n$, the node-list matches $regexp_i$ with respect to x.

`(regexp-and` $regexp_1$ $regexp_2$ … $regexp_n$`)`

Returns a regexp that matches a node-list with respect to a search list x if, for every i such that $1 \le i \le n$, the node-list matches $regexp_i$ with respect to x.

`(regexp-rep regexp)`

Returns a regexp that matches a node-list with respect to a search list x if the node-list is empty or if there is some integer $n \ge 1$ such that the node-list can be split into sublists $s_1, s_2, \ldots, s_n$ such that s_i matches *regexp* for each i such that $1 \le i \le n$ and such that s_i immediately precedes s_{i+1} with respect to x for each i such that $1 \le i \le n-1$.

`(regexp-plus regexp)`

Returns a regexp that matches a node-list with respect to a search list x if there is some integer $n \ge 1$ such that the node-list can be split into sublists $s_1, s_2, \ldots, s_n$ such that s_i matches *regexp* for each i such that $1 \le i \le n$ and such that s_i immediately precedes s_{i+1} with respect to x for each i such that $1 \le i \le n-1$.

`(regexp-opt regexp)`

Returns a regexp that matches a node-list with respect to a search list x if either the node-list is empty or the node-list matches *regexp* with respect to x.

`(regexp-range regexp` k_1 k_2`)`

Returns a regexp that matches a node-list with respect to a search list x if there is some integer n with $k_1 \le n \le k_2$ such that the node-list can be split into sublists $s_1, s_2, \ldots, s_n$ such that s_i matches `regexp` for each i such that $1 \le i \le n$ and such that s_i immediately precedes s_{i+1} with respect to x for each i such that $1 \le i \le n-1$. If k_1 is zero, then the returned regexp shall match the empty node-list.

`(string->regexp string)`

Returns the regexp represented by `string`. It shall be an error if `string` is not a valid representation of an extended regular expression as defined in ISO 9945-2. A normal character in `string` matches a node with a `char` property whose value is that character.

NOTE 35 This could be implemented in terms of the above primitives.

10.3.4 Regular Expression Searching Procedures

The procedures in this clause use regexp objects to create a new auxiliary grove using the Regular Expression Property Set as follows:.

```
<propset psn=regexp fullnm="Regular Expression Property Set">
<classdef rcsnm=root conprop=groups sd=DSSSL>
<desc>
The root of the grove.
<propdef rcsnm=groups datatype=nodelist ac=group sd=DSSSL>
<classdef rcsnm=group sd=DSSSL>
```

`(regexp-search nl regexp)`

Returns a new auxiliary grove built using the regexp property set. The grove contains one group node for each sublist of `nl` that matches `regexp` with respect to `nl`. The source property of each group node contain the nodes in the matching sublist.

NOTE 36 The source property is an intrinsic property of every node in an auxiliary grove.

`(regexp-search-disjoint nl regexp)`

This is the same as `regexp-search` except that the sublists are disjoint. When two sublists overlap, if one sublist has a member that occurs in `nl` before all members of the other sublist, then the first sublist is preferred. If one sublist contains another sublist as a proper sublist, then the containing sublist is preferred.

11 Transformation Language

This clause describes the DSSSL transformation language. Syntactically, the DSSSL transformation language is a data content notation as defined by ISO 8879. The content of an element in this notation is parsed as a *transformation-language-body*.

[151] transformation-language-body = [[*unit-declaration** | *added-char-properties-declaration** | *character-property-declaration** | *transliteration-map-definition** | *language-definition** | *default-language-declaration*? | *definition** | *association**]]

The transformation language uses the expression language defined in clause 8 and SDQL defined in clause 10.

A transformation process requires a single grove as input, which is transformed as specified by the *associations*. An association may cause other groves to be transformed. The grove being transformed is referred to as the current grove.

11.1 Features

The following features are optional in the transformation language:

— The `combine-char` feature allows the `combine-char` element type form.

— The `keyword` feature allows `#!key` in *formal-argument-lists*.

— The `multi-source` feature allows use of the `transform-grove` procedure.

— The `multi-result` feature allows multiple result groves.

— The `regexp` feature allows the use of node regular expressions described in 10.3.2.

— The `word` feature allows the use of the facilities for word searching described in 10.3.1.

— The `hytime` feature allows the use of the facilities for HyTime location addressing described in 10.2.1.

— The `charset` feature allows the use of the declaration element type forms other than `char-repertoire`, `combine-char`, `features`, and `sgml-grove-plan`.

11.2 Associations

The transformation process is specified by a collection of associations.

[152] association = (=> *query-expression transform-expression priority-expression*?)

[153] query-expression = *expression*

[154] transform-expression = *expression*

[155] priority-expression = *expression*

Each association has up to three components:

— a *query-expression* returning a node-list; an association is *potentially applicable* to any node in the node-list returned by its query-expression.

— a *transform-expression* that is evaluated for each of the nodes to which the association is applicable. The value returned describes the node or nodes in the result grove corresponding to the selected node in the source grove.

— an optional *priority-expression* that affects whether the association actually applies to a node to which it is potentially applicable.

A *query-expression* shall evaluate to a node-list. All the nodes in the node-list returned by a *query-expression* shall be nodes in the current grove or shall be nodes in an auxiliary grove whose source grove is the current grove. Auxiliary groves are described in 9.5. In a *query-expression*, the `current-root` procedure and `current-node` procedure return a singleton node-list containing the root of the current grove.

A *priority-expression* shall evaluate to an integer. The number specifies the priority of the association. If the *priority-expression* is omitted for an *association*, the priority of the association is 0. Larger numbers indicate higher priorities.

Each node to which an association is potentially applicable has a *constituent set* of nodes in the current grove. When the node is in the current grove, the constituent set contains just that node. When the node is in an auxiliary grove, then the constituent set contains the nodes in the current grove that occur in the value of the source property of the node in the auxiliary grove. An association is actually applicable to any node, n, to which it is potentially applicable unless some higher priority association applies to a node whose constituent set contains a node that is in the constituent set of n.

11.3 Transform-expression

Within a *transform-expression*, the `current-node` procedure returns a singleton node-list containing the node that is being transformed.

Each *transform-expression* shall return an object of type create-spec or of type transform-grove-spec or a (possibly empty) list of objects each of type create-spec or transform-grove-spec. Each create-spec describes a subgrove to be created at a specified place in the result grove. The subgrove may consist either of a single node or of multiple nodes forming a subgrove rooted in a single node. The place at which the subgrove is to be created may be specified as the root of a result grove, or it may be specified relative to some other node in the result grove.

For each node that is created in the result grove, links are created from each of the constituent nodes of the node whose transformation resulted in creation of the node in the result grove to the created node. These links are referred to as *arrows*. An arrow is labeled with an expression language object. The start-point of an arrow is called the *transformation origin* of its end-point. The arrow for a node in the source grove says where that node was transformed to. The labels on the arrows distinguish between different transformations that were applied to a node. The transform-expression for a node either specifies that the created subgrove shall be the root of a

result grove or specifies the position of the created subgrove in the result grove relative to a node in the result grove to which some other node in the source grove was transformed.

11.3.1 Subgrove-spec

The subgrove to be created is described using an object of type subgrove-spec.

```
(subgrove-spec #!key node: subgrove: class: add: null: remove:
children: sub: label: sort-children:)
```

Returns an object of type subgrove-spec.

The `node:` argument shall be a singleton node-list; it specifies that the node at the root of the created subgrove shall have the same class as the value of `node:`, the same non-nodal, non-intrinsic properties as the value of `node:` (as modified by the `add:` and `remove:` arguments), and the same null-valued properties as the value of `node:` (except as modified by the `null:` and `remove:` arguments).

The `subgrove:` argument shall be a singleton node-list; it specifies the creation of a subgrove that is a copy of the subgrove rooted in the argument node.

The `class:` argument is a symbol specifying the class of the node to be created. Exactly one of the `node:`, `subgrove:`, and `class:` arguments shall be specified.

The `add:` argument specifies non-nodal, non-intrinsic properties with non-null values that shall be added to the node. The `add:` argument shall be a list of two-element lists whose first member is the name of a property and whose second member is the value of that property. The property shall be a non-nodal, non-intrinsic property of the node's class. The value for a property specified in the `add:` argument replaces any value for that property that the node specified by the `node:` argument had.

The `null:` argument is a list of symbols specifying the names of additional non-intrinsic properties of the node which shall have null values. This replaces any non-null property which the node would have by virtue of the `node:` argument.

The `remove:` argument is a list of non-intrinsic properties which the node specified by the `node:` argument has and which the node to be created should not have; it defaults to the empty list. This may be used to remove properties with both null and non-null values.

The `sub:` argument is a list specifying subnodes for the node at the root of the subgrove returned by `subgrove-spec`. The members of the list shall be lists whose first member is a symbol specifying the name of the subnode property and the rest of whose members are subgrove-specs specifying the nodes in the value of the property. This argument defaults to the empty list.

The `children:` argument is a list of subgrove-specs specifying the nodes in the value of the children property of the node at the root of the subgrove returned by `subgrove-spec`.

NOTE 37 These can also be specified using the `sub:` argument, but using `children:` is often more convenient.

This argument defaults to the empty list.

The `label:` argument specifies the label for the arrow which shall be created from the transformed node in the source grove to the node at the root of the subgrove being created in the result grove. It may be any expression language object. The default value is #f.

The `sort-children:` argument is a procedure that affects the ordering of the children of the root node. See 11.3.2.

Classes and properties are named by their application names as defined in the SGML property set, with the usual transformation described in 10.1.5.

11.3.2 Create-spec

`(create-spec? obj)`

Returns `#t` if *obj* is of type `create-spec`, and otherwise returns `#f`.

`(create-root obj sg)`

Returns a create-spec specifying the creation of the root of a result grove. *sg* is a subgrove-spec for the root of the result grove. *obj* is an identifier for the result grove.

```
(create-sub snl sg #!key property: label: result-path: optional:
unique:)
(create-preced snl sg #!key label: result-path: optional: unique:)
(create-follow snl sg #!key label: result-path: optional: unique:)
```

`create-sub`, `create-preced`, and `create-follow` return a create-spec specifying that for each arrow labeled `label:` with a start-point of *snl* the subgrove specified by *sg* shall be created in the result grove. The evaluation of the `create-sub`, `create-preced`, or `create-follow` procedures does not of itself cause the creation of nodes in the result grove; a create-spec that is not returned by a *transform-expression* shall be ignored.

`label:` can be any expression language object; it defaults to `#f`.

If `optional:` is `#f`, then it shall be an error if there never is any such arrow; `optional:` defaults to `#f`.

`result-path:` is a procedure that for each arrow is applied to a result-node-list whose only member is the end-point of the arrow. `result-path:` may be applied to this result-node-list at various points in the construction of the grove. At some point in the construction of the grove, it shall return a result-node-list that contains exactly one member. This is the *creation origin*. At no point shall it return a result-node-list that contains more than one member. If `result-path:` is not specified, it defaults to the identity procedure.

For `create-sub`, `property:` is a symbol or string specifying a property name. This property shall be a subnode property of the creation origin, and the subgrove shall be created as a member

of that property of the creation origin. If the `property:` argument is omitted, it defaults to the children property of the creation origin; it shall not be omitted if the creation origin has no children property. For `create-preced`, the subgrove shall be created as a preceding sibling of the creation origin. For `create-follow`, the subgrove shall be created as a following sibling of the creation origin.

Two subgroves are said to have the *same creation method* if and only if the roots of the subgroves were created with the same creation origin and same creation procedure and, if the creation procedure was `create-sub`, the same *propname*.

If `unique:` is not `#f`, then this subgrove shall be the only one that is ever created with the same creation method as this one. `unique:` defaults to `#f`.

When `unique:` is `#f`, the relative order of subgroves created with the same creation method is determined in a way that is independent of the order in which the subgroves are created. Let the *immediately dependent siblings* of a node be those siblings of the node that were created with a creation origin of that node using the `create-follow` or `create-preced` procedures. Let the *dependent siblings* of a node be the immediately dependent siblings of the node together with the dependent siblings of the immediately dependent siblings. Let the *creation siblings* of a subgrove to be inserted be those nodes that were created with the same creation procedure and with the same creation origin. In addition, if a subgrove is to be inserted using `create-sub`, then any nodes that will be siblings of the inserted subgrove and were created as part of the same subgrove as the origin node shall be treated as creation siblings. The position of a subgrove to be inserted is first determined relative to its creation siblings. It is then inserted in such a way that it follows all the dependent siblings of all those creation siblings that it is to follow and precedes all the dependent siblings of all those creation siblings that it is to precede so that there is no node between it and its creation origin that is neither a creation sibling nor a dependent sibling of a creation sibling.

When the node at the root of the subgrove is a child of the node that will be the origin of the subgrove, the position of the subgrove among its creation siblings is determined by the ordering predicate of the origin node. The ordering predicate is the procedure specified by the `sort-children:` argument to the `subgrove-spec` procedure. The ordering predicate is passed the transformation origins of two nodes in the result grove that are to be compared. It shall return true if the first is before the second. If no ordering predicate was specified, then the `tree-before?` procedure shall be used as an ordering predicate. In this case, it shall be an error if the transformation origins of the subgrove and its creation siblings are not all in the same tree. When the node at the root of the subgrove is not a child of the origin node, then the position of the subgrove among its creation siblings is determined in the same way as for the children of a node with an ordering predicate of `grove-before?`.

An arrow *triggers* another arrow if the second arrow was created by a call to a create procedure that specified the start-point of the first arrow as the first argument and specified the label of the first arrow as the `label:` argument. It shall be an error if there is a sequence of arrows where each arrow triggers the next arrow and where the last arrow has the same start-point and label as the first arrow.

NOTE 38 This requirement avoids the possibility of an infinite loop.

11.3.3 Result-node-list

A result-node-list represents a list of nodes in the result grove. A subset of the operations permitted on node-lists are permitted on result-node-lists. In a prototype, an argument name *rnl* shall be of type result-node-list.

NOTE 39 The allowed operations are designed to ensure that if a node in the result grove is contained in the result-node-list that results from evaluating an expression at some point in the construction of the result grove, then that node shall be contained in the result-node-list that results from evaluating that expression at any subsequent point in the construction of the result grove.

```
(node-list-union rnl …)
(node-list-intersection rnl …)
(children rnl)
(attributes rnl)
(preced rnl)
(follow rnl)
(parent rnl)
(ancestors rnl)
(descendants rnl)
(origin rnl)
(select-by-class rnl sym)
(select-by-property rnl sym proc)
(select-by-null-property rnl sym)
(select-by-missing-property rnl sym)
```

These procedures behave in the same way as the corresponding operations on node-lists except that the return value is of type result-node-list rather than node-list.

```
(select-by-relation rnl i proc)
```

Returns a result-node-list containing those nodes contained in *rnl* which are such that *proc* applied to a result-node-list containing exactly that node returns a result-node-list containing *i* or more nodes. For example,

```
(lambda (x)
  (select-by-relation (children x)
                      1
                      (lambda (y)
                        (select-elements (descendants y) "para"))))
```

selects those children of a node that have a descendant element with a gi of para.

```
(select-by-attribute-token rnl string1 string2)
```

Returns a result-node-list containing those nodes in *rnl* that have an attribute named $string_1$ and that have an attribute with a child of class attribute-value-token with a token property equal to $string_2$ after any applicable string normalization.

11.3.4 Transform-grove-spec

An object of type transform-grove-spec represents a grove to be transformed in addition to the current grove.

`(transform-grove-spec? obj)`

Returns `#t` if *obj* is of type `transform-grove-spec`, and otherwise returns `#f`.

`(transform-grove snl obj ...)`

snl shall be the root of a grove. `transform-grove` creates a new grove from *snl* by adding a `transform-args` property to the grove root whose value is a list containing *obj*, ..., and returns an object of type transform-grove-spec specifying the transformation of that new grove.

`(select-grove nl obj)`

Returns a node-list containing those members of *nl* whose grove root has a `transform-args` property that contains a member equal to *obj*.

11.3.5 SGML Prolog Parsing

`(sgml-parse-prolog string)`

Returns a node-list containing a single node that is the root of a grove built by parsing the prolog of an SGML document. *string* is the system identifier of the SGML document entity. This is built using the default grove plan modified to exclude the instabs module.

NOTE 40 This procedure is typically used to specify the `subgrove:` argument to the `subgrove-spec:` procedure when the source and result groves have different DTDs.

11.4 SGML Document Generator

The SGML document generator generates an SGML document or subdocument from a result grove. The operation of the SGML document generator is specified in terms of a *verification grove*, which is the grove that would be built by parsing the SGML document or subdocument generated from the result grove using a grove plan that included all classes and properties of the SGML property set.

NOTE 41 An implementation is not required to build a verification grove.

A result grove is *valid* if it is possible to generate a conforming SGML document or subdocument from the result grove such that there is a verification mapping from the result grove to the verification grove which meets the requirements specified in 11.4.1. If the result grove is valid, an implementation shall generate such a document or subdocument. An implementation shall report that a result grove is not valid if and only if the result grove is not valid.

11.4.1 Verification Mapping

Any result grove satisfies the following requirements:

— A node in the result grove does not exhibit a value for a property with a declared data type that is nodal unless the property is a subnode property.

— A node in the result grove never exhibits a value for a property that is in the derived category.

The verification mapping, V, maps each node in the result grove to a node in the verification grove. $V(n)$ denotes the result of applying V to the node n; $n[p]$ denotes the value that n exhibits for property p. A node n' in the verification grove is said to be *grounded* if and only if there is a node n in the result grove such that $V(n)$ is n'.

V shall satisfy the following requirements:

— If n is the root of the result grove, then $V(n)$ shall be the root of the verification grove.

— For each distinct m and n in the result grove, $V(m)$ shall be distinct from $V(n)$.

— For each n in the result grove, $V(n)$ shall have the same class as n.

— For each node n in the result grove, and each non-intrinsic property p with a non-nodal declared data type for which $V(n)$ exhibits a null value, n shall exhibit a null value for p unless p is in the derived or optional category.

— For each node n in the result grove, and each non-intrinsic property p for which $V(n)$ exhibits a non-null, non-nodal value, n shall exhibit a value for p unless p is in the derived or optional category.

— A node in the verification grove shall be grounded if its class is not in the mayadd category and either

  – any of its siblings are grounded, or

    • the origin of the node is grounded, and

    • the origin-to-subnode relationship property of its origin is not in the optional category.

— For each node n in the result grove, and for each non-intrinsic property p for which n exhibits a null value, $V(n)$ shall exhibit a null value for p.

— For each node n in the result grove, and for each non-intrinsic non-nodal property p for which n exhibits a non-null value, $n[p]$ shall be equal, after any applicable string normalization specified for the property by the property set, to $V(n)[p]$.

— For each node n in the result grove and each subnode property p with a declared data type of node for which n exhibits a non-null value, $V(n[p])$ shall be equal to $V(n)[p]$.

— For each node *n* in the result grove and each subnode property *p* with a declared data type of nodelist or nmndlist for which *n* exhibits a non-null value, and for each node *s* in *n*[*p*], *V*(*s*) shall be in *V*(*n*)[*p*].

— For each node *n* in the result grove and each subnode property *p* with a declared data type of nodelist for which *n* exhibits a value, and for any nodes *r* and *s* in *n*[*p*], if *r* precedes *s* in the result grove, *V*(*r*) shall precede *V*(*s*) in the verification grove.

The transliteration property described in 11.4.2 is not considered in the verification mapping.

As an exception to these rules, a node in the verification grove of class attribute-assignment need not be grounded if the rules of ISO 8879 that apply with an SGML declaration that specified SHORTTAG YES would not require the attribute to be specified.

11.4.2 Transliteration

[156] transliteration-map-definition = `(define-transliteration-map` *variable transliteration-entry*`)`

[157] transliteration-entry = `(`*character character-list*`)`

[158] character-list = `(`*character*`+)`

A *transliteration-map-definition* binds *variable* to an object of type transliteration-map. The transliteration-map specifies a transliteration in which certain characters are represented by sequences of one or more other characters. Each transliteration entry specifies that the first character is represented by the sequence of characters in the *character-list*.

`(transliteration-map?` *obj*`)`

Returns #t if *obj* is of type transliteration-map, and otherwise returns #f.

Each node in a result grove can have a non-nodal transliteration property whose value is an object of type transliteration-map. If no transliteration property is specified for a node, the value of the transliteration property is the value of the transliteration property of the origin of the node. If no transliteration property is specified for the root node of a result grove, then the value shall be an empty transliteration map.

For each consecutive sequence of data-char nodes in the result grove with the same transliteration property, the sequence of characters that the sequence of characters in the result grove represents with respect to the transliteration-map shall be output instead of the sequence of characters in the result grove. In case of ambiguity, the longest *transliteration-entry* shall be used.

12 Style Language

This clause describes the DSSSL style language. Syntactically, the style language is a data content notation, as defined in ISO 8879. The content of an element in this notation is parsed as a *style-language-body*.

[159] style-language-body = [[*unit-declaration** | *definition** | *construction-rule** | *mode-construction-rule-group** | *application-flow-object-class-declaration** | *application-characteristic-declaration** | *application-char-characteristic+property-declaration** | *initial-value-declaration** | *reference-value-type-declaration** | *page-model-definition** | *column-set-model-definition** | *added-char-properties-declaration** | *character-property-declaration** | *language-definition** | *default-language-declaration*?]]

The style language described in this International Standard uses the core expression language described in 8.6 or, optionally, the full expression language described in clause 8, and the core query language described in 10.2.4 or, optionally, the full query language (SDQL) described in clause 10.

[160] style-language-expression = *make-expression* | *style-expression* | *with-mode-expression*

Within a *style-language-body*, an *expression* may be a *style-language-expression*.

NOTE 42 A *style-expression* is used to specify the values for inherited characteristics.

12.1 Features

The following features are optional in the style language:

— The `expression` feature allows the full expression language. Without this feature only the core expression language shall be used.

— The `multi-process` feature allows the unrestricted use of `process-children` and related procedures as described in 12.4.4.

— The `query` feature allows use of the full query language described in 10 and related facilities described in this clause. Without this feature only the core query language shall be used. This implies the `multi-process` feature.

— The `regexp` feature allows the use of node regular expressions described in 10.3.2.

— The `word` feature allows the use of the facilities for word searching described in 10.3.1.

— The `hytime` feature allows the use of the facilities for HyTime location addressing described in 10.2.1.

— The `combine-char` feature allows the `combine-char` element type form.

— The `keyword` feature allows `#!key` in *formal-argument-lists*.

— The `side-by-side` feature allows use of the side-by-side and side-by-side-item flow object classes.

— The `sideline` feature allows use of the sideline flow object class.

— The `aligned-column` feature allows use of the aligned-column flow object class.

— The `bidi` feature allows use of the right-to-left writing-mode and the embedded-text flow object class.

— The `vertical` feature allows use of the top-to-bottom writing-mode.

— The `math` feature allows use of the flow object classes for mathematical formulae described in 12.6.26.

— The `table` feature allows use of the flow object classes for tables described in 12.6.27.

— The `table-auto-width` feature allows the widths of table columns to be computed automatically. This implies the `table` feature.

— The `simple-page` feature allows use of the facilities for simple page layout described in 12.6.3.

— The `page` feature allows use of the page-sequence and column-set-sequence flow object classes and related features.

— The `multi-column` feature allows use of column-sets containing more than one column. This implies the `page` feature.

— The `nested-column-set` feature allows use of a column-set-sequence flow object with a column-set-sequence flow object ancestor. This implies the `multi-column` and `page` features.

— The `general-indirect` feature allows use of the `general-indirect-sosofo` procedure.

— The `inline-note` feature allows use of the inline-note flow object class.

— The `glyph-annotation` feature allows use of the glyph-annotation flow object class.

— The `emphasizing-mark` feature allows use of the emphasizing-mark flow object class.

— The `included-container` feature allows use the included-container flow object class.

— The `actual-characteristic` feature allows use of the `actual-`*c* procedures for each inherited characteristic *c*.

— The `online` feature allows use of the facilities described in 12.6.28.

— The `font-info` feature allows use of the facilities described in the 12.5.7.

— The `cross-reference` feature allows the use of the `process-element-with-id` procedure.

— The `charset` feature allows the use of the declaration element type form other than `char-repertoire`, `combine-char`, `features`, and `sgml-grove-plan`.

12.2 Flow Object Tree

A flow object tree is an abstract representation of the merger of the formatting specification and the source document. The nodes of the flow object tree are flow objects. Each flow object is of a type called a *flow object class*. A flow object is said to be an *instance* of its class. A flow object also has a set of characteristics. The characteristics that are applicable to a flow object depend on the flow object's class. A flow object's class and characteristics together constitute a specification of the desired formatting behavior of the flow object.

Each flow object has a set of ports to each of which an ordered list of flow objects can be attached. The set of ports may be empty. One port of each flow object that has any ports may be distinguished as the *principal port*. The principal port is unnamed. Every other port has a name which uniquely identifies it in the context of its flow object. The list of flow objects attached to a port is known as a *stream*, and the members of the list are called members of the stream. There is a single flow object in the flow object tree that is not a member of any stream. This flow object is called the *root* of the flow object tree. Every other flow object in the flow object tree is a member of exactly one stream. This stream is referred to as the flow object's stream. The flow object to which a flow object's stream is attached is called the *flow parent* of the flow object. The set of ports that a flow object has is controlled by its class, and for some classes also by its characteristics. A flow object that has no ports is called an *atomic flow object*, and a flow object class whose instances are always atomic is an *atomic flow object class*. The relative positioning of flow objects in different streams can be constrained by *synchronizing* the flow objects. In addition, the value of a characteristic may result in the creation of a flow object.

12.3 Areas

The concept of an area is used to give semantics to flow objects. The result of formatting a flow object other than the root flow object is a sequence of *areas*. The nature of these areas is not fully specified by this International Standard. An area is a rectangular box with a fixed width and height. An area is also a specification of a set of marks that can be imaged on a presentation medium. An area may contain other areas. In particular, an area may contain a glyph. Information may be attached to areas depending on the flow object that produced the area and the context in which it is to be used. Areas are of two types: display areas and inline areas. Each type of area is placed in a different way. For an illustration of the concept of displayed and inlined areas, see Figure 4.

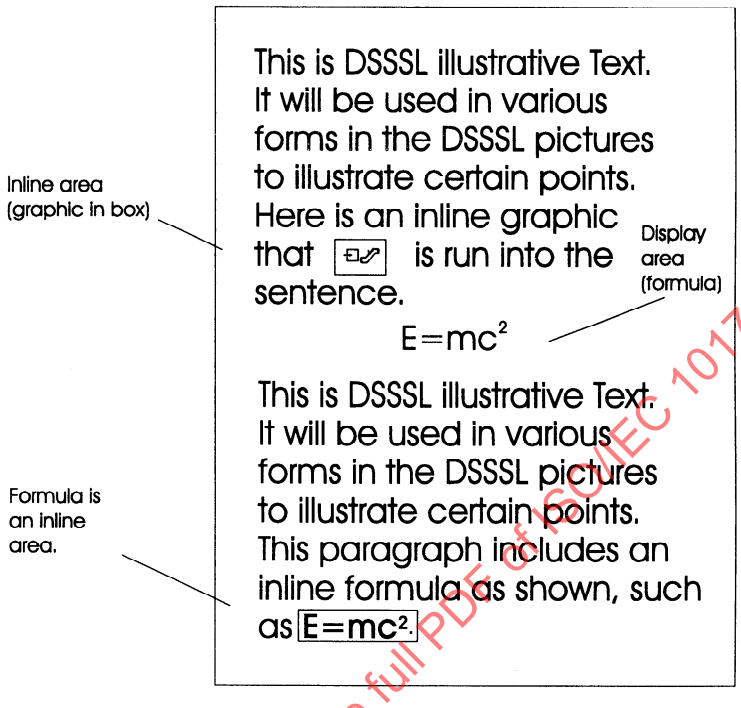

Figure 4 – Displayed and Inlined Areas

12.3.1 Display Areas

Display areas are areas that are not directly parts of lines. A display area has an inherent absolute orientation.

NOTE 43 Informally, the box has an arrow on it saying 'this way up'.

The positioning of display areas is specified by *area containers*. An area container has its own coordinate system with its origin at the lower left corner, the positive x-axis extending horizontally to the right and the positive y-axis extending vertically upward.

An area container has a filling-direction specified in terms of its own coordinate system. The filling-direction gives a starting edge and an ending edge which are opposite to each other. The size of an area container is always fixed in the direction perpendicular to the filling-direction. This means that the lengths of the starting and ending edges are always fixed and equal to each other.

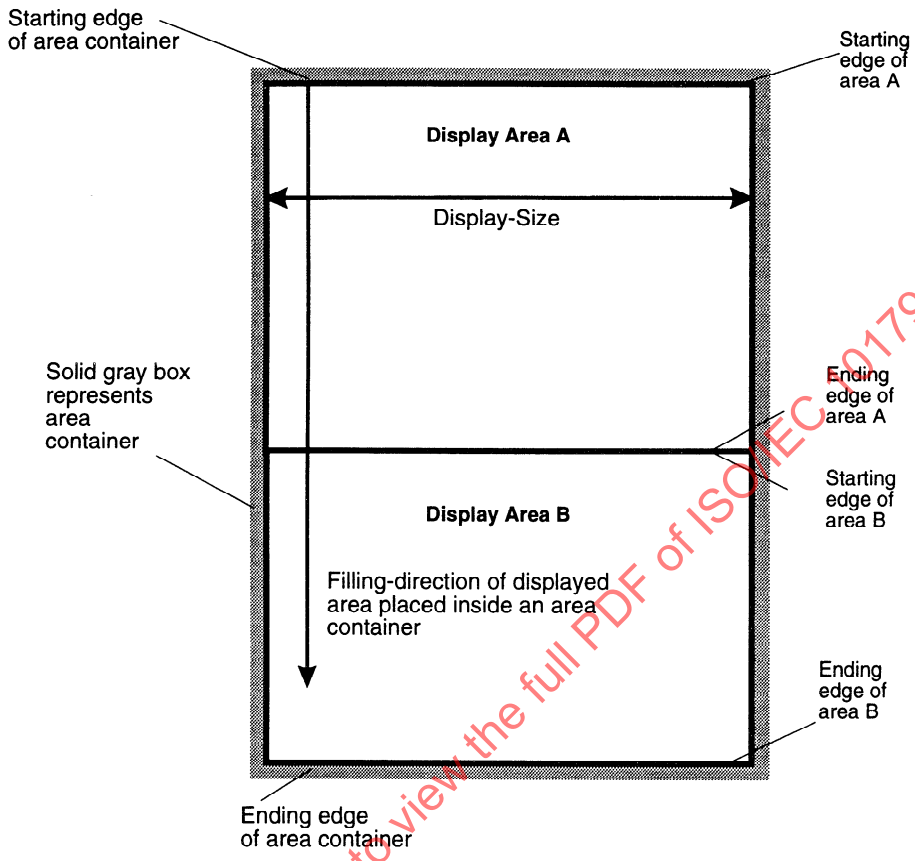

Figure 5 – Area Containers and Display Areas

The size of an area container in the filling-direction may be fixed or it may be specified to grow as necessary to contain the areas with which it is filled. The display areas with which an area container is filled are always created so that their size in the direction perpendicular to the filling-direction is equal to the size of the area container in that direction. This is called the *display-size* of the area. An area container is filled with a sequence of display areas as follows. The first display area is positioned with its starting edge aligned with the area container's starting edge. The next display area is then positioned with its starting edge on the previous area's ending edge, and so on. This is illustrated in Figure 5.

An area container resulting from an included-container-area flow object may also specify a rotation to be applied to each of the display areas with which it is to be filled. The angle of rotation is restricted to be a multiple of 90 degrees. This rotation is applied to each display area, thus changing the display area's starting and ending edges.

NOTE 44 It is possible to have paragraphs with lines with different placement directions on the same page without using rotation. See Figure 15.

The direction between a display area's starting and ending edges is the *placement direction* of the display area. A display area also has an associated writing-mode that is perpendicular to the area's placement direction. This is illustrated in Figure 6.

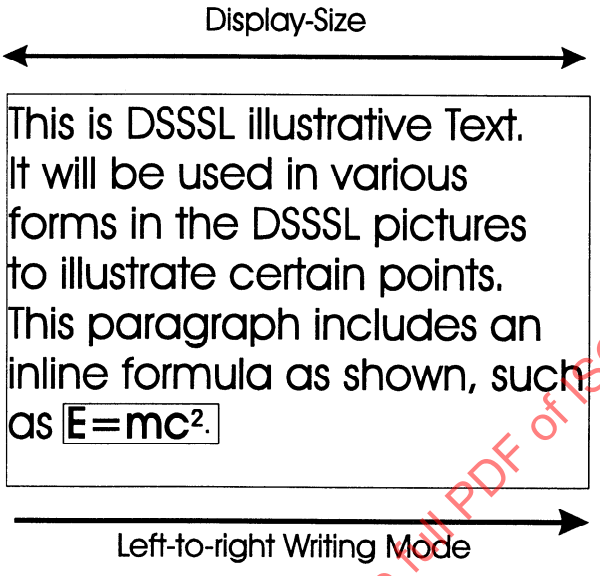

Figure 6 – Placement Direction for Left-to-Right Writing-Mode

Writing-mode may be left-to-right, right-to-left, or top-to-bottom. See Figure 7.

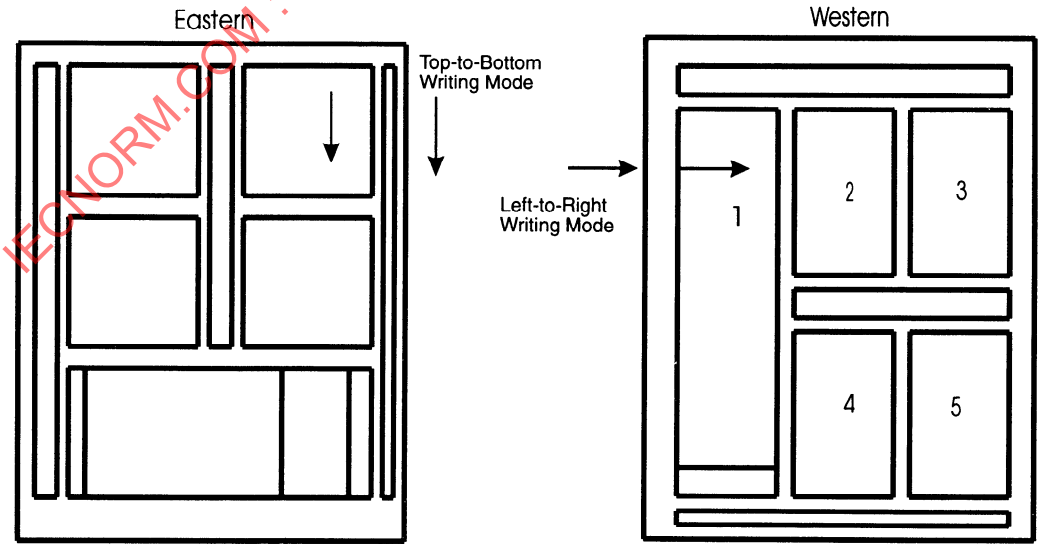

Figure 7 – Different Writing-modes

12.3.2 Inline Areas

Inline areas are areas that are parts of lines. An inline area has a *position point* that lies on one edge of its box and an orientation called the *escapement direction*, which is perpendicular to the edge of the box on which the position point lies. The point on the box which lies in the escapement direction from the position point and is on the opposite edge of the box is called the *escapement point* of the inline area.

NOTE 45 Informally the box has an arrow pointing from the position point that says 'place me so that the arrow lies parallel to the line I'm in'.

Inline areas are positioned to form lines in the following manner. The writing-mode for a paragraph gives an *inline-progression direction* for the paragraph. There is a *placement point* associated with the process of constructing a line. The first inline area is oriented so that its escapement direction is the same as the inline-progression direction of the paragraph, and the point on the inline area's box opposite to the position point becomes the current placement point. The next area is placed so that its position point is coincident with the current placement point and oriented so that its escapement direction is the same as the inline-progression direction of the paragraph. The point on the inline area's box opposite to the position point becomes the current placement point for placing the next area. This is illustrated in Figure 8.

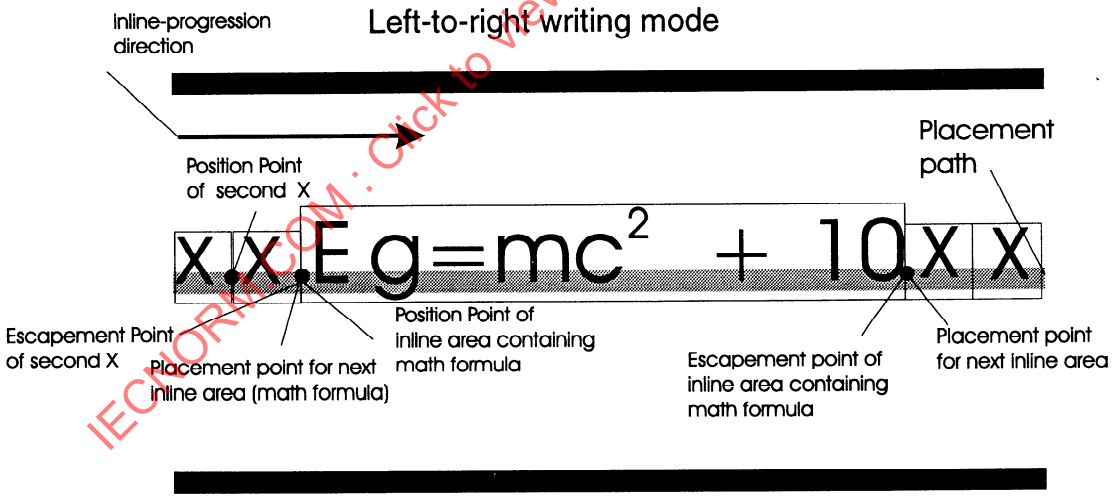

Figure 8 – Inline Area Placement and Positioning

The use of kerning modifies this positioning as illustrated in Figure 9.

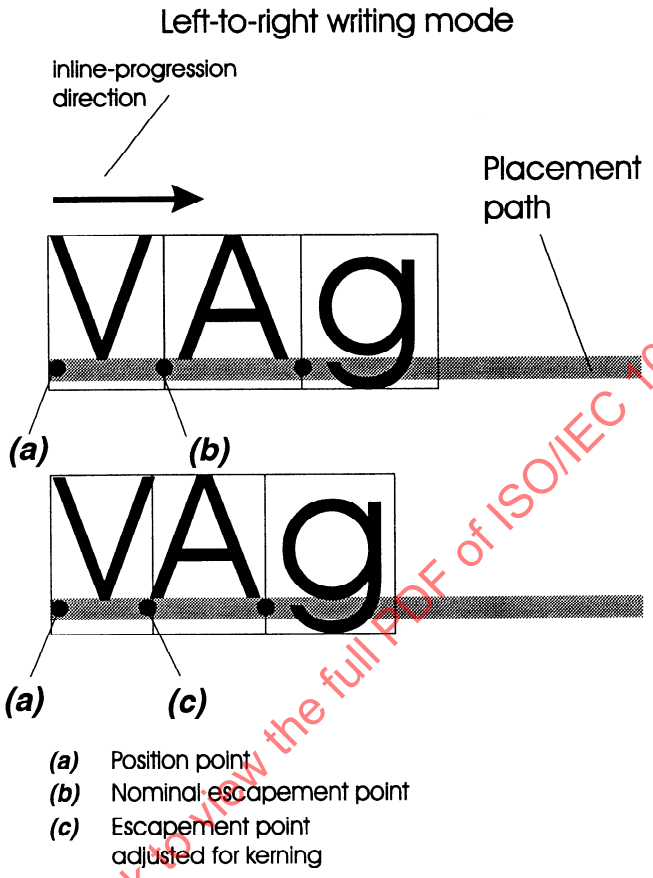

Figure 9 – Positioning with Kerning

The path containing the position points of the inline areas, which have the direction determined by the paragraph's writing-mode, is known as the *placement path*. This is illustrated in Figure 10 for the left-to-right writing-mode and in Figure 11 for the right-to-left writing-mode.

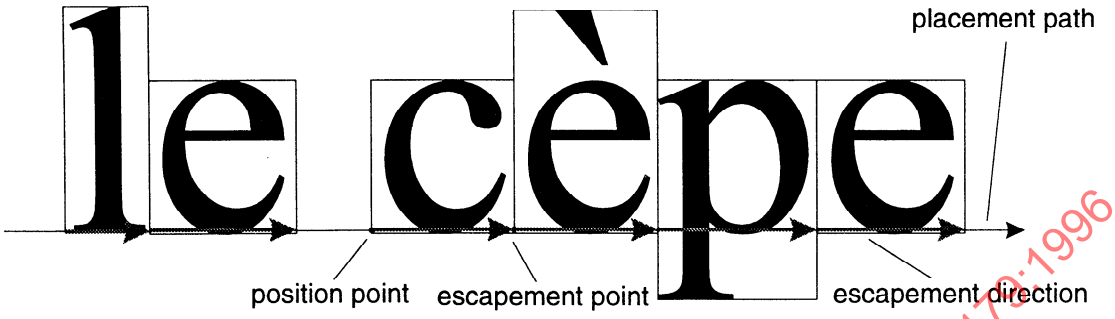

Figure 10 – Glyph Positioning for the Left-to-Right Writing-Mode

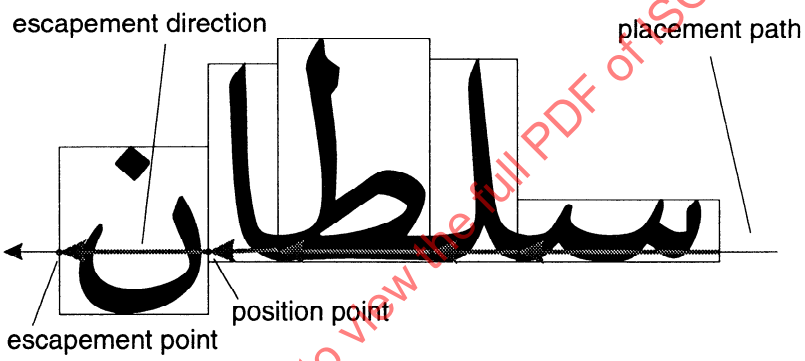

Figure 11 – Glyph Positioning for the Right-to-Left Writing-Mode

There are additional steps in the process when the paragraph uses more than one writing-mode. For example, in Figure 12, there is an inline-progression direction of left-to-right for the English text and an inline-progression direction of right-to-left for the Hebrew text. In addition, line breaking becomes more complex in this case.

Standard generalized תקן בינלאומי זה מגדיר שפת הצגת מסמכים הנקראת
Markup Language (SGML) ניתן להשתמש ב-SGML להוצאה לאור במובן הרחב
ביותר. החל מאמצעי ההוצאה לאור המקובלים, ועד להוצאה לאור רב-אמצעית
של מאגרי מידע. SGML שימושי גם בעיבוד מסמכים משרדי, אשר בו נחוץ
שילוב היתרונות של קלות הקריאה ע"י האדם ושל העברת המידע למערכות
ההוצאה לאור.

Figure 12 – Mixed Writing-Mode for Hebrew and English

The alignment mode specified by the alignment mode property for the font resource also influences how glyphs are positioned, as illustrated in Figure 13. There are characteristics on inlined flow objects that can modify this process.

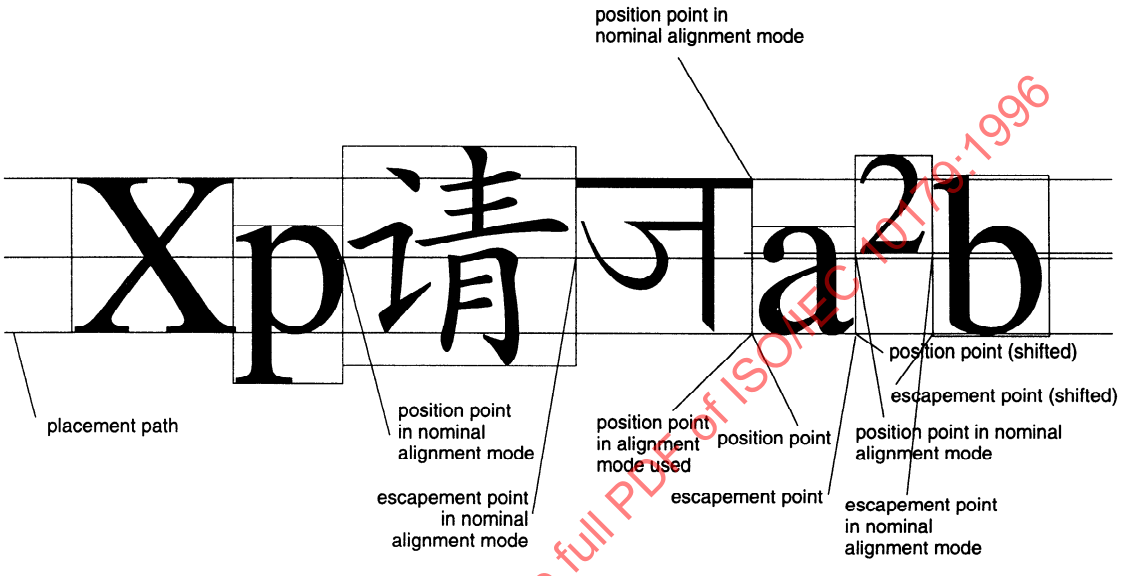

Figure 13 – Scripts with Mixed Alignment Modes

An inline area also has a *line-progression direction*, which is perpendicular to the inline-progression direction for its paragraph. Certain characteristics of inline areas are specified in terms of the line-progression direction.

12.3.3 Inlined and Displayed Flow Objects

A flow object that is to be formatted so as to produce a sequence of inline areas is said to be *inlined*. A flow object that is to be formatted so as to produce a sequence of display areas is said to be *displayed*. Instances of some flow object classes can only be inlined; instances of others can only be displayed; and instances of others can be either inlined or displayed. In the last case, whether a flow object is to be inlined or displayed is controlled by the characteristics of the flow object or by whether the flow objects attached to its ports are themselves inlined or displayed. The class of a flow object determines for each port of that flow object whether the flow objects associated with that port shall be inlined, or whether they shall be displayed, or whether they may be either inlined or displayed.

NOTE 46 The included-container-area flow object described in 12.6.16 allows a flow object that can only be displayed to occur indirectly in a line without causing a break. For example, one may wish to mix vertical Japanese in a line of English text without causing a break.

12.3.4 Attachment Areas

A display area can have a number of associated inline areas called *attachment areas*. These are illustrated in Figure 14 which shows the use of sidelines and graphics as attachment areas on either side of the display area.

NOTE 47 Attachment areas are used for sidelines, line numbers, and marginalia.

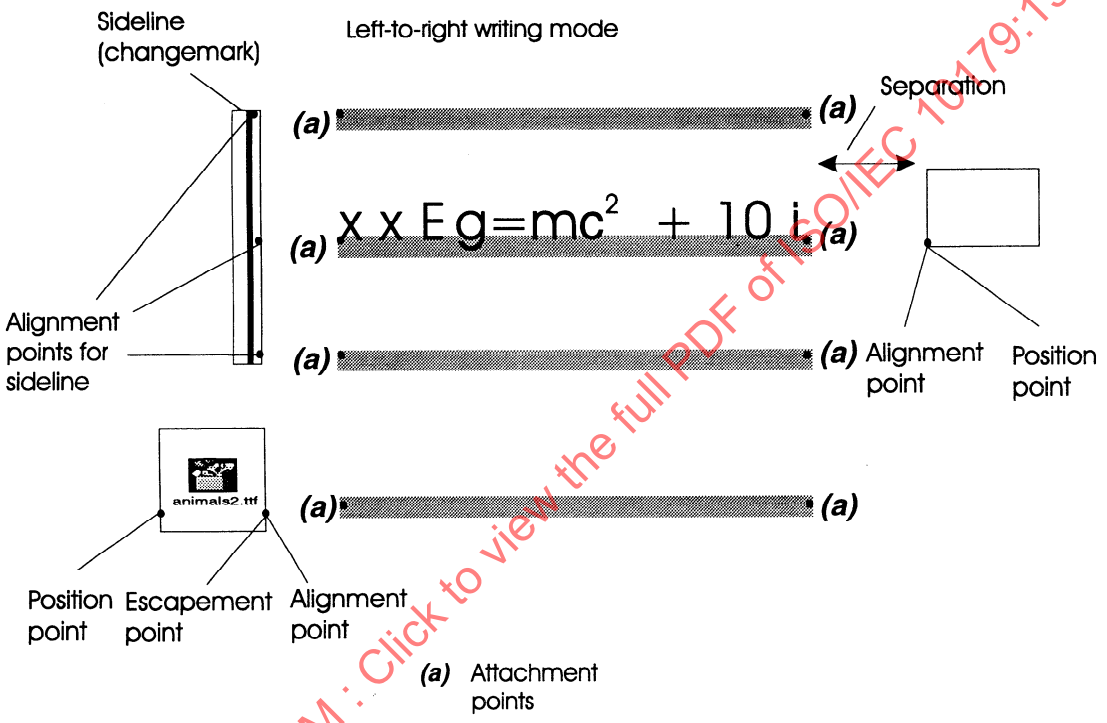

Figure 14 – Attachment Areas

Each attachment area is positioned relative to a point on the display area's box called the *attachment point* for the attachment area. The attachment point may be different for each of the attachment areas of the display area. An attachment point lies on an edge of the display area that is parallel to the placement direction.

There is a specification for each attachment area that indicates which such edge of the display area it is attached to. Each attachment area has an alignment point and is positioned so that the attachment area's alignment point is at the same position in the placement direction as the corresponding attachment point on the display area.

Each attachment area has a specified separation from the display area. If the attachment point is on the edge that is at the start in the direction determined by the writing-mode, then the separation is the distance in that direction from the attachment area's alignment point to the attachment point, and the attachment area's alignment point is its escapement point. If the

attachment point is on the edge that is at the end in the direction determined by the writing-mode, then the separation is the distance in that direction from the attachment point to attachment area's alignment point, and the attachment area's alignment point is its position point.

NOTE 48 A negative value for the separation means that the attachment point is inside the display area.

12.4 Flow Object Tree Construction

12.4.1 Construction Rules

[161] construction-rule = *query-construction-rule* | *id-construction-rule* | *element-construction-rule* | *root-construction-rule* | *default-element-construction-rule*

The construction-rules in a `style-specification` (see 7.1) specify how a node in the source grove is to be processed. Each *construction-rule* matches some (possibly empty) set of the nodes in a source grove. Refer to 9 for information about groves and their use in this International Standard.

A *construction-rule* includes a *construct-expression*, which is an *expression* returning an object of type sosofo. A *sosofo* is a specification of a sequence of flow objects to be added to the flow object tree. See 12.4.3. When a *construction-rule* is applied to a node, its *construct-expression* is evaluated. The node to which it is applied becomes the current node for the evaluation of the *construct-expression*.

The most specific *construction-rule* (as defined below) that matches the node is applied to the node.

NOTE 49 Processing a node has no side-effects; it just returns a value.

A node is processed with respect to a current processing mode. In addition to named processing modes that are specified with *mode-construction-rule-groups*, there is an initial processing mode that is unnamed. *construction-rules* not in any *mode-construction-rule-group* can match nodes both when the processing mode is the initial processing mode and when it is a named processing mode.

A flow object tree is constructed from a source grove by processing the root node of the source grove in the initial processing mode; the flow objects specified by the resulting sosofo are added as children of the root of the flow object tree. The flow objects specified by this sosofo shall all be unlabeled, and shall either be all of class scroll, or shall be all of class page-sequence or simple-page-sequence.

[162] mode-construction-rule-group = `(mode` *mode-name construction-rule**`)`

[163] mode-name = *identifier*

A *construction-rule* in a *mode-construction-rule-group* matches a node only when the current processing mode is *mode-name*.

The relative specificity of *construction-rules* is determined as follows:

— A *construction-rule* in a *mode-construction-rule-group* is more specific than any *construction-rule* not in a *mode-construction-rule-group*.

— Among *construction-rules* that have the same specificity according to the preceding rule, a *construction-rule* in one part of a `style-specification` is more specific than any *construction-rule* in a subsequent part (see 7.1).

— Among construction-rules that have the same specificity according to the preceding rules, each of the following is more specific than the next:

  – *query-construction-rule*

  – *id-construction-rule*

  – *element-construction-rule*

  – *default-element-construction-rule*

  – *root-construction-rule*

— A *query-construction-rule* is more specific than another *query-construction-rule* with a lesser priority.

— An *element-construction-rule* with a *qualified-gi* containing two or more *gis* is more specific than another *element-construction-rule* with no *qualified-gi* or with a *qualified-gi* containing fewer *gis*.

It shall be considered an error if there are two or more equally specific construction rules that match the node.

In addition to *construction-rules* explicitly specified in *style-language-bodys*, there is an implicit *default construction-rule*. The default construction rule matches any node in a source grove but is less specific than any explicitly specified *construction-rule*. The result returned by the default construction-rule shall depend on the type of node to which it is applied:

— for a node of class `sgml-document`, it shall return `(process-children)`.

— for a node of class `element`, it shall return `(process-children)`.

— for a node with a `char` property, it shall return `(make character)`.

— for a node of class `attribute-assignment`, it shall return `(process-children)`.

— for any other kind of node, it shall return `(empty-sosofo)`.

[164] query-construction-rule = (`query` *style-query-expression construct-expression priority-expression*?)

A *query-construction-rule* matches any node in the node-list returned by the *style-query-expression*. *query-construction-rule*s require the `query` feature.

[165] style-query-expression = *expression*

A *style-query-expression* shall return an object of type node-list. Within a *style-query-expression*, the `current-root` and `current-node` procedures both return the grove root of the grove being processed.

[166] construct-expression = *expression*

A *construct-expression* shall return an object of type sosofo. When the `query` feature is enabled, within a *construct-expression*, the `current-node` procedure shall return the current node.

[167] priority-expression = *expression*

The *priority-expression* specifies the priority of the *query-construction-rule*. It shall evaluate to a number. If the *priority-expression* is omitted, then the priority shall be 0. Bigger numbers indicate higher priorities.

[168] element-construction-rule = (`element` (*gi* | *qualified-gi*) *construct-expression*)

[169] gi = *string* | *symbol*

[170] qualified-gi = (*gi*+)

An *element-construction-rule* matches any node of class `element` that matches the *gi* or *qualified-gi*. A node matches a *gi* if its generic identifier is equal to the *string* or *symbol*. A node matches a *qualified-gi* if it matches the last *gi* in the *qualified-gi*, and its parent matches the next to last *gi*, and so on for each *gi* in the *qualified-gi*.

[171] default-element-construction-rule = (`default` *construct-expression*)

A *default-element-construction-rule* matches any node of class `element`.

[172] root-construction-rule = (`root` *construct-expression*)

A *root-construction-rule* matches any node of class `sgml-document`.

[173] id-construction-rule = (`id` *unique-id construct-expression*)

[174] unique-id = *symbol* | *string*

An *id-construction-rule* matches any node of class `element` that has a unique identifier equal to *unique-id*.

12.4.2 Primary Flow Object

A flow object is *associated with* a node in a source grove if it was constructed when that node was the current node and the flow object occurs in the flow object tree, that is, not within a reference value or a characteristic value. Flow objects constructed using the implicit default construction rule are considered to be associated with the nodes in the source grove for which the rule was applied, just as for flow objects constructed using explicit construction rules.

One flow object associated with a node is more closely associated with the node than another flow object associated with the node if:

— the one flow object was constructed when the current processing mode was the initial processing mode, and the other flow object was constructed when the current processing mode was some mode other than the initial processing mode, or

— the one flow object contains directly or indirectly the other flow object.

If there is a flow object associated with a node that is more closely associated with the node than any other flow object associated with the node, then that flow object is the *primary flow object* for the node.

12.4.3 Sosofos

An object of type sosofo is a specification of a sequence of flow objects to be added to the flow object tree.

NOTES

50 The expression language never operates on flow objects directly; it only operates on their specifications using the sosofo data type.

51 An implementation will use the information in a sosofo to construct portions of the flow object tree when a sosofo is returned by a *construct-expression* in a *construction-rule* that has been applied to some node in a source grove.

Each flow object specified by a sosofo may be labeled with a symbol. A sosofo whose members are all unlabeled is called an *unlabeled sosofo*.

NOTE 52 A flow object is labeled by specifying a `label:` argument in a *make-expression*.

`(sosofo? obj)`

Returns #t if `obj` is a sosofo, and otherwise returns #f.

[175] make-expression = `(make` *flow-object-class-name keyword-argument-list content-expression**`)`

[176] content-expression = *expression*

The result of evaluating a *make-expression* is a sosofo (the *result sosofo*) whose first specified member is a flow object of the class named by the *flow-object-class-name*. This flow object is called the *constructed flow object*. Each *content-expression* shall return an object of type sosofo. The sosofos returned by the *content-expression*s are concatenated to form the *content sosofo*. No *content-expression*s shall be specified if the *flow-object-class-name* is of an atomic flow object class. If the *flow-object-class-name* is not of an atomic flow object class and the *make-expression* contains no *content-expression*s, then a *content-expression* with the effect of `(process-children)` shall be used.

Each *make-expression* has a *content map* that maps labels to ports. Each flow object specified in the content sosofo is considered in turn. If it is unlabeled, it is appended to the stream attached to the principal port of the constructed flow object, if the constructed flow object has a principal port, otherwise this shall be an error. If it is labeled, and the label is one that is mapped by the content map, then the flow object is appended to the stream attached to the port of the flow object to which that label is mapped. Otherwise, the flow object is appended to the result sosofo; these flow objects are after the constructed flow object in the result sosofo.

A *keyword* shall be treated as part of the *keyword-argument-list* rather than as a *content-expression*. If the same keyword occurs more than once in the *keyword-argument-list*, it shall not be an error, but all except the first occurrence shall be ignored. The following keywords are allowed in the *keyword-argument-list*:

— A keyword that is the name of a characteristic and specifies the value of that characteristic for the flow object (unless it is an inherited characteristic that is overridden) as described in 12.4.6. If the characteristic is not inherited, then the characteristic shall be one that is applicable to the constructed flow object.

— A keyword `force-c:` where `c` is the name of an inherited characteristic that specifies the value of that characteristic for the flow object and prevents overriding of that value as described in 12.4.6.

— A keyword that is the name of a reference value type and specifies that the constructed flow object has a reference value of that type with the specified value.

— `use:` specifying a style to be used for the constructed flow object as described in 12.4.6. The value shall be a style object or #f indicating that no style shall be used.

— `content-map:` specifying the content map for the *make-expression*. The value shall be a list of lists of two objects, where the first object is a symbol that specifies a label and the second object is either a symbol specifying the name of a port or #f specifying the principal port. No label shall occur more than once in a content map.

  If the `content-map:` argument is not specified, then a content map shall be used that for each non-principal port of the flow object contains a list of two symbols both equal to the name of the port.

— `label:` specifying the label for the constructed flow object in the result sosofo. This argument shall be a symbol.

[177] flow-object-class-name = *identifier*

Any identifier that is the name of a flow object class is a *flow-object-class-name*.

[178] application-flow-object-class-declaration = `(declare-flow-object-class` *identifier string*`)`

This declares *identifier* to be a *flow-object-class-name* for a class with a public identifier specified by *string*.

[179] with-mode-expression = `(with-mode` *mode-specification expression*`)`

[180] mode-specification = *mode-name* | `#f`

A *with-mode-expression* evaluates *expression* with the processing mode specified by *mode-specification*. A *mode-specification* of `#f` indicates the initial unnamed processing mode. The *mode-name* in *mode-specification* shall have been specified in a *mode-construction-rule-group*.

`(empty-sosofo)`

Returns an empty sosofo.

`(literal` *string* …`)`

Returns a sosofo containing one flow object of class character for every char in *string*, … in the same order. Each character flow object is constructed as if by evaluating a *make-expression* with `character` as the *flow-object-class-name* and a `char:` argument specifying the character.

`(process-children)`

Returns the sosofo that results from appending the sosofos that result from processing in order the children of the current node. When the current node is of class `sgml-document`, the value of the `document-element` property is treated as being the children of the node.

`(process-children-trim)`

Returns the sosofo that results from appending the sosofos that result from processing in order the children of the current node after removing any leading and trailing sequence of nodes that have a `char` property with the `input-whitespace` property true.

`(process-matching-children` *pattern* …`)`

Returns the sosofo that results from appending the sosofos that result from processing in order those children of the current node that match any of *pattern*, ... A *pattern* shall be an object

that is allowed as the second argument to the `match-element?` procedure. It is interpreted as it is by `match-element?`.

`(process-first-descendant` *pattern* …`)`

Returns the sosofo that results from processing the first descendant in tree order of the current node that matches any of *pattern*, ... A *pattern* shall be an object that is allowed as the second argument to the `match-element?` procedure. It is interpreted as it is by `match-element?`.

`(process-element-with-id` *string*`)`

Returns the sosofo that results from processing the element in the same grove as the current node whose unique identifier is *string*, if there is such an element, and otherwise returns an empty sosofo. This procedure requires the `cross-reference` feature.

`(process-node-list` *ndlist*`)`

Returns the sosofo that results from appending the sosofos that result from processing the members of the *ndlist* in order. This requires the `query` feature.

`(map-constructor` *procedure node-list*`)`

For each node in *node-list*, *procedure* is evaluated with that node as a current node. *procedure* shall be a procedure of no arguments and shall return a sosofo. `map-constructor` shall return the sosofo that results from concatenating the results of evaluating the procedure. This requires the `query` feature.

`(sosofo-append` *sosofo* …`)`

Returns the sosofo that results from appending *sosofo* ....

`(sosofo-label` *sosofo symbol*`)`

Returns a sosofo that results from labeling with *symbol* each member of *sosofo* that is currently unlabeled. A new sosofo is constructed; neither the *sosofo* nor its members are modified.

`(sosofo-discard-labeled` *sosofo symbol*`)`

Returns a sosofo that results from discarding from *sosofo* any flow object that is labeled with *symbol*. A new sosofo is constructed; the *sosofo* is not modified.

`(next-match)`
`(next-match` *style*`)`

Returns the sosofo that results from applying the next most specific construction rule that matches the current node. If *style* is specified, then that style shall become the current overriding style for the evaluation of that construction rule.

12.4.4 Multi-process Feature

A call to any of the procedures `process-children`, `process-children-trim`, `process-matching-children`, or `process-first-descendant` is a *descending recursive call* if:

— it does not occur during the evaluation of a call to `process-node-set` or `process-element-with-id`, and

— it does not occur during the evaluation of the value of a reference value.

Unless the `multi-process` feature is enabled, it shall be an error if there occur two descending recursive calls both made when the same node was the current node and when the same processing mode was the current processing mode.

12.4.5 Styles

A style object contains a set of expressions specifying values for inherited characteristics.

[181] style-expression = `(style` *keyword-argument-list*`)`

Evaluates to an object of type style. The following keywords are allowed in the *keyword-argument-list*:

— A keyword that is the name of an inherited characteristic and specifies the value of that characteristic for the style (unless overridden) as described in 12.4.6.

— A keyword `force!`*c*`:` where *c* is the name of an inherited characteristic that specifies the value of that characteristic for the style and prevents overriding of that value as described in 12.4.6.

— `use:` specifying another style whose characteristics are to be added to this style as described in 12.4.6.

NOTE 53 A *style-expression* is interpreted in a similar manner to a *make-expression* with an atomic flow object class that has only inherited characteristics.

`(style?` *obj*`)`

Returns #t if *obj* is of type style, and otherwise returns #f.

`(merge-style` *style* …`)`

Returns a style object constructed by merging `style`, ... The expression for a characteristic in the returned style object is the expression for that characteristic in the first of the argument style objects that contains an expression for that characteristic.

12.4.6 Characteristic Specification

Every characteristic is *inherited* unless it is explicitly specified not to be in this International Standard. For each inherited characteristic, there is an expression in this International Standard specifying the initial value for that characteristic. Each non-inherited characteristic has a default value.

While a *construct-expression* is being evaluated, a current overriding style is in effect. When the processing of a node starts, the current overriding style is empty. The `next-match` procedure can change the current overriding style during the evaluation of a *construct-expression*. That *construct-expression* may, in turn, call `next-match` to change the current overriding style, and so on.

The expression specifying an inherited characteristic `c` for a flow object is determined when the *make-expression* is evaluated using the first of the following rules that is applicable:

— If a keyword of `force!c:` was specified, then the corresponding expression shall be used.

— If the current overriding style contains an expression for `c`, then that expression shall be used.

— If a keyword of `c:` was specified, then the corresponding expression shall be used.

— If `use:` was specified on the flow object, and the corresponding style object specifies an expression for `c`, then that expression shall be used.

— Otherwise, an expression `(inherited-c)` shall be used.

The set of characteristics and corresponding expressions for a style object is determined in a similar manner during the evaluation of the *style-expression*. For each inherited characteristic `c`, the expression that the style object has for `c` is determined using the first of the following rules that is applicable:

— If a keyword of `force!c:` was specified, then the corresponding expression shall be used.

— If the current overriding style contains an expression for `c`, then that expression shall be used.

— If a keyword of `c:` was specified, then the corresponding expression shall be used.

— If `use:` was specified on the flow object, and the corresponding style object specifies an expression for `c`, then that expression shall be used.

If none of these rules are applicable, then the style object contains no expression for `c`.

For each non-inherited characteristic *c* applicable to some flow object, if the *make-expression* for that flow object specifies the `c:` keyword, then the corresponding expression shall be evaluated and used; otherwise, the default for that characteristic shall be determined as specified for that characteristic and flow object class.

The expression specifying the value of a characteristic in a *make-expression* or *style-expression* shall not be evaluated immediately; instead the expression shall be associated with the characteristic in the created flow object or style object. The values of the free variables in the expression are remembered and are used when the expression is evaluated, as with a lambda expression. The current node is also remembered and restored for the evaluation of the expression.

When the flow object tree has been sufficiently constructed so that the position of a flow object in the flow object tree has been determined, then the expressions specifying the values for the characteristics applicable to that flow object shall be evaluated.

An expression specifying the value of a characteristic shall be evaluated with respect to two flow objects, which are referred to as the *value flow object* and the *specification flow object*. The value of a characteristic for a flow object is determined by evaluating the expression specifying that characteristic with both the value flow object and the specification flow object equal to that flow object.

`(inherited-c)`

For any inherited characteristic, *c*, there is a procedure `inherited-c`. This procedure shall be used only in the evaluation of an expression specifying a value for a characteristic. The procedure returns the result of evaluating the expression that specifies *c* for the flow parent of the specification flow object; this expression is evaluated with the value flow object unchanged and with the specification flow object equal to the flow parent of the current specification flow object. If the current specification flow object has no flow parent because it occurs as a characteristic value of some flow object, then that flow object shall be treated as the flow parent for this purpose. If the current specification flow object has no flow parent because it is used in a *generate-specification* or a *decoration-specification*, then the page-sequence or column-set-sequence flow object that is using the page-model or column-set-model in which that *generate-specification* or *decoration-specification* occurs shall be treated as the flow parent for this purpose. Otherwise, if the current specification flow object has no flow parent then `inherited-c` returns the result of evaluating the expression specifying the initial value of *c*; there is no specification flow object during the evaluation of this specification, and it shall be an error if it calls `inherited-c` for any inherited characteristic *c*.

The procedure `inherited-c` behaves differently when:

— the flow parent of the specification flow object is a table or a table-part;

— the value flow object is a table-cell of that table or table-part or is in a table-cell of that table or table-part; and

— the table or table-part contains a table-column flow object that specifies *c* and has the same column number as that table-cell.

In this case, `inherited-`*c* shall return the result of evaluating the specification of *c* in the table-column; this expression shall be evaluated with the value flow object unchanged and with the specification flow object equal to the table flow object.

`(actual-`*c*`)`

For each inherited characteristic *c*, `actual-`*c* shall return the value of *c* for the value flow object. This procedure shall be used only in the evaluation of an expression specifying a value for a characteristic. It shall be an error to call `actual-`*c* with a value flow object of *f* in the course of determining the value of *c* for *f*. Use of this procedure requires the `actual-characteristic` feature.

`(char-script-case` $string_1$ obj_1 … $string_{n-1}$ obj_{n-1} obj_n`)`

This procedure shall be used only in the evaluation of an expression specifying a value for an inherited characteristic. There shall be an odd number of arguments. All arguments other than the last shall be interpreted as a series of pairs, where the first member of the pair is a string specifying a public identifier, and the second member is any object. If the value flow object is not a character flow object or is a character flow object that has a script property that is not #f, then `char-script-case` shall return its last argument. Otherwise, the value of the script characteristic shall be compared in turn against the first member of each argument pair; if it matches, then the second member shall be returned; if there is no match, then the last argument shall be returned.

NOTE 54 For example, in formatting Japanese text, it is common to use different fonts for the Katakana, Han, and Latin portions of the text.

[182] application-characteristic-declaration = `(declare-characteristic` *identifier string expression* `)`

This declares *identifier* to be an additional inherited characteristic. It also has the effect of declaring procedures `inherited-`*identifier* and `actual-`*identifier*. The *string* is a public identifier specifying the semantics of the characteristic. If an implementation does not recognize the specified public identifier, it shall ignore uses of the characteristic. The *expression* is the specification of the initial value of the characteristic.

[183] application-char-characteristic+property-declaration = `(declare-char-characteristic+property` *identifier string expression*`)`

This declares *identifier* to be an additional non-inherited characteristic of a character flow object and also declares *identifier* to be an additional character property. The *string* shall be a public identifier specifying the semantics of the characteristic. The default value of the characteristic is the value of the *identifier* property of the character that is the value of the `char:` characteristic of the flow object. The default value of the property is the value of *expression*. This expression

shall be evaluated normally; it shall not be evaluated in the special way that the values of characteristics are evaluated, nor shall it be evaluated with respect to a current node.

[184] initial-value-declaration = `(declare-initial-value` *identifier expression*`)`

This declares the initial value of the inherited characteristic *identifier* to be an *expression*. This shall not be used for characteristics declared with an *application-characteristic-declaration*.

12.4.7 Synchronization of Flow Objects

Facilities in this clause require the `page` feature.

It is sometimes necessary to constrain the relative positioning of flow objects in different streams. For example, a footnote might be constrained to be on the same page as the corresponding reference, or a sidenote might be constrained to be at the same vertical position as its reference. Such constraints are specified by creating a *synchronization set*. A synchronization set is a set of flow objects whose relative positioning is constrained. A flow object contains information describing the synchronization sets to which it belongs. A flow object can belong to any number of synchronization sets. For every synchronization set, there shall be a flow object, the *synchronizing flow object*, that is a flow ancestor of all the flow objects in the synchronization set. In addition, each stream of that flow object can contain (either directly or as a descendant) at most one flow object in the synchronization set.

```
(sync sosofo1 sosofo2
      #!key type: min: max:)
```

Creates a synchronization set whose members are the first member of $sosofo_1$ and the first member of $sosofo_2$. `sync` returns a sosofo comprising:

a) a copy of the first flow object of $sosofo_1$ with added synchronization information,

b) any remaining flow objects of $sosofo_1$,

c) a copy of the first flow object of $sosofo_2$ with added synchronization information, and

d) any remaining flow objects of $sosofo_2$.

The `type:` argument is a symbol specifying the type of constraint on the areas created by formatting the synchronized flow objects. The `min:` and `max:` arguments are integers that further specify the type of constraint. The value of `max:` shall be greater than or equal to that of `min:`. `min:` and `max:` default to 0. The permitted values for `type:` are:

— `page` specifying that the number of pages separating

a) the first of the areas created from the first synchronized flow object from

b) the first of the areas created from the second synchronized flow object

shall not be less than `min:` nor greater than `max:`. The synchronizing flow object shall be a page-sequence flow object or a column-set-sequence flow object with a page-sequence flow object as an ancestor. The number of pages from one area to another area is defined to be the index, among all the pages of the page-sequence, of the page on which the second area lies minus the index of the page on which the first area lies.

NOTE 55 If `min:` were -1 and `max:` were 2, then the first of the areas created from the second synchronized flow object would be constrained to be either on the page before the first of the areas created from the first synchronized flow object, on the same page as the first of the areas created from the first synchronized flow object, on the page after the first of the areas created from the first synchronized flow object, or on the next page after that.

— `spread` specifying that the number of spreads from the first of the areas created from the first synchronized flow object to the first of the areas created from second synchronized flow object shall not be less than `min:` nor greater than `max:`. The synchronizing flow object shall be a page-sequence flow object or a column-set-sequence flow object with a page-sequence flow object as an ancestor.

— `column` specifying that the first of the areas created from the first synchronized flow object and the first of the areas created from the second synchronized flow object shall be in the same column-subset and that the number of columns from the first of the areas created from the first synchronized flow object to the first of the areas created from the second synchronized flow object shall be between `min:` and `max:`. The synchronizing flow object shall be of class column-set-sequence.

The default value of `type:` is `page`.

`(side-sync` *`list`*`)`

Creates a synchronization set containing the first members of each of the members of *`list`*, which shall be a list of two or more sosofos. `side-sync` returns the sosofo that results from concatenating the members of the list except that the first member of each sosofo is replaced by a copy with added synchronization information. The first areas produced by each member of the synchronization set are constrained to be positioned in the same column-set so that the position of their placement paths is the same in the filling-direction, possibly adjusted for any difference in alignment mode.

12.5 Common Data Types and Procedures

12.5.1 Layout-driven Generated Text

This clause describes the facilities for generating text when the value of the text to be generated at some point in the flow object tree may not be known until some formatting has been done. The facilities in this clause require the `page` feature.

NOTE 56 Examples of layout-driven generated text include page numbers, per-page footnote numbers, and dictionary heads.

Each such piece of generated text is represented by an indirect flow object. An indirect flow object contains a specification for a list of flow objects. The result of formatting an indirect flow object is the result of formatting the list of flow objects it specifies. Indirect flow objects are created only by using the procedures in 12.5.1.1 and are not created using the normal flow object creation mechanism. The *content* of the indirect flow object is defined to be the list of flow objects that it specifies. For the purposes of inheritance, the contents of an indirect flow object have the indirect flow object as their flow parent.

The generated-object data type is the specification of an expression-language object. The *kernel* of a generated-object is defined to be the object that is specified. The kernel of a generated-object is not available directly but only through the procedures in 12.5.1.1.

`(generated-object?` *`obj`*`)`

Returns #t if *`obj`* is of type generated-object, and otherwise returns #f.

12.5.1.1 Constructing Indirect Sosofos

`(general-indirect-sosofo` *`procedure generated-object`* `...)`

Returns a sosofo containing a single indirect flow object, the content of which is an unlabeled sosofo that is the result of applying the *`procedure`* to a list of the kernels of the *`generated-objects`*. This requires the `general-indirect` feature.

`(asis-indirect-sosofo` *`generated-object`*`)`

Returns a sosofo containing a single indirect flow object whose content is the kernel of *`generated-object`*. The kernel of *`generated-object`* shall be a sosofo.

NOTE 57 Typically, the generated-object is created by one of the procedures in 12.5.1.3.

`(number-indirect-sosofo` *`generated-object`* `#!key format: add: multiple:)`

Returns a sosofo containing a single indirect flow object whose content is the kernel of *`generated-object`*, which shall be an integer converted to a string and then to a sosofo. The keyword arguments control the conversion of the integer to a string as follows:

— `format:` is a string specifying the format to use for conversion of the number as in the `format-number` procedure. The default is 1.

— `add:` is an integer to be added to the kernel of *`generated-object`* before conversion. The default is 0.

— `multiple:` is an integer. The integers to be converted that are not multiples of this integer shall be converted to the empty string. The integer specified in the `add:` argument shall be added to the kernel of *`generated-object`* before testing whether it is a multiple. The default is 1.

12.5.1.2 Layout Numbering

The following procedures all return a generated-object whose kernel is a number that may depend on the result of formatting. When the `first-area-of-node:` and `last-area-of-node:` arguments are allowed, the number is specified relative to a reference area. At most one of the `first-area-of-node:` and `last-area-of-node:` arguments shall be supplied. If the `first-area-of-node:` argument is supplied, then its value shall be a node, and the reference area is the first area resulting from the primary flow object of that node. If the `last-area-of-node:` argument is supplied, then its value shall be a node, and the reference area is the last area resulting from the primary flow object of that node. One of `first-area-of-node:` or `last-area-of-node:` shall be supplied unless either:

— there is a current node when the procedure is evaluated, in which case the reference area is the first area resulting from the primary flow object of the current node, or

— the procedure is used within a *generate-specification*, in which case the reference area is the generated area, or

— the procedure is used in the construction of a decoration area, in which case the reference area is the decorated area.

Although a column is not an area, in this clause it is treated as an area, and an area is deemed to be in a particular column if it is in the column-set of that column and if that column is the first column in the column-set that the area spans.

It shall be an error to use one of the procedures defined in this clause in such a way that it requires the primary flow object of a node that has no primary flow object.

`(page-number #!key first-area-of-node: last-area-of-node:)`

Returns a generated-object whose kernel is the number of pages before or the same as the reference area.

`(category-page-number #!key first-area-of-node: last-area-of-node:)`

Returns a generated-object whose kernel is the number of pages before or the same as the reference area that has the same category as the page that is or that contains the reference area.

`(page-number-in-node` *nd*`)`

Returns a generated-object whose kernel is the number of pages that:

— are before or contain the first of the areas generated by the indirect-sosofo in which the generated-object is used, and

— contain areas from the flow object that corresponds to *nd*.

NOTE 58 This procedure could be used within a table header or footer.

`(total-node-page-numbers` *nd*`)`

Returns a generated-object whose kernel is the total number of pages that contain an area from the primary flow object associated with *nd*.

`(column-number #!key first-area-of-node: last-area-of-node:)`

Returns a generated-object whose kernel is the number of columns in the same column-subset as the reference area that is before or the same as the reference area.

`(footnote-number` *symbol* `#!key first-area-of-node: last-area-of-node:)`

Returns a generated-object whose kernel is the number of footnote areas that are before or the same as the reference area and are descendants of the nearest ancestor of the reference area that is of the type specified by *symbol*, which is one of `page`, `page-region`, or `column`. For this purpose, a footnote area is an area which is the first in the sequence of areas produced from a flow object whose stream is directed into the footnote zone of a column-set-sequence flow object.

`(line-number` *symbol* `#!key first-area-of-node: last-area-of-node:)`

Returns a generated-object whose kernel is the number of line areas that are before or the same as the reference area and are descendants of the nearest ancestor of the reference area that is of the type specified by *symbol*, where *symbol* is one of `page`, `page-region`, `column`, or `paragraph`. Line areas from paragraphs for which the `numbered-lines?:` characteristic was #f shall not be counted.

12.5.1.3 Reference Values

A flow object may have a number of named objects associated with it called *reference values*.

[185] reference-value-type-declaration = `(declare-reference-value-type` *identifier*`)`

A *reference-value-type-declaration* declares *identifier* to be the name of a reference-value type. The *identifier* shall not be the name of a characteristic or of any other keyword argument accepted by a *make-expression*.

```
(first-area-reference-value symbol #!key default: inherit:)
(last-area-reference-value symbol #!key default: inherit:)
(last-preceding-area-reference-value symbol  #!key default:)
(all-area-reference-values symbol #!key unique: inherit:)
```

Each of these procedures may be used only in a *generate-specification* or in the construction of a decoration area. The context in which these procedures are used determines a list of areas, the *associated-areas list*, on which these procedures operate.

When the procedures are used in the construction of a decoration area, the associated-areas list contains just the decorated area. When the procedures are used in a *generate-specification* in a

header-specification, *footer-specification*, or *footnote-separator-specification* in a *column-specification*, then the associated-area list contains the areas that are placed in the same column-set area container and that are in the body-text zone and that overlap the column. When the procedures are used in a *generate-specification* in a *header-specification* or *footer-specification*, or in a *page-region-specification*, then the associated-area list contains the areas that are placed in the same page-region area container as the generated area.

A flow object is *eligible* if

— it has a reference value `symbol`, or

— it has an ancestor with a reference value `symbol`, and `inherit:` is specified and is not #f.

The *relevant* reference value for an eligible flow object is the reference value `symbol` of the eligible flow object, if the eligible flow object has the reference value `symbol`, and otherwise is the reference value `symbol` of the nearest ancestor of the eligible flow object that has the reference value `symbol`.

`first-area-reference-value` does a pre-order traversal of the flow object tree searching for the first eligible flow object that produces an area that

— is one of the areas in the associated-area list, or

— is contained in one of the areas in the associated-area list

and returns a generated-object whose kernel is the value of the relevant reference value for that flow object. When a flow object has more than one stream, then each stream is searched separately. If the search finds flow objects in more than one stream, then the flow object that is earlier in the layout order of the area is returned. If the search finds no flow object, the value of the `default:` argument is returned, which shall be a generated-object.

`last-area-reference-value` behaves the same as `first-area-reference-value` except that the order of the search is reversed.

`last-preceding-area-reference-value` does a pre-order traversal of the flow object tree searching for the last eligible flow object, all of whose areas are before all the areas in the associated-areas list, and returns a generated-object whose kernel is the value of the relevant reference value for that flow object. If no flow object is found, the value of the `default:` argument is returned, which shall be a generated-object.

NOTE 59 This procedure might be used in the `default:` argument for the `first-area-reference-value` procedure.

`all-area-reference-values` does a pre-order traversal of the flow object tree searching for all eligible flow objects that produce an area that is, or is contained in, one of the areas in the associated-area list; it returns a generated-object whose kernel is a list containing the value of the relevant reference value for each such eligible flow object in the order in which it was found. If `unique:` is not #f, then duplicate (in the sense of `equal?`) values shall be discarded.

12.5.2 Length Specification

An object of type `length-spec` specifies a length as a linear combination of other lengths that may not be currently known. Whenever a value of type length-spec is required, a length (a quantity of dimension 1) may always be used.

```
(+ length-spec …)
(- length-spec …)
(* length-spec x)
(* x length-spec)
(/ length-spec x)
(/ x length-spec)
```

These procedures behave in the same way as their counterparts on quantities, except that they shall return a length-spec if any of their arguments is a length-spec (as opposed to just a length).

```
(display-size)
```

This procedure shall be used only in the evaluation of an expression specifying a value for a characteristic. The value flow object shall be a displayed flow object. It returns a length-spec specifying the display-size of the value flow object.

12.5.3 Decoration Areas

Facilities in this clause require the `page` feature.

An area container may be 'decorated' with one or more other areas called *decoration areas*. Decoration areas do not affect how parent areas treat the decorated area; in particular, they shall not change the width or height of the decorated area.

```
(decoration-area sosofo #!key placement-point-x:
                 placement-point-y: placement-direction:)
```

Returns an object of type decoration-area. The *sosofo* can specify a single flow object of any class that can be used inline. The result of formatting the *sosofo* is used as the decoration area. The decoration area has a placement point and a placement direction specified by the other arguments. The inline area produced by the *sosofo* is placed so that its position point lies on the placement point of the decoration area and its escapement direction is in the placement direction of the decoration area.

`placement-point-x:` is a length-spec specifying the distance between the bottom left corner of the decorated area and the placement point of the decoration area in the x-direction of the decorated area. `placement-point-y:` is a length-spec specifying the distance between the bottom left corner of the decorated area and the placement point of the decoration area in the y-direction of the decorated area. `placement-direction:` is one of the symbols `left-to-right`, `right-to-left`, or `top-to-bottom` giving the placement direction of the

decoration area relative to the orientation of the decorated area. In this case, the line-progression direction of the decoration area is the placement direction of the decorated area.

```
(decorated-area-width)
(decorated-area-height)
```

`decorated-area-width` and `decorated-area-height` return a length-spec specifying, respectively, the width and height of the area to be decorated. They may be used in the specification for the `placement-point-x:` and `placement-point-y:` arguments of a decoration-area.

12.5.4 Spaces

12.5.4.1 Display Spaces

Objects of type display-space are used to describe the desired space between displayed areas.

`(display-space? obj)`

Returns #t if *obj* is an object of type display-space, and otherwise returns #f.

`(display-space length-spec #!key min: max: conditional?: priority:)`

Returns an object of type display-space. *length-spec* specifies the nominal size of the space. `min:` and `max:` are length-specs specifying the minimum and maximum size of the space. These both default to the nominal size. `priority:` is either an integer or the symbol `force`. The default is 0. Higher integers indicate higher priorities. When two display-spaces are adjacent, then if one has a higher priority than the other, the minimum, nominal, and maximum values from the higher priority space shall be used, and the lower priority space shall be ignored. If the priorities are equal, but one display-space has a higher nominal value than the other, then the minimum, nominal, and maximum values from the space with the higher nominal value shall be used, and the other space shall be ignored. Otherwise, the priorities and nominal values are both equal; in this case, that nominal value, the lesser of the maximum values, and the greater of the minimum values shall be used. A priority of `force` is considered greater than any other priority. However, if both priorities are `force`, then the nominal, minimum, and maximum values shall be added together. The `conditional:` argument is a boolean; if true, the space shall be discarded if it starts an area. The default is #t.

NOTE 60 This allows spaces to disappear at page or column breaks.

12.5.4.2 Inline Spaces

Objects of type inline-space are used to describe the desired space between inline areas.

`(inline-space? obj)`

Returns #t if *obj* is an object of type inline-space, and otherwise returns #f.

`(inline-space` *`length-spec`* `#!key min: max:)`

Returns an object of type inline-space. *`length-spec`* specifies the nominal size of the space. `min:` and `max:` are length-specs specifying the minimum and maximum size of the space. These both default to the nominal size.

12.5.5 Glyph Identifiers

Glyph identifiers are represented by objects of type glyph-id.

`(glyph-id?` *`obj`*`)`

Returns #t if *`obj`* is a glyph-id, and otherwise returns #f.

`(glyph-id` *`string`*`)`

Returns a glyph-id with public identifier *`string`*.

[186] glyph-identifier = *afii-glyph-identifier*

[187] afii-glyph-identifier = `#A`*digit-10+*

An *afii-glyph-identifier* is a single token; therefore, no whitespace is allowed between the `#A` and the digits. An *afii-glyph-identifier* represents the glyph-id returned by

`(glyph-id "ISO/IEC 10036/RA//Glyphs::`*n*`")`

where *n* is the same sequence of digits occurring in the *afii-glyph-identifier* with leading zeros removed. The value represented by the digits shall be between 1 and 2^{32}-1.

12.5.6 Glyph Substitution Tables

An object of type glyph-subst-table represents a one-to-one mapping from glyph-ids to glyph-ids.

`(glyph-subst-table?` *`obj`*`)`

Returns #t if *`obj`* is of type glyph-subst-table, and otherwise returns #f.

`(glyph-subst-table` *`list`*`)`

Returns an object of type glyph-subst-table. *`list`* shall contain a list of pairs of glyph-ids. In the resulting glyph-subst-table, the substitution for the first member of each pair is the second member. The substitution for any glyph-id that does not occur as the first member of a pair is itself. If a glyph-id occurs as the first member of more than one pair, then the substitution for that glyph-id is the second member of the first pair that has that glyph-id as its first member.

`(glyph-subst` *`glyph-subst-table glyph-id`*`)`

Returns the glyph-id that substitutes for *glyph-id* in the glyph-subst-table.

12.5.7 Font Information

Facilities in this clause require the `font-info` feature.

```
(font-property string list
#!key size: name: family-name: weight: posture: structure:
proportionate-width: writing-mode:)
```

Returns the value of a property in a font resource. The arguments `name:`, `family-name:`, `weight:`, `posture:`, `structure:`, or `proportionate-width:` select the font in the same manner as the corresponding characteristics, with a prefix of `font-` added, of a character flow object. The `size:` argument is a length specifying the size of the font, which shall be supplied if the ISO/IEC 9541-1 data type of the value is REL-RATIONAL. *string* is a string representing a public identifier specifying the name of the property. *list* is a list, each of whose members is either:

— a string, or

— a list of three strings and an object.

The property value to be returned shall be determined as follows. Initially, the active property-list is the font-resource property-list. Each member of *list* in turn shall set the active property-list to a property-list nested in the active property-list, as follows:

— If the member is a string, then it shall set the property-list to the property-list that is the value of the property of that name in the active property-list.

— Otherwise, the active property-list shall be searched for a property whose name is equal to the first string. The value of the property shall be a property-list. The active property-list shall be set to the value of the property in that list whose name is equal to the second string and whose value is a property-list that contains a property whose name is equal to the third string and whose value is equal to the fourth member of the list.

Finally, the value of the property whose name is *string* in the active property-list shall be returned.

The optional `writing-mode:` argument shall have one of the values `left-to-right`, `right-to-left`, or `top-to-bottom`. The value `left-to-right` is equivalent to prefixing *list* with the list

```
("ISO/IEC 9541-1//WRMODES"
 "ISO/IEC 9541-1//WRMODE"
 "ISO/IEC 9541-1//WRMODENAME"
 "ISO/IEC 9541-1//LEFT-TO-RIGHT")
```

and so on for the other allowed values.

The object returned shall depend on the data type of the value of the property as defined in ISO/IEC 9541-1:

— for a BOOLEAN property, a boolean value shall be returned.

— for a STRUCTURED-NAME, a string containing the ISO 9070 canonical representation shall be returned.

— for MATCH-STRING or MESSAGE, a string shall be returned.

— for OCTET, INTEGER, CARDINAL, or CODE, a number shall be returned.

— for REL-RATIONAL, a length shall be returned which is obtained by scaling the font size.

— for ANGLE, a number shall be returned corresponding to the angle in degrees.

— for an OCTET-STRING, a list of integers shall be returned.

— for a value-list or an ordered-value-list, a list containing the result of converting the members of the value-list or ordered-value-list shall be returned.

Other types of values shall cause an error to be signaled.

12.5.8 Addresses

An address object shall be used as the destination of a hypertext link. An address object represents the address of one or more objects.

`(address? obj)`

Returns #t if *obj* is an object of type address, and otherwise returns #f.

`(address-local? address)`

Returns #t if the *address* is local to the current document, and otherwise returns #f.

`(address-visited? address)`

Returns #t if *address* has been visited, and otherwise returns #f.

`(hytime-linkend)`

Returns an object of type address. The current node shall be an element conforming to the clink architectural form as defined in ISO/IEC 10744. The address identifies the linkend of the current node.

`(idref-address string)`

The *string* is divided into one or more space-separated tokens, and an object of type address shall be returned representing the elements whose unique ID is one of the tokens.

`(current-node-address)`

Returns an address object representing the current node.

`(entity-address` *string*`)`

The *string* is divided into one or more space-separated tokens, and an object of type address shall be returned representing the entities whose names are the tokens.

`(sgml-document-address` $string_1$ $string_2$`)`

$string_1$ shall be the system identifier of an SGML document entity and $string_2$ shall be a unique ID in that SGML document. Returns an address object representing the element in the SGML document that has that unique ID.

`(node-list-address` *node-list*`)`

Returns an address object representing the nodes in *node-list*. This procedure requires the `query` feature.

NOTE 61 External procedures may be used to allow other addressing mechanisms.

12.5.9 Color

A color shall always be specified with respect to a color-space.

`(color-space` *string arg* ...`)`

Returns an object of type color-space. The *string* specifies a public identifier identifying the color-space family. The remaining arguments specify parameters to the color-space family. The type and number of the remaining arguments depend on the color-space family as described below.

`(color-space?` *obj*`)`

Returns #t if *obj* is a color-space, and otherwise returns #f.

`(color` *color-space arg* ...`)`

Returns an object of type color. *color-space* is the color-space relative to which color is to be specified. The type and number of the remaining arguments depend on the color-space family to which *color-space* belongs. If no arguments other than *color-space* are specified, then the default color in *color-space* is returned.

NOTE 62 This is normally black.

`(color? obj)`

Returns #t if *obj* is a color, and otherwise returns #f.

This International Standard defines the following color-space families:

— ISO/IEC 10179:1996//Color-Space Family::Device Gray

— ISO/IEC 10179:1996//Color-Space Family::Device RGB

— ISO/IEC 10179:1996//Color-Space Family::Device CMYK

— ISO/IEC 10179:1996//Color-Space Family::Device KX

— ISO/IEC 10179:1996//Color-Space Family::CIE LAB

— ISO/IEC 10179:1996//Color-Space Family::CIE LUV

— ISO/IEC 10179:1996//Color-Space Family::CIE Based ABC

— ISO/IEC 10179:1996//Color-Space Family::CIE Based A

The semantics of each of these color-space families is that of the corresponding color-space family in ISO/IEC 10180. The additional arguments required by `color-space` when one of these color-space families is specified as the first argument are determined by the parameters of the corresponding Color-Space Object in ISO/IEC 10180. When the ISO/IEC 10180 Color-Space Object has no parameters, `color-space` takes no additional arguments. When the ISO/IEC 10180 Color-Space Object has a single parameter of type Dictionary, `color-space` accepts a keyword argument for each key allowed in the Dictionary. The name of each keyword is derived from the name of the Dictionary key by inserting a hyphen before each upper-case letter in the name that is not the first letter and that is followed by a lower-case letter, and by then mapping all characters to lower-case. The type of each keyword argument shall be determined by the type of the corresponding Dictionary value:

— If the ISO/IEC 10180 type is a number, then the argument type shall be a number.

— If the ISO/IEC 10180 type is a procedure, then the argument type shall be a procedure.

— If the ISO/IEC 10180 type is a reference to a vector of numbers, then the argument type shall be a list of numbers of the same length.

— If the ISO/IEC 10180 type is a reference to a vector of procedures, then the argument type shall be a list of procedures of the same length.

The number and type of the additional arguments required by the `color` procedure when the first argument is a color-space that belongs to one of these families shall be determined by the number and type of the argument required by the ISO/IEC 10180 SetColor operator to specify a color in the corresponding ISO/IEC 10180 color-space. These additional arguments are all

required arguments (not keyword arguments). Their types are determined from the ISO/IEC 10180 types in the same manner as the arguments for `color-space`. The default color for each color-space is determined by the value that ISO/IEC 10180 defines the CurrentColor Graphics State Variable to have immediately after execution of the SetColorSpace operator for the corresponding ISO/IEC 10180 color-space.

NOTE 63 A color specified in a color-space with a procedure argument may be transformed in a device-independent manner to a color specified in a color-space without any procedure arguments. There is, therefore, no need when implementing the style language with output to an ISO/IEC 10180 device to be able to compile an arbitrary expression into the language defined in ISO/IEC 10180.

12.6 Flow Object Classes

12.6.1 Sequence Flow Object Class

A sequence flow object class is formatted to produce the concatenation of the areas produced by each of its children. It has a single principal port. Its children may be inlined or displayed.

NOTE 64 A sequence flow object is useful for specifying inherited characteristics. For example, a sequence flow object with a specification of a `font-posture:` characteristic may be constructed for an emphasized phrase element in a paragraph.

A port of a flow object shall accept a sequence flow object if and only if it would accept each of the flow objects in that sequence.

12.6.2 Display-group Flow Object

A display-group flow object class is formatted to produce the concatenation of the areas produced by each of its children. It has a single principal port. Its children shall all be displayed, and it is itself displayed.

NOTE 65 It will, therefore, cause a line break in a paragraph even if the display-group has no content.

The following characteristics are applicable:

— `coalesce-id:` is a string specifying the coalesce-id of the flow object, or #f if the flow object has no coalesce-id. This characteristic is not inherited. The default value is #f. If the areas from two or more flow objects with the same coalesce-id are flowed into the same `top-float`, `bottom-float`, or `footnote` zone of a column-set area, then the areas from the second and subsequent such flow objects shall be discarded. A value other than #f is allowed for this characteristic only if the flow object is flowed into a `top-float`, `bottom-float`, or `footnote` zone of a column-set.

— `position-preference:` is either #f or one of the symbols `top` or `bottom`. This applies if the flow object is directed into a port on a column-set-sequence flow object that is flowed into both the top-float and bottom-float zones of a column-subset and indicates whether the areas from this flow object may be flowed into only one of the zones. This characteristic is not inherited. The default value is #f.

— `space-before:` is an object of type display-space specifying space to be inserted before, in the placement direction, the areas produced by the flow object. This characteristic is not inherited. The default is for no space before to be inserted.

— `space-after:` is an object of type display-space specifying space to be inserted after, in the placement direction, the areas produced by the flow object. This characteristic is not inherited. The default is for no space after to be inserted.

— `keep-with-previous?:` is a boolean specifying whether the flow object shall be kept in the same area as the previous flow object. This characteristic is not inherited. The default value is #f.

— `keep-with-next?:` is a boolean specifying whether the flow object shall be kept in the same area as the next flow object. This characteristic is not inherited. The default value is #f.

— `break-before:` is #f or one of the symbols `page`, `page-region`, `column`, or `column-set` specifying that the flow object shall start an area of that type. This characteristic is not inherited. The default is #f.

— `break-after:` is #f or one of the symbols `page`, `page-region`, `column`, or `column-set` specifying that the flow object shall end an area of that type. This characteristic is not inherited. The default is #f.

— `keep:` is one of the following:

  – #t meaning that the areas produced by this flow object shall be kept together within the smallest possible area.

  – the symbol `page` indicating that the areas produced by the flow object shall lie within the same page; in this case, the flow object shall have an ancestor flow object of class page-sequence.

  – the symbol `column-set` indicating that the areas produced by the flow object shall lie within the same column set; in this case, the flow object shall have an ancestor of class column-set-sequence.

  – the symbol `column` indicating that the areas produced by the flow object shall lie within the same column set, and that the first column that each area spans in the column set shall be the same.

  – #f indicating that this characteristic is to be ignored.

  This characteristic is not inherited. The default value is #f.

— `may-violate-keep-before?:` is a boolean which, if true, specifies that constraints imposed by the `keep:` characteristics of ancestor flow objects on the relative positioning of this flow object and its previous flow object may not be respected. This characteristic is not inherited. The default value is #f.

— `may-violate-keep-after?:` is a boolean which, if true, specifies that constraints imposed by `keep:` characteristics of ancestor flow objects on the relative positioning of this flow object and its next flow object may not be respected. This characteristic is not inherited. The default value is #f.

12.6.3 Simple-page-sequence Flow Object Class

The facilities in this clause require the `simple-page` feature.

A simple-page-sequence flow object class is formatted to produce a sequence of page areas. A simple-page-sequence flow object has a single principal port that accepts any displayed flow object.

NOTE 66 The simple-page-sequence flow object is intended for systems that wish to provide a very simple page layout facility. More complex page layouts can be obtained with the page-sequence and column-set-sequence flow object classes.

A simple-page-sequence flow object shall not be allowed within the content of any other flow object class.

A simple-page-sequence may have a single-line header and footer containing text that is constant except for a page number.

NOTE 67 A document can contain multiple simple-page-sequences. For example, each chapter of a document could be a separate simple-page-sequence; this would allow the chapter title within a header or footer line.

The page shall be filled from top to bottom. The display-size for the contents of the simple-page-sequence shall be the value of the `page-width:` less the value of the `left-margin:` and `right-margin:` characteristics.

A simple-page-sequence flow object has the following characteristics:

— `page-width:` is a length specifying the total width of the page. The initial value is system-dependent.

— `page-height:` is a length specifying the total height of the page. The initial value is system-dependent.

— `left-margin:` is a length specifying the left margin. The initial value is 0pt.

— `right-margin:` is a length specifying the right margin. The initial value is 0pt.

— `top-margin:` is a length specifying the distance from the top of the page to the top of the area container used for the content of the simple-page-sequence. The initial value is 0pt.

NOTE 68 The header line is within the top margin.

— `bottom-margin`: is a length specifying the distance from the bottom of the page to the bottom of the area container used for the content of the simple-page-sequence. The initial value is 0pt.

NOTE 69 The footer line is within the bottom margin.

— `header-margin`: is a length specifying the distance from the top of the page to the placement path for the header line. The initial value is 0pt.

— `footer-margin`: is a length specifying the distance from the bottom of the page to the placement path for the footer line. The initial value is 0pt.

— `left-header`: is an unlabeled sosofo containing only inline flow objects that is aligned with the left margin of the page in the header line. This characteristic is not inherited. The default value is an empty sosofo.

— `center-header`: is an unlabeled sosofo containing only inline flow objects that is centered between the left and right margins of the page in the header line. This characteristic is not inherited. The default value is an empty sosofo.

— `right-header`: is an unlabeled sosofo containing only inline flow objects that is aligned with the right margin of the page in the header line. This characteristic is not inherited. The default value is an empty sosofo.

— `left-footer`: is an unlabeled sosofo containing only inline flow objects that is aligned with the left margin of the page in the footer line. This characteristic is not inherited. The default value is an empty sosofo.

— `center-footer`: is an unlabeled sosofo containing only inline flow objects that is centered between the left and right margins of the page in the footer line. This characteristic is not inherited. The default value is an empty sosofo.

— `right-footer`: is an unlabeled sosofo containing only inline flow objects that is aligned with the right margin of the page in footer line. This characteristic is not inherited. The default value is an empty sosofo.

— `writing-mode`: is one of the symbols `left-to-right` or `right-to-left`. This determines the writing-mode of the header and footer lines. The initial value is `left-to-right`.

`(page-number-sosofo)`

Returns an indirect-sosofo whose content is a sequence of character flow objects representing the page number of the page on which the first area resulting from the indirect flow object specified by the indirect-sosofo occurs.

`(current-node-page-number-sosofo)`

Returns an indirect-sosofo whose content is a sequence of character flow objects representing the page number of the primary flow object of the current node.

NOTE 70 This is intended to handle cross references in conjunction with `process-element-with-id`.

12.6.4 Page-sequence Flow Object Class

A page-sequence flow object is formatted to produce a sequence of page areas. The structure and positioning of the page areas shall be controlled by page-models.

A page-sequence flow object has the following characteristics:

— `initial-page-models:` is a list of page-models used for the initial pages. The initial value is the empty list.

— `repeat-page-models:` is a list of page-models used for pages after the initial pages. The initial value is the empty list.

— `force-last-page:` is either #f or one of the symbols `front` or `back` specifying the required type of the last page of the page-sequence. If the last page is not of the required type, then an additional blank page shall be generated. A value of #f indicates that the last page may be of either type. The initial value is #f.

— `force-first-page:` is either #f or one of the symbols `front` or `back` specifying the required type of the first page of the page-sequence. If the value is not #f, then the parent flow object shall be of type root; if there is a preceding flow object, then it shall be of type page-sequence. If the value of the `force-last-page:` characteristic of the preceding page-sequence is not #f, it shall have the opposite type to the specified value of the characteristic. If the last page of the preceding page-sequence is not of the opposite type to the value specified for this characteristic, then the preceding page-sequence shall have an additional blank page added. If there is no preceding flow object and the value is not #f, then it shall be an error if the specified type of the first page is not the actual type as determined by the `first-page-type:` characteristic. The initial value is #f.

— `first-page-type:` is either one of the symbols `front` or `back` indicating that the first page of the page-sequence is a front or back page, or the symbol `parent` indicating that the type of the first page shall be determined by the parent flow object. The initial value is `parent`. A value of `parent` shall be allowed only if the parent flow object is the root flow object. In this case, if there is a preceding flow object, then it shall be of type page-sequence, and the first page shall be a front or back page if the last page of the preceding page-sequence was a back or front page; if there is no preceding flow object, then the first page shall be a front page. This characteristic does not cause additional pages to be generated; it merely states that this page will be of the specified type when it is printed and bound. The value shall be `parent` unless the value of the `force-first-page:` characteristic is #f.

NOTE 71 This information makes it possible to determine which pairs of pages are spreads.

— `blank-back-page-model:` is a page-model that shall be used for the final page if it was a back page and was required only because of the `force-last-page:` or `force-first-page:` characteristics, or it is #f if the normal page-model should be used for the final page. The initial value is #f.

— `blank-front-page-model:` is a page-model that shall be used for the final page if it was a front page and was required only because of the `force-last-page:` or `force-first-page:` characteristics, or it is #f if the normal page-model should be used for the final page. The initial value is #f.

— `justify-spread?:` is a boolean specifying whether the bottom of each page in a spread shall be justified. The initial value is #f.

— `page-category:` specifies the category of the page areas resulting from this page-sequence flow object. It may be any expression language object for which the `equal?` procedure is defined. The category of an area is used by procedures defined in 12.5.1.2.

— `binding-edge:` is one of the symbols `left`, `right`, `top`, or `bottom` specifying the edge of a front page to be bound. This affects whether a side of the page is considered to be on the inside or outside. The initial value is `left`.

There shall be an applicable page-model for every page produced by the page-sequence.

The ports of a page-sequence flow object are determined by the page-models.

12.6.4.1 Page-model

A page-model is the specification of a set of possible hierarchies of areas.

`(page-model?` *obj*`)`

Returns #t if *obj* is of type page-model, and otherwise returns #f.

[188] page-model-definition = `(define-page-model` *page-model-name* [[*page-region-specification*+ | *width-specification* | *height-specification* | *filling-direction-specification*? | *decoration-specification**]])

[189] page-model-name = *variable*

`define-page-model` binds *page-model-name* to a page-model object.

The top-level area is the page area. The page area contains a number of sub-areas called *page-regions*. The layout order of the page-regions corresponds to the order of their specification in the *page-model-definition*. Page-regions may overlap.

[190] page-region-specification = `(region` [[*x-origin-specification* | *y-origin-specification* | *width-specification* | *height-specification* | *decoration-specification** | *filling-direction-specification*? | *header-specification*? | *footer-specification*? | *page-region-flow-map*?]])

A *page-region-specification* specifies an area container with fixed dimensions that is filled to produce a page-region area. Each page-region has a single predominant filling-direction.

NOTE 72 Included-container-area flow objects may use a different filling direction.

It is possible to have display areas with different placement directions on the same page using multiple page-regions, as illustrated in Figure 15.

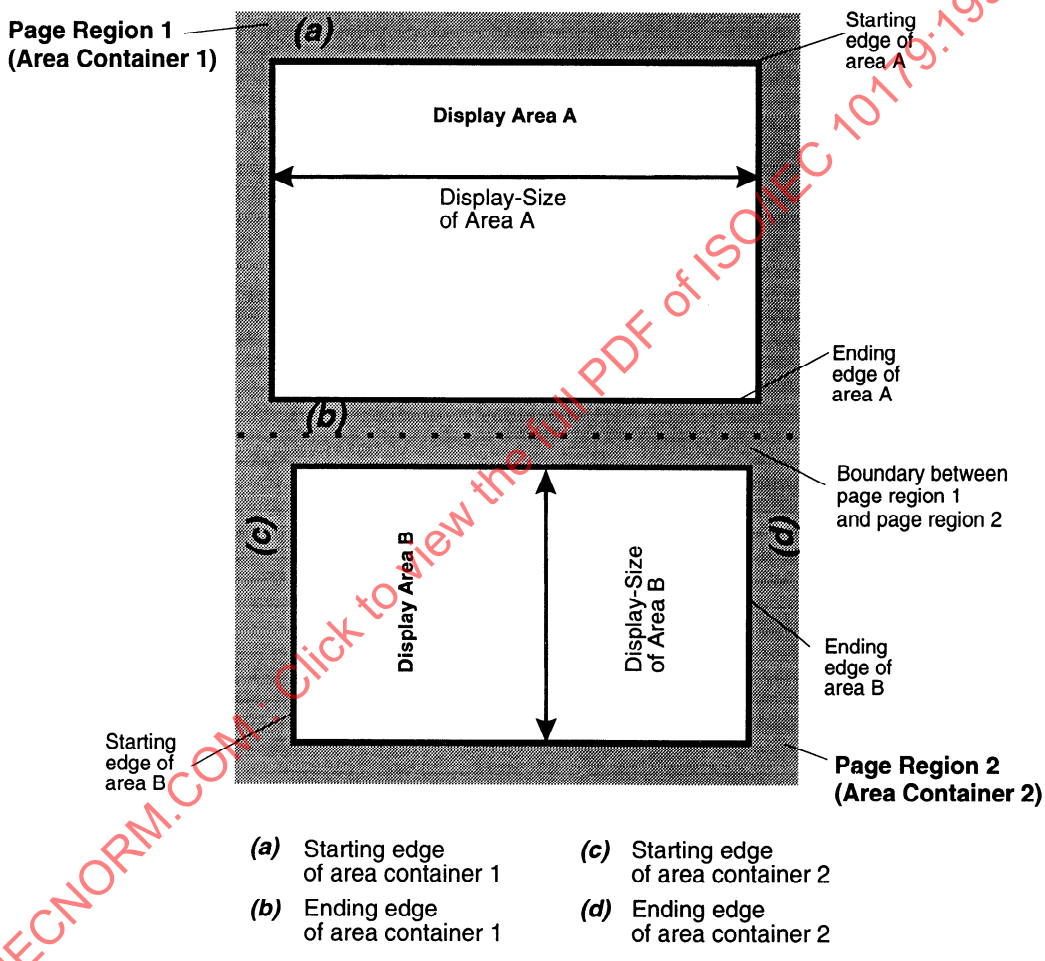

Figure 15 – Multiple Filling Directions on a Single Page

[191] page-region-flow-map = `(flow` *port-specifier+*`)`

A *page-region-flow-map* specifies that areas resulting from formatting flow objects directed into any of the ports identified by one of the *port-specifiers* may be assigned to this page-region.

If there is no *page-region-flow-map*, then `(flow #f)` is the default.

If a *port-specifier* occurs in more than one *page-region-flow-map* in a *page-region-specification* in a *page-model-definition*, then the page-regions shall be filled in the order in which their *page-region-specifications* occur in the *page-model-definition*.

[192] port-specifier = *identifier* | `#f`

A *port-specifier* that is an *identifier* specifies a port with that name; a *port-specifier* of `#f` specifies the principal port.

[193] header-specification = `(header` *generated-area-clauses* `)`

A *header-specification* specifies areas to be generated at the beginning of a page-region or column.

[194] footer-specification = `(footer` *generated-area-clauses* `)`

A *header-specification* specifies areas to be generated at the end of a page-region or column.

[195] generated-area-clauses = [[*height-specification*? | *width-specification*? | *filling-direction-specification*? | *contents-alignment-specification*? | *generate-specification*]]

generated-area-clauses specifies areas to be generated.

[196] generate-specification = `(generate` *expression*`)`

The *expression* shall evaluate to an unlabeled sosofo specifying only displayed flow objects.

[197] x-origin-specification = `(x-origin` *expression*`)`

The *expression* shall evaluate to a length which specifies the x component of the origin of the area container with respect to its parent's coordinate system.

[198] y-origin-specification = `(y-origin` *expression*`)`

The *expression* shall evaluate to a length which specifies the y component of the origin of the area container with respect to its parent's coordinate system.

[199] width-specification = `(width` *expression*`)`

The *expression* shall evaluate to a length which specifies the width (size in the positive x direction) of the area container with respect to its parent's coordinate system.

[200] height-specification = `(height` *expression*`)`

The *expression* shall evaluate to a length which specifies the height (size in the positive y-direction) of the area container with respect to its parent's coordinate system.

[201] decoration-specification = `(decorate` *expression*`)`

The *expression* shall evaluate to a decoration-area object. The area is decorated by the object as explained in 12.5.3.

[202] filling-direction-specification = `(filling-direction` *expression* `)`

The *expression* shall evaluate to one of the symbols `left-to-right`, `right-to-left`, or `top-to-bottom` specifying the filling-direction of the area container.

If the filling-direction is not specified on the page-region, it shall be inherited from the page-model. It shall be an error if it is not specified on either the page-region or the page-model.

[203] contents-alignment-specification = `(contents-alignment` *expression* `)`

The *expression* shall evaluate to one of the symbols `start`, `end`, `center`, or `justify` specifying the alignment of the child areas within the area container in the filling-direction of the area container. The default is `start`.

12.6.5 Column-set-sequence Flow Object Class

A column-set-sequence flow object is formatted to produce a sequence of column-set areas. A column-set area is a display area. A column-set area is produced by creating and filling an area container. A column-set area contains a set of parallel columns. Typically, column-set areas may be used to fill page-regions; however, column-set areas may also be used to fill other column-set areas. The structure and positioning of each column-set area shall be controlled by the column-set-model to which it conforms. A column-set-sequence flow object shall only be displayed.

A column-set-sequence has the following characteristics.

— `column-set-model-map:` is a list of lists each with two members, the first a page-model and the second a column-set-model; whenever an area from this column-set-sequence is placed in an area whose nearest ancestor of type page-region uses the specified page-model, then the specified column-set-model shall be used. The initial value is the empty list.

— `column-set-model:` is a column-set-model specifying the default column-set-model to use if none of the column-set-models specified in the `column-set-model-map:` characteristic are applicable or #f if there is no default column-set-model. If the value is #f, then it shall be an error if a result area is to be placed within a page-region whose page-model is not listed in the value of the `column-set-model-map:` characteristic. The initial value is #f.

— `position-preference:` is either #f or one of the symbols `top` or `bottom`. This applies if the flow object is directed into a port on a column-set-sequence flow object that is flowed into both the top-float and bottom-float zones of a column-subset and indicates whether the areas from this flow object may be flowed into only one of the zones. This characteristic is not inherited. The default value is #f.

— `span`: is a strictly positive integer specifying the number of columns that the areas resulting from this flow object shall span. This characteristic shall apply if the flow object is directed into a port on a column-set-sequence flow object that is flowed into the top-float, bottom-float, or body-text zone of a spannable column-subset. The initial value is 1.

— `span-weak?`: is a boolean specifying whether the areas resulting from this flow object span weakly rather than strongly. See 12.6.5.1. This characteristic applies if the flow object is directed into a port on a column-set-sequence flow object that is flowed into the top-float, bottom-float, or body-text zone of a spannable column-subset and has a `span`: characteristic with a value greater than 1. The initial value is #f.

— `space-before`: is an object of type display-space specifying space to be inserted before, in the placement direction, the areas produced by the flow object. This characteristic is not inherited. The default is for no space before to be inserted.

— `space-after`: is an object of type display-space specifying space to be inserted after, in the placement direction, the areas produced by the flow object. This characteristic is not inherited. The default is for no space after to be inserted.

— `keep-with-previous?`: is a boolean specifying whether the flow object shall be kept in the same area as the previous flow object. This characteristic is not inherited. The default value is #f.

— `keep-with-next?`: is a boolean specifying whether the flow object shall be kept in the same area as the next flow object. This characteristic is not inherited. The default value is #f.

— `break-before`: is #f or one of the symbols `page`, `page-region`, `column`, or `column-set` specifying that the flow object shall start an area of that type. This characteristic is not inherited. The default is #f.

— `break-after`: is #f or one of the symbols `page`, `page-region`, `column`, or `column-set` specifying that the flow object shall end an area of that type. This characteristic is not inherited. The default is #f.

— `keep`: is one of the following:

  – #t meaning that the areas produced by this flow object shall be kept together within the smallest possible area.

  – the symbol `page` indicating that the areas produced by the flow object shall lie within the same page; in this case, the flow object shall have an ancestor flow object of class page-sequence.

  – the symbol `column-set` indicating that the areas produced by the flow object shall lie within the same column set; in this case, the flow object shall have an ancestor of class column-set-sequence.

- the symbol `column` indicating that the areas produced by the flow object shall lie within the same column set, and that the first column that each area spans in the column set shall be the same.

- #f indicating that this characteristic is to be ignored.

This characteristic is not inherited. The default value is #f.

— `may-violate-keep-before?:` is a boolean which, if true, specifies that constraints imposed by the `keep:` characteristics of ancestor flow objects on the relative positioning of this flow object and its previous flow object may not be respected. This characteristic is not inherited. The default value is #f.

— `may-violate-keep-after?:` is a boolean which, if true, specifies that constraints imposed by `keep:` characteristics of ancestor flow objects on the relative positioning of this flow object and its next flow object may not be respected. This characteristic is not inherited. The default value is #f.

A column-set-sequence flow object has a port for each port listed in a *column-subset-flow-map* for any of its column-set-models.

12.6.5.1 Column-set-model

A column-set-model specifies the possible hierarchy of areas for each column-set. For some possible examples of column-sets and column-subset configurations, see Figures 16 and 17.

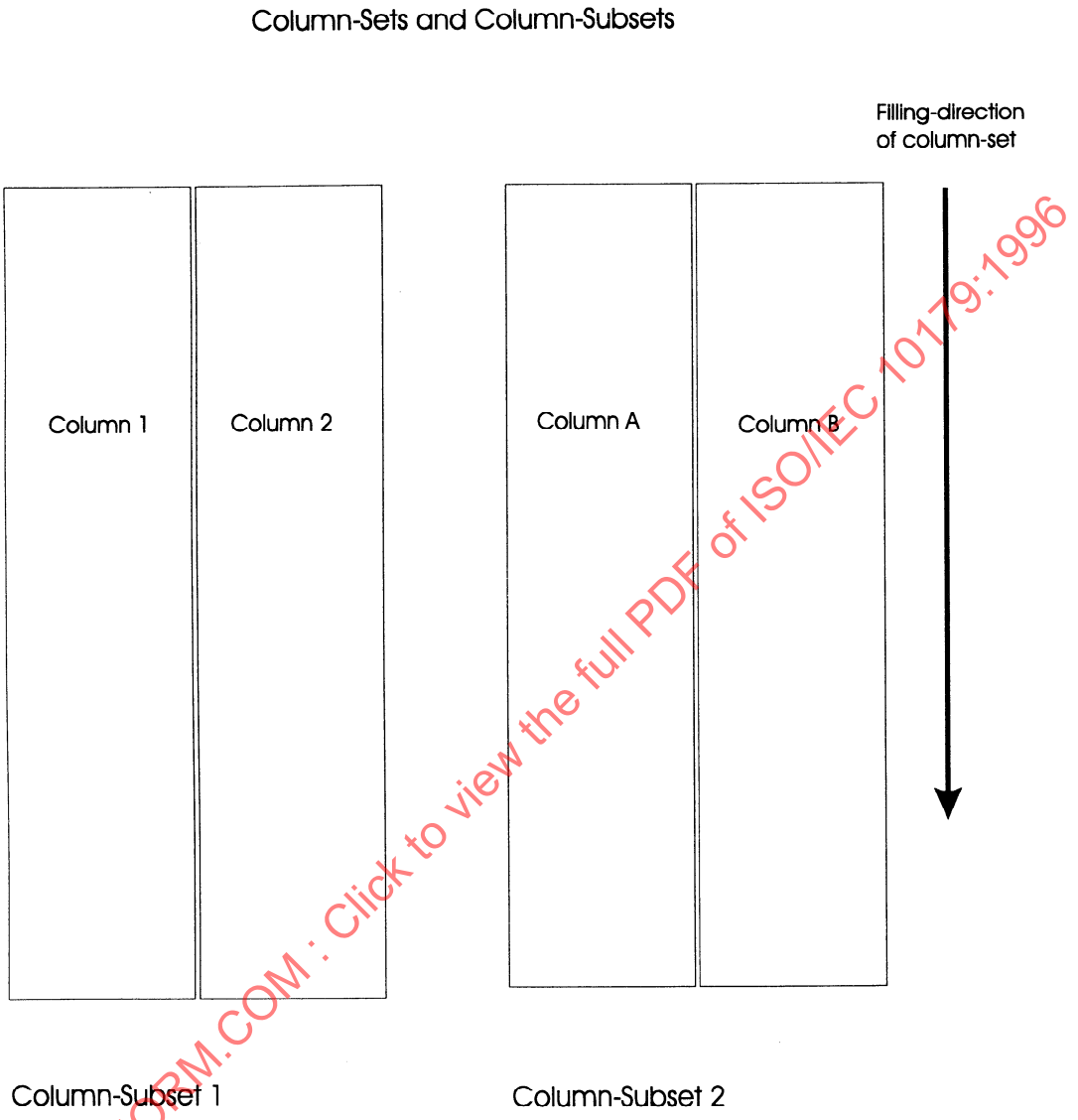

Figure 16 – An Example of Column-Subsets

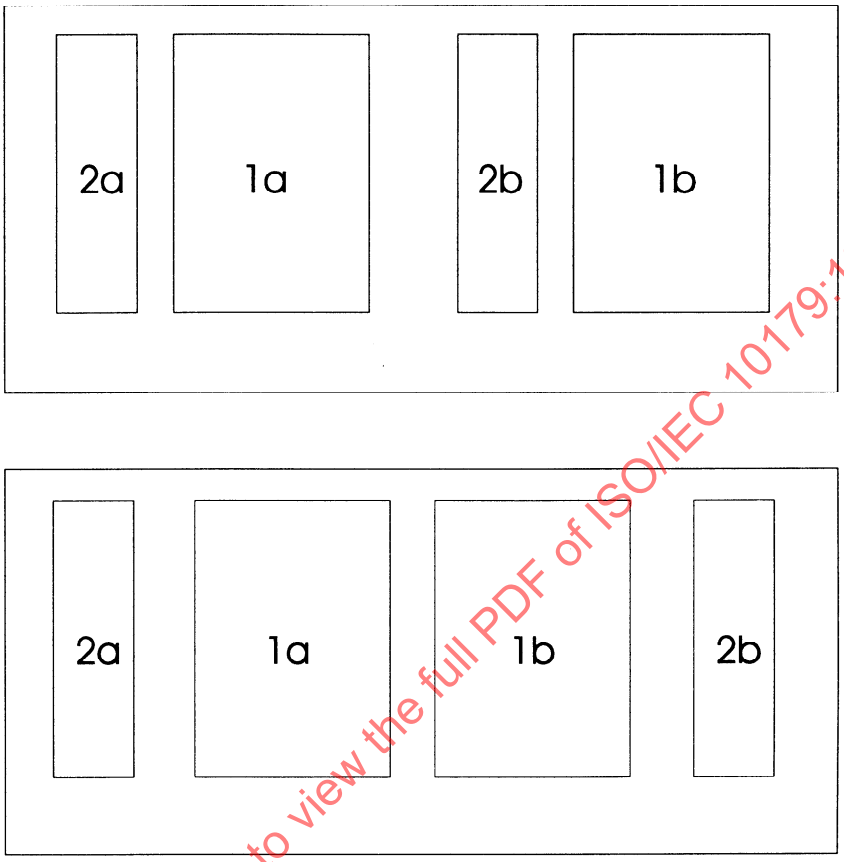

Figure 17 – Another Example of Column-Subsets

The top-level area in the hierarchy is the column-set area. A column-set area shall have a filling-direction. If the *column-set-model-definition* does not contain a *filling-direction-specification*, then the filling-direction of the parent area shall be used. The size of the column-set area shall be fixed in the direction perpendicular to the filling-direction. It can be fixed either by a *width-specification* or a *height-specification* or because this direction is the direction perpendicular to the area's placement direction. The size of a column-set area in the filling-direction may be fixed, or it may grow according to the areas flowed into it.

The area container that produces the column-set shall be filled in a more complicated way than normal area containers. Areas are placed in the column-set area in such a way that they satisfy a number of different constraints.

The most basic constraint is that the areas shall not overlap. This constraint does not apply to decoration areas.

There is a partial ordering defined on the areas that have been placed in a column-set area. This is called the *layout order*.

NOTE 73 The layout order corresponds to the order in which the areas should be read.

A fundamental constraint on the filling of an area container is that if two areas placed in the column-set area container come from the same stream, then they shall be placed so that their layout order is consistent with their order in the stream.

The column-set area is divided geometrically in a direction parallel to the filling-direction into a number of columns.

NOTE 74 When an area is said to be divided in some direction, this means that it is divided in such a way that the dividing line is in that direction.

A column is not an area container. Each column has an extent that is fixed in the direction perpendicular to the filling-direction.

Each column is a member of exactly one column-subset. The layout order of columns in a column-subset is the order of the *column-specification*s in a column-subset specification. There is no layout order defined between columns in different column-sets.

NOTE 75 It is for this reason that the layout order is a partial order.

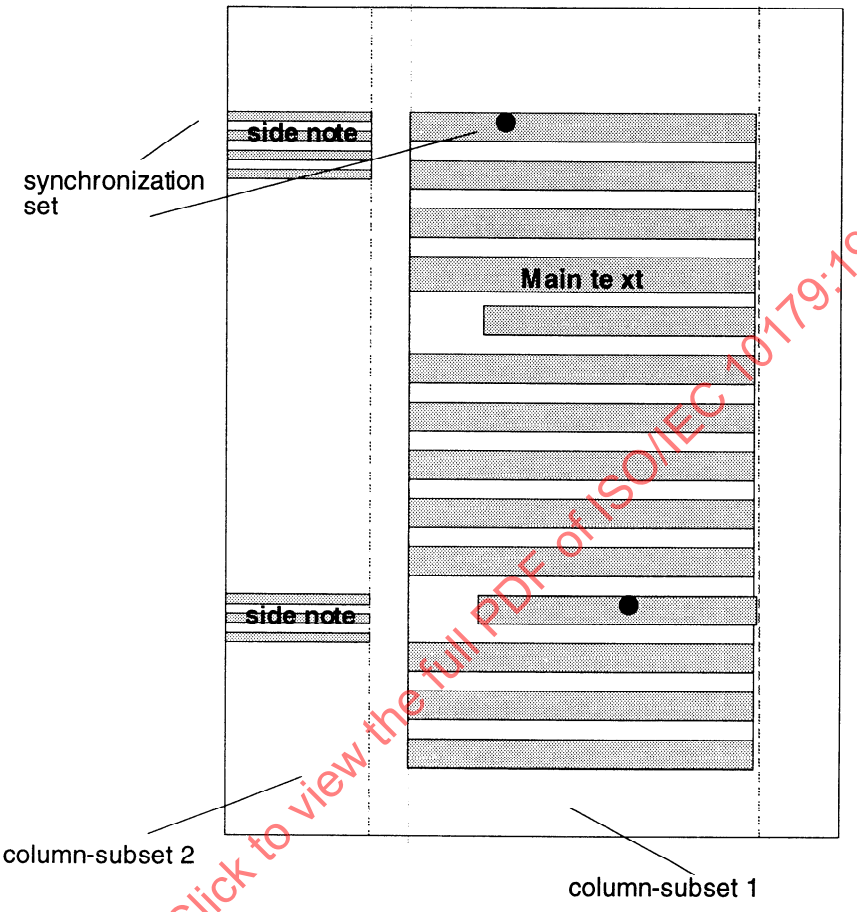

Figure 18 – Multiple Column-Subsets

A column-subset is defined to be *spannable* unless a column in the column-subset is geometrically between any two other consecutive columns in the column-subset. For example, see Figure 19.

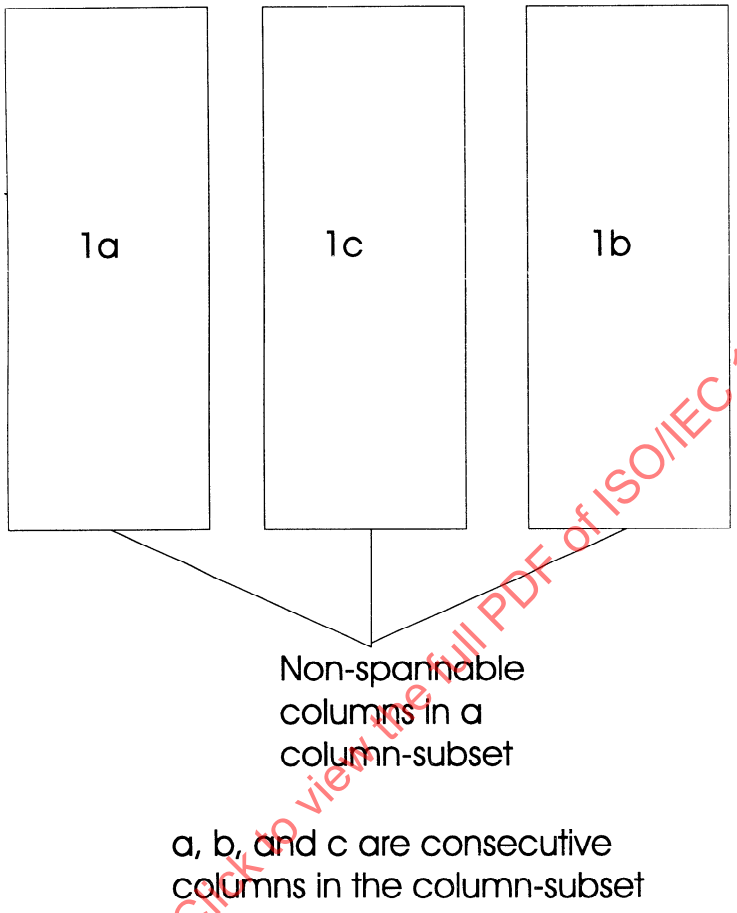

Figure 19 – Non-spannable Column-Subsets

Each area to be placed in a column-set area shall be associated with a single column-subset. If the filling-direction of the column-set area is top-to-bottom, each area that is placed in the column-set area shall be placed so that the left edge is aligned with the left edge of a column in the column-subset and the right edge is aligned with the right edge of a column in the same column-subset. If the filling-direction of the column-set area is left-to-right or right-to-left, each area that is placed in the column-set area shall be placed so that its top edge is aligned with the top edge of a column in the column-subset and its bottom edge is aligned with the bottom edge of a column in the same column-subset. An area may span more than one column only if the column-subset is spannable. The number of columns in the column-subset that an area spans shall be equal to the value of the `span:` characteristic of the flow object from which the area comes.

An area that is to be placed in a column-set area shall be created in such a way that its size in the direction perpendicular to the filling direction is such that it exactly spans the required number of columns. In other words, the display-size of the area shall be equal to the distance between one edge of the first column it spans and the opposite edge of the last column it spans.

NOTE 76 This is an exception to the general principle that an area to be placed in an area container is created so that the area's size in the direction perpendicular to the area's placement direction is equal to the size of the area container in the direction perpendicular to the area container's filling-direction.

Each area that is to be placed in a column-set area container is labeled with a *zone*, which constrains the placement of the area relative to other areas. The allowed zones are `top-float`, `body-text`, `bottom-float`, and `footnote`. An area labeled with one zone shall be positioned so that it precedes, in the filling-direction, an area that is labeled with a zone that is later in the list, unless there is no column that is spanned by both areas. For example, see Figure 20.

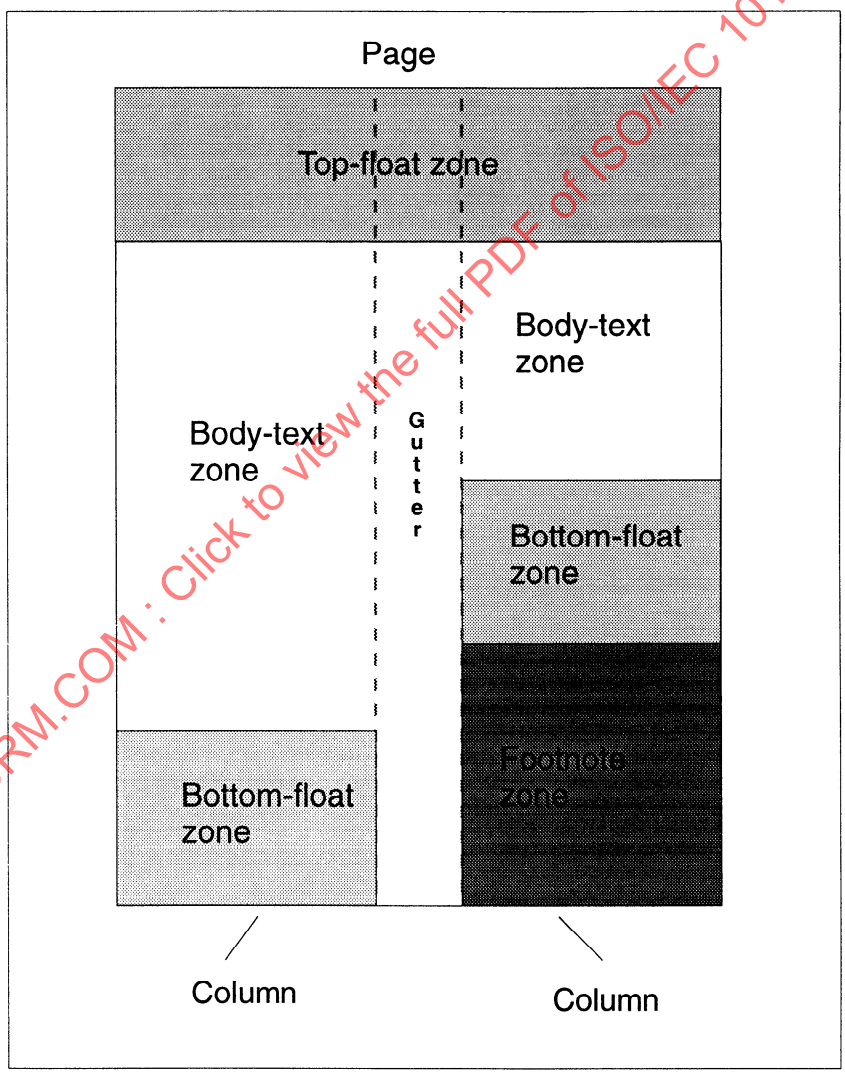

Figure 20 – Column-set areas

An area labeled with the footnote zone shall span exactly one column.

NOTE 77 Full-width footnotes in a multi-column layout may be achieved using a nested-column-set.

An area that spans more than one column may span either *weakly* or *strongly* depending on the value of the `span-weak?:` characteristic on the flow object from which the area comes. An area that spans more than one column strongly is defined to follow in the layout order any areas that:

— are in the same column-subset as the area,

— precede the area geometrically in the filling-direction,

— have a span that is completely included in the span of the area, and

— are labeled with the same zone as the area.

An area that spans more than one column weakly is defined to follow in the layout order exactly those areas that it would follow if it occupied only the first of the columns that it spans.

Two or more column-subsets may be *tied together*. Column-subsets that are tied together shall have the same number of columns. When an area spans strongly more than one column of a column-subset, then the layout order of each column-subset that is tied to that column-subset shall be modified as if an empty area had been created and placed at the same position in the filling-direction as the spanning area and with the same size in the filling-direction as the spanning area so that it spans the corresponding columns of the tied-column-subset; this area can overlap the spanning area.

NOTE 78 A sequence of columns containing sidenotes is usually tied to the sequence of columns containing the text to which the sidenotes refer.

When the spanning area is synchronized using the `side-sync` procedure with an area in a tied-column-subset that does not span, then it shall be placed in the first column in the tied column-subset:

— whose corresponding column in the other column-subset is spanned by the spanning area, and

— which is not covered by the spanning area.

[204] column-set-model-definition = `(define-column-set-model` *variable* [[*column-subset-specification** | *fill-out-specification*? | *tied-column-subset-specification** | *filling-direction-specification*? | *width-specification*? | *height-specification*? | *decoration-specification**]])

A *column-set-model-definition* defines *variable* to be an object of type column-set-model.

`(column-set-model?` *obj*`)`

Returns #t if *obj* is of type column-set-model, and otherwise returns #f.

[205] fill-out-specification = `(fill-out` *expression*`)`

The *expression* shall evaluate to a boolean. If it is #t, then each column-set area shall be filled out in the filling-direction to the maximum size allowed by the area in which it is placed.

[206] column-subset-specification = `(column-subset` [[*column-specification*+ | *column-subset-flow-map* | *top-float-space-below-specification*? | *bottom-float-space-above-specification*? | *balance-specification*? | *justify-specification*? | *justify-limit-specification*? | *justify-last-limit-specification*? | *length-deviation-specification*? | *length-decrease-order-specification*? | *align-lines-specification*?]])

For each column-subset in the column-set-model, there shall be a *column-subset-specification*.

[207] column-subset-flow-map = `(flow (`(*port-specifier zone-name*+)`)+ )`

[208] zone-name = `top-float` | `body-text` | `bottom-float` | `footnote`

A *column-subset-flow-map* specifies that areas resulting from flow objects directed in *port-specifier* shall be labeled with one of the specified *zone-names*. Multiple *zone-names* may be specified for a single *port-specifier* only if the *zone-names* are `top-float` and `bottom-float`.

[209] top-float-space-below-specification = `(top-float-space-below` *expression*`)`

The *expression* shall evaluate to an object of type display-space specifying the size of a space to be added. For each column in the column-set that is spanned by an area in the top-float zone, a space of the specified size shall be added immediately after all the areas that span the column and that are in the top-float zone.

[210] bottom-float-space-above-specification = `(bottom-float-space-above` *expression*`)`

The *expression* shall evaluate to an object of type display-space specifying the size of a space to be added. For each column in the column-set that is spanned by an area in the bottom-float zone, a space of the specified size shall be added immediately before all the areas that span the column and that are in the bottom-float zone.

[211] balance-specification = `(balance?` *expression*`)`

The *expression* shall evaluate to a boolean. A value of #t indicates that a column-subset in the last column-set produced by a column-set-sequence shall be balanced. A value of #f indicates that it shall not be. If a column-subset is balanced, then free space shall be allocated evenly among all the columns in the column-subset. If a column-subset is not balanced, then free space shall be allocated to the columns in reverse order. The default is for the column-subset not to be balanced.

[212] justify-specification = `(justify?` *expression*`)`

The *expression* shall evaluate to a boolean specifying whether the column-subset is to be justified. If a column subset is to be justified, the free space shall be distributed before and after the areas in the column-subset according to the minimum and maximum allowed space specified in the display spaces. Otherwise, all free space shall be distributed at the end of each column. The default is for the column-subset not to be justified. A column-subset may only be justified if the *fill-out-specification* specifies that the column-set is to be filled out.

[213] justify-limit-specification = `(justify-limit` *expression*`)`

The *expression* shall evaluate to a number between 0 and 100. If the amount of free space in a column as a percentage of the total size of the column exceeds this, then that column shall not be justified. The default is 100.

[214] justify-last-limit-specification = `(justify-last-limit` *expression*`)`

The *expression* shall evaluate to a number between 0 and 100. A column shall not be justified if the amount of free space in a column in the last column-set in a column-set-sequence as a percentage of the total size of the column exceeds the number returned by the expression. The default is 0.

[215] length-deviation-specification = `(length-deviation` *expression*`)`

The *expression* shall evaluate to a positive length. When a column-subset is being justified or balanced, then the lengths of the columns may differ by up to this amount. The default is 0pt.

[216] length-decrease-order-specification = `(length-decrease-order` *expression*`)`

The *expression* shall evaluate to one of the following symbols:

— `forward` specifying that as columns progress in the forward direction their length shall not increase,

— `backward` specifying that as columns progress in the backward direction their length shall not increase,

or `#f` implying no additional constraint on the relative length of the columns.

[217] align-lines-specification = `(align-lines?` *expression*`)`

The *expression* shall evaluate to a boolean specifying, if true, that an attempt shall be made in the course of distributing free space to keep lines in different columns aligned.

[218] column-specification = `(column` [[*width-specification*? | *height-specification*? | *x-origin-specification*? | *y-origin-specification*? | *footnote-separator-specification*? | *header-specification*? | *footer-specification*?]]`)`

If the column-set filling-direction is top-to-bottom, then the *column-specification* shall contain a *width-specification* and an *x-origin-specification*. If the column-set filling-direction is right-to-

left or left-to-right, then the *column-specification* shall contain a *height-specification* and a *y-origin-specification*. These specifications give the geometry of the column.

[219] footnote-separator-specification = (`footnote-separator` *generated-area-clauses*)

A *footnote-separator-specification* specifies areas that shall be generated immediately before the areas in the footnote zone if the footnote zone contains any areas.

[220] tied-column-subset-specification = (`tie` *column-subset-specification column-subset-specification*+)

A *tied-column-subset-specification* specifies two or more column-subsets that are tied together. See Figure 18.

NOTE 79 This may be used, for example, with sidenotes.

12.6.6 Paragraph Flow Object Class

A paragraph flow object represents a paragraph. It has a single principal port. The contents of this port may be either inlined or displayed. Inline flow objects are formatted to produce line areas. Displayed flow objects implicitly specify a break, and their areas shall be added to the resulting sequence of areas. A paragraph flow object may only be displayed.

NOTE 80 Typically, a break implies that a new line is to be started.

The following characteristics are applicable:

— `lines:` is a symbol specifying how the content of the paragraph shall be broken into lines in the formatted output, as follows:

  – `wrap` specifying that lines shall be broken so that they fit in the available space.

  – `asis` specifying that lines shall be broken only after character flow objects for which the `record-end?:` characteristic is true.

  – `asis-wrap` specifying that lines shall be broken after character flow objects for which the `record-end?:` characteristic is true, and as necessary to make lines fit in the available space.

  – `asis-truncate` specifying that lines shall be broken only after character flow objects for which the `record-end?:` characteristic is true, and that lines that do not fit the in the available space shall be truncated.

  – `none` specifying that lines shall not be broken at all.

    NOTE 81 This is useful in tables when the `table-auto-width` feature is present to ensure that the width of a column is made large enough so that the content of a cell fits on a single line.

In all cases, line breaks shall also be allowed where explicitly specified with the `break-before:` or `break-after:` characteristics. The initial value is `wrap`.

— `asis-truncate-char:` is either #f or a char object that determines the glyph to be inserted when the `lines:` characteristic has the value `asis-truncate` and a line is truncated. The initial value is #f.

— `asis-wrap-char:` is either #f or a char object that determines the glyph to be inserted at the end of a line when the `lines:` characteristic has the value `asis-wrap` and the line is broken other than after a character flow object for which the `record-end?:` characteristic is true. The initial value is #f.

— `asis-wrap-indent:` is a length-spec giving an indent to be added to the start-indent when the `lines:` characteristic has the value `asis-wrap` for a line following a break other than after a character flow object for which the `record-end?:` characteristic is true. The initial value is #f.

— `first-line-align:` is either #f, #t, or a char object. If it is not #f, then the `quadding:` and `last-line-quadding:` characteristics are ignored for the first line of the paragraph, and the first line shall be aligned using an alignment point in the line. If the value is a char object, then the alignment point shall be the position point of the first area produced by the first occurrence on the line of a character flow object with a `char:` characteristic equal to that char object; otherwise, the alignment point shall be the position of the first alignment-point flow object in the line. If `alignment-point-offset:` is not #f, then the first line of the paragraph shall be aligned so that the percentage of the line length (that is, the display-size less the applicable start and end indents) before the alignment point is equal to the value of `alignment-point-offset:`. If `alignment-point-offset:` is #f, then the paragraph is an *externally aligned* paragraph and shall have an ancestor of class table-cell or aligned-column. Furthermore, the area container in which the areas from this paragraph are placed shall be the same as the area container in which the areas from that ancestor are placed; in this case, the paragraph shall be aligned so that its alignment point is aligned with other such paragraphs in the table-column or aligned-column. If an externally aligned paragraph occurs in a table-cell, then the `table-auto-width` feature shall be enabled. The initial value is #f.

— `alignment-point-offset:` is either #f or a number between 0 and 100 specifying the percentage of the line length (that is, the display-size less the start and end indents) before the alignment point. The initial value is 50.

— `ignore-record-end?:` is a boolean specifying whether a record-end shall be ignored. If this characteristic is true, then a character with the `record-end?` property true shall be ignored. The initial value is #f.

— `expand-tabs?:` is either #f or a strictly positive integer specifying the tab interval. When a tab interval is specified, each character flow object that has the `input-tab?:` characteristic true shall be treated as equivalent to the smallest strictly positive number of spaces that when added to the number of character flow objects following the last preceding record-end character flow object shall be a multiple of the tab interval. The initial value is 8.

— `line-spacing:` is a length-spec giving the normal spacing between the placement paths of lines in the paragraph as described in 12.6.6.1. The initial value is 12pt.

— `line-spacing-priority:` is either an integer or the symbol `force` specifying the priority of any conditional space before the line. This shall be interpreted in the same manner as the `priority:` argument for the `display-space` procedure. The initial value is 0.

— `min-pre-line-spacing:` is a length-spec specifying the minimum size of the line in the placement direction before the placement path as described in 12.6.6.1. A value of #f shall also be allowed, specifying that the value is determined from the paragraph's font. The initial value is #f.

— `min-post-line-spacing:` is a length-spec specifying the minimum size of the line in the placement direction after the placement path as described in 12.6.6.1. A value of #f shall also be allowed, specifying that the value is determined from the paragraph's font. The initial value is #f.

— `min-leading:` is either #f or a length-spec specifying the minimum space between the line areas in the placement direction as described in 12.6.6.1. A value of #f means that the line spacing shall not be automatically adjusted to take into account the size of the content of the lines. The initial value is #f.

— `first-line-start-indent:` is a length-spec giving an indent to be added to the start-indent for the first line. The length may be negative. The initial value is 0pt.

— `last-line-end-indent:` is a length-spec giving an indent to be added to the end-indent for the last line. The length may be negative. The initial value is 0pt.

— `hyphenation-char:` is a char that is used to determine the glyph that is inserted when hyphenation is performed. The characteristics of the character flow object preceding the hyphenation point shall determine the mapping of the character to a glyph, as well as the font resource and font-size of the glyph. The initial value is `#\-` (the hyphen character).

— `hyphenation-ladder-count:` is a strictly positive integer specifying the maximum number of consecutive lines ending with the same glyph as the glyph determined by the value of the `hyphenation-char:` characteristic, or #f indicating that there is no limit. The initial value is #f.

— `hyphenation-remain-char-count:` is a positive integer specifying the minimum number of characters in a hyphenated word before the hyphenation character. This is the minimum number of characters in the word left on the line ending with the hyphenation character. The initial value is 2.

— `hyphenation-push-char-count:` is a positive integer specifying the minimum number of characters in a hyphenated word after the hyphenation character. This is the minimum number of characters in the word pushed to the next line after the line ending with the hyphenation character. The initial value is 2.

— `hyphenation-keep:` is either #f or one of the following symbols:

  – `spread` means that both parts of a hyphenated word shall lie within a single spread.

  – `page` means that both parts of a hyphenated word shall lie within a single page.

  – `column` means that both parts of a hyphenated word shall lie within a single column.

  The initial value is #f.

— `hyphenation-exceptions:` is a list of strings. Each string is a word which may contain hyphen characters, `#\-`, indicating where hyphenation may occur. If a word to be hyphenated occurs in the list, it may only be hyphenated in the specified places. The initial value is the empty list.

NOTE 82 The determination of a word is system-dependent.

— `line-breaking-method:` is #f or a string specifying a public identifier for the line-breaking-method to be used for this paragraph. The initial value is #f.

— `line-composition-method:` is #f or a string specifying a public identifier for the line-composition-method to be used for this paragraph. The initial value is #f.

NOTE 83 Typically, the `line-composition-method:` uses characteristics declared using an *application-characteristic-declaration* or an *application-char-characteristic+property-declaration*.

— `implicit-bidi-method:` is #f or a string specifying a public identifier for the method to be used for implicitly determining the directionality of the content of the paragraph. This includes both the writing-mode of characters, which, when this characteristic is #f, is specified with the writing-mode characteristic, and how portions of content with a common writing-mode are nested within each other, which, when this characteristic is #f, is specified with embedded-text flow objects. It is part of the semantics of the method which characteristics of character flow objects, if any, it uses. A method may be specific to a particular character repertoire, in which case, it may not make use of any characteristics. It may be part of the semantics of a method for certain glyph substitutions to be applied depending on the writing-mode that is determined for a character, and possibly also on characteristics of the character. The initial value is #f.

— `glyph-alignment-mode:` is one of the symbols `base`, `center`, `top`, `bottom`, or `font` specifying the alignment mode to be used for glyphs. `font` means that the nominal alignment mode of the font in the flow object's writing-mode should be used. The initial value is `font`.

— `font-family-name:` is either #f, indicating that any font family is acceptable, or a string giving the font family name property of the desired font resource. The initial value is `iso-serif`.

NOTE 84 ISO/IEC 10180 defines a mandatory font set for interchange comprising the font families `iso-serif`, `iso-sanserif`, and `iso-monospace`.

This characteristic is applicable when the glyph-alignment-mode: is font or when min-pre-line-spacing: or min-post-line-spacing: are #f.

— font-weight: is either #f, indicating that any font weight is acceptable, or one of the symbols not-applicable, ultra-light, extra-light, light, semi-light, medium, semi-bold, bold, extra-bold, or ultra-bold, giving the weight property of the desired font resource. The initial value is medium. This characteristic is applicable when the glyph-alignment-mode: is font or when min-pre-line-spacing: or min-post-line-spacing: is #f.

— font-posture: is either #f, indicating that any posture is acceptable, or one of the symbols not-applicable, upright, oblique, back-slanted-oblique, italic, or back-slanted-italic, giving the posture property of the desired font resource. The initial value is upright. This characteristic is applicable when the glyph-alignment-mode: is font or when min-pre-line-spacing: or min-post-line-spacing: is #f.

— font-structure: is either #f, indicating that any structure is applicable, or one of the symbols not-applicable, solid, or outline. The initial value is solid. This characteristic is applicable when the glyph-alignment-mode: is font or when min-pre-line-spacing: or min-post-line-spacing: is #f.

— font-proportionate-width: is either #f, indicating that any proportionate width is acceptable, or one of the symbols not-applicable, ultra-condensed, extra-condensed, condensed, semi-condensed, medium, semi-expanded, expanded, extra-expanded, or ultra-expanded. The initial value is medium. This characteristic is applicable when the glyph-alignment-mode: is font or when min-pre-line-spacing: or min-post-line-spacing: is #f.

— font-name: is either #f, indicating that any font name is acceptable, or a string which is the public identifier for the font name property of the desired font resource. When the value is a string, the values of the font-family-name:, font-weight:, font-posture:, font-structure:, and font-proportionate-width: characteristics are not used in font selection. The initial value is #f. This characteristic is applicable when the glyph-alignment-mode: is font or when min-pre-line-spacing: or min-post-line-spacing: is #f.

— font-size: is a length specifying the body size to which the font resource should be scaled. The initial value is 10pt. This characteristic is applicable when min-pre-line-spacing: or min-post-line-spacing: is #f.

— numbered-lines?: is #t if the lines produced by this paragraph shall be considered for the purposes of line numbering, and #f otherwise. The initial value is #t.

— line-number: is either #f or an unlabeled sosofo containing only inline flow objects. If it is a sosofo, then for each line in the paragraph, the sosofo is formatted to produce a single inline area that is positioned as an attachment area for the line. See 12.3.4. The initial value is #f.

NOTES

85 The sosofo may include indirect flow objects that refer to the line's number by using the `line-number` procedure.

86 The rules for the positioning of an attachment area mean that line numbers are usually positioned so that the edges nearest the line are aligned. Different alignments can be achieved by using the line-field flow object class.

— `line-number-side:` is one of the symbols `start`, `end`, `spread-inside`, `spread-outside`, `page-inside`, or `page-outside` specifying the side of the line for the attachment specified with the `line-number:` characteristic. A value of `spread-inside` or `spread-outside` shall be allowed only if the flow object has an ancestor of class page-sequence. A value of `page-inside` or `page-outside` shall be allowed only if the flow object has an ancestor of column-set-sequence.

— `line-number-sep:` is a length-spec specifying the separation for the attachment specified with the `line-number:` characteristic.

— `quadding:` is one of the symbols `start`, `end`, `spread-inside`, `spread-outside`, `page-inside`, `page-outside`, `center`, or `justify` specifying the alignment of lines other than the last line in the paragraph in the direction determined by the writing-mode. A value of `spread-inside` or `spread-outside` shall be allowed only if the flow object has an ancestor of class page-sequence. A value of `page-inside` or `page-outside` shall be allowed only if the flow object has an ancestor of column-set-sequence. The initial value is `start`.

— `last-line-quadding:` is one of the symbols `relative`, `start`, `end`, `spread-inside`, `spread-outside`, `page-inside`, `page-outside`, `center`, or `justify` specifying the alignment of the last line of the paragraph in the direction determined by the writing-mode. This shall apply also to any line in the paragraph that immediately precedes a break. A value of `relative` means that the value of the `quadding:` characteristic shall be used, except when that value is `justify`, in which case, a value of `start` shall be used. A value of `spread-inside` or `spread-outside` shall be allowed only if the flow object has an ancestor of class page-sequence. A value of `page-inside` or `page-outside` shall be allowed only if the flow object has an ancestor of column-set-sequence. The initial value is `relative`.

— `last-line-justify-limit:` is a length-spec specifying the maximum amount of free space in the last line that shall cause the last line to be justified rather than aligned as specified by the `last-line-quadding:` characteristic. The initial value is 0.

— `justify-glyph-space-max-add:` is a length-spec specifying the maximum space that may be added between glyphs in order to justify a line. The initial value is 0pt.

— `justify-glyph-space-max-remove:` is a length-spec specifying the maximum space that may be removed between glyphs in order to justify a line. The initial value is 0pt.

— `hanging-punct?:` is a boolean specifying whether the paragraph shall be formatted with the punctuation characters hanging into the margin or gutter of a column. The initial value is #f.

— `widow-count:` is a positive integer specifying the minimum number of lines of the paragraph that shall be kept together at the beginning of an area. If the `widow-count:` is *n*, then no break shall be allowed between the last *n* lines of the paragraph. The initial value is 2.

— `orphan-count:` is a positive integer specifying the minimum number of lines of the paragraph that shall be kept together at the end of an area. If the `orphan-count:` is *n*, then no break shall be allowed between the first *n* lines of the paragraph. The initial value is 2.

— `language:` is #f or a symbol specifying the ISO 639 language code in upper-case. This affects line composition in a system-dependent way. The initial value is #f.

— `country:` is #f or a symbol specifying the ISO 3166 country code in upper-case. This affects line composition in a system-dependent way. The initial value is #f.

— `position-preference:` is either #f or one of the symbols `top` or `bottom`. This applies if the flow object is directed into a port on a column-set-sequence flow object that is flowed into both the top-float and bottom-float zones of a column-subset and indicates whether the areas from this flow object may be flowed into only one of the zones. This characteristic is not inherited. The default value is #f.

— `writing-mode:` is one of the symbols `left-to-right`, `right-to-left`, or `top-to-bottom`. The direction determined by the writing-mode shall be perpendicular to the placement direction. The initial value is `left-to-right`. This controls the orientation of the placement path of the lines.

— `start-indent:` is a length-spec specifying the indent for the edge of the area at the start in the direction of the writing-mode. The initial value is 0pt. This applies only to lines from the paragraph itself.

— `end-indent:` is a length-spec specifying the indent for the edge of the area at the end in the direction of the writing-mode. The initial value is 0pt. This applies only to lines from the paragraph itself.

— `span:` is a strictly positive integer specifying the number of columns that the areas resulting from this flow object shall span. This characteristic shall apply if the flow object is directed into a port on a column-set-sequence flow object that is flowed into the top-float, bottom-float, or body-text zone of a spannable column-subset. The initial value is 1.

— `span-weak?:` is a boolean specifying whether the areas resulting from this flow object span weakly rather than strongly. See 12.6.5.1. This characteristic applies if the flow object is directed into a port on a column-set-sequence flow object that is flowed into the top-float, bottom-float, or body-text zone of a spannable column-subset and has a `span:` characteristic with a value greater than 1. The initial value is #f.

— `space-before:` is an object of type display-space specifying space to be inserted before, in the placement direction, the areas produced by the flow object. This characteristic is not inherited. The default is for no space before to be inserted.

— `space-after:` is an object of type display-space specifying space to be inserted after, in the placement direction, the areas produced by the flow object. This characteristic is not inherited. The default is for no space after to be inserted.

— `keep-with-previous?:` is a boolean specifying whether the flow object shall be kept in the same area as the previous flow object. This characteristic is not inherited. The default value is #f.

— `keep-with-next?:` is a boolean specifying whether the flow object shall be kept in the same area as the next flow object. This characteristic is not inherited. The default value is #f.

— `break-before:` is #f or one of the symbols `page`, `page-region`, `column`, or `column-set` specifying that the flow object shall start an area of that type. This characteristic is not inherited. The default is #f.

— `break-after:` is #f or one of the symbols `page`, `page-region`, `column`, or `column-set` specifying that the flow object shall end an area of that type. This characteristic is not inherited. The default is #f.

— `keep:` is one of the following:

  – #t meaning that the areas produced by this flow object shall be kept together within the smallest possible area.

  – the symbol `page` indicating that the areas produced by the flow object shall lie within the same page; in this case, the flow object shall have an ancestor flow object of class page-sequence.

  – the symbol `column-set` indicating that the areas produced by the flow object shall lie within the same column set; in this case, the flow object shall have an ancestor of class column-set-sequence.

  – the symbol `column` indicating that the areas produced by the flow object shall lie within the same column set, and that the first column that each area spans in the column set shall be the same.

  – #f indicating that this characteristic is to be ignored.

  This characteristic is not inherited. The default value is #f.

— `may-violate-keep-before?:` is a boolean which, if true, specifies that constraints imposed by the `keep:` characteristics of ancestor flow objects on the relative positioning of this flow object and its previous flow object may not be respected. This characteristic is not inherited. The default value is #f.

— `may-violate-keep-after?:` is a boolean which, if true, specifies that constraints imposed by `keep:` characteristics of ancestor flow objects on the relative positioning of this flow object and its next flow object may not be respected. This characteristic is not inherited. The default value is #f.

The line-progression direction for inline areas in the paragraph is the placement direction of the paragraph.

12.6.6.1 Line Spacing

The size of the line areas produced by the paragraph shall be `min-pre-line-spacing:` before the placement path and `min-post-line-spacing:` after the placement path. If `min-leading:` is not #f, the size of the line shall be increased to cover all the areas in the line. If the previous area is a line, then conditional space shall be added, if necessary, before the line so that the total distance between the previous line's placement path and this placement path is the value of the `line-spacing:` characteristic. If the previous area is not a line, then conditional space shall be added, if necessary, before the line so that the total distance between the end of the previous area and this placement path is the value of the `line-spacing:` characteristic less the value of the `min-post-line-spacing:` characteristic. If `min-leading:` is not #f, then additional conditional space shall be added, if required, to make the space between the previous area and this one no less than the value of `min-leading:`. The conditional space has the priority specified by the `line-spacing-priority:` characteristic.

12.6.7 Paragraph-break Flow Object Class

Paragraph-break flow objects can be used to make a paragraph flow object represent a sequence of paragraphs. The paragraphs are separated by paragraph-break flow objects, which are atomic. Paragraph-break flow objects are allowed only in paragraph flow objects. All the characteristics that are applicable to a paragraph flow object are also applicable to a paragraph-break flow object. The characteristics of a paragraph-break flow object determine how the portion of the content of the paragraph flow object following that paragraph-break flow object up to the next paragraph-break flow object, if any, is formatted.

NOTE 87 The paragraph-break flow object inherits from its containing paragraph flow object in the usual way.

The `first-line-start-indent:` characteristic is applicable to the line following a paragraph-break flow object, and the `last-line-end-indent:` characteristic is applicable to the line preceding a paragraph-break flow object.

NOTE 88 It is recommended that paragraph-break flow objects be used only if there is no other way of specifying the desired formatting.

12.6.8 Line-field Flow Object Class

The line-field flow object class is inlined and has inline content. It produces a single inline area. The width of this area is equal to the value of the `field-width:` characteristic. If the content of a line-field area cannot fit in this width, then the area grows to accommodate the content and, if the line-field occurs in a paragraph, there shall be a break after the line-field.

It has a single principal port.

It has the following characteristics:

— `field-width:` is a length-spec specifying the width of the area produced by the flow object. The initial value is 0pt.

— `field-align:` is one of the symbols `start`, `end`, or `center` specifying the alignment of the contents of the field. The initial value is `start`.

— `writing-mode:` is one of the symbols `left-to-right`, `right-to-left`, or `top-to-bottom`. The direction determined by the writing-mode shall be perpendicular to the placement direction. The initial value is `left-to-right`.

— `inhibit-line-breaks?:` is a boolean specifying whether line breaks shall be inhibited before and after each area produced by this flow object. This applies only to line breaks introduced by the formatter to make lines fit in the available space. The initial value is #f.

— `break-before-priority:` is an integer that affects whether a break is allowed before this flow object. The *break priority* of a potential breakpoint is the maximum of the break-after-priority of the flow object immediately preceding the potential breakpoint and the break-before-priorities of the flow object immediately following the potential breakpoint, and any characters immediately following that character for which the `drop-after-line-break?:` characteristic is true. A break shall be allowed at a potential breakpoint only if the break priority is even. This characteristic is not inherited. The default value is 0.

— `break-after-priority:` is an integer that affects whether a break is allowed after this flow object as described in the specification of the `break-before-priority:` characteristic. This characteristic is not inherited. The default value is 0.

A line-break shall be allowed immediately before and after a line-field used in a paragraph.

12.6.9 Sideline Flow Object Class

Use of this flow object requires the `sideline` feature.

A sideline flow object is used to contain flow objects that have an attachment area (see 12.3.4) consisting of a line parallel to the placement direction. A sideline flow object has a single principal port which can contain both inlined and displayed flow objects. For each display area produced by its content, the sideline flow object adds an attachment. For each inline area produced by its content, the sideline flow object annotates that area so as to cause the paragraph in which the flow object occurs to add an attachment area to the line in which that inline area occurs.

NOTE 89 Sidelines are often used to mark changes.

This is illustrated in Figure 14.

A sideline flow object has the following characteristics:

— `sideline-side`: is one of the symbols `start`, `end`, `both`, `spread-inside`, `spread-outside`, `page-inside`, or `page-outside`, specifying the side of the line area for the sideline attachment. A value of `spread-inside` or `spread-outside` is allowed only if the flow object has an ancestor of class page-sequence. A value of `page-inside` or `page-outside` is allowed only if the flow object has an ancestor of column-set-sequence. A value of `both` means that there shall be a sideline attachment on both sides of the line area containing the text.

— `sideline-sep`: is a length-spec specifying the separation for the sideline attachment. A negative value is allowed.

— `color`: is an object of type color that specifies the color in which the flow object's marks should be made. The initial value is the default color in the Device Gray color space.

— `layer`: is an integer specifying the layer of the marks of the areas resulting from the flow object. An area shall be imaged after any area whose layer has a lower value. The initial value is 0.

— `line-cap`: is one of the symbols `butt`, `round`, or `square` specifying the cap style for the line. The initial value is `butt`.

— `line-dash`: is a list of one or more lengths that specifies the dash pattern of the line. The first length specifies the number component of the CurrentDashPattern graphics state variable in ISO/IEC 10180. The remaining lengths specify the vector component of the CurrentDashPattern graphics state variable. The initial value is a list containing the length 0pt.

— `line-thickness`: is a length that specifies the thickness of the line or lines. The initial value is 1pt.

— `line-repeat`: is a strictly positive integer that specifies the number of parallel lines to be drawn. For example, a value of 2 indicates a double line. The initial value is 1.

— `line-sep`: is a length that gives the distance between the centers of parallel lines. The initial value is 1pt.

Sidelines on consecutive areas in a single area container which have no space between them should be drawn as a single line.

12.6.10 Anchor Flow Object Class

Use of this flow object requires the `page` feature.

An anchor flow object is atomic and serves only as a flow object to be synchronized. It may be either inlined or displayed. If inlined, it produces a single area with zero size in the escapement direction. If displayed, it produces a single area with zero size in the placement direction. The

resulting area will be kept with the first area resulting from the flow object that follows unless the `anchor-keep-with-previous?:` characteristic is true.

Anchor flow objects have the following characteristics:

— `anchor-keep-with-previous?:` is a boolean specifying whether the resulting area shall be kept with the last area of the previous flow object instead of the first area resulting from the following flow object. The initial value is #f.

— `display?:` is a boolean specifying whether the flow object is displayed rather than inlined. This characteristic is not inherited. The default value is #f.

— `span:` is a strictly positive integer specifying the number of columns that the areas resulting from this flow object shall span. This characteristic shall apply if the flow object is directed into a port on a column-set-sequence flow object that is flowed into the top-float, bottom-float, or body-text zone of a spannable column-subset. The initial value is 1.

— `span-weak?:` is a boolean specifying whether the areas resulting from this flow object span weakly rather than strongly. See 12.6.5.1. This characteristic applies if the flow object is directed into a port on a column-set-sequence flow object that is flowed into the top-float, bottom-float, or body-text zone of a spannable column-subset and has a `span:` characteristic with a value greater than 1. The initial value is #f.

— `inhibit-line-breaks?:` is a boolean specifying whether line breaks shall be inhibited before and after each area produced by this flow object. This applies only to line breaks introduced by the formatter to make lines fit in the available space. The initial value is #f.

— `break-before-priority:` is an integer that affects whether a break is allowed before this flow object. The *break priority* of a potential breakpoint is the maximum of the break-after-priority of the flow object immediately preceding the potential breakpoint and the break-before-priorities of the flow object immediately following the potential breakpoint, and any characters immediately following that character for which the `drop-after-line-break?:` characteristic is true. A break shall be allowed at a potential breakpoint only if the break priority is even. This characteristic is not inherited. The default value is 0.

— `break-after-priority:` is an integer that affects whether a break is allowed after this flow object as described in the specification of the `break-before-priority:` characteristic. This characteristic is not inherited. The default value is 0.

12.6.11 Character Flow Object Class

A character flow object is atomic. Flow objects of this class can only be inlined. Flow objects of this class have the following characteristics:

— `char:` is an object of type char specifying the character. This characteristic is not inherited. If it is not specified, and there is a current node, and the current node has a `char` property, then the value of the `char` property shall be used as the value of this characteristic. If the value of the `char-map:` characteristic is not #f, then it is applied to the value of the `char`

property, and the result is used as the value of the characteristic. This characteristic may be used to control hyphenation as well as possibly being used in the selection of the glyph.

— `char-map:` is either #f or a procedure that is applied in the construction of the default value of the `char:` characteristic. The initial value is #f.

— `glyph-id:` is an object of type glyph-id specifying the glyph that shall be imaged in the resulting area or #f if no image is associated with the resulting area. This characteristic is not inherited. If this characteristic is not specified, it is computed using the value of the `char:` characteristic: if the `blank?` property of the character is true, then the value of the characteristic shall be #f; otherwise, the value of the characteristic shall be the value of the `glyph-id` property of the character, which shall not be #f in this case.

— `glyph-subst-table:` is either #f or a glyph-subst-table or a list of glyph-subst-tables specifying substitutions to be performed on the glyph-id specified by the `glyph-id:` characteristic. If the value is a list, then the substitutions shall be performed in the specified order. The initial value is #f.

— `glyph-subst-method:` is either #f or a string or a list of strings. Each string shall be a public identifier specifying a method for performing glyph substitution. The initial value is #f.

NOTE 90 This allows for context-dependent glyph substitution and for glyph substitutions that involve multiple glyphs.

— `glyph-reorder-method:` is either #f or a string or a list of strings. Each string shall be a public identifier specifying a method for reordering glyphs. The initial value is #f.

NOTE 91 This is typically used for Indic scripts.

— `writing-mode:` is one of the symbols `left-to-right`, `right-to-left`, or `top-to-bottom`. The direction determined by the writing-mode shall be perpendicular to the placement direction. The initial value is `left-to-right`. This controls which writing-mode of the font resource is used for the metrics of the glyph.

— `font-family-name:` is either #f, indicating that any font family is acceptable, or a string giving the font family name property of the desired font resource. The initial value is `iso-serif`.

NOTE 92 ISO/IEC 10180 defines a mandatory font set for interchange comprising the font families `iso-serif`, `iso-sanserif`, and `iso-monospace`.

— `font-weight:` is either #f, indicating that any font weight is acceptable, or one of the symbols `not-applicable`, `ultra-light`, `extra-light`, `light`, `semi-light`, `medium`, `semi-bold`, `bold`, `extra-bold`, or `ultra-bold`, giving the weight property of the desired font resource. The initial value is `medium`.

— `font-posture:` is either #f, indicating that any posture is acceptable, or one of the symbols `not-applicable`, `upright`, `oblique`, `back-slanted-oblique`, `italic`, or

back-slanted-italic, giving the posture property of the desired font resource. The initial value is upright. In addition, the value math is allowed specifying that the font posture shall be the value of the math-font-posture: characteristic.

— math-font-posture: specifies the posture property of the desired font resource to be used when the font-posture: characteristic has the value math. It shall have the value #f or one of the symbols not-applicable, upright, oblique, back-slanted-oblique, italic, or back-slanted-italic. This characteristic is not inherited. The default value is the value of the math-font-posture character property of the char: characteristic.

— font-structure: is either #f, indicating that any structure is applicable, or one of the symbols not-applicable, solid, or outline. The initial value is solid.

— font-proportionate-width: is either #f, indicating that any proportionate width is acceptable, or one of the symbols not-applicable, ultra-condensed, extra-condensed, condensed, semi-condensed, medium, semi-expanded, expanded, extra-expanded, or ultra-expanded. The initial value is medium.

— font-name: is either #f, indicating that any font name is acceptable, or a string which is the public identifier for the font name property of the desired font resource. When the value is a string, the values of the font-family-name:, font-weight:, font-posture:, font-structure:, and font-proportionate-width: characteristics are not used in font selection. The initial value is #f.

— font-size: is a length specifying the body size to which the font resource should be scaled. The initial value is 10pt.

— stretch-factor: is a number specifying the factor by which the character should be stretched. This characteristic is not inherited. The default is 1.

NOTES

93 It is implementation- and font-dependent how this is achieved.

94 This is designed primarily for math delimiters of various kinds. The size of the delimiter is determined by the product of the font-size and the stretch-factor, but the visual appearance is designed to be consistent with glyphs with that font-size.

— hyphenate?: is a boolean specifying whether hyphenation is allowed. The initial value is #f.

— hyphenation-method: is a string specifying a public identifier for a hyphenation method or #f. The initial value is #f.

— kern?: is a boolean specifying whether kerning (escapement adjustment) is allowed. If true, then kerning shall be performed as specified in 8.8.1.6 of ISO 9541-1 according to the kern-

mode: characteristic. Escapement adjustment is not performed for glyphs whose escapement adjustment indicator property has the value non-adjusting. The initial value is #f.

— kern-mode: is one of the symbols loose, normal, kern, tight, or touch specifying the escapement adjustment mode. The initial value is normal.

— ligature?: is a boolean specifying whether ligatures are allowed. The initial value is #f.

— allowed-ligatures: is a list of allowed ligatures. Each member of the list shall be either a glyph-id or a char. Only ligatures whose result is one of the glyph-ids in the list or is equal to the glyph-id property of one of the chars in the list shall be used. The initial value is the empty list.

— space?: is a boolean specifying whether the flow object is a space. This characteristic is not inherited. This affects only whether the inline-space specified as the value of the inline-space-space: characteristic is applicable to this flow object. The default value is the value of the space? character property of the char: characteristic.

— inline-space-space: is an object of type inline-space which is applicable to the flow object if it is a space. This is in addition to any space from the escapement-space-before: and escapement-space-after: characteristics.

— escapement-space-before: is an object of type inline-space specifying space to be added before the first result area in the escapement direction. The initial value is (inline-space 0pt).

— escapement-space-after: is an object of type inline-space specifying space to be added after the last result area in the escapement direction. The initial value is (inline-space 0pt).

— record-end?: is a boolean specifying whether the flow object is a record-end. Flow objects for which the record-end?: characteristic is true shall be treated differently by paragraphs for which the lines: characteristic has the value asis or for which the ignore-record-end?: characteristic is true. This characteristic is not inherited. The default value is the value of the record-end? character property of the char: characteristic.

— input-tab?: is a boolean specifying whether the flow object is a tab on input. This characteristic is not inherited. Character flow objects that are tabs shall be treated differently by paragraphs for which the expand-tabs property is not #f. The default value is the value of the input-tab? character property of the char: characteristic if the char: characteristic was not explicitly specified, and otherwise #f.

— input-whitespace-treatment: is one of the following symbols:

  – preserve specifying no special action.

- `collapse` specifying that a character flow object for which the `input-whitespace?:` characteristic is true shall be ignored if the preceding flow object was a character flow object also with the `input-whitespace?:` characteristic true.

- `ignore` specifying that any character flow object for which the `input-whitespace?:` characteristic is true shall be ignored.

The initial value is `preserve`.

— `input-whitespace?:` is a boolean specifying whether the character shall be considered as whitespace on input. This characteristic is not inherited. The default value is the value of the `input-whitespace?` character property of the `char:` characteristic if the `char:` characteristic was not explicitly specified, and otherwise #f.

— `punct?:` is a boolean specifying whether the character should be treated as punctuation for the purposes of formatting the paragraph with hanging punctuation. This shall only take effect if the `hanging-punct?:` characteristic of the paragraph is true. This characteristic is not inherited. The default value is the value of the `punct?` character property of the `char:` characteristic.

— `break-before-priority:` is an integer that affects whether a break is allowed before this character. The *break priority* of a potential breakpoint is the maximum of the break-after-priority of the character immediately preceding the potential breakpoint and the break-before-priorities of the character immediately following the potential breakpoint, and any characters immediately following that character for which the `drop-after-line-break?:` characteristic is true. A break is allowed at a potential breakpoint only if the break priority is even. This characteristic is not inherited. The default value is the value of the `break-before-priority` character property of the `char:` characteristic.

NOTE 95 For example, for ideographs, the `break-before-priority:` and `break-after-priority:` characteristics would typically be 0 and 0, for a Latin letter 1 and 1, and for a space character 2 and 3.

— `break-after-priority:` is an integer that affects whether a break is allowed after this character as described in the specification of the `break-before-priority:` characteristic. This characteristic is not inherited. The default value is the value of the `break-after-priority` character property of the `char:` characteristic.

— `drop-after-line-break?:` is a boolean specifying whether this character should be discarded if it follows a line break. This characteristic is not inherited. The default value is the value of the `drop-after-line-break?` character property of the `char:` characteristic.

— `drop-unless-before-line-break?:` is a boolean specifying whether this character shall be discarded unless it precedes a line break. This characteristic is not inherited. The default value is the value of the `drop-unless-before-line-break?` character property of the `char:` characteristic.