

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)

AVA Documents
Document Sharing Hub

Contents

Page

1 Scope	1
2 Conformance	2
3 Normative References	3
4 Definitions	4
5 Notation and Conventions	7
5.1 Syntax Productions	7
5.2 Procedure Prototypes	8
6 DSSSL Overview	8
6.1 Areas of Standardization	9
6.2 Conceptual Model	10
6.3 DSSSL Languages	11
6.3.1 The Transformation Language	11
6.3.1.1 Components of the Transformation Process	12
6.3.1.2 Model for Coded Characters, Characters, and Glyph Identifiers	13
6.3.2 The Style Language	14
6.3.2.1 Components of the Formatting Process	15
6.3.2.2 Grove Building	15
6.3.2.3 Flow Object Tree	15
6.3.2.4 Flow Object Classes	16
6.3.2.5 Areas	17
6.3.2.6 Page and Column Geometry	18
6.3.2.7 Expression Language	18
6.3.2.8 Model for Coded Characters, Characters, and Glyph Identifiers	19
7 DSSSL Specifications	19
7.1 DSSSL Document Architecture	20
7.1.1 Features	24
7.1.2 SGML Grove Plan	24
7.1.3 Character Repertoire	25
7.1.4 Standard Characters	25
7.1.5 Other Characters	26
7.1.6 Baset Encoding	26
7.1.7 Literal Described Character	26
7.1.8 Sdata Entity Mapping	27
7.1.9 Separator Characters	27
7.1.10 Name Characters	27
7.1.11 Character Combination	27
7.2 Public Identifiers	27
7.3 Lexical Conventions	27
7.3.1 Case Sensitivity	27

© ISO/IEC 1996

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

7.3.2 Identifiers	28
7.3.3 Tokens, Whitespace, and Comments	28
8 Expression Language	29
8.1 Overview of the Expression Language	30
8.2 Basic Concepts	30
8.2.1 Variables and Regions	30
8.2.2 True and False	31
8.2.3 External Representations	31
8.2.4 Disjointness of Types	31
8.3 Expressions	32
8.3.1 Primitive Expression Types	32
8.3.1.1 Variable Reference	32
8.3.1.2 Literals	33
8.3.1.3 Procedure Call	34
8.3.1.4 Lambda Expression	34
8.3.1.5 Conditional Expression	36
8.3.2 Derived Expression Types	36
8.3.2.1 Cond-expression	36
8.3.2.2 Case-expression	37
8.3.2.3 And-expression	37
8.3.2.4 Or-expression	38
8.3.2.5 Binding expressions	38
8.3.2.6 Named-let	39
8.3.2.7 Quasiquotation	40
8.4 Definitions	41
8.5 Standard Procedures	43
8.5.1 Booleans	43
8.5.1.1 Negation	43
8.5.1.2 Boolean Type Predicate	44
8.5.2 Equivalence	44
8.5.3 Pairs and Lists	45
8.5.3.1 Pair Type Predicate	46
8.5.3.2 Pair Construction Procedure	46
8.5.3.3 car Procedure	46
8.5.3.4 cdr Procedure	47
8.5.3.5 c...r Procedures	47
8.5.3.6 Empty List Type Predicate	48
8.5.3.7 List Type Predicate	48
8.5.3.8 List Construction	48
8.5.3.9 List Length	48
8.5.3.10 Lists Appendence	49
8.5.3.11 List Reversal	49
8.5.3.12 Sublist Extraction	49
8.5.3.13 List Access	49
8.5.3.14 List Membership	50
8.5.3.15 Association Lists	50
8.5.4 Symbols	50
8.5.4.1 Symbol Type Predicate	51

8.5.4.2	Symbol to String Conversion	51
8.5.4.3	String to Symbol Conversion	51
8.5.5	Keywords.....	51
8.5.5.1	Keyword Type Predicate	52
8.5.5.2	Keyword to String Conversion	52
8.5.5.3	String to Keyword Conversion	52
8.5.6	Named Constants.....	52
8.5.7	Quantities and Numbers	52
8.5.7.1	Numerical Types.....	52
8.5.7.2	Exactness	53
8.5.7.3	Implementation Restrictions.....	54
8.5.7.4	Syntax of Numerical Constants	55
8.5.7.5	Number Type Predicates	56
8.5.7.6	Exactness Predicates.....	56
8.5.7.7	Comparison Predicates	56
8.5.7.8	Numerical Property Predicates	57
8.5.7.9	Maximum and Minimum.....	57
8.5.7.10	Addition	57
8.5.7.11	Multiplication	58
8.5.7.12	Subtraction.....	58
8.5.7.13	Division	58
8.5.7.14	Absolute Value	58
8.5.7.15	Number-theoretic Division	59
8.5.7.16	Real to Integer Conversion	59
8.5.7.17	e^n and Natural Logarithm	60
8.5.7.18	Trigonometric Functions	60
8.5.7.19	Inverse Trigonometric Functions.....	60
8.5.7.20	Square Root	61
8.5.7.21	Exponentiation.....	61
8.5.7.22	Exactness Conversion.....	61
8.5.7.23	Quantity to Number Conversion.....	61
8.5.7.24	Number to String Conversion.....	61
8.5.7.25	String to Number Conversion.....	63
8.5.8	Characters	63
8.5.8.1	Character Properties	64
8.5.8.2	Language-dependent Operations	64
8.5.8.3	Character Type Predicate.....	67
8.5.8.4	Character Comparison Predicates.....	67
8.5.8.5	Case-insensitive Character Predicates	67
8.5.8.6	Character Case Conversion	68
8.5.8.7	Character Properties	68
8.5.9	Strings.....	68
8.5.9.1	String Type Predicate	69
8.5.9.2	String Construction.....	69
8.5.9.3	String Length	69
8.5.9.4	String Access	69
8.5.9.5	String Equivalence.....	69
8.5.9.6	String Comparison.....	69

8.5.9.7 Substring Extraction.....	70
8.5.9.8 String Appendance.....	70
8.5.9.9 Conversion between Strings and Lists.....	70
8.5.10 Procedures.....	70
8.5.10.1 Procedure Type Predicate.....	70
8.5.10.2 Procedure Application.....	71
8.5.10.3 Mapping Procedures over Lists.....	71
8.5.10.4 External Procedures.....	71
8.5.11 Date and Time.....	72
8.5.12 Error Signaling.....	72
8.6 Core Expression Language.....	72
8.6.1 Syntax.....	72
8.6.2 Procedures.....	74
9 Groves.....	75
9.1 Nodal Properties.....	76
9.2 Grove Plans.....	77
9.3 Property Set Definition.....	78
9.3.1 Common Attributes.....	78
9.3.1.1 Component Names.....	78
9.3.1.2 Specification Documents.....	79
9.3.2 Modules.....	79
9.3.3 Data Type Definition.....	80
9.3.4 Class Definition.....	81
9.3.5 Property Definition.....	81
9.3.6 Normalization Rule Definition.....	82
9.4 Intrinsic Properties.....	83
9.5 Auxiliary Groves.....	84
9.6 SGML Property Set.....	84
9.7 DSSSL SGML Grove Plan.....	122
10 Standard Document Query Language.....	123
10.1 Primitive Procedures.....	123
10.1.1 Application Binding.....	123
10.1.2 Node Lists.....	124
10.1.3 Named Node Lists.....	124
10.1.4 Error Reporting.....	125
10.1.5 Application Name Transformation.....	125
10.1.6 Property Values.....	125
10.1.7 SGML Grove Construction.....	126
10.2 Derived Procedures.....	126
10.2.1 HyTime Support.....	126
10.2.2 List Operations.....	130
10.2.3 Generic Property Operations.....	137
10.2.4 Core Query Language.....	143
10.2.4.1 Navigation.....	143
10.2.4.2 Counting.....	143
10.2.4.3 Accessing Attribute Values.....	144
10.2.4.4 Testing Current Location.....	145
10.2.4.5 Entities and Notations.....	146

10.2.4.6 Name Normalization.....	147
10.2.5 SGML Property Operations.....	147
10.3 Auxiliary Parsing.....	149
10.3.1 Word Searching.....	149
10.3.2 Node Regular Expressions.....	150
10.3.3 Regexp Constructors.....	151
10.3.4 Regular Expression Searching Procedures.....	152
11 Transformation Language.....	152
11.1 Features.....	153
11.2 Associations.....	153
11.3 Transform-expression.....	154
11.3.1 Subgrove-spec.....	155
11.3.2 Create-spec.....	156
11.3.3 Result-node-list.....	158
11.3.4 Transform-grove-spec.....	159
11.3.5 SGML Prolog Parsing.....	159
11.4 SGML Document Generator.....	159
11.4.1 Verification Mapping.....	160
11.4.2 Transliteration.....	161
12 Style Language.....	162
12.1 Features.....	162
12.2 Flow Object Tree.....	164
12.3 Areas.....	164
12.3.1 Display Areas.....	165
12.3.2 Inline Areas.....	168
12.3.3 Inlined and Displayed Flow Objects.....	171
12.3.4 Attachment Areas.....	172
12.4 Flow Object Tree Construction.....	173
12.4.1 Construction Rules.....	173
12.4.2 Primary Flow Object.....	176
12.4.3 Sosofos.....	176
12.4.4 Multi-process Feature.....	180
12.4.5 Styles.....	180
12.4.6 Characteristic Specification.....	181
12.4.7 Synchronization of Flow Objects.....	184
12.5 Common Data Types and Procedures.....	185
12.5.1 Layout-driven Generated Text.....	185
12.5.1.1 Constructing Indirect Sosofos.....	186
12.5.1.2 Layout Numbering.....	187
12.5.1.3 Reference Values.....	188
12.5.2 Length Specification.....	190
12.5.3 Decoration Areas.....	190
12.5.4 Spaces.....	191
12.5.4.1 Display Spaces.....	191
12.5.4.2 Inline Spaces.....	191
12.5.5 Glyph Identifiers.....	192
12.5.6 Glyph Substitution Tables.....	192
12.5.7 Font Information.....	193

12.5.8	Addresses	194
12.5.9	Color.....	195
12.6	Flow Object Classes.....	197
12.6.1	Sequence Flow Object Class	197
12.6.2	Display-group Flow Object.....	197
12.6.3	Simple-page-sequence Flow Object Class	199
12.6.4	Page-sequence Flow Object Class	201
12.6.4.1	Page-model.....	202
12.6.5	Column-set-sequence Flow Object Class.....	205
12.6.5.1	Column-set-model.....	207
12.6.6	Paragraph Flow Object Class	217
12.6.6.1	Line Spacing.....	225
12.6.7	Paragraph-break Flow Object Class.....	225
12.6.8	Line-field Flow Object Class	225
12.6.9	Sideline Flow Object Class	226
12.6.10	Anchor Flow Object Class	227
12.6.11	Character Flow Object Class.....	228
12.6.11.1	Character Properties	234
12.6.12	Leader Flow Object Class	236
12.6.13	Embedded-text Flow Object Class.....	237
12.6.14	Rule Flow Object Class.....	238
12.6.15	External-graphic Flow Object Class	242
12.6.16	Included-container-area Flow Object Class	247
12.6.17	Score Flow Object Class	251
12.6.18	Box Flow Object Class.....	253
12.6.19	Side-by-side Flow Object Class	258
12.6.20	Side-by-side-item Flow Object Class.....	260
12.6.21	Glyph-annotation Flow Object Class	261
12.6.22	Alignment-point Flow Object Class.....	262
12.6.23	Aligned-column Flow Object Class	262
12.6.24	Multi-line-inline-note Flow Object Class	265
12.6.25	Emphasizing-Mark Flow Object Class	266
12.6.26	Flow Object Classes for Mathematical Formulae	267
12.6.26.1	Math-sequence Flow Object Class.....	267
12.6.26.2	Unmath Flow Object Class	268
12.6.26.3	Subscript Flow Object Class	269
12.6.26.4	Superscript Flow Object Class.....	269
12.6.26.5	Script Flow Object Class.....	269
12.6.26.6	Mark Flow Object Class.....	271
12.6.26.7	Fence Flow Object Class.....	272
12.6.26.8	Fraction Flow Object Class.....	272
12.6.26.9	Radical Flow Object Class	273
12.6.26.10	Math-operator Flow Object Class	274
12.6.26.11	Grid Flow Object Class.....	275
12.6.26.12	Grid-cell Flow Object Class.....	276
12.6.27	Flow Object Classes for Tables.....	276
12.6.27.1	Table Flow Object Class	277
12.6.27.2	Table-part Flow Object Class.....	280

12.6.27.3	Table-column flow object.....	282
12.6.27.4	Automatic Table-width Computation.....	284
12.6.27.5	Table-row Flow Object Class.....	284
12.6.27.6	Table-cell Flow Object Class.....	284
12.6.27.7	Table-border Flow Object Class.....	287
12.6.28	Flow Object Classes for Online Display.....	289
12.6.28.1	Scroll Flow Object Class.....	289
12.6.28.2	Multi-mode Flow Object Class.....	290
12.6.28.3	Link Flow Object Class.....	290
12.6.28.4	Marginalia Flow Object Class.....	291
Annex A:	Further Information.....	292

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, constitute 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, R⁴RS. 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.

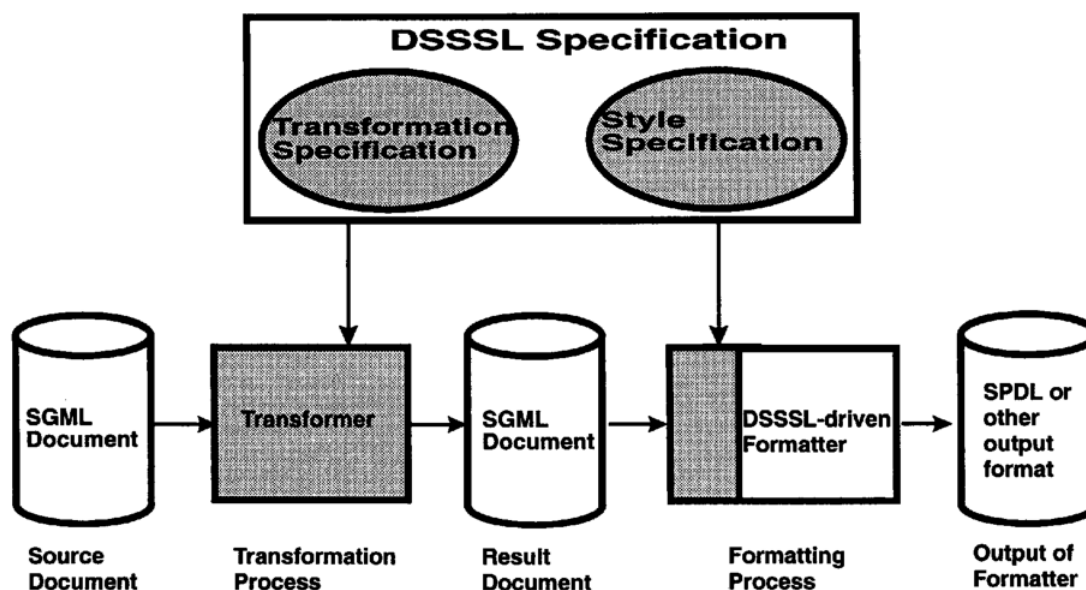


Figure 1 – DSSSL Conceptual Model

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.

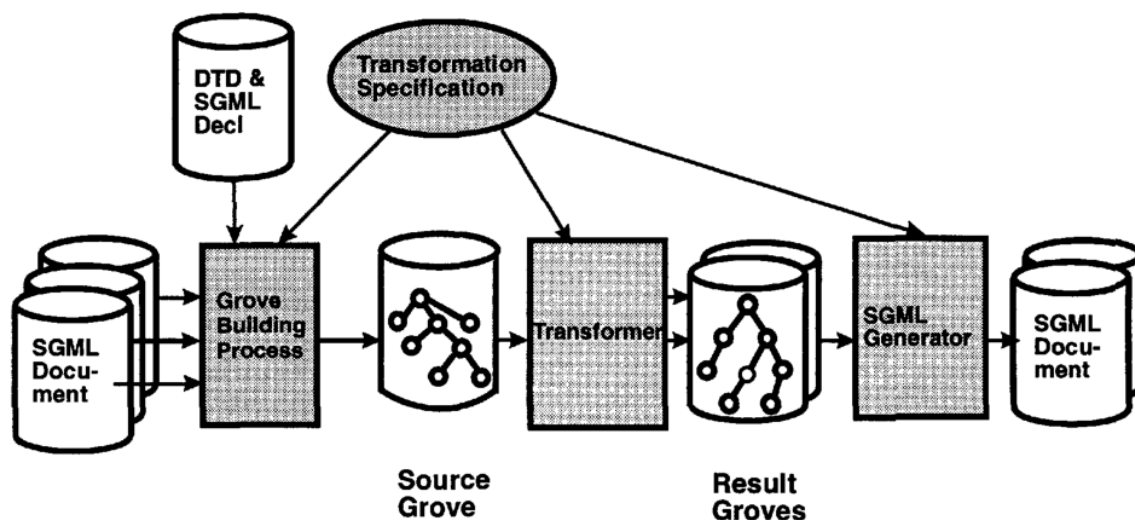


Figure 2 – The Transformation Process

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 ´;

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.

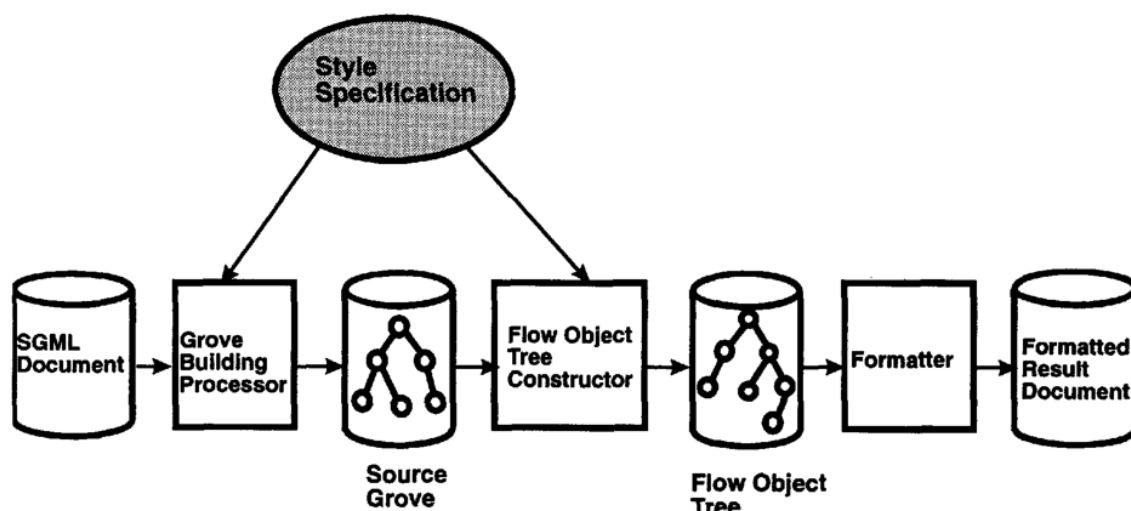


Figure 3 – Formatting Process

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 | basisset-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 Baset Encoding

The `baset-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 `baset-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 `baset-encoding` elements are resolved separately for each character number. There can be multiple `baset-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

```
<baset-encoding name="Character set for the Viking age runic script">
31 runic-f
32 runic-u
</baset-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-characters* 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-characters* 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-characters* 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-characters*. 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, R⁴RS. 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 R⁴RS 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) *Variables* in *required-formal-arguments* 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-arguments*.
- b) Next *variables* in *optional-formal-arguments* are bound to remaining actual arguments. If there are fewer remaining actual arguments than *optional-formal-arguments*, 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-arguments*. If there is no *rest-formal-argument* and there are no *keyword-formal-arguments*, 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* | *quasiquote*

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 *tests* 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 *datums* 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
```