



ISO/IEC JTC 1/SC 32 "Data management and interchange"
Secretariat: ANSI
Committee Manager: Ash Bill Mr



Text for CD 39075 Information Technology — Database Languages — GQL

Document type	Related content	Document date	Expected action
Project / Draft		2021-11-22	VOTE by 2022-02-15

Description

Please submit your ballot via the online balloting system.

ISO/IEC JTC 1/SC 32

Date: 2021-11-22

CD 39075:2021(E)

ISO/IEC JTC 1/SC 32/WG 3

The United States of America (ANSI)

Information technology — Database languages — GQL

Technologies de l'information — Langages de base de données — GQL

Document type: International Standard

Document subtype: Committee Draft (CD)

Document stage: (30) CD under consideration

Document language: English

Document name: 39075_1CD-GQL_2021-11-22

Copyright notice

This ISO document is a Draft International Standard and is copyright-protected by ISO. Except as permitted under the applicable laws of the user's country, neither this ISO draft nor any extract from it may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, photocopying, recording, or otherwise, without prior written permission being secured.

Requests for permission to reproduce should be addressed to ISO at the address below or ISO's member body in the country of the requester.

*ISO copyright office
Case Postale 46 • CH-1211 Geneva 20
Switzerland
Tel. +41 22 749 01 11
Fax +41 22 734 09 47
E-mail: copyright@iso.org
Web: www.iso.org*

Reproduction may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Contents

	Page
Foreword	xxxiii
Introduction	xxxiv
1 Scope.....	1
2 Normative references.....	2
3 Terms and definitions.....	4
3.1 Introduction to terms and definitions.....	4
3.2 General terms and definitions.....	4
3.3 Graph terms and definitions.....	5
3.4 GQL-environment terms and definitions.....	7
3.5 GQL-catalog terms and definitions.....	10
3.6 Procedure terms and definitions.....	12
3.7 Procedure syntax terms and definitions.....	16
3.8 Value terms and definitions.....	18
3.9 Type terms and definitions.....	20
3.10 Temporal terms and definitions.....	23
3.11 Definitions taken from ISO/IEC 14651:2019.....	23
4 Concepts.....	24
4.1 Use of terms.....	24
4.2 Informative elements.....	24
4.3 GQL-environments and their components.....	24
4.3.1 General description of GQL-environments.....	24
4.3.2 GQL-agents.....	25
4.3.3 GQL-implementations.....	25
4.3.3.1 Introduction to GQL-implementations.....	25
4.3.3.2 GQL-clients.....	25
4.3.3.3 GQL-servers.....	26
4.3.4 Basic Security Model.....	26
4.3.4.1 Principals.....	26
4.3.4.2 Authorization identifiers.....	26
4.3.5 GQL-catalog.....	26
4.3.5.1 General description of the GQL-catalog.....	26
4.3.5.2 GQL-directories.....	27
4.3.5.3 GQL-schemas.....	28
4.3.6 GQL-data.....	28
4.4 GQL-objects.....	29
4.4.1 General information about GQL-objects.....	29
4.4.2 Static and dynamic GQL-objects.....	30
4.4.3 Graphs.....	30

4.4.3.1	Introduction to graphs.....	30
4.4.3.2	Graph descriptors.....	32
4.4.3.3	Subgraphs.....	32
4.4.4	Procedure definitions.....	33
4.4.5	Binding tables.....	33
4.4.6	Libraries.....	34
4.4.7	Aliases.....	35
4.4.8	Constant objects.....	35
4.5	Values.....	35
4.5.1	General information about values.....	35
4.5.2	Boxed values.....	35
4.5.3	Reference values.....	35
4.5.4	Value classifications.....	36
4.5.4.1	Values classified by their computational dependencies.....	36
4.5.4.2	Values classified by the kinds of sites in which they occur.....	36
4.6	GQL-sessions.....	36
4.6.1	General description of GQL-sessions.....	36
4.6.2	Session contexts.....	37
4.6.2.1	Introduction to session contexts.....	37
4.6.2.2	Session context creation.....	37
4.6.2.3	Session context modification.....	38
4.7	GQL-transactions.....	38
4.7.1	General description of GQL-transactions.....	38
4.7.2	Transaction demarcation.....	38
4.7.3	Transaction isolation.....	39
4.7.4	Encompassing transaction belonging to an external agent.....	39
4.8	GQL-requests.....	40
4.8.1	General description of GQL-requests.....	40
4.8.2	GQL-request contexts.....	40
4.8.2.1	Introduction to GQL-request contexts.....	40
4.8.2.2	GQL-request context creation.....	41
4.8.2.3	GQL-request context modification.....	41
4.8.3	Execution stack.....	41
4.9	Execution contexts.....	41
4.9.1	General description of execution contexts.....	41
4.9.2	Execution context creation.....	42
4.9.3	Execution context modification.....	43
4.9.4	Execution outcomes.....	43
4.10	Diagnostics.....	44
4.10.1	Introduction to diagnostics.....	44
4.10.2	Conditions.....	44
4.10.3	GQL-status object.....	45
4.11	Procedures and commands.....	46
4.11.1	General description of procedures and commands.....	46
4.11.2	Procedures.....	46
4.11.2.1	General description of procedures.....	46
4.11.2.2	Procedure descriptors.....	46

4.11.2.3	Procedure signatures.....	46
4.11.2.4	Procedure signature descriptor.....	47
4.11.2.5	Type signature descriptor.....	47
4.11.2.6	Procedure execution.....	47
4.11.2.7	Procedures classified by type of computation.....	48
4.11.2.8	Procedures classified by type of provisioning.....	48
4.11.3	Commands.....	48
4.11.3.1	General description of commands.....	48
4.11.3.2	Command execution.....	48
4.11.4	Procedures in GQL.....	49
4.11.4.1	Introduction to procedures in GQL.....	49
4.11.4.2	Variables and parameters.....	49
4.11.4.3	Statements.....	50
4.11.4.4	Statements classified by function.....	50
4.11.4.5	Scope of names.....	50
4.11.5	Operations.....	52
4.11.5.1	Introduction to operations.....	52
4.11.5.2	Operations classified by execution outcome.....	52
4.11.5.3	Operations classified by type of side effects.....	52
4.12	Graph pattern matching.....	53
4.12.1	Summary.....	53
4.12.2	Paths.....	53
4.12.3	Path patterns.....	54
4.12.4	Path pattern matching.....	55
4.12.5	<path mode>.....	55
4.12.6	Selective <path search prefix>.....	56
4.13	Data types.....	56
4.13.1	General information about data types.....	56
4.13.2	Naming of predefined value types and associated base types.....	57
4.13.3	Data type descriptors.....	58
4.13.4	Instance conformance.....	58
4.13.5	Data type relations.....	58
4.13.5.1	Subtype relation.....	58
4.13.5.2	Supertype relation.....	59
4.13.5.3	Extensional type equality.....	59
4.13.5.4	Type cardinality.....	59
4.13.6	Data type operations.....	59
4.13.6.1	Coercion.....	59
4.13.6.2	Type join.....	59
4.13.6.3	Type meet.....	59
4.14	Type hierarchy outline.....	60
4.14.1	Introduction to type hierarchy outline.....	60
4.14.2	The any type.....	60
4.14.3	The any object type.....	60
4.14.4	The any value type.....	60
4.14.5	Union types.....	60
4.14.6	The any property value type.....	60

4.14.7	The nothing type.....	60
4.15	Graph types.....	61
4.15.1	Introduction to graph types.....	61
4.15.2	Graph type descriptors.....	62
4.15.3	Graph element types.....	62
4.15.3.1	Graph element base type.....	62
4.15.3.2	Node base type.....	63
4.15.3.3	Node types.....	63
4.15.3.4	Node type descriptor.....	63
4.15.3.5	Edge base type.....	64
4.15.3.6	Edge types.....	64
4.15.3.7	Edge type descriptor.....	65
4.15.3.8	Property types.....	65
4.15.3.8.1	Introduction to property types.....	65
4.15.3.8.2	Property type descriptor.....	66
4.16	Binding table types.....	66
4.17	Value types.....	66
4.17.1	Constructed types.....	66
4.17.1.1	Introduction to constructed types.....	66
4.17.1.2	Multiset type.....	66
4.17.1.3	Set type.....	67
4.17.1.4	List types.....	67
4.17.1.4.1	List types.....	67
4.17.1.4.2	Regular lists.....	68
4.17.1.4.3	Distinct lists.....	68
4.17.1.5	Path type.....	68
4.17.1.6	Map type.....	68
4.17.1.7	Record type.....	69
4.17.1.8	Unit type.....	69
4.17.2	Reference value types.....	69
4.17.3	Predefined value types.....	70
4.17.3.1	Boolean types.....	70
4.17.3.2	Character string types.....	70
4.17.3.3	Byte string types.....	71
4.17.3.4	Numeric types.....	71
4.17.3.4.1	Introduction to numbers.....	71
4.17.3.4.2	Characteristics of numbers.....	72
4.17.3.4.3	Binary exact numeric types.....	74
4.17.3.4.4	Decimal exact numeric types.....	75
4.17.3.4.5	Approximate numeric types.....	75
4.17.3.5	Temporal types.....	76
4.18	Sites and operations on sites.....	77
4.18.1	Sites.....	77
4.18.2	Assignment.....	77
4.18.3	Nullability.....	77
5	Notation and conventions.....	78
5.1	Notation taken from ISO/IEC 10646.....	78

5.2	Notation.....	78
5.3	Conventions.....	80
5.3.1	Specification of syntactic elements.....	80
5.3.2	Specification of the Information Graph Schema.....	80
5.3.3	Use of terms.....	81
5.3.3.1	Syntactic containment.....	81
5.3.3.2	Terms denoting rule requirements.....	81
5.3.3.3	Rule evaluation order.....	81
5.3.3.4	Conditional rules.....	82
5.3.3.5	Syntactic substitution.....	83
5.3.4	Descriptors.....	83
5.3.5	Subclauses used as subroutines.....	84
5.3.6	Index typography.....	84
5.3.7	Feature ID and Feature Name.....	84
6	GQL-requests.....	85
6.1	<GQL-request>.....	85
6.2	<request parameter set>.....	87
6.3	<GQL-program>.....	88
6.4	<preamble>.....	89
7	Session management.....	90
7.1	<session set command>.....	90
7.2	<session remove command>.....	92
7.3	<session clear command>.....	93
7.4	<session close command>.....	94
8	Transaction management.....	95
8.1	<start transaction command>.....	95
8.2	<end transaction command>.....	96
8.3	<transaction characteristics>.....	97
8.4	<rollback command>.....	98
8.5	<commit command>.....	99
9	Procedures.....	100
9.1	<procedure specification>.....	100
9.2	<query specification>.....	103
9.3	<function specification>.....	104
9.4	<procedure body>.....	105
10	Variable and parameter declarations and definitions.....	107
10.1	Static variable definitions.....	107
10.2	Procedure variable definition.....	108
10.3	Query variable definition.....	109
10.4	Function variable definition.....	110
10.5	Binding variable and parameter declarations and definitions.....	111
10.6	Graph variable and parameter declaration and definition.....	113
10.7	Binding table variable and parameter declaration and definition.....	115
10.8	Value variable and parameter declaration and definition.....	117
11	Object expressions.....	119

11.1	Introduction to object expressions.....	119
11.2	<primary result object expression>.....	120
11.3	<graph expression>.....	121
11.4	<graph type expression>.....	122
11.5	<binding table type expression>.....	124
12	Statements.....	125
12.1	<statement>.....	125
12.2	<call procedure statement>.....	127
12.3	Statement classes.....	128
13	Catalog-modifying statements.....	130
13.1	<linear catalog-modifying statement>.....	130
13.2	<create schema statement>.....	131
13.3	<drop schema statement>.....	132
13.4	<create graph statement>.....	133
13.5	<graph specification>.....	136
13.6	<drop graph statement>.....	138
13.7	<create graph type statement>.....	139
13.8	<graph type specification>.....	141
13.9	<node type definition>.....	143
13.10	<edge type definition>.....	145
13.11	<label set definition>.....	150
13.12	<property type set definition>.....	151
13.13	<drop graph type statement>.....	152
13.14	<create procedure statement>.....	153
13.15	<drop procedure statement>.....	154
13.16	<create query statement>.....	155
13.17	<drop query statement>.....	156
13.18	<create function statement>.....	157
13.19	<drop function statement>.....	158
13.20	<call catalog-modifying procedure statement>.....	159
14	Data-modifying statements.....	160
14.1	<linear data-modifying statement>.....	160
14.2	<conditional data-modifying statement>.....	162
14.3	<do statement>.....	164
14.4	<insert statement>.....	165
14.5	<merge statement>.....	167
14.6	<set statement>.....	168
14.7	<remove statement>.....	170
14.8	<delete statement>.....	172
14.9	<call data-modifying procedure statement>.....	174
15	Query statements.....	175
15.1	<composite query statement>.....	175
15.2	<conditional query statement>.....	176
15.3	<composite query expression>.....	178
15.4	<linear query expression>.....	181
15.5	<linear query statement>.....	182

15.6	Data-reading statements.....	184
15.6.1	<match statement>.....	184
15.6.2	<call query statement>.....	186
15.7	Data-transforming statements.....	187
15.7.1	<mandatory statement>.....	187
15.7.2	<optional statement>.....	188
15.7.3	<filter statement>.....	189
15.7.4	<let statement>.....	190
15.7.5	<aggregate statement>.....	192
15.7.6	<for statement>.....	193
15.7.7	<order by and page statement>.....	196
15.7.8	<call function statement>.....	198
15.8	Result projection statements.....	199
15.8.1	<primitive result statement>.....	199
15.8.2	<return statement>.....	200
15.8.3	<select statement>.....	204
15.8.4	<project statement>.....	205
16	Common elements.....	206
16.1	<from graph clause>.....	206
16.2	<use graph clause>.....	207
16.3	<at schema clause>.....	208
16.4	Named elements.....	209
16.5	<type signature>.....	211
16.6	<graph pattern>.....	213
16.7	<path pattern expression>.....	217
16.8	<path pattern prefix>.....	227
16.9	<simple graph pattern>.....	230
16.10	<label expression>.....	232
16.11	<simplified path pattern expression>.....	234
16.12	<where clause>.....	239
16.13	<procedure call>.....	240
16.14	<inline procedure call>.....	241
16.15	<named procedure call>.....	242
16.16	<yield clause>.....	244
16.17	<group by clause>.....	246
16.18	<order by clause>.....	248
16.19	<aggregate function>.....	249
16.20	<sort specification list>.....	251
16.21	<limit clause>.....	254
16.22	<offset clause>.....	255
17	Object references.....	256
17.1	Schema references.....	256
17.2	Graph references.....	258
17.3	Graph type references.....	261
17.4	Binding table references.....	263
17.5	Procedure references.....	266
17.6	Query references.....	268

17.7	Function references.....	270
17.8	<catalog object reference>.....	272
17.9	<qualified object name>.....	275
17.10	<url path parameter>.....	277
17.11	<external object reference>.....	279
17.12	<element reference>.....	280
18	Functions.....	281
19	Predicates.....	282
19.1	<search condition>.....	282
19.2	<predicate>.....	283
19.3	<comparison predicate>.....	284
19.4	<exists predicate>.....	287
19.5	<null predicate>.....	289
19.6	<normalized predicate>.....	290
19.7	<directed predicate>.....	291
19.8	<labeled predicate>.....	292
19.9	<source/destination predicate>.....	293
19.10	<all_different predicate>.....	295
19.11	<same predicate>.....	296
20	Value expressions.....	297
20.1	<value specification>.....	297
20.2	<value expression>.....	300
20.3	<boolean value expression>.....	302
20.4	<numeric value expression>.....	304
20.5	<value expression primary>.....	306
20.6	<numeric value function>.....	307
20.7	<string value expression>.....	313
20.8	<string value function>.....	316
20.9	<datetime value expression>.....	322
20.10	<datetime value function>.....	323
20.11	<duration value expression>.....	326
20.12	<duration value function>.....	328
20.13	<graph element value expression>.....	329
20.14	<graph element function>.....	330
20.15	<collection value constructor>.....	331
20.16	<list value expression>.....	332
20.17	<list value function>.....	334
20.18	<list value constructor>.....	336
20.19	<multipset value expression>.....	337
20.20	<multipset value function>.....	340
20.21	<multipset value constructor>.....	341
20.22	<set value constructor>.....	342
20.23	<ordered set value constructor>.....	343
20.24	<map value constructor>.....	344
20.25	<record value constructor>.....	346
20.26	<property reference>.....	348

20.27	<value query expression>.....	349
20.28	<case expression>.....	350
20.29	<cast specification>.....	353
20.30	<element_id function>.....	364
21	Lexical elements.....	365
21.1	<literal>.....	365
21.2	<value type>.....	375
21.3	Names and identifiers.....	385
21.4	<token> and <separator>.....	388
21.5	<GQL terminal character>.....	396
22	Additional common rules.....	400
22.1	Satisfaction of a <label expression> by a label set.....	400
22.2	Machinery for graph pattern matching.....	402
22.3	Evaluation of a <path pattern expression>.....	407
22.4	Evaluation of a selective <path pattern>.....	412
22.5	Applying bindings to evaluate an expression.....	415
22.6	Equality operations.....	418
22.7	Ordering operations.....	419
22.8	Result of value type combinations.....	420
22.9	Bag element grouping operations.....	423
23	Diagnostics.....	424
24	Status codes.....	427
24.1	GQLSTATUS.....	427
25	Conformance.....	431
25.1	Introduction to conformance.....	431
25.2	Minimum conformance.....	431
25.3	Conformance to features.....	431
25.4	Requirements for GQL-programs.....	432
25.4.1	Introduction to requirements for GQL-programs.....	432
25.4.2	Claims of conformance for GQL-programs.....	432
25.5	Requirements for GQL-implementations.....	432
25.5.1	Introduction to requirements for GQL-implementations.....	432
25.5.2	Claims of conformance for GQL-implementations.....	432
25.5.3	Extensions and options.....	433
25.6	Requirements for graphs.....	433
25.6.1	Introduction to requirements for graphs.....	433
25.6.2	Claims of conformance for graphs.....	433
25.7	GQL Flagger.....	433
25.8	Implied feature relationships.....	434
Annex A (informative) GQL Conformance Summary.....	435	
Annex B (informative) Implementation-defined elements.....	441	
Annex C (informative) Implementation-dependent elements.....	456	
Annex D (informative) GQL feature taxonomy.....	460	
Annex E (informative) Maintenance and interpretation of GQL.....	462	

Annex F (informative) Differences with SQL	463
Annex G (informative) Graph serialization format	467
Bibliography	468
Index	469

Tables

Table		Page
1 Symbols used in BNF.....		78
2 Conversion of simplified syntax delimiters to default edge delimiters.....		236
3 Truth table for the AND Boolean operator.....		303
4 Truth table for the OR Boolean operator.....		303
5 Truth table for the IS Boolean operator.....		303
6 Command codes.....		424
7 GQLSTATUS class and subclass codes.....		427
8 Implied feature relationships.....		434
9 Feature taxonomy for optional features.....		460

Figures

Figure	Page
1 Components of a GQL-environment	24
2 Components of a GQL-catalog	27

BNF non-terminal symbols

BNF non-terminal symbol	Page
<GQL language character>.....	396
<GQL special character>.....	396
<GQL terminal character>.....	396
<GQL-program>.....	88
<GQL-request>.....	85
<SQL-interval literal>.....	368
<abbreviated edge pattern>.....	218
<abbreviated edge type pattern>.....	145
<abbreviated edge type pattern any direction>.....	146
<abbreviated edge type pattern pointing left>.....	146
<abbreviated edge type pattern pointing right>.....	146
<absolute url path>.....	272
<absolute value expression>.....	307
<accent quoted character representation>.....	366
<aggregate function>.....	249
<aggregate statement>.....	192
<all path search>.....	227
<all shortest path search>.....	227
<all_different predicate>.....	295
<ambient linear data-modifying statement>.....	160
<ambient linear query statement>.....	182
<ampersand>.....	397
<any path search>.....	227
<any shortest path search>.....	227
<approximate numeric literal>.....	367
<approximate numeric type>.....	376
<arc type any direction>.....	145
<arc type filler>.....	145
<arc type pointing left>.....	145
<arc type pointing right>.....	145
<as graph type>.....	122
<as or equals>.....	107
<asterisk>.....	397
<at schema clause>.....	208
<binary digit>.....	396
<binary exact numeric type>.....	375
<binary exact signed numeric type>.....	376
<binary exact unsigned numeric type>.....	376
<binary set function>.....	249
<binary set function type>.....	249
<binding table initializer>.....	115
<binding table name>.....	385
<binding table parameter definition>.....	115
<binding table parent specification>.....	263
<binding table reference>.....	263
<binding table resolution expression>.....	263
<binding table type>.....	124

<binding table type expression>.....	124
<binding table variable>.....	115
<binding table variable declaration>.....	115
<binding table variable definition>.....	115
<binding variable>.....	209
<binding variable declaration>.....	111
<binding variable definition>.....	111
<binding variable definition block>.....	105
<binding variable definition list>.....	111
<binding variable name>.....	386
<boolean factor>.....	302
<boolean literal>.....	365
<boolean predicand>.....	302
<boolean primary>.....	302
<boolean term>.....	302
<boolean test>.....	302
<boolean type>.....	375
<boolean value expression>.....	302
<bracket right arrow>.....	391
<bracket tilde right arrow>.....	391
<bracketed comment>.....	392
<bracketed comment contents>.....	393
<bracketed comment introducer>.....	393
<bracketed comment terminator>.....	393
<byte length expression>.....	307
<byte string concatenation>.....	313
<byte string factor>.....	313
<byte string function>.....	316
<byte string literal>.....	367
<byte string primary>.....	313
<byte string trim function>.....	317
<byte string trim source>.....	317
<byte string type>.....	375
<byte string value expression>.....	313
<byte substring function>.....	317
<call catalog-modifying procedure statement>.....	159
<call data-modifying procedure statement>.....	174
<call function statement>.....	198
<call procedure statement>.....	127
<call query statement>.....	186
<case abbreviation>.....	350
<case expression>.....	350
<case operand>.....	350
<case specification>.....	350
<case-insensitive non-reserved word>.....	389
<case-insensitive reserved word>.....	388
<cast operand>.....	353
<cast specification>.....	353
<cast target>.....	353
<catalog binding table parent and name>.....	263
<catalog binding table reference>.....	263

<catalog function parent and name>.....	270
<catalog function reference>.....	270
<catalog graph parent and name>.....	258
<catalog graph reference>.....	258
<catalog graph type parent and name>.....	261
<catalog graph type reference>.....	261
<catalog object reference>.....	272
<catalog procedure parent and name>.....	266
<catalog procedure reference>.....	266
<catalog query parent and name>.....	268
<catalog query reference>.....	268
<catalog schema parent and name>.....	256
<catalog url path>.....	272
<catalog-modifying procedure specification>.....	100
<catalog-modifying statement>.....	125
<ceiling function>.....	308
<char length expression>.....	307
<character representation>.....	366
<character string concatenation>.....	313
<character string factor>.....	313
<character string function>.....	316
<character string literal>.....	365
<character string primary>.....	313
<character string type>.....	375
<character string value expression>.....	313
<circumflex>.....	397
<collection type>.....	377
<collection value constructor>.....	331
<collection value expression>.....	300
<colon>.....	397
<comma>.....	397
<comment>.....	392
<commit command>.....	99
<common logarithm>.....	308
<common value expression>.....	300
<comp op>.....	284
<compact value variable definition>.....	111
<compact value variable definition list>.....	111
<compact variable declaration>.....	111
<compact variable declaration list>.....	111
<compact variable definition>.....	111
<compact variable definition list>.....	111
<comparison predicate>.....	284
<comparison predicate part 2>.....	284
<composite query expression>.....	178
<composite query statement>.....	175
<concatenation operator>.....	391
<conditional data-modifying statement>.....	162
<conditional query statement>.....	176
<connector any direction>.....	146
<connector pointing right>.....	146

<copy graph expression>.....	121
<copy graph type expression>.....	122
<cost clause>.....	218
<counted shortest group search>.....	227
<counted shortest path search>.....	227
<create function statement>.....	157
<create graph statement>.....	133
<create graph type statement>.....	139
<create procedure statement>.....	153
<create query statement>.....	155
<create schema statement>.....	131
<data-modifying procedure specification>.....	100
<data-modifying statement>.....	125
<date function>.....	323
<date function parameters>.....	323
<date literal>.....	368
<date string>.....	368
<datetime factor>.....	322
<datetime function>.....	323
<datetime function parameters>.....	323
<datetime literal>.....	368
<datetime primary>.....	322
<datetime string>.....	368
<datetime term>.....	322
<datetime value expression>.....	322
<datetime value function>.....	323
<decimal exact numeric type>.....	376
<delete item>.....	172
<delete item list>.....	172
<delete statement>.....	172
<delimited identifier>.....	392
<delimiter token>.....	390
<dependent value expression>.....	249
<destination node type name>.....	146
<destination node type reference>.....	146
<destination predicate part 2>.....	293
<digit>.....	396
<directed predicate>.....	291
<directed predicate part 2>.....	291
<do statement>.....	164
<dollar sign>.....	397
<double colon>.....	391
<double minus sign>.....	391
<double period>.....	391
<double quote>.....	397
<double quoted character representation>.....	366
<double quoted character sequence>.....	365
<double solidus>.....	392
<doubled grave accent>.....	393
<drop function statement>.....	158
<drop graph statement>.....	138

<drop graph type statement>.....	152
<drop procedure statement>.....	154
<drop query statement>.....	156
<drop schema statement>.....	132
<duration absolute value function>.....	328
<duration factor>.....	326
<duration function>.....	328
<duration function parameters>.....	328
<duration literal>.....	368
<duration primary>.....	326
<duration string>.....	368
<duration term>.....	326
<duration term 1>.....	326
<duration term 2>.....	326
<duration value expression>.....	326
<duration value expression 1>.....	326
<duration value function>.....	328
<edge kind>.....	146
<edge pattern>.....	218
<edge reference>.....	293
<edge synonym>.....	393
<edge type definition>.....	145
<edge type filler>.....	145
<edge type label set definition>.....	145
<edge type name>.....	145
<edge type property type set definition>.....	145
<element pattern>.....	217
<element pattern cost clause>.....	218
<element pattern filler>.....	217
<element pattern predicate>.....	218
<element pattern where clause>.....	218
<element property specification>.....	218
<element reference>.....	280
<element type definition>.....	141
<element type definition list>.....	141
<element type name>.....	385
<element variable>.....	385
<element variable declaration>.....	217
<element_id function>.....	364
<else clause>.....	350
<else linear data-modifying statement branch>.....	162
<else linear query branch>.....	176
<empty grouping set>.....	246
<end node function>.....	330
<end transaction command>.....	96
<endpoint definition>.....	146
<endpoint pair definition>.....	146
<endpoint pair definition any direction>.....	146
<endpoint pair definition pointing left>.....	146
<endpoint pair definition pointing right>.....	146
<equals operator>.....	397

<escaped backspace>.....	366
<escaped carriage return>.....	366
<escaped character>.....	366
<escaped double quote>.....	366
<escaped form feed>.....	366
<escaped grave accent>.....	393
<escaped newline>.....	366
<escaped quote>.....	366
<escaped reverse solidus>.....	366
<escaped tab>.....	366
<exact numeric literal>.....	367
<exact numeric type>.....	375
<exclamation mark>.....	397
<exists predicate>.....	287
<exponent>.....	367
<exponential function>.....	308
<extended identifier>.....	388
<external object reference>.....	279
<external object url>.....	279
<factor>.....	304
<field>.....	346
<field list>.....	346
<field name>.....	385
<field type>.....	377
<field type list>.....	377
<field value>.....	346
<filter statement>.....	189
<fixed length>.....	375
<fixed quantifier>.....	219
<floor function>.....	308
<focused linear data-modifying statement>.....	160
<focused linear data-modifying statement body>.....	160
<focused linear query statement>.....	182
<focused linear query statement body>.....	182
<fold>.....	316
<for item>.....	193
<for item alias>.....	193
<for item list>.....	193
<for ordinality or index>.....	193
<for statement>.....	193
<formal parameter declaration>.....	211
<formal parameter declaration list>.....	211
<formal parameter definition>.....	211
<formal parameter definition list>.....	211
<formal parameter list>.....	211
<from graph clause>.....	206
<full edge any direction>.....	218
<full edge left or right>.....	218
<full edge left or undirected>.....	218
<full edge pattern>.....	218
<full edge pointing left>.....	218

<full edge pointing right>.....	218
<full edge type pattern>.....	145
<full edge type pattern any direction>.....	145
<full edge type pattern pointing left>.....	145
<full edge type pattern pointing right>.....	145
<full edge undirected>.....	218
<full edge undirected or right>.....	218
<function initializer>.....	110
<function name>.....	385
<function parent specification>.....	270
<function reference>.....	270
<function resolution expression>.....	270
<function specification>.....	104
<function variable>.....	110
<function variable definition>.....	110
<general literal>.....	365
<general logarithm argument>.....	308
<general logarithm base>.....	308
<general logarithm function>.....	307
<general quantifier>.....	219
<general set function>.....	249
<general set function type>.....	249
<graph element function>.....	330
<graph element primary>.....	329
<graph element type>.....	376
<graph element value expression>.....	329
<graph expression>.....	121
<graph initializer>.....	113
<graph name>.....	385
<graph parameter definition>.....	113
<graph parent specification>.....	258
<graph pattern>.....	213
<graph pattern quantifier>.....	219
<graph pattern where clause>.....	213
<graph reference>.....	258
<graph resolution expression>.....	258
<graph source>.....	133
<graph specification>.....	136
<graph type expression>.....	122
<graph type initializer>.....	139
<graph type name>.....	385
<graph type parent specification>.....	261
<graph type reference>.....	261
<graph type resolution expression>.....	261
<graph type specification>.....	141
<graph type specification body>.....	141
<graph variable>.....	113
<graph variable declaration>.....	113
<graph variable definition>.....	113
<grave accent>.....	397
<greater than operator>.....	391

<greater than or equals operator>.....	391
<group by clause>.....	246
<grouping element>.....	246
<grouping element list>.....	246
<having clause>.....	204
<hex digit>.....	396
<identifier>.....	386
<identifier extend>.....	388
<identifier start>.....	388
<implementation-defined access mode>.....	97
<inDegree function>.....	308
<independent value expression>.....	249
<inline procedure call>.....	241
<insert statement>.....	165
<is label expression>.....	218
<is or colon>.....	218
<keep clause>.....	213
<key word>.....	388
<label>.....	209
<label conjunction>.....	232
<label disjunction>.....	232
<label expression>.....	232
<label factor>.....	232
<label name>.....	385
<label negation>.....	232
<label primary>.....	232
<label set definition>.....	150
<label set expression>.....	168
<label term>.....	232
<labeled predicate>.....	292
<labeled predicate part 2>.....	292
<left angle bracket>.....	398
<left arrow>.....	391
<left arrow bracket>.....	391
<left arrow tilde>.....	391
<left arrow tilde bracket>.....	391
<left brace>.....	397
<left bracket>.....	397
<left minus right>.....	391
<left minus slash>.....	391
<left paren>.....	397
<left tilde slash>.....	391
<length expression>.....	307
<less than operator>.....	391
<less than or equals operator>.....	391
<let statement>.....	190
<like binding table shorthand>.....	124
<like binding table type>.....	124
<like graph expression>.....	122
<like graph expression shorthand>.....	122
<limit clause>.....	254

<linear catalog-modifying statement>.....	130
<linear data-modifying statement>.....	160
<linear query expression>.....	181
<linear query statement>.....	182
<list concatenation>.....	332
<list element>.....	336
<list element list>.....	336
<list literal>.....	368
<list primary>.....	332
<list value constructor>.....	336
<list value constructor by enumeration>.....	336
<list value expression>.....	332
<list value expression 1>.....	332
<list value function>.....	334
<list value type>.....	377
<list value type name>.....	377
<literal>.....	365
<local binding table reference>.....	263
<local datetime function>.....	323
<local function reference>.....	270
<local graph reference>.....	258
<local graph type reference>.....	261
<local procedure reference>.....	266
<local query reference>.....	268
<local time function>.....	323
<lower bound>.....	219
<main activity>.....	88
<mandatory formal parameter list>.....	211
<mandatory statement>.....	187
<mantissa>.....	367
<map element>.....	344
<map element list>.....	344
<map key>.....	344
<map key type>.....	377
<map literal>.....	368
<map value>.....	344
<map value constructor>.....	344
<map value constructor by enumeration>.....	344
<map value expression>.....	300
<map value type>.....	377
<match statement>.....	184
<max length>.....	375
<merge statement>.....	167
<min length>.....	375
<minus left bracket>.....	391
<minus sign>.....	398
<minus slash>.....	392
<modulus expression>.....	307
<multiset alternation operator>.....	390
<multiset element>.....	341
<multiset element list>.....	341

<multiset literal>.....	368
<multiset primary>.....	337
<multiset set function>.....	340
<multiset term>.....	337
<multiset value constructor>.....	341
<multiset value constructor by enumeration>.....	341
<multiset value expression>.....	337
<multiset value function>.....	340
<multiset value type>.....	377
<named procedure call>.....	242
<natural logarithm>.....	308
<nested ambient data-modifying procedure specification>.....	136
<nested catalog-modifying procedure specification>.....	100
<nested data-modifying procedure specification>.....	100
<nested function specification>.....	104
<nested graph query specification>.....	136
<nested graph type specification>.....	141
<nested procedure specification>.....	100
<nested query specification>.....	103
<newline>.....	393
<node pattern>.....	217
<node reference>.....	293
<node synonym>.....	393
<node type definition>.....	143
<node type filler>.....	143
<node type label set definition>.....	143
<node type name>.....	143
<node type property type set definition>.....	143
<non-delimited identifier>.....	388
<non-delimiter token>.....	388
<non-parenthesized value expression primary>.....	306
<non-reserved word>.....	389
<normal form>.....	316
<normalize function>.....	316
<normalized predicate>.....	290
<normalized predicate part 2>.....	290
<not equals operator>.....	392
<null literal>.....	368
<null ordering>.....	251
<null predicate>.....	289
<null predicate part 2>.....	289
<number of groups>.....	227
<number of paths>.....	227
<numeric literal>.....	367
<numeric primary>.....	304
<numeric type>.....	375
<numeric value expression>.....	304
<numeric value expression base>.....	308
<numeric value expression dividend>.....	307
<numeric value expression divisor>.....	307
<numeric value expression exponent>.....	308

<numeric value function>.....	307
<object name>.....	385
<octal digit>.....	396
<of binding table type>.....	124
<of graph type>.....	122
<of type prefix>.....	375
<of type signature>.....	211
<of value type>.....	375
<offset clause>.....	255
<offset synonym>.....	255
<optional binding table variable definition>.....	115
<optional binding variable definition>.....	111
<optional binding variable definition list>.....	111
<optional formal parameter list>.....	211
<optional graph variable definition>.....	113
<optional parameter cardinality>.....	211
<optional statement>.....	188
<optional value variable definition>.....	117
<order by and page statement>.....	196
<order by clause>.....	248
<ordered set element>.....	343
<ordered set element list>.....	343
<ordered set literal>.....	368
<ordered set value constructor>.....	343
<ordered set value constructor by enumeration>.....	343
<ordered set value expression>.....	300
<ordered set value type>.....	377
<ordering specification>.....	251
<other digit>.....	396
<other language character>.....	398
<outDegree function>.....	308
<parameter>.....	209
<parameter cardinality>.....	211
<parameter definition>.....	111
<parameter name>.....	385
<parameter value specification>.....	297
<parameterized url path>.....	272
<parent catalog object reference>.....	272
<parent object relative url path>.....	272
<parenthesized Boolean value expression>.....	302
<parenthesized formal parameter list>.....	211
<parenthesized label expression>.....	232
<parenthesized path pattern cost clause>.....	219
<parenthesized path pattern expression>.....	219
<parenthesized path pattern where clause>.....	219
<parenthesized value expression>.....	306
<path concatenation>.....	217
<path factor>.....	217
<path length expression>.....	307
<path mode>.....	227
<path mode prefix>.....	227

<path multiset alternation>.....	217
<path or paths>.....	227
<path pattern>.....	213
<path pattern expression>.....	217
<path pattern list>.....	213
<path pattern name>.....	385
<path pattern prefix>.....	227
<path pattern union>.....	217
<path primary>.....	217
<path search prefix>.....	227
<path term>.....	217
<path variable>.....	385
<percent>.....	398
<period>.....	398
<plus sign>.....	398
<power function>.....	308
<preamble>.....	89
<preamble option>.....	89
<preamble option identifier>.....	89
<precision>.....	376
<predefined graph parameter>.....	297
<predefined parameter>.....	297
<predefined parent object parameter>.....	297
<predefined schema parameter>.....	297
<predefined table parameter>.....	297
<predefined type>.....	375
<predefined type literal>.....	365
<predicate>.....	283
<primary result object expression>.....	120
<primitive catalog-modifying statement>.....	128
<primitive data-modifying statement>.....	128
<primitive data-transforming statement>.....	128
<primitive result statement>.....	199
<procedure argument>.....	242
<procedure argument list>.....	242
<procedure body>.....	105
<procedure call>.....	240
<procedure initializer>.....	108
<procedure name>.....	385
<procedure parent specification>.....	266
<procedure reference>.....	266
<procedure resolution expression>.....	266
<procedure result type>.....	211
<procedure specification>.....	100
<procedure variable>.....	108
<procedure variable definition>.....	108
<project statement>.....	205
<property key value pair>.....	218
<property key value pair list>.....	218
<property name>.....	385
<property reference>.....	348

<property type definition>.....	151
<property type definition list>.....	151
<property type set definition>.....	151
<qualified binding table name>.....	263
<qualified function name>.....	270
<qualified graph name>.....	258
<qualified graph type name>.....	261
<qualified name prefix>.....	275
<qualified object name>.....	275
<qualified procedure name>.....	266
<qualified query name>.....	268
<quantified path primary>.....	217
<query conjunction>.....	178
<query initializer>.....	109
<query name>.....	385
<query parent specification>.....	268
<query reference>.....	268
<query resolution expression>.....	268
<query specification>.....	103
<query statement>.....	125
<query variable>.....	109
<query variable definition>.....	109
<question mark>.....	398
<questioned path primary>.....	217
<quote>.....	398
<record literal>.....	368
<record value constructor>.....	346
<record value constructor by enumeration>.....	346
<record value expression>.....	300
<record value type>.....	377
<reference value expression>.....	300
<regular identifier>.....	388
<relative url path>.....	272
<remove item>.....	170
<remove item list>.....	170
<remove label item>.....	170
<remove property item>.....	170
<remove statement>.....	170
<request parameter>.....	87
<request parameter set>.....	87
<reserved word>.....	388
<result>.....	350
<result expression>.....	350
<return item>.....	200
<return item alias>.....	200
<return item list>.....	200
<return statement>.....	200
<return statement body>.....	200
<reverse solidus>.....	398
<right angle bracket>.....	397
<right arrow>.....	392

<right brace>.....	398
<right bracket>.....	398
<right bracket minus>.....	392
<right bracket tilde>.....	392
<right paren>.....	398
<rollback command>.....	98
<same predicate>.....	296
<scale>.....	376
<schema name>.....	385
<schema reference>.....	256
<search condition>.....	282
<searched case>.....	350
<searched when clause>.....	350
<select graph match>.....	204
<select graph match list>.....	204
<select item>.....	204
<select item alias>.....	204
<select item list>.....	204
<select query specification>.....	204
<select statement>.....	204
<select statement body>.....	204
<semicolon>.....	398
<separated identifier>.....	386
<separator>.....	392
<session activity>.....	88
<session clear command>.....	93
<session close command>.....	94
<session parameter>.....	90
<session parameter command>.....	88
<session parameter flag>.....	90
<session remove command>.....	92
<session set command>.....	90
<session set graph clause>.....	90
<session set parameter clause>.....	90
<session set schema clause>.....	90
<session set time zone clause>.....	90
<set all properties item>.....	168
<set element>.....	342
<set element list>.....	342
<set item>.....	168
<set item list>.....	168
<set label item>.....	168
<set literal>.....	368
<set operator>.....	178
<set property item>.....	168
<set quantifier>.....	249
<set statement>.....	168
<set time zone value>.....	90
<set value constructor>.....	342
<set value constructor by enumeration>.....	342
<set value expression>.....	300

<set value type>.....	377
<shortest path search>.....	227
<sign>.....	367
<signed decimal integer>.....	367
<signed numeric literal>.....	367
<simple Latin letter>.....	396
<simple Latin lower-case letter>.....	396
<simple Latin upper-case letter>.....	396
<simple case>.....	350
<simple catalog-modifying statement>.....	128
<simple comment>.....	392
<simple comment character>.....	392
<simple comment introducer>.....	392
<simple data-accessing statement>.....	128
<simple data-modifying statement>.....	128
<simple data-reading statement>.....	128
<simple data-transforming statement>.....	128
<simple graph pattern>.....	230
<simple linear query statement>.....	182
<simple path pattern>.....	230
<simple path pattern list>.....	230
<simple query statement>.....	128
<simple relative url path>.....	272
<simple url path>.....	272
<simple when clause>.....	350
<simplified concatenation>.....	234
<simplified conjunction>.....	235
<simplified contents>.....	234
<simplified defaulting any direction>.....	234
<simplified defaulting left>.....	234
<simplified defaulting left or right>.....	234
<simplified defaulting left or undirected>.....	234
<simplified defaulting right>.....	234
<simplified defaulting undirected>.....	234
<simplified defaulting undirected or right>.....	234
<simplified direction override>.....	235
<simplified factor high>.....	235
<simplified factor low>.....	234
<simplified multiset alternation>.....	234
<simplified negation>.....	235
<simplified override any direction>.....	235
<simplified override left>.....	235
<simplified override left or right>.....	235
<simplified override left or undirected>.....	235
<simplified override right>.....	235
<simplified override undirected>.....	235
<simplified override undirected or right>.....	235
<simplified path pattern expression>.....	234
<simplified path union>.....	234
<simplified primary>.....	235
<simplified quantified>.....	235

<simplified questioned>.....	235
<simplified secondary>.....	235
<simplified term>.....	234
<simplified tertiary>.....	235
<single quoted character representation>.....	366
<single quoted character sequence>.....	365
<slash minus>.....	392
<slash minus right>.....	392
<slash tilde>.....	392
<slash tilde right>.....	392
<solidus>.....	398
<sort key>.....	251
<sort specification>.....	251
<sort specification list>.....	251
<source node type name>.....	146
<source node type reference>.....	146
<source predicate part 2>.....	293
<source/destination predicate>.....	293
<space>.....	397
<square root>.....	308
<standard digit>.....	396
<start node function>.....	330
<start position>.....	317
<start transaction command>.....	95
<statement>.....	125
<statement block>.....	105
<statement mode>.....	127
<static variable>.....	209
<static variable definition>.....	107
<static variable definition block>.....	105
<static variable name>.....	386
<string length>.....	317
<string literal character>.....	366
<string value expression>.....	313
<string value function>.....	316
<subpath variable>.....	386
<subpath variable declaration>.....	219
<substring function>.....	316
<tail list function>.....	334
<temporal literal>.....	367
<temporal type>.....	376
<term>.....	304
<then statement>.....	105
<tilde>.....	398
<tilde left bracket>.....	392
<tilde right arrow>.....	392
<tilde slash>.....	392
<time function>.....	323
<time function parameters>.....	323
<time literal>.....	368
<time string>.....	368

<token>.....	388
<transaction access mode>.....	97
<transaction activity>.....	88
<transaction characteristics>.....	97
<transaction mode>.....	97
<trigonometric function>.....	307
<trigonometric function name>.....	307
<trim byte string>.....	317
<trim character string>.....	316
<trim function>.....	316
<trim list function>.....	334
<trim source>.....	316
<trim specification>.....	316
<truth value>.....	302
<type name>.....	385
<type signature>.....	211
<unbroken accent quoted character sequence>.....	366
<unbroken character string literal>.....	365
<unbroken double quoted character sequence>.....	366
<unbroken single quoted character sequence>.....	365
<underscore>.....	398
<unicode 4 digit escape value>.....	366
<unicode 6 digit escape value>.....	367
<unicode escape value>.....	366
<unsigned binary integer>.....	367
<unsigned decimal integer>.....	367
<unsigned hexadecimal integer>.....	367
<unsigned integer>.....	367
<unsigned integer specification>.....	297
<unsigned literal>.....	365
<unsigned numeric literal>.....	367
<unsigned octal integer>.....	367
<unsigned value specification>.....	297
<untyped value expression>.....	300
<upper bound>.....	219
<url path parameter>.....	277
<url segment>.....	272
<use graph clause>.....	207
<value expression>.....	300
<value expression primary>.....	306
<value initializer>.....	117
<value name>.....	385
<value parameter definition>.....	117
<value query expression>.....	349
<value specification>.....	297
<value type>.....	375
<value variable>.....	117
<value variable declaration>.....	117
<value variable definition>.....	117
<variable name>.....	386
<verbose binary exact numeric type>.....	376

<vertical bar>.....	398
<when clause>.....	162
<when operand>.....	350
<when operand list>.....	350
<when then linear data-modifying statement branch>.....	162
<when then linear query branch>.....	176
<where clause>.....	239
<whitespace>.....	392
<wildcard label>.....	232
<yield clause>.....	244
<yield item>.....	244
<yield item alias>.....	244
<yield item list>.....	244
<yield item name>.....	244

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.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents), or the IEC list of patent declarations received (see patents.iec.ch).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This is the first edition of ISO/IEC 39075.

Any feedback or questions on this document should be directed to the user's national standards body. www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

This document defines the data structures and basic operations on the GQL-catalog and GQL-data for the GQL language.

The GQL language is a composable declarative database language for working with property graphs that can be used both as an independent language as well as in conjunction with SQL or other languages.

As a declarative language, GQL is intended to allow multiple, different implementation strategies (e.g., by pattern matching using relational algebra, linear algebra, or automata theory) using different storage representations (e.g., index-free adjacency storage, classic tabular storage techniques, matrix representations) and targeting various classes of workloads (e.g., OLTP, OLAP).

Information technology — Database languages — GQL

1 Scope

The GQL language provides functional capabilities for

- querying, modifying, and projecting property graphs,
- querying, modifying, and projecting property graph views,
- transforming binding tables,
- working with primitive data values,
- working with nested collections and maps of data values,
- composing requests from procedures and commands and composing optionally parameterized procedures from nested procedures, statements, patterns, expressions, and similar constructs,
- managing named GQL-directories, GQL-schemas and catalog objects in the GQL-catalog,
- working with catalog objects such as property graphs, property graph types (comprising graph element and constraint definitions), user-defined and built-in procedures, queries,
- session management,
- transaction demarcation,
- interacting with other languages and systems,
- handling errors, and
- reporting related diagnostic information.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEEE Std 754:2019, *IEEE Standard for Floating-Point Arithmetic*

ISO 8601-1:2019, *Date and time — Representations for information interchange — Part 1: Basic rules*

ISO 8601-2:2019, *Date and time — Representations for information interchange — Part 2: Extensions*

ISO/IEC 9075-2:202x, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

ISO/IEC 10646, *Information technology — Universal Multi-Octet Coded Character Set (UCS)*

ISO/IEC 14651:2019, *Information technology — International string ordering and comparison — Method for comparing character strings and description of the common template tailorable ordering*

POSIX, ISO/IEC/IEEE 9945, *Information technology — Portable Operating System Interface (POSIX®) Base Specifications*

Berners-Lee, T., Fielding, R., Masinter, L.. *Uniform Resource Identifier (URI): Generic Syntax* [online]. Wilmington, Delaware, USA: Network Working Group, January 2005 . Available at <http://www.ietf.org/rfc/rfc3986.txt>

Duerst, M., Suignard, M.. *Internationalized Resource Identifiers (IRI)* [online]. Wilmington, Delaware, USA: Network Working Group, January 2005 . Available at <http://www.ietf.org/rfc/rfc3987.txt>

IANA. *Time Zone Database* [online]. Los Angeles, California, USA: Internet Assigned Numbers Authority, The Time Zone Database (often called tz or zoneinfo) contains code and data that represent the history of local time for many representative locations around the globe. It is updated periodically to reflect changes made by political bodies to time zone boundaries, UTC offsets, and daylight-saving rules. Available at <https://www.iana.org/time-zones>

Kuhn, Markus. *Coordinated Universal Time with Smoothed Leap Seconds (UTC-SLS)* [online]. University of Cambridge: IETF, January 2006 . Available at <https://tools.ietf.org/html/draft-kuhn-leapsecond-00>

The Unicode Consortium. *The Unicode Standard (Information about the latest version of the Unicode standard can be found by using the "Latest Version" link on the "Enumerated Versions of The Unicode Standard" page.)* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <http://www.unicode.org/versions/enumeratedversions.html>

The Unicode Consortium. *Unicode Collation Algorithm* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <http://www.unicode.org/reports/tr10/>

The Unicode Consortium. *Unicode Normalization Forms* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <http://www.unicode.org/reports/tr15/>

The Unicode Consortium. *Unicode Identifier and Pattern Syntax* [online]. Mountain View, California, USA: The Unicode Consortium, Available at <http://www.unicode.org/reports/tr31/>

van Kesteren, A.. *URL Living Standard* [online]. [Place of publication unknown]: WHATWG, Available at <https://url.spec.whatwg.org>

3 Terms and definitions

3.1 Introduction to terms and definitions

For the purposes of this document, the terms and definitions given in ISO 8601-1:2019, ISO 8601-2:2019, ISO/IEC 14651:2019 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- IEC Electropedia: available at <http://www.electropedia.org/>
- ISO Online browsing platform: available at <http://www.iso.org/obp>

In this document, the definition of a verb defines every voice, mood, and tense of that verb.

3.2 General terms and definitions

3.2.1

atomic

incapable of being subdivided

Note 1 to entry: The antonym is *composite* (3.2.2).

3.2.2

composite

comprising distinguishable elements

Note 1 to entry: The antonym is *atomic* (3.2.1).

3.2.3

identify

to reference without ambiguity

3.2.4

identifier

value (3.8.1) by which something is identified

3.2.5

object

<“X” object> *thing* of some type “X” that is separately identifiable, has a definition, and possibly, contents

Note 1 to entry: The contents of objects is mutable unless explicitly defined otherwise.

Note 2 to entry: Objects are not *values* (3.8.1).

3.2.6

descriptor

<“X” descriptor> coded description of a *GQL-object* (3.4.20) that defines the *metadata* of a *GQL-object* (3.4.20) of a specified type

Note 1 to entry: A descriptor includes all information about its GQL-object that is required by a conforming implementation.

Note 2 to entry: See Subclause 5.3.4, “Descriptors”.

3.2.7

persistent

continuing to exist indefinitely, until destroyed deliberately

3.2.8

dictionary

<“X” dictionary> mapping defined by a collection of key-value pairs

Note 1 to entry: Different pairs never have the same key.

3.2.9

multiset

unordered collection of elements that are not necessarily distinguishable

3.2.10

set

unordered collection of distinguishable elements

3.2.11

sequence

ordered collection of elements that are not necessarily distinguishable

3.2.12

ordered set

ordered collection of distinguishable elements

3.2.13

temporary

lasting for only a limited period of time

3.2.14

variable

<“X” variable> *identifier* (3.2.4) assigned to a *site* (3.7.5)

Note 1 to entry: A variable represents a site.

3.3 Graph terms and definitions

3.3.1

graph

property graph

data *object* (3.2.5) comprising zero or more *labels* (3.3.20), zero or more *properties* (3.3.21), zero or more *nodes* (3.3.10) and zero or more *edges* (3.3.11)

Note 1 to entry: In the context of working with property graphs, the term graph is commonly used in the real world as a shorthand for property graph. Without judgment or prejudice, this document prefers the term graph in descriptive text. The term property graph is only used to establish the right context in introductory text.

3.3.2

multigraph

graph (3.3.1) that allows more than one *edge* (3.3.11) connecting two *nodes* (3.3.10)

3.3.3

directed graph

graph (3.3.1) in which every *edge* (3.3.11) is directed

3.3.4**undirected graph**

graph (3.3.1) in which every *edge* (3.3.11) is an *undirected edge* (3.3.14)

3.3.5**mixed graph**

graph (3.3.1) that allows both *directed edges* (3.3.13) and *undirected edges* (3.3.14)

3.3.6**empty graph**

graph (3.3.1) with zero *nodes* (3.3.10) and zero *edges* (3.3.11)

3.3.7**path**

non-empty *sequence* (3.2.11) of *graph elements* (3.3.9)

Note 1 to entry: A path always starts and ends with a *node* (3.3.10) and, alternates between nodes and *edges* (3.3.11) such that each edge resides in the path between its *endpoints* (3.3.16).

Note 2 to entry: A path may comprise a single node.

Note 3 to entry: A node or an edge may be contained multiple times in a path, including via self-loops.

3.3.8**subpath**

path (3.3.7) fully contained in another path

Note 1 to entry: A subpath may be identical to its containing path.

Note 2 to entry: A subpath cannot be longer than its containing path.

3.3.9**graph element**

node (3.3.10) or *edge* (3.3.11)

3.3.10**node****vertex**

fundamental unit of which a *graph* (3.3.1) is formed

Note 1 to entry: Plural: nodes or vertices.

Note 2 to entry: A node has zero or more *labels* (3.3.20) and zero or more *properties* (3.3.21).

Note 3 to entry: Both terms, node and vertex, are used in the real world to mean the same. Without judgment or prejudice, this document uses only the term node in descriptive text. In BNF productions, wherever the keyword NODE is allowed, the keyword VERTEX can be used instead.

3.3.11**edge****relationship**

connection between two *nodes* (3.3.10)

Note 1 to entry: Both terms, edge and relationship, are used in the real world to denote the same concept. Without judgment or prejudice, this document uses only the term edge in descriptive text. In BNF productions, wherever the keyword EDGE is allowed, the keyword RELATIONSHIP can be used instead.

3.3.12**directionality**

<edge> information regarding whether an *edge* (3.3.11) is a *directed edge* (3.3.13) or an *undirected edge* (3.3.14)

3.3.13

directed edge

edge (3.3.11) that distinguishes one of its *endpoints* (3.3.16) as its *source node* (3.3.18) and one of its endpoints as its *destination node* (3.3.19)

Note 1 to entry: A directed edge expresses a relationship that is asymmetric.

Note 2 to entry: The antonym is *undirected edge* (3.3.14).

3.3.14

undirected edge

edge (3.3.11) that does not distinguish between its *endpoints* (3.3.16)

Note 1 to entry: An undirected edge expresses a relationship that is necessarily symmetric.

Note 2 to entry: The antonym is *directed edge* (3.3.13).

3.3.15

any-directed edge

either a *directed edge* (3.3.13) or an *undirected edge* (3.3.14)

3.3.16

endpoint

incident node

<edge> one of the two *nodes* (3.3.10) connected by an *edge* (3.3.11)

Note 1 to entry: Both endpoints of an edge may be the same node.

3.3.17

adjacent

connected by at least one *edge* (3.3.11)

3.3.18

source node

start node

node (3.3.10) that is distinguished as the source of a *directed edge* (3.3.13)

3.3.19

destination node

end node

node (3.3.10) that is distinguished as the destination of a *directed edge* (3.3.13)

3.3.20

label

identifier (3.2.4) associated with a *graph* (3.3.1), a *node* (3.3.10), or an *edge* (3.3.11)

3.3.21

property

pair comprising a property name and a *property value* (3.8.8)

Note 1 to entry: The value of a property is a value of its *property type* (3.8.9).

3.4 GQL-environment terms and definitions

3.4.1

GQL-environment

milieu in which the *GQL-catalog* (3.5.1) and *GQL-data* (3.5.4) exists and *GQL-requests* (3.4.13) are executed

Note 1 to entry: See Subclause 4.3, “GQL-environments and their components”.

3.4 GQL-environment terms and definitions

3.4.2

GQL-server

processor capable of executing a *GQL-request* (3.4.13) that was submitted by a *GQL-client* (3.4.4) and delivering the outcome of that *execution* (3.6.7) back to the GQL-client in accordance with the rules and definitions of the GQL language

Note 1 to entry: See Subclause 4.3.3.3, “GQL-servers”.

3.4.3

principal

object (3.2.5) that represents a user within a GQL-implementation

Note 1 to entry: See Subclause 4.3.4.1, “Principals”.

3.4.4

GQL-client

processor capable of establishing a connection to a *GQL-server* (3.4.2) on behalf of a *GQL-agent* (3.4.5) that is authenticated to represent a *principal* (3.4.3)

Note 1 to entry: See Subclause 4.3.3.2, “GQL-clients”.

3.4.5

GQL-agent

independent process that causes the *execution* (3.6.7) of *procedures* (3.6.28) and *commands* (3.6.40)

Note 1 to entry: See Subclause 4.3.2, “GQL-agents”.

3.4.6

GQL-session

period in which consecutive *GQL-requests* (3.4.13) are executed by a *GQL-client* (3.4.4) on behalf of a *GQL-agent* (3.4.5)

Note 1 to entry: See Subclause 4.6, “GQL-sessions”.

3.4.7

session context

context associated with a *GQL-session* (3.4.6) in which multiple *GQL-requests* (3.4.13) are executed consecutively

3.4.8

current session context

session context (3.4.7) associated with the *GQL-session* (3.4.6) of the currently executing *GQL-request* (3.4.13)

3.4.9

session parameter

context parameter (3.4.19) defined in a *session context* (3.4.7)

3.4.10

session parameter flag

flag controlling whether a *session parameter* (3.4.9) may be overridden by another *request parameter* (3.4.15) with the same name or modified by a later *GQL-request* (3.4.13) in the same *GQL-session* (3.4.6)

3.4.11

session-derived

<GQL-object> defined to include references to *session parameters* (3.4.9)

3.4.12**GQL-transaction**

logical unit of independent *atomic* (3.2.1) *execution* (3.6.7)

Note 1 to entry: See Subclause 4.7, "GQL-transactions".

3.4.13**GQL-request**

request source (3.4.14) and *request parameters* (3.4.15)

Note 1 to entry: See Subclause 4.8, "GQL-requests".

3.4.14**request source**

character string (3.8.21) containing the source text of a GQL-program

3.4.15**request parameter**

context parameter (3.4.19) defined in a *GQL-request* (3.4.13)

3.4.16**GQL-request context**

context augmenting a *session context* (3.4.7) in which an individual *GQL-request* (3.4.13) is executed

Note 1 to entry: See Subclause 4.8.2, "GQL-request contexts".

3.4.17**current request context**

GQL-request context (3.4.16) associated with the currently executing *GQL-request* (3.4.13)

3.4.18**execution stack**

push-down stack of *execution contexts* (3.6.9) associated with a *GQL-request* (3.4.13)

3.4.19**context parameter**

pair comprising a parameter name and a parameter value

3.4.20**GQL-object**

object (3.2.5) capable of being manipulated directly by the *execution* (3.6.7) of a *GQL-request* (3.4.13)

Note 1 to entry: See Subclause 4.4, "GQL-objects".

3.4.21**data object**

GQL-object (3.4.20) comprising data

3.4.22**primary object**

independently definable *GQL-object* (3.4.20)

Note 1 to entry: A primary object may be defined separately from, but may also be contained as a component of another object.

3.4.23**secondary object**

GQL-object (3.4.20) that is an inseparable component of another *GQL-object* (3.4.20)

3.4 GQL-environment terms and definitions**3.4.24****static object type***object type* (3.9.15) comprising *static objects* (3.4.25)**3.4.25****static object***GQL-object* (3.4.20) assignable to a *static site* (3.4.26)**3.4.26****static site***site* (3.7.5) only holding *static* (3.4.27) instances determined at request submission time**3.4.27****static**

always determined at request submission time

Note 1 to entry: The antonym is *dynamic* (3.4.31).**3.4.28****dynamic object type***data type* (3.9.2) comprising *dynamic objects* (3.4.29)**3.4.29****dynamic object***GQL-object* (3.4.20) assignable to a *dynamic site* (3.4.30)**3.4.30****dynamic site***site* (3.7.5) possibly holding *dynamic* (3.4.31) instances**3.4.31****dynamic**possibly only determined at *execution* (3.6.7) timeNote 1 to entry: The antonym is *static* (3.4.27).**3.4.32****boxed value***GQL-object* (3.4.20) comprising a single non-reference *value* (3.8.1)**3.4.33****original**

<GQL-object> defined independently

Note 1 to entry: The antonym is *derived* (3.4.34).**3.4.34****derived**<GQL-object> defined by a computation from other *objects* (3.2.5)Note 1 to entry: The antonym is *original* (3.4.33).**3.5 GQL-catalog terms and definitions****3.5.1****GQL-catalog***persistent* (3.2.7) hierarchically organized collection of *GQL-directories* (3.5.3) and *GQL-schemas* (3.5.5)

Note 1 to entry: See Subclause 4.3.5, “GQL-catalog”.

3.5.2**GQL-catalog root**

GQL-directory (3.5.3) with an empty name or a *GQL-schema* (3.5.5) with an empty name that is the root of the *GQL-catalog* (3.5.1)

Note 1 to entry: The GQL-catalog root indirectly contains every catalog object.

3.5.3**GQL-directory**

persistent (3.2.7) *dictionary* (3.2.8) of *GQL-directories* (3.5.3) and *GQL-schemas* (3.5.5)

Note 1 to entry: See Subclause 4.3.5.2, “GQL-directories”.

3.5.4**GQL-data**

data described by *GQL-schemas* (3.5.5) in the *GQL-catalog* (3.5.1) — data that is under the control of a GQL-implementation in a *GQL-environment* (3.4.1)

Note 1 to entry: See Subclause 4.3.6, “GQL-data”.

3.5.5**GQL-schema**

persistent (3.2.7) *dictionary* (3.2.8) of *primary catalog objects* (3.5.12)

Note 1 to entry: See Subclause 4.3.5.3, “GQL-schemas”.

3.5.6**library**

catalog object (3.5.11) comprising a *dictionary* (3.2.8) of catalog objects

3.5.7**alias**

catalog object (3.5.11) comprising a *reference* (3.7.6) to another catalog object

3.5.8**type object**

catalog object (3.5.11) reifying the existence of a *data type* (3.9.2)

3.5.9**constant object**

catalog object (3.5.11) comprising a single *direct* (3.8.2) value

3.5.10**catalog-derived**

<GQL-object> *derived* (3.4.34) from *catalog objects* (3.5.11)

3.5.11**catalog object**

GQL-object (3.4.20) directly or indirectly defined in the *GQL-catalog* (3.5.1)

3.5.12**primary catalog object**

GQL-object (3.4.20) that is both a *primary object* (3.4.22) and a *catalog object* (3.5.11)

3.5.13**visible**

<GQL-object> capable of being referenced according to effective access control rules

3.5.14**home schema**

default *GQL-schema* (3.5.5) associated with a *principal* (3.4.3)

3.5.15**home graph**

default *graph* (3.3.1) associated with a *principal* (3.4.3)

3.6 Procedure terms and definitions

3.6.1**side effect**

change caused during the *execution* (3.6.7) of a *GQL-request* (3.4.13) that is detectable by the execution of another *operation* (3.6.8) as part of the execution of the same or another GQL-request

3.6.2**catalog-modifying**

<side effect> modifying the *GQL-catalog* (3.5.1)

3.6.3**session-modifying**

<side effect> modifying the *GQL-session* (3.4.6) and its context

3.6.4**transaction-modifying**

<side effect> manipulating *GQL-transactions* (3.4.12)

3.6.5**data-populating**

<side effect> initially populating the contents of newly created *data objects* (3.4.21) using *data-modifying* (3.6.6) *operations* (3.6.8)

3.6.6**data-modifying**

<side effect> modifying the contents of *data objects* (3.4.21)

3.6.7**execution**

computation of a result that may cause *side effects* (3.6.1)

3.6.8**operation**

identifiable action carried out by *execution* (3.6.7)

3.6.9**execution context**

context comprising a *dictionary* (3.2.8) of objects that is associated with and manipulated by the *execution* (3.6.7) of *operations* (3.6.8)

Note 1 to entry: See Subclause 4.9, "Execution contexts".

3.6.10**current execution context**

execution context (3.6.9) of the currently executing *operation* (3.6.8) of the currently executing *procedure* (3.6.28) or *command* (3.6.40)

3.6.11**execution outcome**

component of an *execution context* (3.6.9) representing the outcome of an *execution* (3.6.7)

Note 1 to entry: See Subclause 4.9.4, "Execution outcomes".

3.6.12**current execution outcome**

execution outcome (3.6.11) of the *current execution context* (3.6.10)

3.6.13**successful outcome**

<execution context> *execution outcome* (3.6.11) containing a result determined or unchanged by the successful and complete *execution* (3.6.7) of the last *operation* (3.6.8) executed in its containing *execution context* (3.6.9)

3.6.14**failed outcome**

<execution context> *execution outcome* (3.6.11) containing no result and representing a failure caused or not recovered by the last *operation* (3.6.8) executed in its containing *execution context* (3.6.9)

3.6.15**supported result**

valid result of a *successful outcome* (3.6.13)

3.6.16**regular result**

result that is a *value* (3.8.1) produced by the successful *execution* (3.6.7) of an *operation* (3.6.8)

3.6.17**omitted result**

result indicating the successful *execution* (3.6.7) of an *operation* (3.6.8) that produced no *value* (3.8.1)

3.6.18**result type**

data type (3.9.2) of a result

3.6.19**result object**

object (3.2.5) identified by a result

3.6.20**result object type**

object type (3.9.15) of a *result object* (3.6.19)

3.6.21**current execution result**

result of the *current execution outcome* (3.6.12)

Note 1 to entry: See Subclause 4.9.4, "Execution outcomes".

3.6.22**evaluation**

computation of a result that is not permitted to cause *side effects* (3.6.1)

3.6.23**locally-defined**

<GQL-object> defined internally during *execution* (3.6.7) of an *operation* (3.6.8)

3.6 Procedure terms and definitions

3.6.24

locally-derived

<GQL-object> not *catalog-derived* (3.5.10) and not *session-derived* (3.4.11)

3.6.25

catalog-modifying procedure

procedure (3.6.28) whose *execution* (3.6.7) may perform *catalog-modifying* (3.6.2) *side effects* (3.6.1) and *data-populating* (3.6.5) side effects only

3.6.26

data-modifying procedure

procedure (3.6.28) whose *execution* (3.6.7) is not permitted to perform *catalog-modifying* (3.6.2) *side effects* (3.6.1), *session-modifying* (3.6.3) side effects, or *transaction-modifying* (3.6.4) side effects

3.6.27

stored procedure

persistent (3.2.7) *procedure* (3.6.28) in the *GQL-catalog* (3.5.1)

3.6.28

procedure

description of a computation on input arguments whose *execution* (3.6.7) computes an *execution outcome* (3.6.11) and optionally causes *side effects* (3.6.1)

Note 1 to entry: See Subclause 4.11.2, "Procedures".

3.6.29

procedure signature

<*procedure*> *declaration* (3.7.1) of the list of mandatory *procedure* (3.6.28) parameters required, optional *procedure* (3.6.28) parameters allowed together with default values to be used when they are not given, the kinds of *side effects* (3.6.1) possibly performed, and the *procedure* (3.6.28) result type of the result returned by a complete and successful *execution* (3.6.7) of the *procedure* (3.6.28)

Note 1 to entry: See Subclause 4.11.2.1, "General description of procedures".

3.6.30

procedure logic

<*procedure*> operations (such as statements) together with the order in which they have to be effectively performed to completely execute the algorithm that is implemented by the *procedure* (3.6.28)

3.6.31

formal parameter

(formal parameter name, formal parameter type, parameter cardinality) triple describing an input argument of a *procedure* (3.6.28) call

3.6.32

parameter cardinality

<formal parameter> specification of the number of different parameter values that may be provided for the *formal parameter* (3.6.31) by the input *binding table* (3.8.15) in a *procedure* (3.6.28) call

3.6.33

single parameter cardinality

parameter cardinality (3.6.32) of one

Note 1 to entry: A formal parameter with single parameter cardinality declares a *fixed variable* (3.8.12) in the scope of the procedure.

3.6.34

multiple parameter cardinality

parameter cardinality (3.6.32) of zero, one, or more

Note 1 to entry: A formal parameter with multiple parameter cardinality declares an *iterated variable* (3.8.13) in the scope of the procedure.

3.6.35

simple view

view (3.6.37) that expects no arguments

3.6.36

parameterized view

view (3.6.37) that expects one or more arguments

3.6.37

view

query (3.6.38) that returns a *graph* (3.3.1)

3.6.38

query

procedure (3.6.28) whose *execution* (3.6.7) may perform *data-populating* (3.6.5) *side effects* (3.6.1) only

3.6.39

function

query (3.6.38) whose *execution* (3.6.7) will not access the *GQL-catalog* (3.5.1) or the current *GQL-session* (3.4.6) in any way

3.6.40

command

operation (3.6.8) executing independently and absent a currently executing *procedure* (3.6.28) whose *execution* (3.6.7) computes an *execution outcome* (3.6.11) and may cause *side effects* (3.6.1)

Note 1 to entry: See Subclause 4.11.3, "Commands".

3.6.41

session command

command (3.6.40) that may only perform *session-modifying* (3.6.3) *side effects* (3.6.1)

3.6.42

transaction command

command (3.6.40) that may only perform *transaction-modifying* (3.6.4) *side effects* (3.6.1)

3.6.43

working schema

GQL-schema (3.5.5) that is implicitly accessed or manipulated by the *execution* (3.6.7) of a *procedure* (3.6.28)

3.6.44

working graph

graph (3.3.1) that is implicitly accessed or manipulated by the *execution* (3.6.7) of a *procedure* (3.6.28)

3.6.45

working table

binding table (3.8.15) that is implicitly transformed by the *execution* (3.6.7) of a *procedure* (3.6.28)

3.6.46

working record

record (3.8.16) that is implicitly accessed or manipulated by the *execution* (3.6.7) of a *procedure* (3.6.28)

3.6 Procedure terms and definitions

3.6.47

GQL-procedure

procedure (3.6.28) defined in the GQL language

3.6.48

external procedure

procedure (3.6.28) provided via an implementation-defined mechanism

3.6.49

statement

operation (3.6.8) executed as part of executing a *procedure* (3.6.28) that updates the *current execution context* (3.6.10) and its *current execution outcome* (3.6.12) and that may cause *side effects* (3.6.1)

3.6.50

successful result statement

statement (3.6.49) producing a *successful outcome* (3.6.13)

3.6.51

successful statement

statement (3.6.49) not producing a *failed outcome* (3.6.14)

3.6.52

failed statement

statement (3.6.49) producing a *failed outcome* (3.6.14)

3.7 Procedure syntax terms and definitions

3.7.1

declaration

<“X” declaration> syntax that declares an “X”

3.7.2

definition

<“X” definition> syntax that defines an “X”

3.7.3

scope

one or more BNF non-terminal symbols within which a name introduced by a *declaration* (3.7.1), a *definition*, or implied context is effective

3.7.4

instance

<“X” instance> physical representation of an “X”

Note 1 to entry: Each instance is at exactly one *site* (3.7.5). An “X” instance has a type that is the type of “X”.

3.7.5

site

place occupied by an *instance* (3.7.4) of a specified type (or subtype of that type)

3.7.6

reference

<“X” reference> *instance* (3.7.4) that identifies a *site* (3.7.5) containing an “X”

3.7.7**literal**

<reference> specified literally

Note 1 to entry: The antonym is *opaque* (3.7.8).

3.7.8**opaque**

<reference> specified indirectly

Note 1 to entry: The antonym is *literal* (3.7.7).

3.7.9**referent**

<“X” referent> *instance* (3.7.4) identified by an “X” *reference* (3.7.6)

3.7.10**pattern**

syntax for abstractly specifying a collection of *pattern matches* (3.7.19)

3.7.11**pattern macro**

named template providing syntactic abstraction for a *pattern* (3.7.10)

3.7.12**pattern variable**

<“X” pattern variable> “X” *variable* (3.2.14) that is declared in an “X” *pattern* (3.7.10)

3.7.13**edge variable**

element variable (3.7.14) that is declared in an *edge pattern* (3.7.24)

Note 1 to entry: An edge variable may be bound to an *edge* (3.3.11).

3.7.14**element variable**

variable (3.2.14) that may be bound to a *graph element* (3.3.9)

3.7.15**graph pattern variable**

path variable (3.7.17), *subpath variable* (3.7.18) or *element variable* (3.7.14)

3.7.16**node variable**

element variable (3.7.14) that is declared in a *node pattern* (3.7.23)

Note 1 to entry: A node variable may be bound to a *node* (3.3.10).

3.7.17**path variable**

identifier (3.2.4) assigned to a *path* (3.3.7) that is matched by a *path pattern* (3.7.21)

3.7.18**subpath variable**

variable (3.2.14) that may be bound to a *subpath* (3.3.8) of a *path* (3.3.7) in a *graph* (3.3.1)

3.7.19**pattern match**

dictionary (3.2.8) of *variable* (3.2.14) bindings whose entries are related according to some specified *pattern* (3.7.10)

3.7.20**graph pattern***set* (3.2.10) of one or more *path patterns* (3.7.21)**3.7.21****path pattern**pattern that matches a *path* (3.3.7)**3.7.22****element pattern***node pattern* (3.7.23) or *edge pattern* (3.7.24)**3.7.23****node pattern***path pattern* (3.7.21) that matches a single *node* (3.3.10)**3.7.24****edge pattern***path pattern* (3.7.21) that matches a single *edge* (3.3.11)**3.7.25****label expression**expression composed from *label* (3.3.20) names using disjunction, conjunction, and negation

Note 1 to entry: Disjunction, conjunction, and negation are denoted respectively by a vertical bar "|", ampersand "&" and exclamation mark "!", with parentheses for grouping.

3.7.26**linear composition**composition of a *sequence* (3.2.11) of suboperations that forms a *composite* (3.2.2) *operation* (3.6.8) that effectively executes the suboperations in the order given by the sequence

Note 1 to entry: The linear composition of operations over binding tables corresponds to a left lateral join between those tables.

3.7.27**whitespace***sequence* (3.2.11) of one or more characters that have the Unicode property White_Space

Note 1 to entry: Whitespace is typically used to separate <non-delimiter token>s from one another, and is always permitted between two tokens in text written in the GQL language.

3.8 Value terms and definitions

3.8.1**value**

<"X" value> definite, immutable, and irreducible unit of data of some type "X"

Note 1 to entry: Values stand for themselves. The identity of a value is independent of where it occurs.

Note 2 to entry: Values are not *objects* (3.2.5).**3.8.2****direct**<value> not a *reference value* (3.8.4) and only containing *direct* (3.8.2) valuesNote 1 to entry: The antonym is *indirect* (3.8.3).

3.8.3

indirect

<value> *reference value* (3.8.4) or containing *indirect* (3.8.3) values

Note 1 to entry: The antonym is *direct* (3.8.2).

Note 2 to entry: Reference values are indirect values.

3.8.4

reference value

value (3.8.1) representing a *reference* (3.7.6)

3.8.5

material

<value> not the *null value* (3.8.6)

3.8.6

null value

special *value* (3.8.1) that is used to indicate the absence of other data

Note 1 to entry: The null value of Boolean type and the truth value *Unknown* are the same value. The null values of different data types are indistinguishable.

3.8.7

unit value

value (3.8.1) of the *record type* (3.9.24) that has zero *fields* (3.8.17)

3.8.8

property value

element of the *property value type* (3.9.20)

3.8.9

property type

<node type or edge type definition> pair comprising a property name and a *property value type* (3.9.20)

Note 1 to entry: A *value* (3.8.1) of a *property* (3.3.21) is a value of the property value type of its property type.

3.8.10

local variable

static (3.4.27) *variable* (3.2.14) or *fixed variable* (3.8.12)

3.8.11

binding variable

dynamic (3.4.31) *variable* (3.2.14) corresponding to the *field* (3.8.17) of a *working record* (3.6.46) in the current *execution stack* (3.4.18) at request *execution* (3.6.7) time

Note 1 to entry: A binding variable may be declared as a *fixed variable* (3.8.12) in some scope and by extension in every child scope but in the absence of such a declaration is implicitly assumed to be declared as an *iterated variable* (3.8.13) in those scopes unless explicitly specified otherwise.

3.8.12

fixed variable

<in the scope of a procedure> *binding variable* (3.8.11) that is always bound to the same *value* (3.8.1) when iterating over the current *working table* (3.6.45) during *procedure* (3.6.28) *execution* (3.6.7)

Note 1 to entry: This determination does not consider the concrete records of the working table during execution but is made by the application of Syntax Rules only.

3.8 Value terms and definitions

3.8.13

iterated variable

<in the scope of a procedure> *binding variable* (3.8.11) that may be bound to different values when iterating over the current *working table* (3.6.45) during *procedure* (3.6.28) *execution* (3.6.7)

Note 1 to entry: This determination does not consider the concrete records of the working table during execution but is made by the application of Syntax Rules only.

3.8.14

expression variable

binding variable (3.8.11) introduced by an expression for evaluating an inner expression

3.8.15

binding table

GQL-object (3.4.20) comprising a collection of zero or more records of the same *record type* (3.9.24)

Note 1 to entry: See Subclause 4.4.5, “Binding tables” and in Subclause 4.16, “Binding table types”.

3.8.16

record

value (3.8.1) of a *record type* (3.9.24); possibly empty *set* (3.2.10) of *fields* (3.8.17)

3.8.17

field

pair comprising a field name and a field value

3.8.18

empty binding table

binding table (3.8.15) of zero records

Note 1 to entry: In this document, the record type of a new empty binding table has zero fields by default unless otherwise specified.

3.8.19

unit binding table

binding table (3.8.15) of one *record* (3.8.16) whose *record type* (3.9.24) has zero *fields* (3.8.17)

3.8.20

byte string

an element of the byte string type

3.8.21

character string

an element of the character string type

3.9 Type terms and definitions

3.9.1

declared type

<site or a non-terminal specifying an operation> unique *data type* (3.9.2) common to every *instance* (3.7.4) that may be assigned to a given *site* (3.7.5) or that may result from *execution* (3.6.7) or *evaluation* (3.6.22) of the *operation* (3.6.8) specified by a given non-terminal

Note 1 to entry: Declared types may be optional, i.e., not every site or every non-terminal that specifies an operation has a declared type or at least their declared type is not necessarily fully known prior to execution.

3.9.2

data type

set (3.2.10) of elements with shared characteristics representing data

Note 1 to entry: A data type characterizes the sites that may be occupied by instances of its elements.

3.9.3

supertype

<“X” supertype> data type containing all elements of some data type “X”

Note 1 to entry: The antonym is *subtype (3.9.4)*.

3.9.4

subtype

<“X” subtype> data type comprising only elements contained in some data type “X”

Note 1 to entry: The antonym is *supertype (3.9.3)*.

3.9.5

meta type

set of data types of the same kind

3.9.6

base type

single associated *meta type (3.9.5)* of a *data type (3.9.2)*

3.9.7

most specific type

<of “X”> smallest data type that includes “X”

Note 1 to entry: If a data type *A* comprises fewer elements than a data type *B*, then *A* is smaller than *B*.

3.9.8

constructed

<data type> comprising *composite (3.2.2)* elements

3.9.9

user-defined

<data type> *constructed (3.9.8)* and explicitly defined by the user

3.9.10

predefined

<data type> *atomic (3.9.11)* and provided by the GQL-implementation

3.9.11

atomic

<data type> comprising *atomic (3.2.1)* elements only

3.9.12

material

<data type> excluding the *null value (3.8.6)*

Note 1 to entry: The antonym is *nullable (3.9.13)*.

3.9.13

nullable

<data type> including the *null value (3.8.6)*

Note 1 to entry: The antonym is *material (3.9.12)*.

3.9 Type terms and definitions

3.9.14

any type

data type (3.9.2) comprising every element from every *object type* (3.9.15) and *value type* (3.9.18)

3.9.15

object type

data type (3.9.2) comprising *objects* (3.2.5)

3.9.16

graph type

object type (3.9.15) describing a *graph* (3.3.1) in terms of restrictions on its labels, properties, nodes, edges, and topology

Note 1 to entry: See Subclause 4.15, "Graph types".

3.9.17

binding table type

object type (3.9.15) of every *binding table* (3.8.15) of a specified *record type* (3.9.24)

Note 1 to entry: See Subclause 4.16, "Binding table types".

3.9.18

value type

data type (3.9.2) comprising *values* (3.8.1)

3.9.19

reference value type

value type (3.9.18) comprising *reference values* (3.8.4)

3.9.20

property value type

value type (3.9.18) comprising *property values* (3.8.8)

3.9.21

column name

field (3.8.17) name of a *column* (3.9.23)

3.9.22

column type

field (3.8.17) *value type* (3.9.18) of a *column* (3.9.23)

3.9.23

column

field (3.8.17) of the *record type* (3.9.24) of the records of a *binding table* (3.8.15)

3.9.24

record type

value type (3.9.18) that describes the *set* (3.2.10) of *fields* (3.8.17) of a *record* (3.8.16) in terms of their *field type* (3.9.25)

3.9.25

field type

pair comprising a *field* (3.8.17) name and a *field* (3.8.17) *value type* (3.9.18)

3.9.26

nothing type

empty *data type* (3.9.2)

Note 1 to entry: The nothing type has no instances.

3.10 Temporal terms and definitions

3.10.1

time

[SOURCE: ISO 8601-1:2019, 3.1.1.2]

3.10.2

instant

[SOURCE: ISO 8601-1:2019, 3.1.1.3]

3.10.3

time scale

[SOURCE: ISO 8601-1:2019, 3.1.1.5]

3.10.4

duration

[SOURCE: ISO 8601-1:2019, 3.1.1.8]

3.10.5

time of day

[SOURCE: ISO 8601-1:2019, 3.1.1.16]

3.10.6

Gregorian calendar

[SOURCE: ISO 8601-1:2019, 3.1.1.19]

3.10.7

time shift

[SOURCE: ISO 8601-1:2019, 3.1.1.25]

3.10.8

calendar date

[SOURCE: ISO 8601-1:2019, 3.1.2.7]

3.10.9

representation with reduced precision

[SOURCE: ISO 8601-1:2019, 3.1.3.7]

3.10.10

negative duration

[SOURCE: ISO 8601-2:2019, 3.1.1.7]

3.11 Definitions taken from ISO/IEC 14651:2019

3.11.1

collation

[SOURCE: ISO/IEC 14651:2019, 3.7]

4 Concepts

4.1 Use of terms

The concepts on which this document is based are described in terms of objects, in the usual sense of the word.

Some objects are a component of an object on which they depend. If an object ceases to exist, then every object dependent on that object also ceases to exist. The representation of an object is known as a descriptor. The descriptor of an object represents everything that needs to be known about the object. See also Subclause 5.3.4, "Descriptors".

4.2 Informative elements

In several places in the body of this document, informative notes appear. For example:

NOTE 1 — This is an example of a note.

Those notes do not belong to the normative part of this document. A claim of conformance to material in a Note is not meaningful.

4.3 GQL-environments and their components

4.3.1 General description of GQL-environments

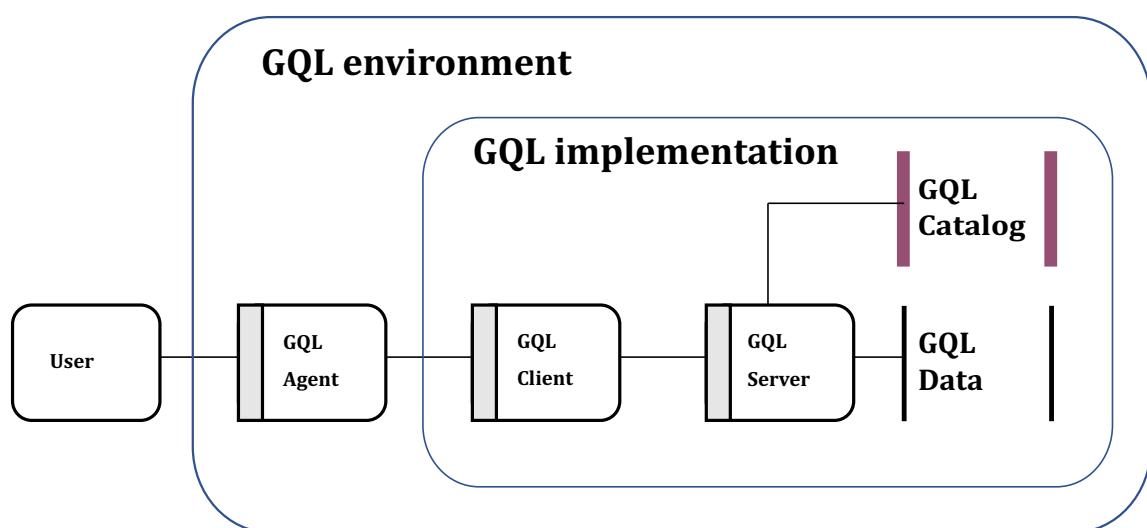


Figure 1 — Components of a GQL-environment

A *GQL-environment* is a milieu in which the *GQL-catalog* and *GQL-data* exists and *GQL-requests* are executed. A pictorial overview is shown in Figure 1, "Components of a GQL-environment". A GQL-environment comprises:

- 1) One GQL-agent.
- 2) One GQL-implementation containing one GQL-client and one GQL-server.

- 3) Zero or more authorization identifiers that identify principals.
- 4) One GQL-catalog that comprises one GQL-catalog root.
- 5) The sites, principally catalog objects, that contain GQL-data, as described by the contents of the GQL-schemas. This data can be thought of as “the database”, but the term is not used in this document, because it has different meanings in the general context.

This document recognizes that an installation of a software that implements GQL can provide multiple GQL-clients, multiple GQL-servers, and multiple GQL-catalogs such that some GQL-agents can use multiple GQL-clients and some GQL-servers can share the same GQL-catalog. In such a scenario, every interaction of one GQL-agent, one GQL-client, one GQL-server, and one GQL-catalog is understood to occur in an isolated GQL-environment. Each such GQL-environment is considered separately in terms of conformance. Any implementation-dependent interaction between multiple such GQL-environments is understood as the activity of additional GQL-agents.

4.3.2 GQL-agents

A *GQL-agent* is that which utilizes an implementation-defined mechanism to instruct a GQL-client to create and destroy GQL-sessions to GQL-servers, and to submit GQL-requests to them, and thus cause the execution of procedures and commands by the GQL-implementation.

4.3.3 GQL-implementations

4.3.3.1 Introduction to GQL-implementations

A *GQL-implementation* is a processor that executes GQL-requests, as required by a GQL-agent. A GQL-implementation, as perceived by a GQL-agent, includes one GQL-client, to which that GQL-agent is bound, and one GQL-server. A GQL-implementation can conform to this document even if it allows more than one GQL-server to exist in a GQL-environment.

Because a GQL-implementation can be specified only in terms of how it manages GQL-sessions and executes GQL-requests, the concept denotes an installed instance of some software (database management system). This document does not distinguish between features of the GQL-implementation that are determined by the software vendor and those determined by the installer.

This document recognizes that there is a possibility that GQL-client and GQL-server software has been obtained from different vendors; it does not specify the method of interaction or communication between GQL-client and GQL-server.

4.3.3.2 GQL-clients

A *GQL-client* is a processor capable of establishing a connection to a *GQL-server* on behalf of a *GQL-agent* that is authenticated to represent a *principal*. A GQL-client is perceived by the GQL-agent as part of the GQL-implementation, that establishes and manages GQL-sessions to GQL-servers, submits GQL-requests via them, and maintains a GQL-status object and other state data relating to interactions between itself, the GQL-agent, and the GQL-server for its current GQL-session.

A GQL-implementation may detect the loss of the connection between the GQL-client and GQL-server during the execution of any statement. When such a connection failure is detected, an exception condition is raised: *transaction rollback — statement completion unknown (40003)*. This exception condition indicates that the results of the actions performed in the GQL-server on behalf of the GQL-client are unknown to the GQL-agent. Similarly, a GQL-implementation may detect the loss of the connection during the execution of a *<commit command>*. When such a connection failure is detected, an exception condition is raised: *connection exception — transaction resolution unknown (08007)*. This exception condition indicates that the GQL-implementation cannot verify whether the GQL-transaction was committed successfully, rolled back, or left active.

4.3.3.3 GQL-servers

A *GQL-server* is a processor capable of executing a *GQL-request* that was submitted by a *GQL-client* and delivering the outcome of that *execution* back to the *GQL-client* in accordance with the rules and definitions of the *GQL* language. A *GQL-server* is perceived by the *GQL-agent* as part of the *GQL-implementation*, that manages the *GQL-catalog* and *GQL-data*.

Each *GQL-server*:

- Manages *GQL-sessions* taking place between itself and *GQL-clients* on behalf of *GQL-agents*.
- Executes *GQL-requests* received from *GQL-clients* to completion and delivers execution outcomes back to *GQL-clients* as required.
- Maintains the state of the *GQL-session*, including the authorization identifier, the *GQL-transaction*, and certain session defaults.

4.3.4 Basic Security Model

4.3.4.1 Principles

A *principal* is an implementation-defined *object* that represents a user within a *GQL-implementation*.

A *principal* is identified by one or more *authorization identifiers*. The means of creating and destroying *authorization identifiers*, and their mapping to *principals*, is implementation-defined.

NOTE 2 — Allowing multiple *authorization identifiers* to map to a single *principal* allows a user to operate with different sets of privileges.

Every *principal* is associated with a *home schema* and a *home graph*. The manner in which this association is specified is implementation-defined.

The *current home schema* is the *home schema* of the *current principal*.

The *current home graph* is the *home graph* of the *current principal*.

NOTE 3 — In cases where it is not desired to provide a *home schema* for a *principal*, but where instead *principals* are required to explicitly select their *GQL-schema* at the start of a *GQL-session*, a *GQL-implementation* may use an empty schema, containing nothing and that no *principal* has rights to modify, as the *home schema* for such a *principal*.

4.3.4.2 Authorization identifiers

An *authorization identifier* identifies a *principal* and a set of privileges for that *principal*.

The set of privileges identified by an *authorization identifier* are implementation-defined.

4.3.5 GQL-catalog

4.3.5.1 General description of the GQL-catalog

The *GQL-catalog* is a *persistent* hierarchically organized collection of *GQL-directories* and *GQL-schemas*.

The *GQL-catalog* is pictorially represented in [Figure 2, “Components of a GQL-catalog”](#).

The *GQL-catalog* comprises the *GQL-catalog root* that is a *GQL-directory* with an empty name or a *GQL-schema* with an empty name that is the root of the *GQL-catalog*. If the *GQL-catalog root* is a *GQL-directory*, it is called the *root directory*. Otherwise the *GQL-catalog root* is a *GQL-schema* that is called the *root schema*.

4.3 GQL-environments and their components

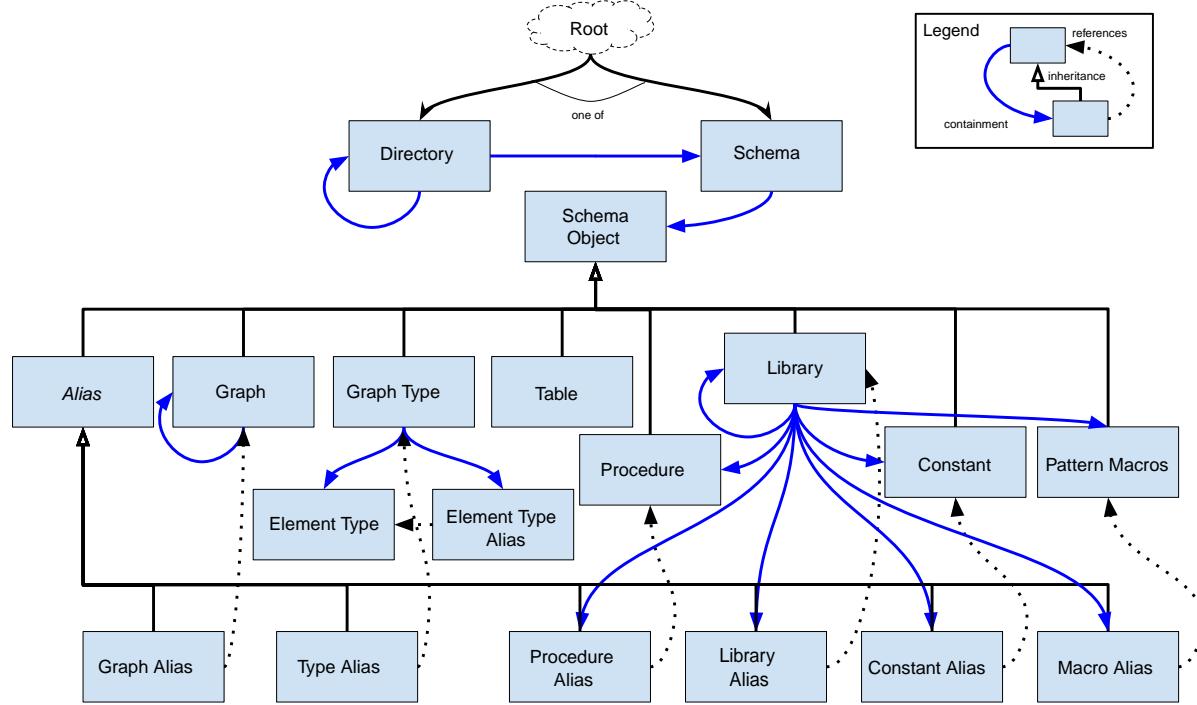


Figure 2 — Components of a GQL-catalog

The nesting structure of GQL-directories and GQL-schemas in the GQL-catalog is implementation-defined.

A GQL-implementation may pre-populate the GQL-catalog with GQL-directories and GQL-schemas during its installation.

NOTE 4 — This supports implementations that provide a GQL-catalog that has only a single GQL-schema as its GQL-catalog root or that has a single GQL-directory that only contains GQL-schemas as its GQL-catalog root or that has a nesting structure that directly matches that of SQL such that its GQL-catalog root is a GQL-directory of SQL-catalogs (represented as GQL-directories) containing GQL-schemas, or any other suitable restriction of the nesting structure.

NOTE 5 — GQL-implementations may provide a pre-configured system principal (a virtual user) that cannot be impersonated by a GQL-client. GQL-implementations that only support a single GQL-schema as the GQL-catalog root and may decide to grant ownership of that GQL-schema to such a system principal but grant full access to the catalog objects in the GQL-schema to regular database administrators.

4.3.5.2 GQL-directories

A *GQL-directory* is a persistent dictionary of GQL-directories and GQL-schemas.

A GQL-directory that directly contains a GQL-directory or GQL-schema *DOS* is called the *parent directory* of *DOS*. The GQL-catalog root has no parent directory.

Every GQL-directory is completely described by its persistent *GQL-directory descriptor* that comprises:

4.3 GQL-environments and their components

- The name of the GQL-directory, an identifier or the empty name. The name of a GQL-directory shall be non-empty and uniquely identifies it within its parent directory unless the GQL-directory is the root directory.

— The descriptors of every GQL-directory and GQL-schema contained in the GQL-directory.

A GQL-directory shall not contain both a GQL-directory and a GQL-schema that have the same name.

A GQL-implementation may automatically create a GQL-directory in an implementation-defined way.

NOTE 6 — For instance, a GQL-implementation may (recursively) create GQL-directories when a GQL-schema is created in a thus far not existing GQL-directory.

4.3.5.3 GQL-schemas

A *GQL-schema* is a [persistent dictionary](#) of [primary catalog objects](#). Every GQL-schema is completely described by its persistent GQL-schema descriptor that comprises:

- The name of the GQL-schema, an identifier or the empty name. The name of a GQL-schema shall be non-empty and uniquely identifies it within its parent directory unless the GQL-schema is the root schema.
- The owner of the GQL-schema, an authorization identifier of a principal.
- The catalog object descriptors of every catalog object directly contained in this GQL-schema.

Some GQL-schemas may be provided by the GQL-implementation and can neither be created nor dropped by the execution of a GQL-request.

A *catalog object* is a [GQL-object](#) directly or indirectly defined in the [GQL-catalog](#). Every catalog object is created directly or indirectly in the context of a GQL-schema, and owned by that GQL-schema. This GQL-schema is called *owning schema* of the catalog object. A catalog object that is directly contained in a GQL-schema is owned by that GQL-schema. A catalog object that is directly contained in another catalog object is owned by the owning schema of its containing catalog object.

Every catalog object is described by a persistent *catalog object descriptor* that has a name that is an identifier that uniquely identifies the catalog object within its containing GQL-schema or container. A catalog object descriptor is one of:

- A named graph descriptor.
- A named graph type descriptor.
- A named procedure descriptor.
- A named library descriptor.
- A named alias descriptor.
- A named constant object descriptor.

NOTE 7 — A GQL-schema cannot contain a GQL-directory or a GQL-schema since they are not catalog objects.

NOTE 8 — In this document, only GQL-schemas, libraries and graphs may contain other catalog objects.

NOTE 9 — Implementations may add support for additional catalog object descriptors.

A GQL-implementation may automatically populate a GQL-schema upon its creation in an implementation-defined way.

4.3.6 GQL-data

GQL-data is data described by [GQL-schemas](#) in the [GQL-catalog](#) — data that is under the control of a GQL-implementation in a [GQL-environment](#).

4.4 GQL-objects

4.4.1 General information about GQL-objects

A *GQL-object* is an [object](#) capable of being manipulated directly by the [execution](#) of a [GQL-request](#). GQL-objects are defined by their respective contents, which are either original (defined independently) or derived (defined by a computation from other [objects](#)).

Every GQL-object is identified by one or more effectively immutable *internal object identifiers* that are unique within the GQL-Environment containing the GQL-object. These internal object identifiers of a GQL-object are determined in an implementation-dependent way when the GQL-object is created. Any two internal object identifiers are considered equivalent if they identify the same GQL-object. Any two separately created GQL-objects shall effectively never be identified by the same internal object identifier during a GQL-session. An internal object identifier of a GQL-object is never exposed directly to the user in this document.

Additionally, a GQL-object may have associated descriptors that define its metadata. This document defines different kinds of GQL-objects. The descriptor of a persistent *data object* describes a catalog object that has a separate, though dependent, existence as GQL-data. Other descriptors describe GQL-objects that have no existence distinct from their descriptors (at least as far as this document is concerned). Hence there is no loss of precision if, for example, the term “path pattern” is used when “path pattern descriptor” would be more strictly correct.

NOTE 10 — Implementations may define additional kinds of GQL-objects in extension to those defined by this document.

Every GQL-object is either a *primary* GQL-object or it is a *secondary* GQL-object.

A *primary object* is an independently definable GQL-object. A primary object always has a descriptor. Procedures or commands may define new primary objects in the GQL-catalog, assign them to *session parameters* in a GQL-session, or bind them to local variables, subject to syntax restrictions. Primary objects may also be passed as request parameters or in procedure arguments. Furthermore, primary objects may be returned as the result of an execution outcome or generally occur as the result of expression evaluation if they are valid result objects, as specified by [Subclause 4.9.4, “Execution outcomes”](#). Procedures and commands interact with the following kinds of primary objects:

- GQL-directories, as defined in [Subclause 4.3.5.2, “GQL-directories”](#).
- GQL-schemas, as defined in [Subclause 4.3.5.3, “GQL-schemas”](#).
- Graphs, as defined in [Subclause 4.4.3, “Graphs”](#).
- Type objects (e.g., representing graph types, as defined in [Subclause 4.15, “Graph types”](#)).

NOTE 11 — Type objects are primary objects that reify the existence of a type in the type system that has the same name as the type object.

- Procedures, queries, and functions, as described by [Subclause 4.11.2, “Procedures”](#).
- Binding tables, as described by [Subclause 4.4.5, “Binding tables”](#).
- Libraries, as described by [Subclause 4.4.6, “Libraries”](#).
- Aliases, as described by [Subclause 4.4.7, “Aliases”](#).
- Constant objects representing boxed values, as described by [Subclause 4.4.8, “Constant objects”](#).

NOTE 12 — A primary object that is defined in a GQL-schema is also a catalog object.

A *secondary object* is a GQL-object that is an inseparable component of another GQL-object. A procedure or command may create, modify, delete, or otherwise interact with secondary objects as long as the primary object that contains them has not been deleted. Secondary objects may be set as session parameters, passed as request parameters or in procedure parameter arguments, bound as local variables,

generally occur as the result of expression evaluation, or may be returned in the result of an execution outcome. Procedures and commands interact with the following kinds of secondary objects:

- Nodes.
- Edges.
- Node types.
- Edge types.

NOTE 13 — A primary object P may be referenced as a component of another GQL-object R . If R is deleted, P keeps existing unless explicitly deleted. P keeps existing in possibly an invalid state — a situation this document has to handle by explicit means. In contrast, a secondary object that is a component of a GQL-object C is implicitly deleted when C is deleted.

4.4.2 Static and dynamic GQL-objects

A *static object* is a [GQL-object](#) assignable to a [static site](#). A *static object type* is an [object type](#) comprising [static objects](#). A *static site* is a [site](#) only holding [static](#) instances determined at request submission time. Static objects are never part of valid results returned by a procedure. Procedures or commands may define new static objects in the GQL-catalog, assign them to *session parameters* in a GQL-session, or bind them to local stable variables, subject to syntactic restrictions. Furthermore, static objects may be passed as request parameters. Procedures and commands interact with the following kinds of static objects:

- GQL-directories, as defined in [Subclause 4.3.5.2, “GQL-directories”](#).
- GQL-schemas, as defined in [Subclause 4.3.5.3, “GQL-schemas”](#).
- Libraries, as described by [Subclause 4.4.6, “Libraries”](#).
- Type objects (e.g., representing graph types, as defined in [Subclause 4.15, “Graph types”](#)).

NOTE 14 — Type objects are primary objects that reify the existence of a type in the type system that has the same name as the type object.

- Procedures, queries, and functions, as described by [Subclause 4.11.2, “Procedures”](#).
- Constant objects representing boxed values, as described by [Subclause 4.4.8, “Constant objects”](#).

A *dynamic object* is a [GQL-object](#) assignable to a [dynamic site](#). A *dynamic object type* is an [data type](#) comprising [dynamic objects](#). A *dynamic site* is a [site](#) possibly holding [dynamic](#) instances. Dynamic objects are never part of valid results returned by a procedure. Procedures or commands may define new dynamic objects in the GQL-catalog, assign them to *session parameters* in a GQL-session, or bind them to local variables, subject to syntactic restrictions. Furthermore, dynamic objects may be passed as request parameters and procedure arguments. Procedures and commands interact with the following kinds of dynamic objects:

- Graphs, as defined in [Subclause 4.4.3, “Graphs”](#).
- Binding tables, as described by [Subclause 4.4.5, “Binding tables”](#).
- Boxed values, as described by [Subclause 4.5.2, “Boxed values”](#).

Aliases exist in the GQL-catalog and shall only be manipulated by procedures via catalog-modifying statements. Any other use of aliases implicitly resolves the alias target.

4.4.3 Graphs

4.4.3.1 Introduction to graphs

Graphs are the primary form of data that is queried and manipulated by *procedures*.

In the GQL language, every graph is a *property graph*. Therefore both terms can be used interchangeably, although the shorter term, graph, is preferred in this document.

A graph is both a mixed graph and a multigraph and comprises:

- A graph label set that comprises a set of zero or more labels. A label has a name that is an identifier that is unique within the graph.

The maximum cardinality of a graph label set is implementation-defined.

Without Feature GA00, “Graph label set support”, the graph label set of a conforming graph shall be empty.

- A graph property set that comprises a set of zero or more properties. Each property comprises:

- Its name, which is an identifier that is unique within the graph.

NOTE 15 — The names of graph labels and of graph properties are in separate namespaces.

- Its value, which shall be of any property value type.

The maximum cardinality of a graph property set is implementation-defined.

Without Feature GA01, “Graph property set support”, the graph property set of a conforming graph shall be empty.

- A set of zero or more nodes. Each node comprises:

- An identifier that is unique within the graph.

NOTE 16 — The value of a node identifier is implementation-dependent and is possibly not accessible to the user. This unique identifier is used for definitional purposes to establish the identity of the node.

- A node label set that comprises a set of zero or more labels. A label has a name, which is an identifier that is unique within the node.

The maximum cardinality of node label sets is implementation-defined.

Without Feature GA02, “Empty node label set support”, a node in a conforming graph shall contain a non-empty node label set.

Without Feature GA03, “Singleton node label set support”, a node in a conforming graph shall not contain a singleton node label set.

Without Feature GA04, “Unbounded node label set support”, a node in a conforming graph shall not contain a node label set that comprises one or more labels.

- A node property set that comprises zero or more properties. Each property comprises:

- Its name, which is an identifier that is unique within the node.

NOTE 17 — The names of node labels and of node properties are in separate namespaces.

- Its value, which can be of any property value type.

The maximum cardinality of node property sets is implementation-defined.

- A set of zero or more edges. Each edge comprises:

- An identifier that is unique within the graph.

NOTE 18 — The value of an edge identifier is implementation-dependent and is possibly not accessible to the user. This unique identifier is used for definitional purposes to establish the identity of the edge.

- An edge label set that comprises a set of zero or more labels. A label has a name, which is an identifier that is unique within the edge.

The maximum cardinality of edge label sets is implementation-defined.

Without Feature GA05, “Empty edge label set support”, an edge in a conforming graph shall contain a non-empty edge label set.

Without Feature GA06, “Singleton edge label set support”, an edge in a conforming graph shall not contain a singleton edge label set.

Without Feature GA07, “Unbounded edge label set support”, an edge in a conforming graph shall not contain an edge label set that comprises one or more labels.

- An edge property set that comprises zero or more properties. Each property comprises:
 - Its name, which is an identifier that is unique within the edge.

NOTE 19 — The names of edge labels and of edge properties are in separate namespaces.

- Its value, which can be of any property value type.

The maximum cardinality of edge property sets is implementation-defined.

- Two (possibly identical) endpoints, which are nodes contained in the same graph.
- An indication whether the edge is a directed edge or an undirected edge.

Additionally, a directed edge identifies one of its endpoints as its source, and the other as its destination. The direction of a directed edge is from its source to its destination.

Without Feature G001, “Undirected edge patterns”, a conforming graph shall not contain undirected edges.

NOTE 20 — A graph may be catalog object that has a name, but graphs may exist that have no name, or at least have no name that is required to be visible to a user of a GQL-implementation.

4.4.3.2 Graph descriptors

A graph is described by a *graph descriptor* that comprises:

- The name of the graph type descriptor.
- The descriptors of every named subgraph contained in the graph.

A *named graph descriptor* is both a catalog object descriptor and a graph descriptor and comprises every component of both kinds of descriptors.

4.4.3.3 Subgraphs

A subgraph SG of a graph G is any graph such that

- The graph label set of SG includes the graph label set of G .
- The graph property set of SG includes the graph property set of G .
- Every node of SG is a node of G .
- Every edge of SG is an edge of G .

A named subgraph NSG of a graph G is any subgraph of G such that:

- NSG is a named subgraph directly contained in G .

NOTE 21 — See [Subclause 4.4.3, “Graphs”](#) for the definition of subgraph.

- The graph type of the named graph descriptor of NSG is a subtype of the graph type of the graph descriptor of G .

NOTE 22 — See Subclause 4.15, “Graph types” for the definition of subtyping between graph types.

A named subgraph shall either be explicitly defined and named, or induced by the element types of the graph. The name of a subgraph induced from an element type is the name of that element type. The graph label set of an induced named subgraph is the graph label set of its containing graph. The graph property type set of an induced named subgraph is the graph property type set of its containing graph.

4.4.4 Procedure definitions

4.4.5 Binding tables

A binding table is a **GQL-object** comprising a collection of zero or more records of the same **record type**. As a graph query language, GQL does not use binding tables to hold persistent data. Instead, binding tables mainly serve as

- primary iteration construct that drives the execution of procedures,
- container that holds intermediary results produced by statements such as the matches found by graph pattern matching, and as
- result table that is returned to the GQL-agent.

Every binding table has an associated binding table descriptor that comprises:

- The declared record type of the binding table.

NOTE 23 — Binding table records may contain reference values to primary or secondary objects.

- An indication whether the binding table is declared as *ordered* or as *unordered*. If a binding table is ordered, the order of its records has been determined according to some <sort specification>.
- An indication whether the binding table is declared as *duplicate-free* or as *allowing duplicates*. A duplicate-free binding table shall not contain duplicates of the same record, that is it either contains some arbitrary record R of its declared record type exactly once or it does not contain R at all.
- An optional preferred column sequence that is a permutation of the column names of the binding table.

NOTE 24 — The preferred column sequence is metadata tracked by this document to allow implementations to provide a column sequence for a binding table that is returned as a result table to the GQL-agent.

In this document, a column is a **field** of the **record type** of the records of a **binding table**, a column name is a **field name** of a **column**, and a column type is a **field value type** of a **column**. Furthermore, the records of the collection of records of a binding table are simply referred to as the records of the binding table.

Let $TABLE$ be an ordered binding table with n records and let $record_i$ be the i -th record of $TABLE$, for $1 \leq i \leq n$. The *record ordinal* of any such $record_i$ of $TABLE$ is i and the record index of $record_i$ of $TABLE$ is $i-1$, for $1 \leq i \leq n$.

The *canonical column sequence* of a binding table BT is defined such that if BT has a preferred column sequence PCS . Otherwise, the canonical column sequence of BT is the sequence of every column name of BT in standard sort order.

In the absence of relevant additional provisions, a new binding table implicitly

- is declared as unordered,
- is declared as allowing duplicates, and
- does not have a preferred column sequence.

Furthermore, if new binding table *NEW_TABLE* is obtained only by selecting a subset of the records of a given binding table *TABLE* that is explicitly declared as duplicate-free, then *NEW_TABLE* is implicitly declared as duplicate-free.

If a General Rule refers to the *i*-th record *RECORD* of an ordered binding table *TABLE*, then *RECORD* is the *i*-th record in the sequence of all records of *TABLE* in the order determined by *TABLE*.

If a General Rule *RULE* refers to the *i*-th record *RECORD* of an unordered binding table *TABLE*, then *RECORD* is the *i*-th record in some sequence of all records of *TABLE* in an implementation-dependent order determined for each application of *RULE* and any of its sub rules.

If the application of a General Rule *RULE* processes all records of a binding table using order-independent terms, then the records are to be processed effectively in a sequential, implementation-dependent order determined for that application of *RULE* and any of its sub rules.

The collection of records comprising a binding table is not modified after the initial construction and population of that binding table.

The *collection value* for a binding table is defined depending on whether the binding table is declared as ordered or as duplicate-free:

- The collection value for a binding table declared as ordered and as allowing duplicates is the regular list of every record of the binding table.
- The collection value for of a binding table declared as ordered and as duplicate-free is the distinct list of every record of the binding table.
- The collection value for a binding table declared as unordered and as duplicate-free is the set of every record of the binding table.
- The collection value for a binding table declared as unordered and as allowing duplicates is the multiset of every record of the binding table.

The *Cartesian product* between a record *R* and a binding table *T* such that no field name of a field of *R* is the column name of a column of *T* is a new binding table *NT* comprising a collection of new records obtained by constructing a new record *NR* for every record *TR* of *T* such that the fields of *NR* are the fields of both *TR* and *R*. If *T* is declared as duplicate-free, then *NT* is declared as duplicate-free. Otherwise, *NT* is declared as allowing duplicates.

Furthermore, the Cartesian product between two binding tables *T1* and *T2* is a new binding table *T3* comprising a collection of every record from the Cartesian products between each record of *T1* with *T2*. *T3* is declared as duplicate-free if both *T1* and *T2* are declared as duplicate-free.

4.4.6 Libraries

A *library* is a [catalog object](#) comprising a [dictionary](#) of catalog objects. Libraries group catalog objects and provide qualified names for them. In this document, libraries are assumed to contain sublibraries, procedures, constant objects, or aliases to any of these catalog objects only.

A library may have a default procedure that is invoked when the library is invoked as if it was a procedure.

Every library is completely described by its named library descriptor. Every named library descriptor is a catalog object descriptor. In addition to the components of a catalog object descriptor, a named library descriptor comprises:

- The optional *library default procedure name* that is the name of a procedure that is contained in the library and that is called the *library procedure object*.
- The catalog object descriptors of every catalog object contained in the library.

4.4.7 Aliases

An *alias* is a [catalog object](#) comprising a [reference](#) to another catalog object. The aliased object of an alias is also called the *alias target*. The kinds of objects an implementation may alias is implementation-defined. *Alias resolution* determines the alias target of an alias. Resolving an alias may fail; whether this is a possibility for a given alias depends on the kind of alias.

There are two kinds of aliases. Aliases may be either *hard links* that are alternative names to the same primary object, or *symbolic links* that are forwarding pointers to another qualified name that is resolved through the catalog.

Every alias is completely described by its named alias descriptor. Every named alias descriptor is a catalog object descriptor. In addition to the components of a catalog object descriptor, a named alias descriptor comprises an alias reference that is one of the following

- 1) If the alias is a hard link, then the alias reference is an internal object identifier to the aliased object.
- 2) If the alias is a symbolic link, then the alias reference is either an <absolute url path> or a <relative url path> to the alias target.

4.4.8 Constant objects

A *constant object* is a [catalog object](#) comprising a single [direct](#) value.

Every constant object is completely described by its named constant object descriptor. Every named constant object descriptor is a catalog object descriptor. In addition to the components of a catalog object descriptor, a named constant object descriptor comprises the value of the constant object.

NOTE 27 — Constant objects are boxed values.

4.5 Values

4.5.1 General information about values

In a general sense, values are definite objects. In GQL, values are used to define

- request parameters,
- session parameters,
- procedure arguments,
- static variables,
- binding variables such as fixed variables, iterated variables, and expression variables,
- result values, and
- some characteristics of GQL-objects and their descriptors.

4.5.2 Boxed values

A *boxed value* is a [GQL-object](#) comprising a single non-reference [value](#). Boxed values may have associated descriptors that define the metadata of the boxed value. A boxed value is implicitly interpreted as the value it represents in contexts that expect a value. In every other context, boxed values are treated as the GQL-objects that they are.

4.5.3 Reference values

In some contexts it is necessary to treat GQL-objects as if they were values. In such situations, GQL-objects are implicitly interpreted as reference values that reference the GQL-object they represent using its

internal object identifier unless this is explicitly specified otherwise for the particular GQL-object or type of GQL-objects. In other contexts, it is necessary to treat a reference value as if it were a GQL-object. In such situations, reference values are implicitly interpreted as standing for their referent.

4.5.4 Value classifications

4.5.4.1 Values classified by their computational dependencies

The following are the broad classes of values according to their computational dependencies:

- Direct values describe themselves and have no dependencies.
- Indirect values are described in terms of other objects:
 - Derived values are indirect values that are defined by a computation over GQL-data and that depend on every object that is indirectly referenced by that computation in any form.
 - Reference values are indirect values that represent some implicitly referenced GQL-object and therefore depend on the existence of their referent.

4.5.4.2 Values classified by the kinds of sites in which they occur

The following are broad classes of values according to the kind of the site in which they occur:

- Constants objects are boxed direct values.
- Property values are values that may occur as the values of properties.

4.6 GQL-sessions

4.6.1 General description of GQL-sessions

A *GQL-session* is an implementation-defined period in which consecutive *GQL-requests* are executed by a *GQL-client* on behalf of a *GQL-agent*. At any one time during a GQL-session, exactly one of these consecutive GQL-requests is being executed and is said to be an *executing GQL-request*.

A GQL-session is created either explicitly by the GQL-client, on behalf of a GQL-agent or implicitly whenever a GQL-client, on behalf of a GQL-agent, initiates a request to a GQL-server and no GQL-session is current. The GQL-session is terminated following the last request from that GQL-client, on behalf of that GQL-agent. The mechanism and rules by which a GQL-implementation determines when the last request has been received are implementation-defined.

The context of a GQL-session can be manipulated by session management commands.

An executing GQL-request *ER* causes a nested sequence of consecutive (inner) procedures and commands to be executed as a direct result of *ER*; during that time, exactly one of these is also an executing procedure or an executing command and it in turn may similarly involve execution of a further nested sequence, and so on, indefinitely. An executing procedure or command *EPC* such that no procedure or command is executing as a direct result of *EPC* is called the *innermost executing procedure or command* of the GQL-session. An executing procedure *EP* such that no procedure is executing as a direct result of *EP* is called the *innermost executing procedure* of the GQL-session. An executing command *EC* such that no command is executing as a direct result of *EC* is called the *innermost executing command* of the GQL-session.

Executing each such individual procedure or command *EPC* causes a nested sequence of consecutive (sub)operations such as statements to be executed as part of executing *EPC*; during that time, exactly one of these is also an executing operation and it in turn may similarly involve execution of a further nested sequence of other operations including procedures, and so on, indefinitely. An executing statement *ES* such that no statement is executing as a direct result of *ES* within the same innermost executing procedure or command is called the *innermost executing statement* of the GQL-session. An executing operation

EO such that no operation is executing as a direct result of *EO* within the same innermost executing procedure or command is called the *innermost executing operation* of the GQL-session.

4.6.2 Session contexts

4.6.2.1 Introduction to session contexts

A *session context* is a context associated with a **GQL-session** in which multiple **GQL-requests** are executed consecutively and that comprises the following characteristics:

- 1) The authorization identifier.
- 2) The principal identified by the authorization identifier.
- 3) The time zone identifier.
- 4) The session schema that is a GQL-schema.
- 5) The session graph that is a graph.
- 6) The session parameters that constitute a context parameter dictionary.
- 7) The session parameter flags that constitute a dictionary that associates each session parameter with its parameter flag.
- 8) The current transaction that is an optional GQL-transaction.
- 9) The request context that is an optional GQL-request context.
- 10) The termination flag that is a Boolean value.

NOTE 28 — The termination flag is initially false but may be set during the execution of a GQL-program by the <session close command> to signal that the current session is to be terminated.

The *current session context* is the *session context* associated with the **GQL-session** of the currently executing **GQL-request**.

As a principle, phrases of the form *current x* are used to refer to the *GQL-x-characteristic* or (if non-existing) to the optionally prefixed *x-characteristic* of the current session context. Concretely, exactly the following non-ambiguous forms are used in this document:

- 1) The *current authorization identifier* is the authorization identifier of the current session context.
- 2) The *current principal* is the principal of the current session context.
- 3) The *current time zone identifier* is the time zone identifier of the current session context.
- 4) The *current session schema* is the session schema of the current session context.
- 5) The *current session graph* is the session graph of the current session context.
- 6) The *current session parameters* are the session parameters of the current session context.
- 7) The *current session parameter flags* are the session parameter flags of the current session context.
- 8) The *current transaction* is the GQL-transaction of the current session context.
- 9) The *current request context* is the request context of the current session context.
- 10) The *current termination flag* is the termination flag of the current session context.

4.6.2.2 Session context creation

A new session context is created and initialized by setting each of its characteristics explicitly.

4.6.2.3 Session context modification

The characteristics of a session context shall only be modified after their initialization as follows:

- Setting the time zone identifier.
- Setting the session schema.
- Setting the session graph.
- Setting the session parameters.
- Setting the current transaction to the active GQL-transaction associated with the GQL-session immediately after that GQL-transaction is initialized.
- Setting the current transaction to no transaction immediately after the active GQL-transaction associated with the GQL-session is terminated.
- Setting the request context upon starting or terminating the execution of a GQL-request.
- Setting the termination flag to signal that the GQL-session is about to be terminated.

4.7 GQL-transactions

4.7.1 General description of GQL-transactions

A *GQL-transaction* (transaction) is a sequence of executions of procedures that is atomic with respect to recovery. That is to say: either the complete execution of each procedure results in a successful outcome, or there is no effect on any objects in the GQL-catalog or on GQL-data.

At any time, there is at most one current transaction between the GQL-agent and the GQL-server.

For the purposes of this specification there is only one current transaction, which is the behavior of a system that only supports serializable transactions.

Statements within a procedure effectively execute serially. The state of GQL-data and the GQL-catalog, as affected by a successfully-completed statement, are visible to a successor statement.

Any relaxation of the assumption of the serializable transactional behavior (for example, less restrictive isolation levels) is an implementation-defined extension.

A GQL-implementation may define different kinds of GQL-transactions to be initiated for the execution of a catalog-modifying procedure, a data-modifying procedure, and a query.

It is implementation-defined whether or not the execution of a data-modifying statement is permitted to occur within the same GQL-transaction as the execution of a catalog-modifying statement. If permitted, there may be additional implementation-defined restrictions, requirements, and conditions. If any such restrictions, requirements, or conditions are violated, then an exception condition or a completion condition warning is raised with an implementation-defined class and/or subclass code.

4.7.2 Transaction demarcation

Starting and terminating of transactions may be accomplished by implementation-defined means (including proprietary agent-server protocols, transaction managers that act out of band to agent-server interactions, auto-starting, and auto-rollback on exception of transactions) but any GQL-implementation shall make available the following explicit transaction demarcation commands, to be issued by agents in GQL-requests to a GQL-server.

A GQL-transaction can be initiated by a GQL-agent submitting a GQL-request that specifies a start transaction command. If no GQL-transaction has been initiated the commencement of execution of a procedure will initiate a GQL-transaction.

The start transaction command allows the specification of the characteristics of the transaction, such as the mode (read only, or read write). If a start transaction command is submitted when there is already an active GQL-transaction then an exception condition is raised: *invalid transaction state — active GQL-transaction (25G01)*.

Every GQL-transaction is terminated by an attempt to either commit or a rollback. A successful rollback causes the transaction to have no effect on the GQL-catalog or GQL-data; a successful commit causes the execution outcome to be completely successful.

The GQL-agent that initiated an agent transaction may request the GQL-implementation to either commit or rollback that transaction by submitting a GQL-request that specifies a transaction commit command or a transaction rollback command to the GQL-server.

The transaction demarcation commands may be submitted in the same GQL-request as a procedure. A GQL-request may state a start transaction command prior to the procedure, allowing specifying characteristics of the transaction started for that procedure. A GQL-request may state a transaction commit command or a transaction rollback command following the procedure. A GQL-request stating a transaction commit command following a procedure causes a request to the GQL-server to terminate the active GQL-transaction by committing upon the successful completion of that procedure. A GQL-request stating a transaction rollback command following a procedure causes a request to the GQL-server to terminate the active GQL-transaction by rolling back upon the successful completion of that procedure.

A GQL-implementation may force the rollback of any GQL-transaction if it becomes blocked, or cannot complete without causing semantic inconsistency, or if the resources required to continue its execution become unavailable.

Any failure within a GQL-request procedure will cause an attempt by the GQL-implementation to rollback the current transaction.

Once a request to commit or rollback has been issued then no further statements will be executed in the sequence of statements within the sequence of procedures executions that make up the transaction, and no further request to terminate (commit or rollback) will be processed.

Once requested, transaction termination shall succeed or fail. If a termination request cannot be processed successfully then the state of the GQL-catalog and GQL-data becomes indeterminate. The ways in which termination success or failure statuses are made available to the GQL-agent or to an administrator is implementation-defined.

4.7.3 Transaction isolation

Provisional (uncommitted) changes to the GQL-catalog or GQL-data state, that are made in the context of a transaction executing a catalog-modifying procedure or a data-modifying procedure, may be visible — at some point or to some degree — to other GQL-agents executing a concurrent transaction. The levels of transaction isolation, their interactions, their granularity of application, and the means of selecting them are implementation-defined.

NOTE 29 — GQL-implementations are therefore free to use the isolation levels defined in SQL, or more recently-defined levels such as Snapshot Isolation or Serializable Snapshot Isolation, or other industrially-applied or theoretical variants.

4.7.4 Encompassing transaction belonging to an external agent

In some environments (e.g., remote database access), a GQL-transaction can be part of an encompassing transaction that is controlled by an agent other than the GQL-agent.

In such environments, an encompassing transaction shall be terminated via that other agent, which in turn interacts with the GQL-implementation via an interface that may be different from GQL (COMMIT or ROLLBACK), in order to coordinate the orderly termination of the encompassing transaction. If the encompassed GQL-transaction is terminated by an implicitly initiated <rollback command>, then the GQL-implementation will interact with that other agent to terminate that encompassing transaction. The

specification of the interface between such agents and the GQL-implementation is beyond the scope of this document. However, it is important to note that the semantics of a GQL-transaction remain as defined in the following sense:

- When an agent that is different from the GQL-agent requests the GQL-implementation to rollback a GQL-transaction, the General Rules of Subclause 8.4, “[<rollback command>](#)”, are performed.
- When such an agent requests the GQL-implementation to commit a GQL-transaction, the General Rules of Subclause 8.5, “[<commit command>](#)”, are performed. To guarantee orderly termination of the encompassing transaction, this commit operation may be processed in several phases not visible to the application; not all of the General Rules of Subclause 8.5, “[<commit command>](#)”, need to be executed in a single phase.

4.8 GQL-requests

4.8.1 General description of GQL-requests

A GQL-request comprises:

- The request source that is a GQL-program.
- The request parameter set that is effectively a [<request parameter set>](#). It is not expected that many, if any, implementations will support the syntax of [<request parameter set>](#) but every implementation shall provide an implementation-defined mechanism that is functionally equivalent to the [<request parameter set>](#) specification.

4.8.2 GQL-request contexts

4.8.2.1 Introduction to GQL-request contexts

A *GQL-request context* is a context augmenting a [session context](#) in which an individual [GQL-request](#) is executed and that comprises the following characteristics:

- 1) The request preamble that is the explicit or implicit [<preamble>](#).
- 2) The request parameters that constitute a context parameter dictionary.
- 3) The execution stack that is a push-down stack of execution contexts.
- 4) The request outcome that is an execution outcome.

A preamble is a construct that allows for the passing of settings that may influence the way in which the GQL-request is executed e.g., which query planner options to use, which syntactic transformations are to be allowed, which features are required/allowed/disallowed.

If an explicit preamble is not provided then an implementation-defined preamble is assumed.

Some preamble options are standardized but implementation-defined options are explicitly permitted.

The *current request context* is a [GQL-request context](#) associated with the currently executing [GQL-request](#).

As a principle, phrases of the form *current x* are used to refer to the *GQL-x-characteristic* or (if non-existing) to the optionally prefixed *x-characteristic* of the current request context. Concretely, exactly the following non-ambiguous forms are used in this document:

- 1) The *current request preamble* is the request preamble of the current request context.
- 2) The *current request parameters* are the request parameters of the current request context.
- 3) The *current execution stack* is the execution stack of the current request context.

- 4) The *current request outcome* is the request outcome of the current request context.

4.8.2.2 GQL-request context creation

A new GQL-request context is created and initialized by setting each of its characteristics explicitly.

4.8.2.3 GQL-request context modification

The characteristics of a GQL-request context shall only be modified after their initialization as follows:

- Setting the request preamble.
- Setting the request outcome.

4.8.3 Execution stack

The *execution stack* is a push-down stack of **execution contexts** associated with a **GQL-request**. There is one cell on this stack that is initialized with a new execution context when execution of the GQL-request begins. This execution context is always the lowest execution context in the execution stack.

An additional execution context is pushed on the stack for each procedure or command that is executed, and is removed when that procedure or command completes execution. An additional child execution context may be pushed on the stack for the execution of a command or a statement, the evaluation of an expression, or the processing of some other non-terminal that specifies an operation to be executed, and if such an execution context has been created, it is removed when that execution, evaluation, or processing completes.

NOTE 30 — Changes to the execution stack are always explicitly specified by General Rules.

The highest execution context in the execution stack of the currently executing GQL-request is called the *current execution context*.

4.9 Execution contexts

4.9.1 General description of execution contexts

An *execution context* is a context comprising a **dictionary** of objects that is associated with and manipulated by the **execution of operations**. It provides access to visible objects, allowing their manipulation by the execution of a procedure or a command.

An execution context comprises the following characteristics:

- 1) The working schema that is a GQL-schema.
- 2) The working graph that is a graph.
- 3) The working table that is a binding table.
- 4) The working record that is a record.
- 5) The execution outcome that is an execution outcome.

The atomicity of multiple operations that are executed in an execution context depends on the type of the operation.

The *current execution context* is an **execution context** of the currently executing **operation** of the currently executing **procedure** or **command**.

As a principle, phrases of the form *current x* are used to refer to the *GQL-x-characteristic* or (if non-existing) to the optionally prefixed *x-characteristic* of the current execution context. Concretely, exactly the following non-ambiguous forms are used in this document:

4.9 Execution contexts

- 1) The *current working schema* is the working schema of the current execution context.
- 2) The *current working graph* is the working graph of the current execution context.
- 3) The *current working table* is the working table of the current execution context.
- 4) The *current working record* is the working record of the current execution context.
- 5) The *current execution outcome* is the execution outcome of the current execution context.
- 6) Additionally, the *current execution result* is the result of the current execution outcome.

4.9.2 Execution context creation

A new execution context may be created by one of the following approaches:

- 1) Create a new execution context that is initialized by setting each of its characteristics explicitly or to implementation-defined defaults.
- 2) Create a new *child execution context* that is initialized by:
 - a) copying each of its characteristics from the corresponding characteristics of the current execution context, and
 - b) setting its working table to a unit binding table.

The creation of an execution context may be further modified by overriding some of its characteristics using a phrase such as “with ... as its *CHARACTERISTIC*” or similar grammatical variants and by resetting some of its characteristics to implementation-defined defaults using a phrase such as “with the default *CHARACTERISTIC*” or similar grammatical variants.

The execution of an *ACTION* in a new child execution context using a phrase such as “perform *ACTION* in a new child execution context” or similar grammatical variants is defined as follows:

- a) Create a new child execution context *CTX*.
- b) Push *CTX* on the current execution stack.
- c) Perform *ACTION*.
- d) Identify the outcome of *ACTION* with the current execution outcome.
- e) Identify the result of *ACTION* with the current execution result.
- f) Pop the current execution stack and destroy *CTX*.

NOTE 31 — An implementation may choose to implement an instruction to execute an *ACTION* in a new child execution context by instead executing it directly in the current execution context as long as that direct execution is indistinguishable from the execution in a new child execution context that was specified by the instruction.

The evaluation *EVAL* of an element *ELT* in a new child execution context for a record *R* using a phrase such as “evaluate *ELT* in a new child execution context amended with *R*” or similar grammatical variants is defined to effectively perform the following steps in a new child execution context *CHILD*:

- a) Amend the current working record with *R*.
- b) Identify the result of *EVAL* with the result of evaluating *ELT* in *CHILD*.

To amend the current working record with *R*, the current working record is set to a new record that comprises:

- 1) Every field *FO* of the original current working record for which it holds that the field name of every field *FR* in *R* differs from the field name of *FO*.

2) Every field of R .

NOTE 32 — This implements the shadowing of fields in the original current working record by fields in R that have the same field name as a fall-back provision. This document generally aims to prevent this case from arising by other means such as syntax restrictions on variable name declarations.

4.9.3 Execution context modification

The characteristics of an execution context shall only be modified after their initialization as follows:

- Setting the working schema.
- Setting the working graph.
- Setting the working table.
- Setting the working record.
- Setting the execution outcome.

An application A of the General Rules for an element ELT in this document only modifies the current execution context temporarily so that its characteristics are restored to their initial state from the start of A when A finishes unless ELT is one of the root elements defined by a Subclause of:

- Subclause 6.3, “<GQL-program>”,
- Clause 7, “Session management”,
- Clause 8, “Transaction management”,
- Clause 9, “Procedures”,
- Clause 10, “Variable and parameter declarations and definitions”,
- Clause 12, “Statements”,
- Clause 13, “Catalog-modifying statements”,
- Clause 14, “Data-modifying statements”, or
- Clause 15, “Query statements”.

4.9.4 Execution outcomes

An *execution outcome* is a component of an *execution context* representing the outcome of an *execution* and comprises:

- 1) A GQL-status object.
- 2) An optional result that is always a value. Every result is required to be a *supported result*, which is a valid result of a *successful outcome*. Supported results are direct values or indirect values that possibly reference GQL-objects.

NOTE 33 — Dynamically constructed objects are represented by reference values that identify them.

An execution outcome is one of:

- A *successful outcome* that is an *execution outcome* containing a result determined or unchanged by the successful and complete *execution* of the last *operation* executed in its containing *execution context* and whose result shall be further distinguished to be one of:
 - A *regular result* is a result that is a *value* produced by the successful *execution* of an *operation*.
 - An *omitted result* is a result indicating the successful *execution* of an *operation* that produced no *value*.

- A *failed outcome* that is an **execution outcome** containing no result and representing a failure caused or not recovered by the last operation executed in its containing **execution context**.

The GQL-status object of a successful outcome has a GQLSTATUS that is a completion condition and indicates that any previous errors have been recovered and no new error was raised by the last operation executed in the execution context.

The GQL-status object of a failed outcome has a GQLSTATUS that is an exception condition and indicates that a previously raised error was not recovered or that a new error was raised by the last operation executed in the execution context.

4.10 Diagnostics

4.10.1 Introduction to diagnostics

Every GQL-program returns some diagnostic information to the GQL-client that originated the GQL-request of which the GQL-program was a part.

This diagnostic information is contained in a GQL-status object that minimally comprises a condition code but may also include additional diagnostic information.

It is implementation-defined how a GQL-status object is presented to a GQL-client.

NOTE 34 — For example, in a Java environment a GQL-status object with a GQLSTATUS value whose category in “X” may cause an exception to be thrown.

4.10.2 Conditions

There are two types of conditions:

- completion conditions
- exception conditions

A *completion condition* is one that permits a statement to have an effect other than that associated with raising the condition. These correspond to a GQLSTATUS class code of *successful completion (00000)*, *warning (01000)*, or *no data (02000)*.

The completion condition *warning (01000)* is broadly defined as completion in which the effects are correct, but there is reason to caution the user about those effects. It is raised for implementation-defined conditions as well as conditions specified in this document. The completion condition *no data (02000)* has special significance and is used to indicate an empty result. The completion condition *successful completion (00000)* is defined to indicate a completion condition that does not correspond to *warning (01000)* or *no data (02000)*.

If no other completion or exception condition has been specified, then completion condition *successful completion (00000)* is returned. This includes conditions in which the GQLSTATUS subclass provides implementation-defined information of a non-cautionary nature.

An *exception condition* is one that causes a statement to have no effect other than that associated with raising the condition (that is, not a completion condition).

If a GQL-request does not conform to the Format, Syntax Rules and Conformance Rules of <GQL-request>, then an exception condition is raised. Unless a Syntax Rule is violated that specifies the explicit exception condition *syntax error or access rule violation — invalid reference (42002)*, the exception condition *syntax error or access rule violation — invalid syntax (42001)* is raised.

Except where otherwise specified, the phrase “an exception condition is raised:”, followed by the name of a condition, is used in General Rules and elsewhere to indicate that:

- The execution of a statement is unsuccessful.

- The application of the General Rules is terminated, unless explicitly stated otherwise.
- Diagnostic information is to be made available.
- Execution of the statement is to have no effect on GQL-data or the GQL-catalog.

The phrase “a completion condition is raised:”, followed by the name of a condition, is used in General Rules and elsewhere to indicate that application of General Rules is not terminated and diagnostic information is to be made available; unless an exception condition is also raised, the execution of the statement is successful.

If more than one condition could have occurred as a result of a statement, it is implementation-dependent whether diagnostic information pertaining to more than one condition is made available. Those addition conditions, if any, are placed in a separate GQL-status object and chained to the GQL-status object containing the GQLSTATUS with the greatest precedence.

For the purpose of choosing the GQLSTATUS value to be returned:

- Every exception condition for transaction rollback has precedence over every other exception condition.
- Every exception conditions has precedence over every completion conditions.
- The completion condition *no data (02000)* has precedence over the completion condition *warning (01000)*.
- The completion condition *warning (01000)* has precedence over the completion condition *successful completion (00000)*.

The values assigned to GQLSTATUS shall obey these precedence requirements.

4.10.3 GQL-status object

A *GQL-status object* is one in which one or more *conditions* are recorded as they arise.

Whenever a statement is executed, it sets values, representing one or more conditions resulting from that execution, in a GQL-status object. These values give some indication of what has happened. When a GQL-program terminates, the diagnostic information from the current execution outcome is returned to the GQL-client.

Each GQL-status object comprises:

- A GQLSTATUS character string.
- A character string describing the GQLSTATUS character string. This shall be either:
 - The concatenation of the Condition and Subcondition fields of the associated row in [Table 7, “GQLSTATUS class and subclass codes”](#), or
 - An implementation-defined translation of that concatenation appropriate for the locale of the GQL-client.
- A map value with diagnostics information as defined in [Clause 23, “Diagnostics”](#).
- A chain of nested GQL-status objects.
- An optional map of implementation-defined diagnostic information.

NOTE 35 — For example, this may include a stack-trace.

At the beginning of any execution the current GQL-status object is emptied. An implementation places information about a completion condition or an exception condition reported by GQLSTATUS into this

GQL-status object. If other conditions are raised, the extent to which further GQL-status objects are chained is implementation-defined.

4.11 Procedures and commands

4.11.1 General description of procedures and commands

Procedures and commands are executed as part of executing a GQL-request in order to:

- read or modify the GQL-catalog and the catalog objects it contains,
- read or modify GQL-data,
- read or modify session context characteristics such as their session graph, session schema, session parameters, or GQL-transactions,
- read request parameters,
- or otherwise read, modify, or manipulate GQL-objects that are reachable via the current execution context, the current request context, or the current session context.

4.11.2 Procedures

4.11.2.1 General description of procedures

A procedure is a description of a computation on input arguments whose [execution](#) computes an [execution outcome](#) and optionally causes [side effects](#). In this document, a procedure is represented by a primary object that defines the *procedure logic* of the procedure; the procedure logic are operations (such as statements) together with the order in which they have to be effectively performed to completely execute the algorithm that is implemented by the [procedure](#).

These operations may read, modify, or otherwise manipulate the GQL-catalog, the GQL-data, the GQL-session, or other objects reachable via the current execution context, the current request context, or the current session context that are available during their execution.

4.11.2.2 Procedure descriptors

A procedure is described by a *procedure descriptor* that comprises:

- A <schema reference> to the working schema of the procedure.
- A <catalog graph reference> to the working graph of the procedure.
- The procedure signature descriptor of the procedure.
- Zero or more of the types of side effects described in [Subclause 4.11.5.3, “Operations classified by type of side effects”](#) that may be performed when the procedure is executed.

A *named procedure descriptor* is both a catalog object descriptor and a procedure descriptor and comprises every component of both kinds of descriptors.

4.11.2.3 Procedure signatures

The *procedure signature* of a procedure is a [declaration](#) of the list of mandatory [procedure](#) parameters required, optional [procedure](#) parameters allowed together with default values to be used when they are not given, the kinds of [side effects](#) possibly performed, and the [procedure](#) result type of the result returned by a complete and successful [execution](#) of the [procedure](#).

A procedure signature *A* matches a procedure signature *B* if and only if the procedure signature descriptor of *A* matches the procedure signature descriptor of *B*.

4.11.2.4 Procedure signature descriptor

A procedure signature is described by a *procedure signature descriptor* that comprises:

- One type signature descriptor.
- One of the types of computation described in Subclause 4.11.2.7, “Procedures classified by type of computation” specified as follows:
 - CATALOG PROCEDURE specifies that this procedure signature is the signature of a catalog-modifying procedure.
 - PROCEDURE specifies that this procedure signature is the signature of a data-modifying procedure.
 - QUERY specifies that this procedure signature is the signature of a query.
 - FUNCTION specifies that this procedure signature is the signature of a function.

A procedure signature descriptor *A* matches a procedure signature descriptor *B* if and only if the type signature descriptor of *A* matches the type signature descriptor of *B* and the type of computation of *A* is included in the type of computation of *B*.

4.11.2.5 Type signature descriptor

A *type signature descriptor* comprises:

- One list of required mandatory procedure parameter declarations.
- One list of allowed optional procedure parameter definitions.
- One procedure result type.

A type signature descriptor *A* matches a type signature descriptor *B* if and only if the mandatory parameters of *A* match the mandatory parameters of *B*, the optional parameters of *A* match the optional parameters of *B*, and the result type of *A* is subtype of the result type of *B*.

The mandatory parameters of a type signature descriptor *A* match the mandatory parameters of a type signature descriptor *B* if and only if both *A* and *B* expect the same number *NUM* of mandatory parameters and every mandatory parameter of *A* at position *i* is a super type of the mandatory parameter of *B* at position *i* for $1 \leq i \leq NUM$.

The optional parameters of a type signature descriptor *A* match the optional parameters of a type signature descriptor *B* if and only if *A* expects a smaller number *NUM* of optional parameters than *B* and every optional parameter of *A* at position *i* is a super type of the optional parameter of *B* at position *i* for $1 \leq i \leq NUM$.

4.11.2.6 Procedure execution

Procedures are executed by calling them in a child execution context on procedure call arguments that are passed via the working table of that execution context. The provided procedure call arguments need to fulfil the requirements of the procedure signature of the called procedure regarding the cardinality, data type, and optionality of positionally corresponding formal parameters. Every optional formal parameter of the procedure signature of the called procedure that is not included in the provided procedure call arguments is defaulted as specified by the procedure signature of the called procedure.

The parameter cardinality of binding variables is determined syntactically in the scope of the procedure containing the procedure call. A formal parameter of the procedure signature of the called procedure that is specified as requiring single parameter cardinality shall not be provided by the result of an argument expression that references iterated variables introduced prior to the procedure call.

4.11 Procedures and commands

The current working schema, working graph, and other execution context characteristics shall only be modified during the execution of the procedure as specified by its procedure descriptor.

The outcome of execution of a procedure is the execution outcome present in the specified execution context after execution has terminated. The result of execution of a procedure is the result of the execution outcome present in the specified execution context after execution has terminated.

4.11.2.7 Procedures classified by type of computation

Procedures are classified by the type of computation that their execution may perform. For this purpose, this document distinguishes between:

- A procedure that is a description of a computation on input arguments whose **execution** computes an **execution outcome** and optionally causes **side effects**.
- A catalog-modifying procedure that is a **procedure** whose **execution** may perform **catalog-modifying side effects** and **data-populating** side effects only. Every catalog-modifying procedure is a procedure by definition.
- A data-modifying procedure that is a **procedure** whose **execution** is not permitted to perform **catalog-modifying side effects**, **session-modifying** side effects, or **transaction-modifying** side effects.
- A query that is a **procedure** whose **execution** may perform **data-populating side effects** only. Every query is a procedure by definition.
- A function, i.e., a **query** whose **execution** will not access the **GQL-catalog** or the current **GQL-session** in any way. Every function is a query by definition.

Furthermore, a **view** is a **query** that returns a **graph**, a **simple view** is a **view** that expects no arguments, and a **parameterized view** is a **view** that expects one or more arguments.

4.11.2.8 Procedures classified by type of provisioning

Procedures are classified by how they have been provided to the GQL-server. For this purpose, this document distinguishes between:

- A GQL-procedure that is a **procedure** defined in the GQL language that was provided directly by using the GQL language only.
- An external procedure that is a procedure provided via an implementation-defined mechanism.

4.11.3 Commands

4.11.3.1 General description of commands

A **command** is an **operation** executing independently and absent a currently executing **procedure** whose **execution** computes an **execution outcome** and may cause **side effects**.

Every supported command is one of the following:

- A **session command**, i.e., a **command** that may only perform **session-modifying side effects**.
- A **transaction command**, i.e., a **command** that may only perform **transaction-modifying side effects**.

NOTE 36 — Implementations may extend the set of supported commands.

4.11.3.2 Command execution

Commands are executed by invoking them in an execution context.

NOTE 37 — All arguments required by an invocation need to be supplied implicitly via the current execution context.

The outcome of execution of a command is the execution outcome present in the specified execution context after execution has terminated. The result of execution of a command is the result of the execution outcome present in the specified execution context after execution has terminated.

4.11.4 Procedures in GQL

4.11.4.1 Introduction to procedures in GQL

The procedure signature of a procedure that is written in the GQL language is either specified explicitly by providing a <type signature> or inferred implicitly from the <procedure body> when the procedure is defined.

The procedure logic of a procedure that is written in the GQL language is specified by the <procedure body> provided when the procedure is defined.

A <procedure body> comprises:

- 1) A (possibly empty) sequence of local variable definitions that are specified by a <static variable definition block>, or a <binding variable definition block>, or both.
- 2) A sequence of statements that are specified by the <statement block> of the <procedure body>.

The procedure logic of a procedure written in the GQL language is executed in the current execution context by first executing its variable definitions followed by executing its statements. This is specified completely by Subclause 9.4, “<procedure body>”.

4.11.4.2 Variables and parameters

Procedures interact with the following kinds of variables and parameters:

- 1) A context parameter that is a pair comprising a parameter name and a parameter value, which is defined in some context. Context parameters hold supported result values only. A context parameter shall be one of the following:
 - a) A session parameter that is a **context parameter** defined in a **session context**.
 - b) A request parameter that is a **context parameter** defined in a **GQL-request**.
- 2) A static variable. A variable is static if it is always determined at request submission time. Every static variable holds references to GQL-objects or boxed values.
- 3) A binding variable is a **dynamic variable** corresponding to the **field** of a **working record** in the current **execution stack** at request **execution** time. Every binding variable is a dynamic variable that holds a supported result value. A variable is dynamic if it is possibly only determined at **execution** time. A binding variable shall be one of the following:
 - a) A fixed variable that is a **binding variable** that is always bound to the same **value** when iterating over the current **working table** during **procedure execution**.
 - b) An iterated variable that is a **binding variable** that may be bound to different values when iterating over the current **working table** during **procedure execution**.

An expression variable is a **binding variable** introduced by an expression for evaluating an inner expression.

Procedures define variables and parameters as follows:

- 1) Catalog-modifying statements define catalog objects and optionally their named subobjects in the GQL-catalog. Catalog objects may be bound as session parameters, static variables, and binding variables subject to syntactic restrictions depending on the type of the catalog object.

4.11 Procedures and commands

- 2) The *<type signature>* declares every mandatory formal parameter for which arguments need to be provided by a *<named procedure call>*.
- 3) The *<type signature>* declares every optional formal parameter for which arguments may be provided by a *<named procedure call>* and defines the default value of every such optional formal parameter that is to be used in the absence of an explicitly provided argument.
- 4) A local variable is a **static variable** or **fixed variable**. Every static variable defined by a *<static variable definition block>* is defined by a *<static variable definition>*. Every binding variable defined by a *<binding variable definition block>* is defined by a *<binding variable definition>*.
- 5) The *<statement block>* of a procedure defines binding variables.
- 6) A *<value expression>*, *<graph pattern>*, *<simple graph pattern>* may define additional binding variables.

This document permits GQL-implementations to provide additional, implementation-defined types of local variable definitions subject to the following rules:

- 1) Execution of local variable definitions shall always amend the current execution context with additional local variables and shall not change the current execution context in any other way.
- 2) Execution of local variable definitions shall only perform data-reading and data-populating side effects.

4.11.4.3 Statements

A statement defines an **operation** executed as part of executing a **procedure** that updates the **current execution context** and its **current execution outcome** and that may cause **side effects** (e.g., *<match statement>* for pattern matching).

This document permits GQL-implementations to provide additional, implementation-defined statements.

4.11.4.4 Statements classified by function

A statement is classified according to its function as:

- *catalog-modifying statement* that modifies the catalog structure to create, alter, and drop catalog objects and that cause catalog-modifying side effects to this end.
- *data-modifying statement* that performs insert, update, and delete operations on data objects and that cause data-modifying side effects to this end.
- *query statement* that in turn shall be either
 - 1) a *data-reading statement* that reads persistent data, or
 - 2) a *data-transforming statement* that performs filtering, aggregating, and projecting operations.

All statements may cause data-populating side effects during the construction of new data objects.

4.11.4.5 Scope of names

The execution of procedures, statements, and commands interacts with instances such as objects and values stored at various sites. This interaction is specified indirectly by addressing those sites by the names assigned to them by their definition in the GQL-catalog, the current session context, the current request context, the current execution context, or syntactic elements such as local variable definitions or the formal parameter list of a procedure. A name shall only be used in its scope that is one or more BNF non-terminal symbols within which a name introduced by a **declaration**, a **definition**, or implied context is effective. In this document, names are always either *<identifier>*s or *<identifier>*s prefixed with some operator symbol.

There are five main namespaces in the GQL language:

- 1) The *catalog namespace* of the names of every GQL-directory, GQL-schema, catalog object or named component of a catalog object in the GQL-catalog.
- 2) The *parameter namespace* of the names of <parameter>s resolved in order as request parameters and session parameters.
- 3) The *local namespace* comprising
 - <binding variable> names resolved in order as expression variables introduced by the evaluation of a <value expression>, a <graph pattern>, or a <simple graph pattern>, iterated variables introduced by iterating over working tables, fixed variables introduced by local variable definitions, formal parameters bound by a named procedure call, and finally catalog objects of the current working schema that are supported results.
 - <static variable> names resolved in order as static variables introduced by local definitions and catalog objects of the current working schema that are static objects.
- 4) The *label namespace* of the names corresponding to labels of catalog objects.
- 5) The *property namespace* of the names corresponding to the properties of catalog objects and subobjects of catalog objects that are exposed as properties.

Catalog namespace The name of a defined GQL-directory, GQL-schema, and catalog object or named component of a catalog object is valid in the scope of every <catalog object reference>.

Parameter namespace The name of a request parameter is valid in the scope of the containing GQL-request. The name of a session parameter is valid in the scope of a GQL-request that is executed within a session that contains such a parameter and that does not include a request parameter with the same name.

Local namespace Resolution of names in the local namespace is subject to the following rules:

- 1) The name of a <binding variable> introduced by a <value expression> is only valid inside that expression as defined for that particular expression.
 - 2) The name of a <binding variable> introduced by a <graph pattern> or <simple graph pattern> is valid inside the statement that contains the <graph pattern> as defined for that particular statement. The exact scope of graph pattern variables is defined in Subclause 16.6, “<graph pattern>” and in Subclause 16.9, “<simple graph pattern>”.
- NOTE 38 — This includes <binding variable>s defined by <graph pattern>s and <simple graph pattern>s that are implicitly added to the current working table by the execution of a statement.
- 3) The name of a <binding variable> introduced by a statement is valid in the scope of every sibling statement within their containing <linear query statement> or <linear data-modifying statement> until the scope is explicitly closed by a <primitive result statement>.
 - 4) The execution of a <named procedure call> explicitly interrupts the scope of every <binding variable> until after the call has finished.
 - 5) The name of a local variable definition introduced in a <static variable definition block>, or a <binding variable definition block> that is immediately contained in some <procedure body> is valid in the scope of the <statement block> immediately contained in the same <procedure body>.
 - 6) The name of a <binding variable> introduced by a formal parameter list of a procedure definition is valid in the scope of the <procedure body> that specifies the procedure logic of the procedure when the procedure is executed and interrupts the scope of names of catalog objects in the current working schema.

4.11 Procedures and commands

- 7) The name of a catalog object in some schema is valid in the scope of every BNF non-terminal symbol that is specified to be executed with that schema as its working schema until that scope is explicitly interrupted by a formal parameter list, local variable definition, statement, <value expression>, <graph pattern>, or <simple graph pattern> as specified above.

Label namespace Labels are valid in the scope of every BNF non-terminal symbol that is specified to be executed within a context that identifies the GQL-object whose definition implies their possible existence.

Property namespace Properties are valid in the scope of every BNF non-terminal symbol that is specified to be executed within a context that identifies the GQL-object whose definition implies their possible existence.

4.11.5 Operations

4.11.5.1 Introduction to operations

Procedures, commands, statements, and auxiliary elements included in their specification such as <value expression>s determine the operations required for their execution and are implicitly understood to represent them where this is unambiguous in this document. Operations are either atomic or comprise suboperations. This document is only concerned with the effective execution of operations to the degree that any side effect they may cause or any result they may produce can be determined by the GQL-agent through the execution of GQL-requests or by examining the GQL-session and associated objects such as the active GQL-transaction.

NOTE 39 — This intentionally allows for a wide range of conforming implementations of this document.

Operations are classified either by their execution outcome (See Subclause 4.11.5.2, “Operations classified by execution outcome”) or by the type of side effects causes (See Subclause 4.11.5.3, “Operations classified by type of side effects”).

4.11.5.2 Operations classified by execution outcome

Operations such as statements and commands that are performed as part of executing a GQL-request are classified by the type of produced execution outcome as:

- Successful operations that produce or preserve a successful outcome. Such operations are further classified as:
 - Operations that produce or preserve a successful outcome with a material result.
 - Operations that produce or preserve a successful outcome with an omitted result.
- Failed operations that produce a failed outcome.

Execution of an operation will generally either preserve the execution outcome that was determined just prior to the start of its execution as part of executing its containing GQL-request or procedure or will explicitly produce a different execution outcome.

4.11.5.3 Operations classified by type of side effects

Operations such as statements and commands that are performed as part of executing a GQL-request are classified by the type of side effects their execution may cause as having:

- *Session-modifying side effects* that are modifying the GQL-session and its context.
- *Transaction-modifying side effects* that are manipulating GQL-transactions.
- *Catalog-modifying side effects* that are modifying the GQL-catalog.

- *Data-populating side effects* that are initially populating the contents of newly created **data objects** using **data-modifying operations**.
- *Data-modifying side effects* that are modifying the contents of **data objects**.
- No side effects.

Data-populating side effects are further qualified according to whether they apply to catalog objects, session parameters, or locally-defined objects.

Data-modifying side effects are further qualified according to whether they apply to catalog objects, session parameters, or locally-defined objects.

The expression *may cause* or similar grammatical variants is used to indicate that an operation may cause some side effect depending on a condition to be determined during its execution. The expression *always causes* or similar grammatical variants is used to indicate that an operation always cause some side effect during its successful execution. The side effects of an operation are all side effects it may cause. The side effects of a composite operation includes the side effects of its suboperations, if any.

NOTE 40 — Implementations may extend GQL with additional procedures, commands, or statements that may perform additional kinds of side effects that are not considered here.

4.12 Graph pattern matching

4.12.1 Summary

Graph pattern matching is the process of applying a list of special-purpose regular expressions called a <graph pattern> on a pure property graph PG in order to return a set of reduced matches. Each reduced match is a list of path bindings; each path binding is a function that maps the symbols in a word of a regular language to a path of PG . The regular language and related concepts such as path binding and reduced match are specified in [Subclause 22.2, “Machinery for graph pattern matching”](#), and [Subclause 16.6, “<graph pattern>”](#).

A <graph pattern> is a list of <path pattern>s. Each <path pattern> in the list is matched to PG to detect a possibly-empty set of path bindings in PG that correspond to the <path pattern>.

The cross-product of these sets is reduced by equi-natural joins over those global singleton element variables in each <path pattern> that are bound to the same graph element in PG . The remaining tuples are called reduced matches; the set of these reduced matches may be empty, but may not be infinite because of syntactic restrictions to guard against infinite cycling.

The behaviour of the graph pattern matching is defined in this document.

Additional qualifying parameters (predicates, a selective <path search prefix>, and a <path mode>) that restrict the result may be supplied.

If PG contains cycles then a match to a <path pattern> having an unbounded quantifier might return an infinite set of paths: however, this possibility is prevented by Syntax Rules that require the use of selective <path search prefix>s or restrictive <path mode>s (or both) to prevent infinitely-sized result sets.

4.12.2 Paths

A path P is a sequence of n graph elements of a property graph PG , such that:

- n is 0 (zero) or an odd number.
- If $n = 0$ (zero) or $n = 1$ (one), then P has no edges.
- If $n \geq 1$ (one), then the graph element at each odd index is a node and the graph element at each even index is an edge that connects the pair of nodes immediately before and after the edge in the sequence.

4.12 Graph pattern matching

A path contains a (possibly empty) sequence of nodes and a (possibly empty) sequence of edges. If there are two or more nodes, then the path is a sequence of graph elements that starts with a node and is followed by a sequence of ordered (edge, node) pairs.

If PG is a multigraph then an edge in the path between a pair of nodes is one of possibly several edges between those nodes in PG .

Every edge E in the path has an orientation. Let $V1$ be the node that immediately precedes E in the path, and let $V2$ be the node that immediately follows E . If the source of E is $V1$ and the destination of E is $V2$, then the orientation of E is *left to right*; if the source of E is $V2$ and the destination of E is $V1$, then the orientation of E is *right to left*; and if E is undirected, then the orientation of E is *undirected*.

NOTE 41 — A directed self-edge (i.e., when E is directed and $V1$ and $V2$ are the same node), is oriented both left to right and right to left.

If no edge from PG appears more than once in a path, then the path is called a *trail*. If no node from PG appears more than once in a path, except possibly as the first and last nodes of the path, then the path is called *simple*. If no node from PG appears more than once in a path, then the path is called *acyclic*.

NOTE 42 — The term “path” is used in more than one way in mathematical graph theory and in informal technical presentations and discussions. In this document the term path always denotes what a graph-theoretic work might call a partially oriented walk in a pure property graph. Such a graph is a mixed multigraph: edges may be directed or undirected, and there may be multiple edges between two nodes.

4.12.3 Path patterns

A <path pattern> is an expression built from the following syntactic elements, governed by the Format and Syntax Rules of Subclause 16.6, “<graph pattern>”, and other Subclauses:

- An optional <path variable>, separated from the <path pattern prefix> and <path pattern expression> by an <equals operator>.
- An optional <path pattern prefix>.

NOTE 43 — <path pattern prefix> is described in Subclause 4.12.5, “<path mode>”, and Subclause 4.12.6, “Selective <path search prefix>”, and specified in Subclause 16.8, “<path pattern prefix>”.
- A mandatory <path pattern expression>.

A <path pattern expression> is an expression built recursively from <element pattern>s, governed by the Format and Syntax Rules of Subclause 16.7, “<path pattern expression>”, and other Subclauses.

An <element pattern> is a pattern to match a single graph element. There are two kinds of <element pattern>s:

- <node pattern>. A <node pattern> is a pattern to match a single node. A <node pattern> comprises at a minimum a matching pair of parentheses, which may contain optional <element pattern filler>, described subsequently.
- <edge pattern>. An <edge pattern> is a pattern to match a single edge. An <edge pattern> is either a <full edge pattern> (which optionally permits <element pattern filler>) or an <abbreviated edge pattern> (which does not support <element pattern filler>). These two major classes of <edge pattern> have seven variants each, for the seven possible non-empty combinations of the three edge orientations (the individual edge orientations being directed pointing left, undirected, or directed pointing right). Thus there are fourteen varieties of <edge pattern>.

<element pattern filler> provides three optional components of <node pattern>s and <full edge pattern>s:

- <element variable declaration>, to declare an <element variable> to be bound to a graph element by the <element pattern>.
- <label expression>, a predicate regarding the labels of the graph element that is bound by the <element pattern>. A <label expression> is an expression formed from <label>s and the <wildcard label> “%”

using the operation signs <vertical bar> “|” for disjunction, <ampersand> “&” for conjunction, <exclamation mark> “!” for negation, and balanced pairs of parentheses or square brackets for grouping.

- <element pattern where clause>, a <search condition> to be satisfied by the graph element that is bound by the <element pattern>.

<path pattern expression>s are regular expressions built recursively from <element pattern>s using the following operations, governed by the Format and Syntax Rules of Subclause 16.7, “<path pattern expression>”, and other Subclauses.

- concatenation, indicated syntactically by string concatenation (i.e., no operation sign).

NOTE 44 — <element pattern>s and more complex <path pattern expression>s may be concatenated in ways that appear to juxtapose either two <node pattern>s or two <edge pattern>s. These topologically inconsistent patterns are understood during pattern matching as follows:

- Two consecutive <node pattern>s must bind to the same node.
- Two consecutive <edge pattern>s conceptually have an implicit <node pattern> between them.

- Grouping, using either matching pairs of parentheses or square brackets to form a <parenthesized path pattern expression>. A <parenthesized path pattern expression> may optionally contain a <search condition> to constrain matches.
- Alternation, indicated by <vertical bar> or <multiset alternation operator>.

NOTE 45 — Alternation with <vertical bar> provides set semantics using deduplication of redundant equivalent reduced matches, whereas alternation with <multiset alternation operator> provides multiset semantics, with no deduplication.

- Quantification, indicated by a postfix <graph pattern quantifier>, which may be affixed to an <edge pattern> or a <parenthesized path pattern expression>.
- <questioned path primary>, indicated by a postfix <question mark> affixed to an <edge pattern> or a <parenthesized path pattern expression>.

NOTE 46 — Unlike many regular expression languages, the <question mark> operator is not the same as {0,1}, the difference being that <questioned path primary> exposes its singleton variables as conditional singletons, whereas {0, 1}, in common with all other quantifiers, exposes all variables as group.

4.12.4 Path pattern matching

Path pattern matching is performed by Subclause 16.6, “<graph pattern>”, which in turn may invoke Subclause 22.3, “Evaluation of a <path pattern expression>”, and Subclause 22.4, “Evaluation of a selective <path pattern>”, as well as other Subclauses incidentally invoked for expression evaluation, etc..

4.12.5 <path mode>

A <path mode> may be specified for any <parenthesized path pattern expression> or any <path pattern> from the following choices:

- WALK, the default <path mode>, this being the absence of any filtering implied by the other <path mode>s.
- TRAIL, where path bindings with repeated edges are not returned.
- ACYCLIC, where path bindings with repeated nodes are not returned.
- SIMPLE, where path bindings with repeated nodes are not returned unless these repeated nodes are the first and the last in the path.

Using trail, acyclic or simple matching path modes for all unbounded quantifiers guarantees that the result set of a graph pattern matching will be finite.

4.12.6 Selective <path search prefix>

The set of path bindings resulting from a graph pattern match can be further restricted by a selective <path search prefix> *SPSP*. *SPSP* is defined by partitioning the potentially infinite set of path bindings by the endpoints, that is, the first and last nodes bound by the path bindings.

NOTE 47 — This partitioning is crucial to the definition of *SPSP*. *SPSP* makes a selection of some number of path bindings from each partition. For example, a path binding is “shortest” if its length is minimal within its partition. A “shortest” path binding in one partition may be longer than a “shortest” path binding in another partition.

SPSP can constrain the result set in the following ways:

- <any path search>: to non-deterministically pick some number of path bindings from each partition; the number is specified by an <unsigned integer specification>.
- <all shortest path search>: to pick all the shortest path bindings in each partition.
- <counted shortest path search>: to non-deterministically pick some number of shortest path bindings from each partition; the number is specified by an <unsigned integer specification>.
- <counted shortest group search>: to group each partition into groups of path bindings having the same length, order the groups by path length, and pick all path bindings in some number of groups from the front of each partition; the number is specified by an <unsigned integer specification>.

The specification of *SPSP* guarantees that the result set of a graph pattern matching will be finite.

4.13 Data types

4.13.1 General information about data types

A *data type* is a set of elements with shared characteristics representing data. An *object type* is a data type comprising *objects*, a *value type* is a data type comprising *values*, a *reference value type* is a value type comprising *reference values*, and a *property value type* is a value type comprising *property values*. A data type characterizes the sites that may be occupied by instances of its elements. Every instance belongs to multiple data types. A data type characterizes the sites that may be occupied by instances of its elements. The physical representation of an instance of a data type is implementation-dependent.

A data type *A* may be equivalent to, disjoint from, overlap with, or included in some other data type *B* depending on the sharing of elements between *A* and *B*. In this document, any two data types with the exact same elements are considered equivalent. A data type containing all elements of some data type *X* is a supertype of *X*. A data type comprising elements contained in some data type *X* is a subtype of *X*. All data types form a partial type hierarchy that is implicitly defined by the subtyping relation between them.

A *most specific type* of some element *X* is a smallest supertype of all data types that includes *X*. In this document, this data type is always either uniquely defined or determined to be the normal form of all equivalent data types containing *X*. The *strict elements* of a data type are the elements for which that data type is the most specific type.

A *meta type* is a set of data types of the same kind (such as the meta type of all character string types). Every data type is of exactly one single associated meta type that classifies the data type itself. This meta type is called the base type of that data type. Base types implicitly determine the universe of elements from which each of their data types are drawn (e.g., the integers). Every base type has a name. The name of a base type is a sequence of reserved words that is specified by this document that specifies the base type in specific contexts (such as base type names in data type descriptors). The name of a base type always ends with the reserved word DATA. Two data types of different base types are always disjoint. Elements of a data type of some base type *A* may still be assignable to a site whose declared type is another data type of some other base type *B* that is different from *A* through the application of Rules for implicit type conversion.

This document defines the following kinds of data types:

- A *predefined* data type (such as a *character string type*) is a data type specified by this document that is **atomic** and provided by the GQL-implementation. A data type is predefined even though the user is required (or allowed) to provide certain parameters when specifying it (for example the precision of a number). Predefined data types are sometimes called “built-in data types”, though not in this International Standard.
- A *constructed* data type such as a *record type* is a data type comprising **composite** elements.
- A *user-defined* data type is a data type that is **constructed** and explicitly defined by the user. User-defined data types can be defined by a standard, by an implementation, or by an application.

An *atomic* data type is a data type comprising **atomic** elements only. The existence of an operation (SUBSTRING) that is capable of selecting part of an element of some data type T (such as a character string or a datetime value) does not imply that T is not an atomic data type.

A *material* data type is a data type excluding the **null value** but a *nullable* data type is a data type including the **null value**. In this document, the nullability characteristic of a site is recorded in the data type descriptor of the declared type of the site. The null values of all data types of the same base type are identical. The null values included in any two data types of different base types conceptually are different values but for all practical intents and purposes this difference cannot be observed during the execution of a GQL-request.

4.13.2 Naming of predefined value types and associated base types

Every predefined value type is specified by any of its *declared names*, including its *preferred name*, which is determined subject to implementation-defined provisions. A declared name is a non-empty sequence of reserved words specified by this document. Every predefined value type is classified by its associated base type, which is a meta type. The names of all base types of all predefined value types are specified by one of the following sequences of <key word>s:

- BOOLEAN DATA
- STRING DATA
- BINARY DATA
- INTEGER DATA
- DECIMAL DATA
- FLOAT DATA

For reference purposes:

- The value types specified by BOOL and BOOLEAN are referred to as *Boolean types* and the values of Boolean types are referred to as *truth values*, or, alternatively, as *Booleans*. The base type of all Boolean types is BOOLEAN DATA.
- The value types specified by STRING and VARCHAR are referred to as *character string types* and the values of character string types are referred to as *character strings*. The base type of all character string types is STRING DATA.
- The value types specified by BYTES, BINARY, and VARBINARY are referred to as *byte string types* and the values of byte string types are referred to as *byte strings*. The base type of all byte string types is BINARY DATA.
- Exact numeric types and approximate numeric types are collectively referred to as *numeric types*. Values of numeric types are referred to as *numbers*.
- The signed exact numeric types and the unsigned exact numeric types are collectively referred to as *exact numeric types*. Values of exact numeric types are referred to as *exact numbers*.

4.13 Data types

- The value types specified by DECIMAL, DEC, SMALLINT, INT, INTEGER, SIGNED INTEGER, INT16, INTEGER16, SIGNED INTEGER16, INT32, INTEGER32, SIGNED INTEGER32, INT64, INTEGER64, SIGNED INTEGER64, INT128, INTEGER128, SIGNED INTEGER128, INT256, INTEGER256, SIGNED INTEGER256, and BIGINT are collectively referred to as *signed exact numeric types*. Values of signed exact numeric types are referred to as *signed exact numbers*.
- The value types specified by UINT, UNSIGNED INTEGER, UINT16, UNSIGNED INTEGER16, UINT32, UNSIGNED INTEGER32, UINT64, UNSIGNED INTEGER64, UINT128, UNSIGNED INTEGER128, UINT256, and UNSIGNED INTEGER256 are collectively referred to as *unsigned exact numeric types*. Values of unsigned exact numeric types are referred to as *unsigned exact numbers*.
- Exact numeric types with binary precision in bits and a scale of 0 (zero) are collectively referred to as (signed or unsigned) *integer types*. Values of (signed or unsigned) integer types are referred to as (signed or unsigned) *integer numbers*, or, alternatively as (signed or unsigned) *integers*. The base type of all (signed or unsigned) integer types is INTEGER DATA.
- Exact numeric types with decimal precision and scale in digits are referred to as *decimal types*. Values of decimal types are collectively referred to as *decimal numbers*, or, alternatively as *decimals*. The base type of all decimal types is DECIMAL DATA.
- The value types specified by FLOAT, FLOAT16, FLOAT32, FLOAT64, FLOAT128, FLOAT256, REAL, DOUBLE, and DOUBLE PRECISION are collectively referred to as *approximate numeric types* and the values of approximate numeric types are known as *approximate numbers*, or, alternatively, as *floating point numbers*. The base type of all approximate numeric types is FLOAT DATA.

4.13.3 Data type descriptors

Each data type has an associated data type descriptor; the contents of a data type descriptor are determined by the specific data type that it describes. A data type descriptor includes an identification of the data type and all information needed to characterize an element of that data type.

Subclause 21.2, “*<value type>*”, describes the semantic properties of each value type.

4.13.4 Instance conformance

An instance V *conforms* to a data type T if and only if V is an element of T . In the context of data types and their instances, the verb “to be” (including all its grammatical variants, such as “is”) is defined as follows: An instance V is said to be a (or of) data type T if A conforms to T .

An instance V *strictly conforms* to a data type T if and only if V conforms to T and there is no data type U that is a (non-empty) strict subtype of T such that V is also an element of U . In the context of data types and their instances, the verb “to be” (including all its grammatical variants, such as “is”) is defined as follows: An instance V is said to strictly be a (or of) data type T if V strictly conforms to T .

NOTE 48 — Only the strict elements of a data type strictly conform to that data type.

4.13.5 Data type relations

4.13.5.1 Subtype relation

The *subtype relation* between data types is defined in terms of their shared elements. Any two data types either are or are not subtypes of each other.

Given two data types T and U , T is a subtype of U if and only if every element of T is also an element of U . Given three data types T , U , and V , it is true that if T is a subtype of U and U is a subtype of V then T is a subtype of V . In the context of data types, the verb “to be” (including all its grammatical variants, such as “is”) is defined as follows: A data type T is said to be data type U if T is a subtype of U .

Given two data types T and U , T is a *strict subtype* of U if and only if T is different from U and T is a subtype of U . Given three data types T , U , and V , it is true that if T is a strict subtype of U and U is a strict subtype

of V then T is a strict subtype of V . In the context of data types, the verb “to strictly be” (including all its grammatical variants, such as “is”) is defined as follows: A data type T is said to strictly be data type U if T is a strict subtype of U .

Exactly one of the non-empty data types of an instance V , namely the *most specific* type of V , is a subtype of every data type of V . Every data type is a subtype of itself but it is never a strict subtype of itself.

4.13.5.2 Supertype relation

The *supertype relation* between data types is the inverse of the subtype relation. Any two data types either are or are not supertypes of each other.

Given two data types T and U , T is a *supertype* of U if and only if U is a subtype of T .

Given two data types T and U , T is a *strict supertype* of U if and only if T is different from U and T is a supertype of U .

Every value type is a supertype of itself but never is a strict supertype of itself.

4.13.5.3 Extensional type equality

Two data types are *extensionally equal* if they have the exact same elements.

4.13.5.4 Type cardinality

The *cardinality of a data type* is the cardinality of the set of its elements. A data type T is greater than a data type U , if and only if the cardinality of T is greater than the cardinality of U . Conversely, a data type T is smaller than a data type U , if and only if the cardinality of T is smaller than the cardinality of U .

Given two data types T and U , if T is a strict subtype of U then T is smaller than U .

An *empty data type* is a data type that has no elements.

4.13.6 Data type operations

4.13.6.1 Coercion

Coercion is the automatic conversion of an instance V of data type T into an instance W of data type U through the application of a *coercion rule*. A *coercion rule* explicitly states how instances of a data type T are to be converted into instances of a data type U that is different from T .

NOTE 49 — Coercion may occur when binding an instance V to a site that does not meet the requirements of the site.

4.13.6.2 Type join

Given two data types T and U , the *join* of T and U is the smallest existing data type V that is a supertype of both T and U . The join between two data types T and U is the same as the join between U and T .

NOTE 50 — The join computes the union between two data types in the type hierarchy, i.e., the smallest data type that contains the elements of both argument data types.

The join between a collection of data types is uniquely defined independent of the order in which data types are combined.

4.13.6.3 Type meet

Given two data types T and U , the *meet* between T and U is the largest data type V that is a subtype of both T and U . The meet between two data types T and U is the same as the meet between U and T .

NOTE 51 — The type meet computes the intersection between two data types in the type hierarchy, i.e., the greatest value type whose values are elements of both argument data types.

The meet between a collection of data types is uniquely defined independent of the order in which data types are combined.

4.14 Type hierarchy outline

4.14.1 Introduction to type hierarchy outline

The *type hierarchy* forms an algebraic lattice of data types that puts the any type at the top and the nothing type at the bottom.

4.14.2 The any type

The *any type* is the largest supported data type and therefore is placed at the root of the type hierarchy.

Type characteristics The any type is a taxonomic data type.

Subtyping Every data type is a subtype of the any type.

4.14.3 The any object type

The *any object type* is the supertype of all objects.

Type characteristics The any object type is a taxonomic data type.

Subtyping Every object type is a subtype of the any object type.

4.14.4 The any value type

The *any value type* is the supertype of all values.

Type characteristics The any value type is a taxonomic data type.

Subtyping Every value type is a subtype of the any value type.

4.14.5 Union types

A *union type* between two data types T and U is the union of every element of T and every element of U .

Type characteristics A union type between a data type T and a data type U is the same data type as a union type between U and T . A union type between a data type T and a data type U is the same data type as U if and only if T is a subtype of U .

Subtyping Any data type T is a subtype of a union type between T and some other data type.

NOTE 52 — Every data type is a subtype of every data type between itself and some other data type.

4.14.6 The any property value type

The *any property value type* is the super type of every property value of every graph element of GQL-data.

Type characteristics The any property value type is a taxonomic data type.

Subtyping The any property value type is the super type of all predefined value types and all regular list types of lists of elements that are all subtypes of the property value type.

NOTE 53 — Implementations may extend the definition of the any property value type to encompass additional value types.

4.14.7 The nothing type

The *nothing type* is the only supported empty data type. It is the smallest data type and therefore is placed at the bottom of the type hierarchy. The nothing type has no elements.

Type characteristics The nothing type is an abstract taxonomic data type.

Subtyping The nothing type is a subtype of every other data type.

4.15 Graph types

4.15.1 Introduction to graph types

A *graph type* describes the attributes of and the nodes and edges that may occur in a conforming graph. A graph type comprises:

- A graph type label set that comprises zero or more labels. A label has a name, which is an identifier that is unique within the graph type.

The maximum cardinality of graph type label sets is implementation-defined.

Without Feature GA00, “Graph label set support”, the graph type label set of a conforming graph type shall be empty.

- A graph type property type set that comprises a set of zero or more property types. Each property type comprises:

- Its property name, which is an identifier that is unique within the graph type.

NOTE 54 — The names of graph type labels and of graph type property types are in separate namespaces.

- Its property value type, which is a subtype of the any property value type.

The maximum cardinality of graph type property type sets is implementation-defined.

Without Feature GA01, “Graph property set support”, the graph type property type set of a conforming graph type shall be empty.

- A set of node types.

- A set of edge types.

- A node type name dictionary that maps node type names, which are identifiers, to node types such that each node type name is mapped to a single node type. These node types shall be contained in the same graph type.

Without Feature GA08, “Named node types in graph types”, a conforming graph type shall contain an empty node type name dictionary.

- An edge type name dictionary that maps edge type names, which are identifiers, to edge types such that each edge type name is mapped to a single edge type. These edge types shall be contained in the same graph type.

Without Feature GA09, “Named edge types in graph types”, a conforming graph type shall contain an empty edge type name dictionary.

NOTE 55 — A graph type may be associated with a catalog object that has a name, but graph types can exist that have no name, or at least have no name that is required to be visible to a user of a GQL-implementation. A graph projected by a query has a type, but it is not catalogued or named.

The element types of a graph type may be references to element types defined in another graph type.

NOTE 56 — This document distinguishes between aliases for element types and aliases for primary objects.

Type characteristics Every graph type is a constructed type.

Subtyping Every graph type is a subtype of the any object type.

Any graph type T is a subtype of any graph type U if and only if:

4.15 Graph types

- 1) The graph type label set of T contains the graph type label set of U .
- 2) The graph property type set of T contains the graph property type set of U .
- 3) Every node type in the node type set of T is a subtype of some node type from the node type set of U .
- 4) Every edge type in the edge type set of T is a subtype of some edge type from the edge type set of U .

4.15.2 Graph type descriptors

A graph type is described by a graph type descriptor that includes:

- The name of the graph type (also known as the graph type name) that is the name of the catalog object.
- A set of zero or more labels (also known as a graph type label set). A label has a name that is an identifier that is unique within the graph type label set.
- A set of zero or more property type descriptors (also known as a graph type property type set).
- A set of node type descriptors (also known as a node type set).
- A set of edge type descriptors (also known as an edge type set).
- A node type name dictionary that maps node type names, which are identifiers, to node type such that each node type name is mapped to a single node type. These node types shall be contained in the graph type node type set of the same graph type descriptor.
- An edge type name dictionary that maps edge type names, which are identifiers, to edge type such that each edge type name is mapped to a single edge type. These edge types shall be contained in the graph type edge type set of the same graph type descriptor.

Two graph type descriptors describe equal graph types if they contain

- Equal graph type label sets,
- Equal graph type property type sets,
- Equal node type sets,
- Equal edge type sets,
- Equal node type name dictionaries, and
- Equal edge type name dictionaries.

Two node and edge type name dictionaries are equal if

- They map an equal set of node and edge type name and
- Map each of these node and edge type names to equal node and edge types, respectively.

A *named graph type descriptor* is both a catalog object descriptor and a graph type descriptor and comprises every component of both kinds of descriptors.

4.15.3 Graph element types**4.15.3.1 Graph element base type**

A graph element is either a node or an edge. The *graph element base type* is the data type of all graph elements

Subtyping The graph element base type is the supertype of every graph element type.

4.15.3.2 Node base type

A node is the fundamental unit from which a graph is formed. The *node base type* is the data type of all nodes.

Subtyping The node base type is a subtype of the graph element base type. The base node type is the supertype of every node type.

Identity Two nodes are identical if they represent the same original node from the same base graph.

4.15.3.3 Node types

A *node type* is the data type of nodes that have specific labels and that have specific properties and whose properties have a specific property value type.

Each node type comprises:

- A node type label set that comprises a set of zero or more labels. A label has a name that is an identifier that is unique within the node type.

The maximum cardinality of node type label sets is implementation-defined.

Without Feature GA02, “Empty node label set support”, a node type in a conforming graph type shall contain a non-empty node type label set.

Without Feature GA03, “Singleton node label set support”, a node type in a conforming graph type shall not contain a singleton node type label set.

Without Feature GA04, “Unbounded node label set support”, a node type in a conforming graph type shall not contain a node type label set that comprises one or more labels.

- A node type property type set that comprises a set of zero or more property types. Each property type comprises:

- Its property name that is an identifier that is unique within the node type.

NOTE 57 — The names of node type labels and of node type property types are in separate namespaces.

- Its property value type that is a subtype of the any property value type.

The maximum cardinality of node type property type sets is implementation-defined.

Strict elements Every node that has the exact set of labels and properties required by the node type and whose property values strictly conform to the corresponding property value types required by the node type is a strict of the node type.

Subtyping A node type is a super type of another node type if the other node type requires the same restrictions on labels, properties, and property value types.

4.15.3.4 Node type descriptor

A node type is described by the node type descriptor. The node type descriptor includes:

- The node type name, if specified.
- A set of zero or more labels (also known as a node type label set). A label has a name that is an identifier that is unique within the node type label set.
- A set of zero or more property type descriptors (also known as a node type property type set).

Two node type descriptors describe equal node types if they contain equal node type label sets and equal node type property type.

4.15.3.5 Edge base type

An edge has zero or more labels and zero or more properties and represents a connection from a source node to a destination node in the same graph (if directed) or a connection between two endpoints (if undirected). The *edge base type* is the data type of all edges.

Subtyping The edge base type is a subtype of the graph element base type. The edge base type is the supertype of every edge type.

An edge type that is introduced by an edge type definition of a graph type restricts its elements in terms of their required directionality, required labels and properties, and the required property value types of those properties, as well as the required node types of their endpoints.

Identity Two edges are identical if they represent the same original edge from the same base graph.

4.15.3.6 Edge types

An *edge type* is the data type of edges that have specific labels and that have specific properties and whose properties have a specific property value type and whose endpoints conform to specific node types.

Each edge type comprises:

- An edge type label set that comprises a set of zero or more labels. A label has a name that is an identifier that is unique within the edge type.

The maximum cardinality of edge type label sets is implementation-defined.

Without Feature GA05, “Empty edge label set support”, an edge type in a conforming graph type shall contain a non-empty edge type label set.

Without Feature GA06, “Singleton edge label set support”, an edge type in a conforming graph type shall not contain a singleton edge type label set.

Without Feature GA07, “Unbounded edge label set support”, an edge type in a conforming graph type shall not contain an edge type label set that comprises one or more labels.

- An edge type property type set that comprises a set of zero or more property types. Each property type comprises:

- Its property name that is an identifier that is unique within the edge type.

NOTE 58 — The names of edge type labels and of edge type property types are in separate namespaces.

- Its property value type that is a subtype of the any property value type.

The maximum cardinality of edge type property type sets is implementation-defined.

- Two (possibly identical) endpoint node types that are node types contained in the same graph type.
- An indication whether the edge type is a directed edge type or an undirected edge type.

Additionally, a directed edge type identifies one of its endpoint node types as its source, and the other as its destination. The direction of a directed edge type is from its source to its destination.

Without Feature G001, “Undirected edge patterns”, a conforming graph type shall not contain undirected edge types.

Strict elements Every edge that has the exact set of labels and properties required by the edge type and whose property values strictly conform to the corresponding property value types required by the edge type and whose source nodes strictly conform to the node types required for source nodes of the

edge type and whose destination nodes strictly conform to the node types required for endpoints of the edge type (if directed) or whose endpoints strictly conform to the node types required for endpoints of the edge type (if undirected) is a strict element of the edge type.

Subtyping An edge type is a super type of another edge type if the other edge type requires the same restrictions on labels, properties, property value types, directionality, and the node types of endpoints.

4.15.3.7 Edge type descriptor

An edge type is described by the edge type descriptor. The edge type descriptor includes:

- The edge type name, if specified.
- A set of zero or more labels (also known as an edge type label set). A label has a name that is an identifier that is unique within the edge type label set.
- A set of zero or more property type descriptors (also known as an edge type property type set).
- An indication whether the edge type is directed or undirected.
- Case:
 - If the edge type is directed, then
 - A source node type descriptor,
 - A destination node type descriptor.
 - If the edge type is undirected, then
 - A set of two endpoint node types descriptors.

Two edge type descriptors describe equal edge types if they contain

- Equal edge type label sets,
- Equal node type property type,
- Equal indication whether they are directed or undirected, and
- Case:
 - If the edge types are directed, then
 - Equal source node types and
 - Equal destination node types.
 - If the edge types are undirected, then
 - Equal sets of endpoint node types.

4.15.3.8 Property types

4.15.3.8.1 Introduction to property types

A property type is a pair comprising:

- The property name that is an identifier.
- A declared type that can be any property value type.

Two property types are equal if they have equal property names and equal declared types.

4.15.3.8.2 Property type descriptor

Each property type descriptor comprises:

- The property name that is an identifier.
- A declared type that can be any property value type.

4.16 Binding table types

A binding table type is the object type of all binding tables that are comprised of a collection of records of the same record type.

Type characteristics A binding table type whose collections of records has type T is an abstract object type, if and only if T is an abstract object type.

Strict elements A binding table BT is a strict element of a binding table type BT whose collection of records has type CT if its collection of records is a strict element of CT .

Subtyping A binding table whose collection of records has type T is a subtype of a binding table whose collection of records has type S , if and only if T is a subtype of S .

Identity Let T_1 and T_2 be binding tables. T_1 and T_2 are identical, if their collections of records are identical and are either both ordered or unordered.

NOTE 59 — This defines the identity of objects in terms of their contents but not of references to them.

4.17 Value types

4.17.1 Constructed types

4.17.1.1 Introduction to constructed types

The GQL language supports constructed types that are either collection types or map types.

The GQL language supports 4 types of collections:

- Multisets
- Sets
- Lists (both regular or distinct)

A collection may be empty.

The GQL language supports maps that may be regular maps or records, and that may be empty.

4.17.1.2 Multiset type

A value of *multiset type* with element type ET is an unordered collection of elements of ET that is called a *multiset*.

NOTE 60 — Since a multiset is unordered, there is no ordinal position to reference individual elements of a multiset.

Type characteristics A multiset type with element type T is an abstract value type if and only if T is an abstract value type.

Strict elements A multiset M is a strict element of a multiset type with element type T if and only if T is the join of the most specific types of all elements of M .

Subtyping Every multiset type with element type T is a subtype of a collection type with element type U if and only if T is a subtype of U . Every multiset type with element type T is a subtype of a multiset type with element type U if and only if T is a subtype of U .

If MT is some multiset type with element type EDT , then every value of MT is a multiset of elements of EDT .

Identity Let $B1$ and $B2$ be multisets of EDT . $B1$ and $B2$ are identical if and only if $B1$ and $B2$ have the same cardinality n , and for each element x in $B1$, the number of elements of $B1$ that are identical to x , including x itself, equals the number of elements of $B2$ that are identical to x .

Submultisets $B1$ is a submultiset of $B2$ if, for each element x of $B1$, the number of elements of $B1$ that are not distinct from x , including x itself, is less than or equal to the number of elements of $B2$ that are not distinct from x .

4.17.1.3 Set type

A value of *set type* with element type ET is an unordered, duplicate-free collection of elements of ET that is called a *set*.

NOTE 61 — Since a set is unordered, there is no ordinal position to reference individual elements of a multiset.

Type characteristics A set type with element type T is an abstract value type if and only if T is an abstract value type.

Strict elements A set S is a strict element of the set type with element type T if and only if T is the join of the most specific types of all elements of S .

Subtyping Every set type with element type T is a subtype of a collection type with element type U if and only if T is a subtype of U . Every set type with element type T is a subtype of a set type with element type U if and only if T is a subtype of U .

If ST is some set type with element type EDT , then every value of ST is a set of elements of EDT .

Identity Let $S1$ and $S2$ be sets of EDT . $S1$ and $S2$ are identical if and only if $S1$ and $S2$ have the same cardinality n , and for each element x in $S1$, there is an element of $S2$ that is identical to x .

Subsets $S1$ is a subset of $S2$ if, for each element x of $S1$, $S2$ contains an element identical to x .

4.17.1.4 List types

4.17.1.4.1 List types

A value of *list type* with element type ET is an ordered collection of elements of ET that is called a *list* or an *array*. In a list L , each element is associated with exactly one ordinal position in L . If n is the cardinality of L , then the ordinal position p of an element is an integer in the range $1 \text{ (one)} \leq p \leq n$.

Type characteristics A list type with element type T is an abstract value type if and only if T is an abstract value type.

Strict elements A list L is a strict element of a list type with element type T if and only if T is the join of the most specific types of all elements of L .

Subtyping Every list type with element type T is a subtype of a collection type with element type U if and only if T is a subtype of U . Every list type with element type T is a subtype of a list type with element type U if and only if T is a subtype of U . Every distinct list type with element type T is a subtype of a list type with element type U if and only if T is a subtype of U . Every distinct list type with element type T is a subtype of a distinct list type with element type U if and only if T is a subtype of U .

If EDT is the element type of L , then L can thus be considered as a function of the integers in the range $1 \text{ (one)} \text{ to } n$ into EDT .

If LT is some list type with element type EDT , then every value of LT is a list of EDT . If LT is some distinct list type with element type EDT , then every value of LT is a distinct list of EDT .

Identity Let $L1$ and $L2$ be lists of EDT . $L1$ and $L2$ are identical if and only if $L1$ and $L2$ have the same cardinality n and if, for every i in the range $1 \text{ (one)} \leq i \leq n$, the element at ordinal position i in $L1$ is identical to the element at ordinal position i in $L2$.

Comparison Let $L1$ and $L2$ be lists. $L1$ is less than or equal than $L2$ if and only if $L1$ has a smaller cardinality than $L2$ or $L1$ and $L2$ have the same cardinality and one of the following is true:

- All elements of $L1$ are less than their corresponding elements at the same position in $L2$.
- Some leading elements of $L1$ are less than their corresponding elements at the same position in $L2$ and all following elements in $L1$ are equal to to their corresponding elements in $L2$ at the same position.
- All elements of $L1$ are equal to their corresponding elements at the same position in $L2$.

4.17.1.4.2 Regular lists

A *list* or an *array* may contain duplicate elements.

4.17.1.4.3 Distinct lists

A *distinct list* or a *distinct array* is a *list* that shall not contain duplicate elements, i.e., no two elements of a distinct list are equal.

NOTE 62 — A distinct list may contain multiple null values.

4.17.1.5 Path type

A value of *path type* is called a *path*. A path is an ordered sequence of graph elements that always starts and ends with a node and, alternates between nodes and edges, such that each edge in the sequence is adjacent to the immediately preceding and following nodes.

Strict elements Every path is a strict element of the path type.

Subtyping The path type is a subtype of the collection type whose element type is the graph element type.

Identity Let $P1$ and $P2$ be paths. $P1$ and $P2$ are identical if and only if $P1$ and $P2$ have the same cardinality n and if, for every i in the range $1 \text{ (one)} \leq i \leq n$, the graph element at ordinal position i in $P1$ is identical to the graph element at ordinal position i in $P2$.

Comparison Let $P1$ and $P2$ be paths. $P1$ is less than or equal than $P2$ if and only if $P1$ has a smaller cardinality than $P2$ or $P1$ and $P2$ have the same cardinality and one of the following is true:

- All graph elements of $P1$ are less than the corresponding graph element at the same position in $P2$.
- Some leading graph elements of $P1$ are less than the corresponding graph element at the same position in $P2$ and all following graph elements in $P1$ are equal to to the corresponding graph element in $P2$ at the same position.
- All graph elements of $P1$ are equal to the corresponding graph element at the same position in $P2$.

4.17.1.6 Map type

A value of the *map type* with key type KT and value type VT is called a *map*. A map is an unordered collection of (key, value) pairs that are called entries, such that each possible key appears at most once in the collection. All the keys have the same value type that shall be a predefined value type.

Type characteristics The map type with key type KT and value type VT is an abstract value type if either KT or VT is an abstract value type.

Strict elements A map M is a strict element of a map type with key type KT and value type VT if and only if KT is the join of the most specific types of keys of all map entries of M and VT is the join of the most specific types of values of all map entries M .

Subtyping A map type with key type $KT1$ and value type $VT1$ is a subtype of a map type with key type $KT2$ and value type $VT2$ if and only if $KT1$ is a subtype of $KT2$ and $VT1$ is a subtype of $VT2$.

NOTE 63 — Since a map is unordered, there is no ordinal position to reference individual elements of a map. Individual values in a map are accessed by a look-up of the key instead.

Comparison Maps are not comparable.

4.17.1.7 Record type

A value of *record type* with a set of field types is called a *record*. A *record* has a set of fields, each field has a name that is unique within the record, and a value.

Type characteristics A record type is an abstract data type if and only if the value type of any of the field types of the record type is an abstract data type.

Strict elements A record is a strict element of the record type with a set of field types that exactly are the field types of the fields of the record and whose value types are the most specific types of the corresponding fields of the actual record.

Subtyping Every record type U is a subtype of a record type T if and only if for every field FT of T there is either a corresponding field FU in U with the same name and the value type of FU is a subtype of the value type of FT or the value type of FT is a subtype of the null type.

Comparison A value of a record $R1$ is compared with a value of a record $R2$ by component-wise comparison of fields with the same name in the order obtained by sorting the union of all field names of $R1$ and $R2$ in standard sort order. Missing fields in either $R1$ or $R2$ are assumed to be the null value.

4.17.1.8 Unit type

The *unit value* is a record with zero fields.

Subtyping The unit type is a subtype of every record type.

4.17.2 Reference value types

A value of a reference type R for an object type O is a reference to an object of type O .

Type characteristics A reference value type is a value type.

The GQL language currently only supports reference values to the following kinds of GQL-objects:

- Graphs.
- Binding tables.
- Nodes.
- Edges.

Subtyping A reference type R for an object type O is a subtype of a reference type S for an object type P if and only if O is a subtype of P .

Identity References are identical if and only if they have the same referent.

Comparison References are compared by comparing their referent.

4.17.3 Predefined value types

4.17.3.1 Boolean types

The material values of a *Boolean type* are the distinct truth values *True* or *False*. The truth value *Unknown* is represented by the null value.

This document does not make a distinction between the null value and the truth value *Unknown* that is the result of a GQL <predicate>, <search condition>, or <boolean value expression>; they may be used interchangeably to mean exactly the same value.

Every Boolean type is described by its Boolean data type descriptor. A Boolean data type descriptor comprises:

- The name of the base type of all Boolean types (BOOLEAN DATA).
- The preferred name of the Boolean type (implementation-defined choice of BOOLEAN or BOOL).
- An indication of whether the Boolean type includes the null value (which is indistinguishable from the truth value *Unknown*).

A GQL-implementation regards certain <boolean type>s as equivalent, as determined by the Syntax Rules of Subclause 21.2, “<value type>”. When two or more <boolean type>s are equivalent, the GQL-implementation chooses one of these equivalent <boolean type>s as the normal form representing that equivalence class of <boolean type>s.

All Boolean values and truth values are assignable to a site of at least Boolean type. The values *True* and *False* may be assigned to any site having at least material Boolean type; assignment of *Unknown*, or the null value, is subject to the nullability characteristic of the target.

4.17.3.2 Character string types

A material value of a *character string type* is a possibly-empty variable-length string (sequence) of characters drawn from the Universal Character Set repertoire specified by [The Unicode® Standard](#) and [ISO/IEC 10646](#).

A character string has a length that is the number of characters in the sequence. The length is 0 (zero) or a positive integer. The maximum length of a character string is implementation-defined but shall be greater than or equal to 16383 characters.

With two exceptions, a character string value expression is assignable only to sites of at least a character string type. The exceptions are detection of a noncharacter in a character string and such other cases as are implementation-defined. If a store assignment would result in the loss of characters due to truncation, then an exception condition is raised. If a retrieval assignment or evaluation of a <cast specification> would result in the loss of characters due to truncation, then a warning condition is raised.

A GQL-implementation may assume that all character strings are normalized in one of Normalization Form C (NFC), Normalization Form D (NFD), Normalization Form KC (NFKC), or Normalization Form KD (NFKD), as specified by [The Unicode® Standard](#). <normalized predicate> can be used to verify the normalization form to which a particular character string conforms. Applications can also use <normalize function> to enforce a particular <normal form>. With the exception of <normalize function> and <normalized predicate>, the result of any operation on an unnormalized character string is implementation-defined.

Detection of a noncharacter in a character string causes an exception condition to be raised. The detection of an unassigned code point does not.

Every character string type is described by its character string data type descriptor. A character string data type descriptor contains:

- The name of the base type of all character string types (STRING DATA).
- The preferred name of the character string type (implementation-defined choice of STRING or VARCHAR).
- An indication of whether the type includes the null value.
- The maximum length in characters of the character string type.

The maximum length in characters of a character string type shall be a positive integer.

A GQL-implementation regards certain <character string type>s as equivalent, if they have the same maximum length, as determined by the Syntax Rules of Subclause 21.2, “<value type>”. When two or more <character string type>s are equivalent, the GQL-implementation chooses one of these equivalent <character string type>s as the normal form representing that equivalence class of <character string type>s.

4.17.3.3 Byte string types

A material value of a *byte string type* is a sequence of bytes that is called a *byte string*.

A byte string has a length that is the number of bytes in the sequence. The length is 0 (zero) or a positive integer. The maximum length of a byte string is implementation-defined but shall be greater than or equal to 65534 bytes.

Every byte string type is described by its byte string data type descriptor. A byte string data type descriptor comprises:

- The name of the base type of all byte string types (BINARY DATA).
- The preferred name of the byte string type (implementation-defined choice of BINARY or BYTES for fixed-length byte string types and implementation-defined choice of VARBINARY or BYTES for variable-length byte string types).
- An indication of whether the byte string type includes the null value.
- The minimum length in bytes of the byte string type.
- The maximum length in bytes of the byte string type.

The minimum length in bytes of a byte string type shall be a non-negative integer.

The maximum length in bytes of a byte string type shall be a positive integer.

A GQL-implementation regards certain <byte string type>s as equivalent, if they have the same maximum length, as determined by Subclause 21.2, “<value type>”. When two or more <byte string type>s are equivalent, the GQL-implementation chooses one of these equivalent <byte string type>s as the normal form representing that equivalence class of <byte string type>s.

A byte string is assignable only to sites of at least byte string type. If a store assignment would result in the loss of bytes due to truncation, then an exception condition is raised. If a retrieval assignment would result in the loss of bytes due to truncation, then a warning condition is raised.

4.17.3.4 Numeric types

4.17.3.4.1 Introduction to numbers

GQL supports two classes of numeric data:

- Exact numeric data
- Approximate numeric data

Exact numeric data is either signed or unsigned. Signed numeric data is either non-negative (positive or zero) or negative. Unsigned numeric data is always non-negative. Approximate numeric data is always signed.

Exact numeric types are either of base 2 (binary) or base 10 (decimal) precision. Signed binary exact numeric types are two's complement integers. Decimal exact numeric types may specify a scale factor. Approximate numeric types are of base 2 (binary) precision and may specify a scale factor.

Every *numeric type* is described by a numeric data type descriptor. A numeric data type descriptor comprises the following characteristics:

- The name of the base type of the numeric type (INTEGER DATA for exact numeric types with binary precision, DECIMAL DATA for exact numeric types with decimal precision, and FLOAT DATA for approximate numeric types).
- The preferred name of the specific numeric type, which is the declared name of the normal form of the numeric type.
- An indication of whether the type includes the null value.
- The (implemented) precision of the numeric type.
- For an exact numeric type with decimal precision or an approximate numeric type, the (implemented) scale of the numeric type.
- An indication of whether the precision and the scale are expressed in binary or decimal terms.
- The explicit declared precision, if any, of the numeric type.
- For an exact numeric type with decimal precision or an approximate numeric type, the explicit declared scale, if any, of the numeric type.

If <precision> and <scale> are not specified explicitly, then the corresponding element of the descriptor effectively contains the null value.

A GQL-implementation is permitted to regard certain <exact numeric type>s as equivalent, if they have the same precision, scale, and radix, as permitted by the Syntax Rules of Subclause 21.2, “<value type>”. When two or more <exact numeric type>s are equivalent, the GQL-implementation chooses one of these equivalent <exact numeric type>s as the normal form representing that equivalence class of <exact numeric type>s. The normal form determines the preferred name of the exact numeric type in the numeric data type descriptor.

Similarly, a GQL-implementation is permitted to regard certain <approximate numeric type>s as equivalent, as permitted by the Syntax Rules of Subclause 21.2, “<value type>”, in which case the GQL-implementation chooses a normal form to represent each equivalence class of <approximate numeric type> and the normal form determines the preferred name of the approximate numeric type in the numeric data type descriptor.

For every numeric type, the least material value is less than or equal to zero and the greatest material value is greater than zero.

4.17.3.4.2 Characteristics of numbers

An exact numeric type has a precision P and a scale S . P is a positive integer that determines the number of significant digits in a particular radix R , where R is either 2 or 10. S is a non-negative integer. Every value of an exact numeric type of scale S is of the form $n \times 10^{-S}$, where n is an integer such that $-R^P \leq n < R^P$.

NOTE 64 — Not every value in that range is necessarily a value of the type in question.

An exact numeric value consists of either one or more decimal digits followed by an optional decimal point and zero or more decimal digits or a decimal point followed by one or more decimal digits.

Approximate numeric values consist of a mantissa and an exponent. The mantissa is a signed numeric value, and the exponent is a signed integer that specifies the magnitude of the mantissa. Approximate numeric values have a precision. The precision of approximate numeric values is a positive integer that specifies the number of significant binary digits in the mantissa. The scale of approximate numeric values is a signed integer that specifies the number of significant binary digits of the exponent. The value of an approximate numeric type is the mantissa multiplied by a factor determined by the exponent.

NOTE 65 — An implementation may choose an internal representation for approximate numeric values that implicitly encodes leading bits instead of physically storing them. Any such implied bits are still part of the mantissa of any value represented using such an encoding.

An *<exact numeric literal>* *ENL* consists of either an *<unsigned integer>*, a *<period>* followed by an *<unsigned decimal integer>*, or an *<unsigned decimal integer>* followed by a *<period>* and an optional *<unsigned decimal integer>*. The declared type of *ENL* is an exact numeric type.

An *<approximate numeric literal>* *ANL* consists of a *<mantissa>* that is an *<exact numeric literal>*, the letter 'E' or 'e', and an *<exponent>* that is a *<signed decimal integer>*. The declared type of *ANL* is an approximate numeric type. If *M* is the value of the *<mantissa>* and *E* is the value of the *<exponent>*, then $M * 10^E$ is the *apparent value* of *ANL*. If the declared type of *ANL* is an approximate numeric type, then the actual value of *ANL* is approximately the apparent value of *ANL*, according to implementation-defined rules.

A number is assignable only to sites of at least numeric type. If an assignment of some number would result in a loss of its most significant digit, an exception condition is raised. If least significant digits are lost, implementation-defined rounding or truncating occurs, with no exception condition being raised. The rules for arithmetic are specified in Subclause 20.4, “*<numeric value expression>*”.

Whenever a numeric value is assigned to an exact numeric value site, an approximation of its value that preserves leading significant digits after rounding or truncating is represented in the declared type of the target. The value is converted to have the precision and scale of the target. The choice of whether to truncate or round is implementation-defined.

An approximation obtained by truncation of a numeric value *N* for an exact numeric type *T* is a value *V* in *T* such that *N* is not closer to zero than *V* and there is no value in *T* between *V* and *N*.

An approximation obtained by rounding of a numeric value *N* for an exact numeric type *T* is a value *V* in *T* such that the absolute value of the difference between *N* and the numeric value of *V* is not greater than half the absolute value of the difference between two successive numeric values in *T*. If there is more than one such value *V*, then it is implementation-defined which one is taken.

All numeric values between the smallest and the largest value, inclusive, in a given exact numeric type have an approximation obtained by rounding or truncation for that type; it is implementation-defined which other numeric values have such approximations.

An approximation obtained by truncation or rounding of a numeric value *N* for an approximate numeric type *T* or the decimal floating-point type *T* is a value *V* in *T* such that there is no numeric value in *T* distinct from that of *V* that lies between the numeric value of *V* and *N*, inclusive. >If there is more than one such value *V* then it is implementation-defined which one is taken.

It is implementation-defined which numeric values have approximations obtained by rounding or truncation for a given approximate numeric type or a given decimal floating-point type.

Whenever a numeric value is assigned to an approximate numeric value site, an approximation of its value is represented in the declared type of the target. The value is converted to have the precision of the target.

Operations on numbers are performed according to the normal rules of arithmetic, within implementation-defined limits, except as provided for in Subclause 20.4, “[<numeric value expression>](#)”.

4.17.3.4.3 Binary exact numeric types

The signed binary exact numeric types are:

- The *signed 8-bit integer type* SIGNED INTEGER8 (alternatively: INTEGER8, INT8) with binary precision of 8 bits and with scale 0 (zero).
- The *signed 16-bit integer type* SIGNED INTEGER16 (alternatively: INTEGER16, INT16) with binary precision of 16 bits and with scale 0 (zero).
- The *signed 32-bit integer type* SIGNED INTEGER32 (alternatively: INTEGER32, INT32) with binary precision of 32 bits and with scale 0 (zero).
- The *signed 64-bit integer type* SIGNED INTEGER64 (alternatively: INTEGER64, INT64) with binary precision of 64 bits and with scale 0 (zero).
- The *signed 128-bit integer type* SIGNED INTEGER128 (alternatively: INTEGER128, INT128) with binary precision of 128 bits and with scale 0 (zero).
- The *signed 256-bit integer type* SIGNED INTEGER256 (alternatively: INTEGER256, INT256) with binary precision of 256 bits and with scale 0 (zero).
- The *signed regular integer type* SIGNED INTEGER (alternatively: INTEGER, INT) with implementation-defined binary precision greater than or equal to 32 bits and with scale zero (0).
- The *signed small integer type* SMALLINT with implementation-defined binary precision less than or equal to the precision of the signed regular integer type SIGNED INTEGER and with scale zero (0).
- The *signed big integer type* BIGINT with implementation-defined binary precision greater than or equal to the precision of the signed regular integer type SIGNED INTEGER and with scale zero (0).
- The *signed user-specified integer types* SIGNED INTEGER(*p*) (alternatively: INTEGER(*p*), INT(*p*)) with implementation-defined binary precision greater than or equal to *p* bits and with scale zero (0).

The unsigned binary exact numeric types are:

- The *unsigned 8-bit integer type* UNSIGNED INTEGER8 (alternatively: UINT8) with binary precision of 8 bits and with scale 0 (zero).
- The *unsigned 16-bit integer type* UNSIGNED INTEGER16 (alternatively: UINT16) with binary precision of 16 bits and with scale 0 (zero).
- The *unsigned 32-bit integer type* UNSIGNED INTEGER32 (alternatively: UINT32) with binary precision of 32 bits and with scale 0 (zero).
- The *unsigned 64-bit integer type* UNSIGNED INTEGER64 (alternatively: UINT64) with binary precision of 64 bits and with scale 0 (zero).
- The *unsigned 128-bit integer type* UNSIGNED INTEGER128 (alternatively: UINT128) with binary precision of 128 bits and with scale 0 (zero).
- The *unsigned 256-bit integer type* UNSIGNED INTEGER256 (alternatively: UINT256) with binary precision of 256 bits and with scale 0 (zero).
- The *unsigned regular integer type* UNSIGNED INTEGER (alternatively: UINT) with implementation-defined binary precision greater than or equal to 32 bits and with scale zero (0).
- The *unsigned user-specified integer types* UNSIGNED INTEGER(*p*) (alternatively: UINT(*p*)) with implementation-defined binary precision greater than or equal to *p* bits and with scale zero (0).

4.17.3.4.4 Decimal exact numeric types

The decimal exact numeric types are:

- The *regular decimal exact numeric type* DECIMAL (alternatively: DEC) with implementation-defined decimal precision and with scale 0 (zero).
- The *user-specified decimal exact numeric types* DECIMAL(p) (alternatively: DEC(p)) with implementation-defined decimal precision greater than or equal to p digits and with scale 0 (zero).
- The user-specified decimal exact numeric types DECIMAL(p,s) (alternatively: DEC(p, s)) with implementation-defined decimal precision greater than or equal to p digits and with implementation-defined decimal scale of s digits.

4.17.3.4.5 Approximate numeric types

The approximate numeric types are:

- The *16-bit approximate numeric type* FLOAT16 with binary precision of 11 bits and with binary scale of 5 bits.

NOTE 66 — The material value space of the 16-bit approximate numeric type FLOAT16 is defined to be compatible with the binary16 interchange format of IEEE Std 754:2019.
- The *32-bit approximate numeric type* FLOAT32 with binary precision of 24 bits and with binary scale of 8 bits.

NOTE 67 — The material value space of the 32-bit approximate numeric type FLOAT32 is defined to be compatible with the binary32 interchange format of IEEE Std 754:2019.
- The *64-bit approximate numeric type* FLOAT64 with binary precision of 53 bits and with binary scale of 11 bits.

NOTE 68 — The material value space of the 64-bit approximate numeric type FLOAT64 is defined to be compatible with the binary64 interchange format of IEEE Std 754:2019.
- The *128-bit approximate numeric type* FLOAT128 with binary precision of 113 bits and with binary scale of 15 bits.

NOTE 69 — The material value space of the 128-bit approximate numeric type FLOAT128 is defined to be compatible with the binary128 interchange format of IEEE Std 754:2019.
- The *256-bit approximate numeric type* FLOAT256 with binary precision of 237 bits and with binary scale of 18 bits.

NOTE 70 — The material value space of the 256-bit approximate numeric type FLOAT256 is defined to be compatible with the binary256 interchange format of IEEE Std 754:2019.
- The *regular approximate numeric type* FLOAT with implementation-defined binary precision and with implementation-defined binary scale.
- The *real approximate numeric type* REAL with implementation-defined binary precision less than or equal to the precision of the regular approximate numeric type FLOAT and with implementation-defined binary scale.
- The *double approximate numeric type* DOUBLE (alternatively: DOUBLE PRECISION) with implementation-defined binary precision greater than or equal to the precision of the regular approximate numeric type FLOAT and with implementation-defined binary scale.
- The *user-specified approximate numeric types* FLOAT(p) with implementation-defined binary precision greater than or equal to p bits and with implementation-defined binary scale.

4.17 Value types

- The user-specified approximate numeric types FLOAT(p, s) with implementation-defined binary precision greater than or equal to p bits and with implementation-defined binary scale greater than or equal to s bits.

4.17.3.5 Temporal types

There are two classes of temporal types, temporal instant types and temporal duration.

The time scale used for temporal instant types shall be **UTC-SLS**.

The temporal instant types are:

1) DATETIME

An instant capturing the date, the time, and the time zone.

NOTE 71 — Equivalent to TIMESTAMP WITH TIME ZONE with nanosecond precision in SQL.

2) LOCALDATETIME

An instant capturing the date and the time, but not the time zone.

NOTE 72 — Equivalent to TIMESTAMP WITHOUT TIME ZONE with nanosecond precision in SQL.

3) DATE

An instant capturing the date, but not the time, nor the time zone.

NOTE 73 — Equivalent to DATE in SQL.

4) TIME

An instant capturing the time of day, and the time zone offset, but not the date.

NOTE 74 — Equivalent to TIME WITH TIME ZONE with nanosecond precision in SQL.

5) LOCALTIME

An instant capturing the time of day, but not the date, nor the time zone.

NOTE 75 — Equivalent to TIME WITHOUT TIME ZONE with nanosecond precision in SQL.

The temporal duration type is:

1) DURATION

A temporal amount. This captures the difference in time between two instants. It only captures the amount of time between two instants, it does not capture a start time and end time. A Duration can be negative.

A Duration captures time difference in a few different logical units. These units divide into three groups where conversion of units is possible within a group but not between groups (other than through applying the Duration to a point in time) as follows:

a) Month-based units: months, quarters, years

b) Day-based units: days, weeks

c) Time-based units: hours, minutes, seconds, and subseconds (milliseconds, microseconds, nanoseconds)

NOTE 76 — This has some similarity to INTERVAL in SQL.

4.18 Sites and operations on sites

4.18.1 Sites

A *site* is a place that holds an instance of a specified data type. Every site has a defined degree of persistence, independent of its data type. A site that exists until deliberately destroyed is said to be *persistent*. A site that necessarily ceases to exist on completion of a statement, at the end of a GQL-transaction, or at the end of a GQL-session is said to be *temporary*. A site that exists only for as long as necessary to hold an argument or returned value is said to be *transient*.

4.18.2 Assignment

The instance at a site can be changed by the operation of *assignment*. Assignment replaces the instance at a site with a new (possibly different) instance.

Assignment has no effect on any reference values targeting a site, if any exist.

4.18.3 Nullability

Every site has a *nullability characteristic* that indicates whether it may contain the null value (is *possibly nullable*) or not (is *known not nullable*). In GQL, if a site has a declared type, it is aligned with and specifies the nullability characteristics of the site.

5 Notation and conventions

5.1 Notation taken from ISO/IEC 10646

The notation for the representation of UCS code points is defined in [ISO/IEC 10646](#), Subclause 6.5, “Short identifiers for characters”.

In this document, this notation is used only to unambiguously identify characters and is not meant to imply a specific encoding for any GQL-implementation’s use of that character.

5.2 Notation

The syntactic notation used in this document is an extended version of BNF (“Backus Normal Form” or “Backus Naur Form”).

In a BNF language definition, each syntactic element, known as a *BNF non-terminal symbol*, of the language is defined by means of a *production rule*. This defines the element in terms of a formula consisting of the characters, character strings, and syntactic elements that can be used to form an instance of it.

In the version of BNF used in this document, the following symbols have the meanings shown in [Table 1](#), “Symbols used in BNF”.

Table 1 — Symbols used in BNF

Symbol	Meaning
< >	A character string enclosed in angle brackets is the name of a syntactic element (BNF non-terminal) of the GQL language.
: :=	The definition operator is used in a production rule to separate the element defined by the rule from its definition. The element being defined appears to the left of the operator and the formula that defines the element appears to the right.
[]	Square brackets indicate optional elements in a formula. The portion of the formula within the brackets may be explicitly specified or may be omitted.
{ }	Braces group elements in a formula. The portion of the formula within the braces shall be explicitly specified.
	The alternative operator. The vertical bar indicates that the portion of the formula following the bar is an alternative to the portion preceding the bar. If the vertical bar appears at a position where it is not enclosed in braces or square brackets, it specifies a complete alternative for the element defined by the production rule. If the vertical bar appears in a portion of a formula enclosed in braces or square brackets, it specifies alternatives for the contents of the innermost pair of such braces or brackets.

Symbol	Meaning
...	The ellipsis indicates that the element to which it applies in a formula may be repeated any number of times. If the ellipsis appears immediately after a closing brace "}", then it applies to the portion of the formula enclosed between that closing brace and the corresponding opening brace "{". If an ellipsis appears after any other element, then it applies only to that element. In Syntax Rules, General Rules, and Conformance Rules, a reference to the n -th element in such a list assumes the order in which these are specified, unless otherwise stated.
!!	Introduces normal English text. This is used when the definition of a syntactic element is not expressed in BNF.

Whitespace is used to separate syntactic elements. Multiple whitespace characters are treated as a single space. Apart from those symbols to which special functions were given above, other characters and character strings in a formula stand for themselves. In addition, if the symbols to the right of the definition operator in a production consist entirely of BNF symbols, then those symbols stand for themselves and do not take on their special meaning.

Pairs of braces and square brackets may be nested to any depth, and the alternative operator may appear at any depth within such a nest.

A *predicative production rule* is a production rule that defines a syntactic element that is not only defined in terms of the production rule itself but also in terms of further restrictions imposed on it by additional Syntax Rules such that the production rule cannot sufficiently distinguish the syntactic element from the syntactic elements defined by another production rule in an alternative between those two production rules without considering those restrictions.

Production rules are always explicitly annotated with normal English text if they are predicative production rules. For example, in the production:

```
<predicative rule> ::=  
  !! Predicative production rule.  
  <defining rule>  
  
<defining rule> ::=  
  ...
```

In this example, `<predicative rule>` is a predicative production rule that is defined by the specified `<defining rule>` but is declared as requiring the application of additional Syntax Rules not shown here to be able to distinguish it syntactically from other production rules in an alternative.

A character string that forms an instance of any syntactic element may be generated from the BNF definition of that syntactic element by application of the following steps:

- 1) Select any one option from those defined in the right hand side of a production rule for the element, and replace the element with this option.
- 2) Replace each ellipsis and the object to which it applies with one or more instances of that object.
- 3) For every portion of the character string enclosed in square brackets, either delete the brackets and their contents or change the brackets to braces.
- 4) For every portion of the character string enclosed in braces, apply Step 1) through Step 5) to the substring between the braces, then remove the braces.
- 5) Apply steps Step 1) through Step 5) to any BNF non-terminal symbol that remains in the character string.

The expansion or production is complete when no further non-terminal symbols remain in the character string.

The left normal form derivation of a character string CS in the source language character set from a BNF non-terminal NT is obtained by applying Step 1) through Step 5) above to NT , always selecting in Step 5) the leftmost BNF non-terminal.

5.3 Conventions

5.3.1 Specification of syntactic elements

Syntactic elements are specified in terms of:

- **Function:** A short statement of the purpose of the element.
- **Format:** A BNF definition of the syntax of the element.
- **Syntax Rules:** A specification in English of the syntactic properties of the element. These include rules for the following aspects that are specified in the sequence given:
 - 1) the expansion of syntactic short-hand forms,
 - 2) the normalization of syntactic forms,
 - 3) additional syntactic constraints, not expressed in BNF, that the element shall satisfy,
 - 4) the visibility or scope of identifiers, and
 - 5) specifying or defining the type of elements.
- **General Rules:** A specification in English of the run-time effect of the element. Where more than one General Rule is used to specify the effect of an element, the required effect is that which would be obtained by beginning with the first General Rule and applying the Rules in numeric sequence unless a Rule is applied that specifies or implies a change in sequence or termination of the application of the Rules. Unless otherwise specified or implied by a specific Rule that is applied, application of General Rules terminates when the last in the sequence has been applied.
- **Conformance Rules:** A specification of how the element shall be supported for conformance to GQL. Conformance Rules are effectively a kind of Syntax Rule, differentiated only because of their use to specify conformance to the GQL language. Conformance Rules in a given Subclause are therefore always applied concurrently with Syntax Rules of that Subclause.

The scope of notational symbols is the Subclause in which those symbols are defined. Within a Subclause, the symbols defined in Syntax Rules, or General Rules can be referenced in other rules provided that they are defined before being referenced.

5.3.2 Specification of the Information Graph Schema

The objects of the Information Graph Schema in this document are specified in terms of:

- **Function:** A short statement of the purpose of the definition.
- **Definition:** A definition, in the GQL language, of the object being defined.
- **Description:** A specification of the run-time value of the object, to the extent that this is not clear from the definition.
- **Population:** A specification of the elements of the graph that a GQL-implementation shall provide that are not specified in the General Rules of other Subclauses.
- **Conformance Rules:** A specification of how the element shall be supported for conformance to GQL.

The only purpose of the Information Graph Schema is to provide access to the definitions contained in the schema. The actual objects on which the Information Graph Schema is based are implementation-dependent.

5.3.3 Use of terms

5.3.3.1 Syntactic containment

Let $\langle A \rangle$, $\langle B \rangle$, and $\langle C \rangle$ be syntactic elements; let A_1 , B_1 , and C_1 respectively be instances of $\langle A \rangle$, $\langle B \rangle$, and $\langle C \rangle$.

In a Format, $\langle A \rangle$ is said to *immediately contain* $\langle B \rangle$ if $\langle B \rangle$ appears on the right-hand side of the BNF production rule for $\langle A \rangle$. An $\langle A \rangle$ is said to *contain* or *specify* $\langle C \rangle$ if $\langle A \rangle$ immediately contains $\langle C \rangle$ or if $\langle A \rangle$ immediately contains a $\langle B \rangle$ that contains $\langle C \rangle$.

In GQL language, A_1 is said to *immediately contain* B_1 if $\langle A \rangle$ immediately contains $\langle B \rangle$ and B_1 is part of the text of A_1 . A_1 is said to *contain* or *specify* C_1 if A_1 immediately contains C_1 or if A_1 immediately contains B_1 and B_1 contains C_1 . If A_1 contains C_1 , then C_1 is *contained in* A_1 and C_1 is *specified by* A_1 .

A_1 is said to contain B_1 *with an intervening instance of* $\langle C \rangle$ if A_1 contains B_1 and A_1 contains an instance of $\langle C \rangle$ that contains B_1 . A_1 is said to contain B_1 *without an intervening instance of* $\langle C \rangle$ if A_1 contains B_1 and A_1 does not contain an instance of $\langle C \rangle$ that contains B_1 .

A_1 *simply contains* B_1 if A_1 contains B_1 without an intervening instance of $\langle A \rangle$ or an intervening instance of $\langle B \rangle$. If A_1 *simply contains* B_1 , then B_1 is *simply contained in* A_1 .

If an instance of $\langle A \rangle$ contains an instance of $\langle B \rangle$, then $\langle B \rangle$ is said to be *contained in* $\langle A \rangle$ and $\langle A \rangle$ is said to be a *containing* production symbol for $\langle B \rangle$. If an instance of $\langle A \rangle$ simply contains an instance of $\langle B \rangle$, then $\langle B \rangle$ is said to be *simply contained in* $\langle A \rangle$ and $\langle A \rangle$ is said to be a *simply containing* production symbol for $\langle B \rangle$.

A_1 is the *innermost* $\langle A \rangle$ satisfying a condition C if A_1 satisfies C and A_1 does not contain an instance of $\langle A \rangle$ that satisfies C . A_1 is the *outermost* $\langle A \rangle$ satisfying a condition C if A_1 satisfies C and A_1 is not contained in an instance of $\langle A \rangle$ that satisfies C .

In a Format, the verb “to be” (including all its grammatical variants, such as “is”) is defined as follows: $\langle A \rangle$ is said to be $\langle B \rangle$ if there exists a BNF production rule of the form $\langle A \rangle ::= \langle B \rangle$. If $\langle A \rangle$ is $\langle B \rangle$ and $\langle B \rangle$ is $\langle C \rangle$, then $\langle A \rangle$ is $\langle C \rangle$. If $\langle A \rangle$ is $\langle C \rangle$, then $\langle C \rangle$ is said to *constitute* $\langle A \rangle$. In GQL language, A_1 is said to be B_1 if $\langle A \rangle$ is $\langle B \rangle$ and the text of A_1 is the text of B_1 . Conversely, B_1 is said to *constitute* A_1 if A_1 is B_1 .

5.3.3.2 Terms denoting rule requirements

In the Syntax Rules, the term *shall* defines conditions that are required to be true of syntactically conforming GQL language. When such conditions depend on the contents of one or more schemas, they are required to be true just before the actions specified by the General Rules are performed. The treatment of language that does not conform to the Formats and Syntax Rules is implementation-defined. If any condition required by Syntax Rules is not satisfied when the evaluation of General Rules is attempted and the GQL-implementation is neither processing non-conforming GQL language nor processing conforming GQL language in a non-conforming manner, then an exception condition is raised as specified in Subclause 4.10.2, “Conditions”.

In the Conformance Rules, the term *shall* defines conditions that are required to be satisfied if the named Feature is or Features are not supported.

5.3.3.3 Rule evaluation order

A conforming GQL-implementation is not required to perform the exact sequence of actions defined in the General Rules, provided its effect on GQL-data and the GQL-catalog is identical to the effect of that

5.3 Conventions

sequence. The term *effectively* is used to emphasize actions whose effect may be achieved in other ways by a GQL-implementation.

The Syntax Rules and Conformance Rules for contained syntactic elements are effectively applied at the same time as the Syntax Rules and Conformance Rules for the containing syntactic elements. The General Rules for contained syntactic elements are effectively applied before the General Rules for the containing syntactic elements.

Where the precedence of operators is determined by the Formats of this document or by parentheses, those operators are effectively applied in the order specified by that precedence.

Where the precedence is not determined by the Formats or by parentheses, effective evaluation of expressions is *generally* performed from left to right. However, it is implementation-dependent whether expressions are *actually* evaluated left to right, particularly when the evaluation of operands or operators causes conditions to be raised or if the results of the expressions may be determined without completely evaluating all parts of the expression.

If some syntactic element contains more than one other syntactic element, then the General Rules for contained elements that appear earlier in the production for the containing syntactic element are applied before the General Rules for contained elements that appear later.

For example, in the production:

```
<A> ::=  
    <B> <C>  
  
<B> ::=  
    ...  
  
<C> ::=  
    ...
```

the Syntax Rules and Conformance Rules for *<A>*, **, and *<C>* are effectively applied simultaneously. The General Rules for ** are applied before the General Rules for *<C>*, and the General Rules for *<A>* are applied after the General Rules for both ** and *<C>*.

An exception to this rule is when the General Rules of the containing syntactic element explicitly states when the General Rules of the contained syntactic element are to be evaluated.

NOTE 77 — In this document these exceptions are shown by the presence of a General Rule of the form “The General Rules of *xxx* are evaluated.” where *xxx* is a BNF non-terminal. This indicates that *xxx* is evaluated at this point and not at the point implied by the general “Rule evaluation order”.

If the result of an expression or search condition is not dependent on the result of some part of that expression or search condition, then that part of the expression or search condition is said to be *inessential*.

If evaluation of an inessential part would cause an exception condition to be raised, then it is implementation-dependent whether or not that exception condition is raised.

During the computation of the result of an expression, the GQL-implementation may produce one or more *intermediate results* that are used in determining that result. The declared type of a site that contains an intermediate result is implementation-dependent.

5.3.3.4 Conditional rules

A conditional rule is specified with “If” or “Case” conventions. A rule specified with “Case” conventions includes a list of conditional subrules using “If” conventions. The first such “If” subrule whose condition is true is the effective subrule of the “Case” rule. The last subrule of a “Case” rule may specify “Otherwise”, in which case it is the effective subrule of the “Case” rule if no preceding “If” subrule in the “Case” rule is satisfied. If the last subrule does not specify “Otherwise”, and if there is no subrule whose condition is true, then there is no effective subrule of the “Case” rule.

5.3.3.5 Syntactic substitution

In the Syntax and General Rules, the phrase “*X* is implicit” indicates that the Syntax and General Rules are to be interpreted as if the element *X* had actually been specified. Within the Syntax Rules of a given Subclause, it is known whether the element was explicitly specified or is implicit.

In the Syntax and General Rules, the phrase “the following $\langle A \rangle$ is implicit: *Y*” indicates that the Syntax and General Rules are to be interpreted as if a syntactic element $\langle A \rangle$ containing *Y* had actually been specified.

In the Syntax Rules and General Rules, the phrase “*former* is equivalent to *latter*” indicates that the Syntax Rules and General Rules are to be interpreted as if all instances of *former* in the element had been instances of *latter*.

If a BNF non-terminal is referenced in a Subclause without specifying how it is contained in a BNF production that the Subclause defines, then

Case:

- If the BNF non-terminal is itself defined in the Subclause, then the reference shall be assumed to be to the occurrence of that BNF non-terminal on the left side of the defining production.
- Otherwise, the reference shall be assumed to be to a BNF production in which the particular BNF non-terminal is immediately contained.

5.3.4 Descriptors

A *descriptor* is a coded description of a [GQL-object](#) that defines the *metadata* of a [GQL-object](#) of a specified type. The concept of descriptor is used in specifying the semantics of GQL. It is not necessary that any descriptor exist in any particular form in any GQL-environment.

Some GQL-objects cannot exist except in the context of other GQL-objects. For example, nodes cannot exist except within the context of graphs. Each such object is independently described by its own descriptor, and the descriptor of an enabling object (e.g., graph) is said to *include* the descriptor of each enabled object (e.g., node or edge). Conversely, the descriptor of an enabled object is said to *be included in* the descriptor of an enabling object. The descriptor of some object *A* *generally includes* the descriptor of some object *C* if the descriptor of *A* includes the descriptor of some object *B* and the descriptor of *B* generally includes the descriptor of *C*.

In other cases, certain GQL-objects cannot exist unless some other GQL-object exists, even though there is no inclusion relationship. In general, a descriptor *D*₁ can be said to depend on, or to be dependent on, some descriptor *D*₂.

The descriptor of some object *A* *generally depends on* or *is generally dependent on* the descriptor of some object *C* if the descriptor of *A* depends on the descriptor of some object *B* and the descriptor of *B* generally depends on the descriptor of *C*.

The execution of a statement may result in the creation of many descriptors. A GQL-object that is created as a result of a statement may depend on other descriptors that are only created as a result of the execution of that statement.

There are two ways of indicating dependency of one GQL-object on another. In many cases, the descriptor of the dependent GQL-object is said to “include the name of” the GQL-object on which it is dependent. In this case “the name of” is to be understood as meaning “sufficient information to identify the descriptor of”. Alternatively, the descriptor of the dependent GQL-object can be said to include text of the GQL-object on which it is dependent. However, in such cases, whether the GQL-implementation includes actual text (with defaults and implications made explicit) or its own style of parse tree is irrelevant; the validity of the descriptor is clearly “dependent on” the existence of descriptors of objects that are referenced in it.

An attempt to destroy a GQL-object, and hence its descriptor, may fail if other descriptors are dependent on it, depending on how the destruction is specified. Such an attempt may also fail if the descriptor to be destroyed is included in some other descriptor. Destruction of a descriptor results in the destruction of all descriptors included in it, but has no effect on descriptors on which it is dependent.

The implementation of some GQL-objects described by descriptors requires the existence of objects not specified by this document. Where such objects are required, they are effectively created whenever the associated descriptor is created and effectively destroyed whenever the associated descriptor is destroyed.

5.3.5 Subclauses used as subroutines

In this document, some Subclauses are defined without explicit syntax to invoke their semantics. Such Subclauses, called subroutine Subclauses, typically factor out rules that are required by one or more other Subclauses and are intended to be invoked by the rules of those other Subclauses.

In other words, the rules of these Subclauses behave as though they were a sort of definitional “subroutine” that is invoked by other Subclauses. These subroutine Subclauses are typically specified in a manner that requires information to be passed to them from their invokers. The information that is required to be passed is represented as parameters of these subroutine Subclauses, and that information is required to be passed in the form of arguments provided by the invokers of these subroutine Subclauses.

Every invocation of a subroutine Subclause shall explicitly provide information for every required parameter of the subroutine Subclause being invoked.

NOTE 78 — In this document the invocation will occur in the form “The xxx Rules of Subclause *n.n*, “*aaaaaaa*”, with ... ”.

5.3.6 Index typography

In the Indexes to this document, the following conventions are used:

- An index entry in **boldface** indicates the page where the word, phrase, or BNF non-terminal is defined.
- An index entry in *italics* indicates a page where the BNF non-terminal is used in a Format.
- An index entry in neither boldface nor italics indicates a page where the word, phrase, or BNF non-terminal is not defined, but is used other than in a Format (for example, in a heading, Function, Syntax Rule, General Rule, Conformance Rule, Table, or other descriptive text).

5.3.7 Feature ID and Feature Name

Features are either *standard-defined features* or *implementation-defined features*.

Standard-defined features are defined in this document. Implementation-defined features are defined by GQL-implementations (see [Subclause 25.5.3, “Extensions and options”](#)).

Features are referenced by a Feature ID and by a Feature Name. A Feature ID value comprises a letter and three digits.

Feature IDs whose letter is “V” are reserved for implementation-defined features.

Standard-defined features are optional features that are defined by implicit or explicit Conformance Rules. An implicit Conformance Rule is any normative statement that begins “Without Feature *nnn*.”. The Feature ID of a standard-defined feature is stable and can be depended on to remain constant.

For convenience, all of the features defined in this document are collected together in a non-normative annex.

6 GQL-requests

6.1 <GQL-request>

Function

Define a GQL-request.

Format

```
<GQL-request> ::=  
[ <request parameter set> ] <GQL-program>
```

Syntax Rules

- 1) If <request parameter set> is not specified, then a, possibly empty, <request parameter set> shall be effectively provided in an implementation-defined manner.

General Rules

- 1) Let RPS be the <request parameter set>. Let PR be the <GQL-program>. Let P be the principal identified by the authorization identifier AI associated with the GQL-agent on whose behalf the GQL-request was submitted.
- 2) Case:
 - a) If no GQL-session has been established for the GQL-client, then let S be a new GQL-session with an associated session context SCX that is initialized as follows:
 - i) Set the authorization identifier of SCX to AI .
 - ii) Set the principal of SCX to P .
 - iii) Set the time zone identifier of SCX to an implementation-defined default time zone identifier for S .
 - iv) Set the session schema of SCX to the current home schema.
NOTE 79 — If no current home schema is set, no session schema is set to SCX .
 - v) Set the session graph of SCX to the current home graph.
 - vi) Set the session parameters of SCX to the implementation-defined session default parameters for S .
NOTE 80 — The scope of the session parameters is defined in 4th paragraph of Subclause 4.11.4.5, “Scope of names”.
 - vii) Set the session parameter flags of SCX to the implementation-defined session default parameter flags for S .
 - viii) Set no transaction for SCX .
 - ix) Set no request context for SCX .

- x) Set the termination flag of *SCX* to *False*.
 - b) Otherwise, *S* is the GQL-session that has already been established for the GQL-client and *SCX* is the session context associated with *S*.
- 3) Let *RCX* be a new GQL-request context that is initialized as follows:
- a) Set the execution stack of *RCX* to a new empty execution stack.
 - b) Set the request outcome of *RCX* to a successful outcome with an omitted result.
- 4) Set the current request context to *RCX*.
- 5) Push a new execution context *CTX* onto the current execution stack, initialized as follows:
- a) Set the working schema of *CTX* to the current session schema.
NOTE 81 — If no current session schema is set, no working schema is set to *CTX*.
 - b) Set the working graph of *CTX* to the current session graph.
 - c) Set the working table of *CTX* to a new empty binding table.
 - d) Set the working record of *CTX* to a new empty record.
 - e) Set the execution outcome of *CTX* to a successful outcome with an omitted result.
NOTE 82 — *CTX* becomes the current execution context.
- 6) The General Rules of *RPS* are evaluated.
- 7) The General Rules of *PR* are evaluated.
- 8) Set the current request outcome to the execution outcome of *CTX*.
- 9) Pop the current execution context.
NOTE 83 — This leaves the execution stack empty.
- 10) If the current termination flag is set to *True* and the current transaction is active, then:
- a) The following statement is implicitly executed:
ROLLBACK
 - b) Set a failed outcome as the current request outcome.
- 11) Return the current request outcome to the GQL-client for delivery to the GQL-agent.
- 12) Set no current request context.
- 13) If the current termination flag is set to *True*, then close *S* by disassociating it from the GQL-client and destroying *SCX*.

Conformance Rules

- 1) Without Feature GB10, “Explicit request parameters”, conforming GQL language shall not contain <request parameter set>.
NOTE 84 — This feature only controls the provision of a textual representation of a <request parameter set>. A conforming implementation is required to provide the possibility of specifying a set of request parameters by some means.

6.2 <request parameter set>

Function

Define a set of request parameters.

Format

```
<request parameter set> ::=  
  <request parameter> [ { <comma> <request parameter> }... ]  
  
<request parameter> ::=  
  <parameter definition>
```

Syntax Rules

- 1) The scope of a <request parameter> is the containing <GQL-request>.

NOTE 85 — See 4th paragraph of Subclause 4.11.4.5, “Scope of names” for the interaction between the scopes of <request parameter> and <session parameter>.

General Rules

- 1) Set the request parameters of the current request context to the request parameters of <request parameter set>.

Conformance Rules

None.

6.3 <GQL-program>

Function

Define a GQL-program.

Format

```
<GQL-program> ::=  
  [ <preamble> ] <main activity>  
  
<main activity> ::=  
  <session activity>  
  | [ <session activity> ] { <transaction activity> [ <session activity> ] }...  
  | <session close command>  
  | <session close command>  
  
<session activity> ::=  
  <session clear command> [ <session parameter command>... ]  
  | <session parameter command>...  
  
<session parameter command> ::=  
  <session set command>  
  | <session remove command>  
  
<transaction activity> ::=  
  <start transaction command>  
  | [ <procedure specification> [ <end transaction command> ] ]  
  | <procedure specification> [ <end transaction command> ]  
  | <end transaction command>
```

Syntax Rules

- 1) If no <preamble> is specified, then an implementation-defined <preamble> is implicit.

General Rules

None.

Conformance Rules

None.

6.4 <preamble>

Function

Define the preamble to a GQL-program.

Format

```
<preamble> ::=  
  <preamble option> [ { <comma> <preamble option> }... ]  
  
<preamble option> ::=  
  PROFILE  
  | EXPLAIN  
  | <preamble option identifier> [ <equals operator> <literal> ]  
  
<preamble option identifier> ::=  
  <identifier>
```

NOTE 86 — The syntax of the preamble has been intentionally defined using forward-looking syntax such that the preamble can be parsed greedily and separately before parsing the GQL-request in order to allow the preamble to influence the parsing of the procedure.

Syntax Rules

- 1) <preamble option identifier> shall be an implementation-defined <identifier> that is not a <key word>.

General Rules

- 1) Set the request preamble of the current request context to <preamble>.
- 2) If PROFILE is specified, then the GQL-server records and reports profiling information during the execution of the GQL-request.
- 3) The format, contents and means of reporting the profiling information are implementation-defined.
- 4) If EXPLAIN is specified, then the GQL-server does not execute the GQL-request but reports a description of how it would have executed the GQL-request if EXPLAIN had not been specified, i.e., return a query plan.
- 5) The format, contents and means of reporting the query plan are implementation-defined.
- 6) The effect of other <preamble option>s is implementation-defined.

Conformance Rules

None.

7 Session management

7.1 <session set command>

Function

Set values in the session context.

Format

```

<session set command> ::==
  SESSION SET {
    <session set schema clause>
  | <session set graph clause>
  | <session set time zone clause>
  | <session set parameter clause>
  }

<session set schema clause> ::==
  SCHEMA <schema reference>

<session set graph clause> ::==
  <graph resolution expression>

<session set time zone clause> ::==
  TIME ZONE <set time zone value>

<set time zone value> ::==
  <string value expression>

<session set parameter clause> ::==
  [ <session parameter flag> ] <session parameter> [ IF NOT EXISTS ]

<session parameter> ::==
  [ PARAMETER ] <parameter definition>

<session parameter flag> ::=
  MUTABLE | FINAL

```

Syntax Rules

- 1) If the <session set schema clause> SSSC is specified, then let S be the schema that is the result of the <schema reference> immediately contained in SSSC.
- 2) If <session set parameter clause> does not immediately contain <session parameter flag>, then MUTABLE is implicit.
- 3) The scope of <session parameter> is defined in 4th paragraph of Subclause 4.11.4.5, "Scope of names".

General Rules

- 1) If the <session set schema clause> is specified, then:

- a) Set the current session schema to S .
- b) Set the current execution outcome to a successful outcome with an omitted result.
- 2) If a <session set graph clause> $SSGC$ is specified, then:
 - a) Let GRE be the <graph resolution expression> that is immediately contained in $SSGC$.
 - b) Set the current session graph to the result of GRE .
 - c) Set the current execution outcome to a successful outcome with an omitted result.
- 3) If the <string value expression> immediately contained in <set time zone value>s do not conform to the representation specified in clause 4.3 “Time scale components and units” of ISO 8601-1:2019 with, the optional addition of an IANA Time Zone Database time zone designator enclosed between a <left bracket> and a <right bracket> or, if both an IANA Time Zone Database time zone designator and a ISO 8601-1:2019 time shift specification are present and the IANA Time Zone Database time zone designator does not resolve to the same value as the ISO 8601-1:2019 time shift specification then an exception is raised: *data exception — invalid time zone displacement value (22009)*.
- 4) If a <session set parameter clause> $SSPC$ is specified, then:
 - a) Let PD be the <parameter definition> contained in $SSPC$.
 - b) Let $NAME$ be the <parameter name> simply specified by PD .
 - c) Case:
 - i) If $SSPC$ immediately contains IF NOT EXISTS but a current session parameter with name $NAME$ exists, then do nothing.
 - ii) If the current session parameter with name $NAME$ is set with the session parameter flag FINAL, then an exception condition is raised: *invalid action — attempt to change constant value (G4001)*.
 - iii) Otherwise:
 - 1) The General Rules for PD are evaluated.
 - 2) Set the current session parameter with name $NAME$.
 - 3) Set the current session parameter flag with $NAME$ to the <session parameter flag> specified by $SSPC$.
 - d) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

7.2 <session remove command>

Function

Forget previously declared session parameters.

Format

```
<session remove command> ::=  
[ SESSION ] REMOVE <parameter> [ IF EXISTS ]
```

Syntax Rules

None.

General Rules

- 1) Let *SRC* be the <session remove command>.
- 2) Let *P* be the <parameter> immediately contained in *SRC*.
- 3) Let *PN* be the <parameter name> immediately contained in *P*.
- 4) Case:
 - a) If *PN* is the name of a current session parameter, then:
 - Case:
 - i) If its associated current session parameter flag is FINAL, then an exception condition is raised: *data exception — attempt to modify a FINAL parameter (22G0E)*.
 - ii) Otherwise, remove the current session parameter with the name *PN* and remove its associated current session parameter flag.
 - b) Otherwise, if IF EXISTS is not specified, then an exception condition is raised: *data exception — invalid session parameter name (22G0A)*.
 - 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

7.3 <session clear command>

Function

Clear the current session by forgetting previously declared session parameters.

Format

```
<session clear command> ::=  
[ SESSION ] CLEAR
```

Syntax Rules

None.

General Rules

- 1) For each current session parameter with name *CSPN* for which there is an associated current session parameter flag with name *CSPN* that are not FINAL, remove both the current session parameter with name *CSPN* and its associated current session parameter flag with name *CSPN*.
- 2) For each implementation-defined default session parameter *DSP* for which there is an associated implementation-defined default session parameter flag *DSPF* with the same name as *DSP* that would be present in a newly created session context as specified by [Subclause 6.1, “<GQL-request>”, General Rule 2\)a\)vi\)](#) but for which there is no current session parameter with the same name as *DSP*, add *DSP* to the current session parameters and add *DSPF* to the current session parameter flags.
- 3) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

7.4 <session close command>

Function

Close the current session.

Format

```
<session close command> ::=  
  [ SESSION ] CLOSE
```

Syntax Rules

None.

General Rules

- 1) Set the current termination flag to true.

Conformance Rules

None.

8 Transaction management

8.1 <start transaction command>

Function

Start a new GQL-transaction and set its characteristics.

Format

```
<start transaction command> ::=  
    START TRANSACTION [ <transaction characteristics> ]
```

Syntax Rules

- 1) If <transaction characteristics> is omitted, then READ WRITE is implicit.

General Rules

- 1) If a GQL-transaction is currently active, then an exception condition is raised: *invalid transaction state — active GQL-transaction (25G01)*.
- 2) A new GQL-transaction *TX* is initiated.
- 3) The current transaction is set to *TX*.

NOTE 87 — This determines *TX* as the active GQL-transaction associated with the GQL-session of the currently executing GQL-request

Conformance Rules

None.

8.2 <end transaction command>

Function

Terminate a GQL-transaction.

Format

```
<end transaction command> ::=  
    <commit command>  
  | <rollback command>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

8.3 <transaction characteristics>

Function

Specify transaction characteristics.

Format

```
<transaction characteristics> ::=  
  <transaction mode> [ { <comma> <transaction mode> }... ]  
  
<transaction mode> ::=  
  <transaction access mode>  
  | <implementation-defined access mode>  
  
<transaction access mode> ::=  
  READ ONLY  
  | READ WRITE  
  
<implementation-defined access mode> ::=  
  !! See the Syntax Rules.
```

Syntax Rules

- 1) Let TC be the <transaction characteristics>.
- 2) TC shall contain at most one <transaction access mode>.
- 3) If TC does not contain a <transaction access mode>, then READ WRITE is implicit.
- 4) The Format and Syntax Rules for <implementation-defined access mode> are implementation-defined.

General Rules

None.

Conformance Rules

None.

8.4 <rollback command>

Function

Terminate the current transaction with rollback.

Format

```
<rollback command> ::=  
    ROLLBACK
```

Syntax Rules

None.

General Rules

- 1) If the current transaction is part of an encompassing transaction that is controlled by an agent other than the GQL-agent and the <rollback command> is not being implicitly executed, then an exception condition is raised: *invalid transaction termination (2D000)*.
- 2) All changes to GQL-data or the GQL-catalog that were made by the current GQL-transaction are canceled.
- 3) The current transaction is terminated and set to no transaction.

NOTE 88 — This discards the active GQL-transaction associated with the GQL-session of the currently executing GQL-request.

Conformance Rules

None.

8.5 <commit command>

Function

Terminate the current transaction with commit.

Format

```
<commit command> ::=  
    COMMIT
```

Syntax Rules

None.

General Rules

- 1) If the current transaction is part of an encompassing transaction that is controlled by an agent other than the GQL-agent, then an exception condition is raised: *invalid transaction termination (2D000)*.
- 2) Case:
 - a) If any enforced constraint is not satisfied, then any changes to GQL-data or the GQL-catalog that were made by the current transaction are canceled and an exception condition is raised: *transaction rollback — integrity constraint violation (40002)*.
 - b) If any other error preventing commitment of the GQL-transaction has occurred, then any changes to GQL-data or the GQL-catalog that were made by the current transaction are canceled and an exception condition is raised: *transaction rollback (40000)* with an implementation-defined subclass value.
 - c) Otherwise, any changes to GQL-data or the GQL-catalog that were made by the current GQL-transaction are eligible to be perceived by all subsequent GQL-transactions.
- 3) The current transaction is terminated and set to no transaction.

NOTE 89 — This discards the active GQL-transaction associated with the GQL-session of the currently executing GQL-request.

Conformance Rules

None.

9 Procedures

9.1 <procedure specification>

Function

Define the procedural logic of a procedure.

Format

```

<nested procedure specification> ::==
  <left brace> <procedure specification> <right brace>

<procedure specification> ::==
  <catalog-modifying procedure specification>
  | <data-modifying procedure specification>
  | <query specification>
  | <function specification>

<nested catalog-modifying procedure specification> ::==
  <left brace> <catalog-modifying procedure specification> <right brace>

<catalog-modifying procedure specification> ::==
  !! Predicative production rule.
  <procedure body>

<nested data-modifying procedure specification> ::==
  <left brace> <data-modifying procedure specification> <right brace>

<data-modifying procedure specification> ::==
  !! Predicative production rule.
  <procedure body>

```

Syntax Rules

- 1) A <catalog-modifying procedure specification> shall be a <procedure body> PB such that every <statement> immediately contained in PB is a <catalog-modifying statement>.
- 2) If the <data-modifying procedure specification> $DMPS$ is specified, then:
 - a) Let PB be the <procedure body> immediately contained in $DMPS$.
 - b) PB shall immediately contain at least one <data-modifying statement>.
 - c) Every <statement> immediately contained in PB shall be either a <query statement> or a <data-modifying statement>
 - d) If PB contains a <focused linear query statement> or a <focused linear data-modifying statement> without an intervening <procedure body>, then PB shall not contain an <ambient linear query statement> or an <ambient linear data-modifying statement> without an intervening <procedure body>.
 - e) If PB contains an <ambient linear query statement> or an <ambient linear data-modifying statement> without an intervening <procedure body>, then PB shall not contain a <focused linear

query statement> or a <focused linear data-modifying statement> without an intervening <procedure body>.

General Rules

- 1) If a <nested procedure specification> *NPS* is specified, then the General Rules are applied to the immediately contained <procedure specification> *PS*. The outcome of executing *NPS* is the outcome of executing *PS* and the result of executing *NPS* is the result of executing *PS*.
- 2) If a <procedure specification> *PS* is specified, then:
 - a) If *PS* is immediately contained in a <transaction activity>, then a new child execution context *CONTEXT* is created.
 - b) If no GQL-transaction is active, then a new GQL-transaction *TX* is initiated and the current transaction is set to *TX*.

NOTE 90 — This determines *TX* as the active GQL-transaction associated with the GQL-session of the currently executing GQL-request.
 - c) Case:
 - i) If a <catalog-modifying procedure specification> *CPS* is immediately specified by *PS*, then the General Rules of Subclause 9.4, “<procedure body>” are applied to the <procedure body> immediately contained in *CPS*. The outcome of executing *PS* is the outcome of executing *CPS* and the result of executing *PS* is the result of executing *CPS*.
 - ii) If a <data-modifying procedure specification> *DPS* is immediately specified by *PS*, then the General Rules of Subclause 9.4, “<procedure body>” are applied to the <procedure body> immediately contained in *DPS*. The outcome of executing *PS* is the outcome of executing *DPS* and the result of executing *PS* is the result of executing *DPS*.
 - iii) If a <query specification> *QS* is immediately specified by *PS*, then the General Rules of Subclause 9.2, “<query specification>” are applied. The outcome of executing *PS* is the outcome of executing *QS* and the result of executing *PS* is the result of executing *QS*.
 - iv) If a <function specification> *FS* is immediately specified by *PS*, then the General Rules of Subclause 9.3, “<function specification>” are applied. The outcome of executing *PS* is the outcome of executing *FS* and the result of executing *PS* is the result of executing *FS*.
 - d) If *PS* is immediately contained in a <transaction activity>, then let *OUTCOME* be the current execution outcome available at the end of the application of these General Rules in *CONTEXT* before *CONTEXT* is implicitly popped and destroyed. Set the current execution outcome to *OUTCOME*.
- 3) If a <nested catalog-modifying procedure specification> *NCPS* is specified, then the General Rules of Subclause 9.1, “<procedure specification>” are applied to the <catalog-modifying procedure specification> *CPS* simply contained in *NCPS*. The outcome of executing *NCPS* is the outcome of executing *CPS* and the result of executing *NCPS* is the result of executing *CPS*.
- 4) If a <catalog-modifying procedure specification> *CPS* is specified, then the General Rules of Subclause 9.4, “<procedure body>” are applied to the <procedure body> *PB* immediately contained in *CPS*. The outcome of executing *CPS* is the outcome of executing *PB* and the result of executing *CPS* is the result of executing *PB*.
- 5) If a <nested data-modifying procedure specification> *NDPS* is specified, then the General Rules of Subclause 9.1, “<procedure specification>” are applied to the <data-modifying procedure specification> *DPS* simply contained in *NDPS*. The outcome of executing *NDPS* is the outcome of executing *DPS* and the result of executing *NDPS* is the result of executing *DPS*.

- 6) If a <data-modifying procedure specification> *DPS* is specified, then the General Rules of [Subclause 9.4](#), “[<procedure body>](#)” are applied to the <procedure body> *PB* immediately contained in *DPS*. The outcome of executing *DPS* is the outcome of executing *PB* and the result of executing *DPS* is the result of executing *PB*.

Conformance Rules

None.

9.2 <query specification>

Function

Define the procedural logic of a query.

Format

```
<nested query specification> ::=  
  <left brace> <query specification> <right brace>  
  
<query specification> ::=  
  !! Predicative production rule.  
  <procedure body>
```

Syntax Rules

- 1) Let *QS* be the <query specification>.
- 2) Let *PB* be the <procedure body> immediately contained in *QS*.
- 3) Every <statement> immediately contained in *PB* shall be a <query statement>.
- 4) If *PB* contains a <focused linear query statement> without an intervening <procedure body>, then *PB* shall not contain an <ambient linear query statement> without an intervening <procedure body>.
- 5) If *PB* contains an <ambient linear query statement> without an intervening <procedure body>, then *PB* shall not contain a <focused linear query statement> without an intervening <procedure body>.

General Rules

- 1) If a <nested query specification> *NQS* is specified, then the General Rules of Subclause 9.2, “<query specification>” are applied to the <query specification> *QPS* simply contained in *NQS*. The outcome of executing *NQS* is the outcome of executing *QS* and the result of executing *NQS* is the result of executing *QS*.
- 2) If a <query specification> *QS* is specified, then the General Rules of Subclause 9.4, “<procedure body>” are applied to the <procedure body> immediately contained in *QS*. The outcome of executing *QS* is the outcome of executing *PB* and the result of executing *QS* is the result of executing *PB*.

Conformance Rules

None.

9.3 <function specification>

Function

Define the procedural logic of a function.

Format

```
<nested function specification> ::=  
  <left brace> <function specification> <right brace>  
  
<function specification> ::=  
  !! Predicative production rule.  
  <procedure body>
```

Syntax Rules

- 1) Let FS be the <function specification>.
- 2) Let PB be the <procedure body> immediately contained in FS .
- 3) Every <statement> immediately contained in PB shall be a <query statement>.
- 4) If PB contains a <focused linear query statement> without an intervening <procedure body>, then PB shall not contain an <ambient linear query statement> without an intervening <procedure body>.
- 5) If PB contains an <ambient linear query statement> without an intervening <procedure body>, then PB shall not contain a <focused linear query statement> without an intervening <procedure body>.
- 6) A <function specification> shall not contain a <property reference>, a <from graph clause>, a <use graph clause>, or an <at schema clause> without an intervening <procedure body>.

General Rules

- 1) If a <nested function specification> NFS is specified, then the General Rules of Subclause 9.3, “<function specification>” are applied to the <function specification> FS simply contained in NFS . The outcome of executing NFS is the outcome of executing FS and the result of executing NFS is the result of executing FS .
- 2) If a <function specification> FS is specified, then the General Rules of Subclause 9.4, “<procedure body>” are applied to the <procedure body> PB immediately contained in FS . The outcome of executing FS is the outcome of executing PB and the result of executing FS is the result of executing PB .

Conformance Rules

None.

9.4 <procedure body>

Function

Define a <procedure body>.

Format

```

<procedure body> ::=

  [ <static variable definition block> ] [ <binding variable definition block> ] <statement
  block>

<static variable definition block> ::=

  <static variable definition>...

<binding variable definition block> ::=

  <binding variable definition>...

<statement block> ::=

  <statement> [ <then statement>... ] 

<then statement> ::=

  THEN [ <yield clause> ] <statement>

```

Syntax Rules

- 1) Let PB be the <procedure body>.
- 2) If the <static variable definition block> $SVDBLK$ was specified, then:
 - a) Let $SVDSEQ$ be the sequence of <static variable definition> immediately contained in $SVDBLK$.
 - b) The General Rules of Subclause 10.1, “Static variable definitions” are applied to $SVDSEQ$.

General Rules

- 1) If the <binding variable definition block> $BVDBLK$ was specified, then:
 - a) Let $BVDSEQ$ be the sequence of <binding variable definition> immediately contained in $BVDBLK$.
 - b) The General Rules of Subclause 10.5, “Binding variable and parameter declarations and definitions” are applied to $BVDSEQ$.
- 2) Let $SBLK$ be the <statement block>.
- 3) Let $STMSEQ$ be the sequence of <then statement>s immediately contained in $SBLK$ in the order of their occurrence in $SBLK$ from left to right and let m be the number of elements of $STMSEQ$.
- 4) Let STM_j be the j -th element of $STMSEQ$, for $1 \leq j \leq m$.
- 5) Let CWS be the current working schema.
- 6) Let CWG_0 be the current working graph.
- 7) The General Rules of Clause 12, “Statements” are applied to the <statement> immediately contained in $SBLK$.
- 8) For $1 \leq j \leq m$:

- a) Let CWG_j be the current working graph.
 - b) Case:
 - i) If the current execution outcome has a result graph RG , then:
 - 1) Set the current working schema to CWS .
 - 2) Set the current working graph to RG .
 - 3) Set the current working table to a new unit binding table.
 - ii) If the current execution outcome has a result table RT , then:
 - 1) Set the current working schema to CWS .
 - 2) Set the current working graph to CWG_{j-1} .
 - 3) Set the current working table to RT .
 - iii) Otherwise:
 - 1) Set the current working schema to CWS .
 - 2) Set the current working graph to CWG_{j-1} .
 - 3) Set the current working table to a new unit binding table.
 - 4) Set the current execution outcome to a successful outcome with an unknown result.
 - c) If STM_j contains a <yield clause> YC , then the General Rules for Subclause 16.16, “<yield clause>” are applied to YC ; set the current working table to be the $YIELD$ returned from the application of these General Rules.
 - d) The General Rules of Clause 12, “Statements” are applied to the <statement> contained in STM_j .
- 9) Set the current working schema to CWS .
- 10) Let $OUTCOME$ be the current execution outcome and let $RESULT$ be the current execution result.
- 11) The outcome of the application of these General Rules is $OUTCOME$ and the result of the application of these General Rules is $RESULT$.

Conformance Rules

None.

10 Variable and parameter declarations and definitions

10.1 Static variable definitions

Function

Define static variables.

Format

```
<static variable definition> ::=  
  <procedure variable definition>  
  | <query variable definition>  
  | <function variable definition>  
  
<as or equals> ::=  
  AS | <equals operator>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

10.2 Procedure variable definition

Function

Define a procedure variable.

Format

```
<procedure variable definition> ::=  
  [ CATALOG ] PROCEDURE <procedure variable> <of type signature> <procedure initializer>  
  
<procedure variable> ::=  
  <static variable name>  
  
<procedure initializer> ::=  
  <as or equals> <procedure reference>  
  | [ AS ] <nested procedure specification>  
  | <colon> <catalog procedure reference>
```

Syntax Rules

- 1) If the <procedure variable definition> *PVD* is specified, then:
 - a) Let *SVN* be the <static variable name> immediately contained in *PVD*.
 - b) If a static variable with name *SVN* is defined in scope, an exception condition is raised.
 - c) Let *ROPI* be the result of the <procedure initializer> immediately contained in *PVD*.
 - d) If the <of type signature> *OPS* is immediately contained in *PVD* and the type signature of *ROPI* does not match the procedure signature of *OPS*, then an exception condition is raised.
 - e) If CATALOG is specified, then *ROPI* shall be a catalog-modifying procedure.
 - f) If CATALOG is not specified, then *ROPI* shall not be a catalog-modifying procedure.
 - g) A static variable with name *SVN* is defined to be *ROPI* in the scope of the containing <procedure body>.
- 2) If the <procedure initializer> *PI* is specified, then the result of *PI* is the result of the <procedure reference> or the <nested procedure specification> immediately contained in *PI*.
- 3) If the <procedure initializer> *PI* is specified and *PI* immediately contains the <catalog procedure reference> *CPR*, then *PI* is effectively replaced by:

AS PROCEDURE *CPR*

General Rules

None.

Conformance Rules

None.

10.3 Query variable definition

Function

Define a query variable.

Format

```
<query variable definition> ::=  
  QUERY <query variable> <of type signature> <query initializer>  
  
<query variable> ::=  
  <static variable name>  
  
<query initializer> ::=  
  <as or equals> <query reference>  
  | [ AS ] <nested query specification>  
  | <colon> <catalog query reference>
```

Syntax Rules

- 1) If the <query variable definition> *QVD* is specified, then:
 - a) Let *SVN* be the <static variable name> immediately contained in *QVD*.
 - b) If a static variable with name *SVN* is defined in scope, an exception condition is raised.
 - c) Let *ROQI* be the result of the <query initializer> immediately contained in *QVD*.
 - d) If the <of type signature> *OPS* is immediately contained in *QVD* and the type signature of *ROQI* does not match the procedure signature of *OPS*, then an exception condition is raised.
 - e) A static variable with name *SVN* is defined to be *ROQI* in the scope of the containing <procedure body>.
- 2) If the <query initializer> *QI* is specified, then the result of *QI* is the result of the <query reference> or the <nested query specification> immediately contained in *QI*.
- 3) If the <query initializer> *QI* is specified and *QI* immediately contains the <catalog query reference> *CQR*, then *QI* is effectively replaced by:

AS QUERY *CQR*

General Rules

None.

Conformance Rules

None.

10.4 Function variable definition

Function

Define a function variable.

Format

```
<function variable definition> ::=  
  FUNCTION <function variable> <of type signature> <function initializer>  
  
<function variable> ::=  
  <static variable name>  
  
<function initializer> ::=  
  <as or equals> <function reference>  
  | [ AS ] <nested function specification>  
  | <colon> <catalog function reference>
```

Syntax Rules

- 1) If the <function variable definition> *FVD* is specified, then:
 - a) Let *SVN* be the <static variable name> immediately contained in *FVD*.
 - b) If a static variable with name *SVN* is defined in scope, an exception condition is raised:
 - c) Let *ROFI* be the result of the <function initializer> immediately contained in *FVD*.
 - d) If the <of type signature> *OPS* is immediately contained in *FVD* and the type signature of *ROFI* does not match the procedure signature of *OPS*, then an exception condition is raised:
 - e) A static variable with name *SVN* is defined to be *ROFI* in the scope of the containing <procedure body>.
- 2) If the <function initializer> *PI* is specified, then the result of *PI* is the result of the <function reference> or the <nested function specification> immediately contained in *PI*.
- 3) If the <function initializer> *PI* is specified and *PI* immediately contains the <catalog function reference> *CFR*, then *PI* is effectively replaced by:

AS FUNCTION *CFR*

General Rules

None.

Conformance Rules

None.

10.5 Binding variable and parameter declarations and definitions

Function

Declare and define variables.

Format

```

<compact variable declaration list> ::= 
  <compact variable declaration> [ { <comma> <compact variable declaration> }... ]

<compact variable declaration> ::= 
  <binding variable declaration> | <value variable>

<binding variable declaration> ::= 
  <graph variable declaration>
  | <binding table variable declaration>
  | <value variable declaration>

<compact variable definition list> ::= 
  <compact variable definition> [ { <comma> <compact variable definition> }... ]

<compact variable definition> ::= 
  <compact value variable definition>
  | <binding variable definition>

<compact value variable definition list> ::= 
  <compact value variable definition> [ { <comma> <compact value variable definition> } ] 

<compact value variable definition> ::= 
  <value variable> <equals operator> <value expression>

<binding variable definition list> ::= 
  <binding variable definition> [ { <comma> <binding variable definition> }... ]

<binding variable definition> ::= 
  <graph variable definition>
  | <binding table variable definition>
  | <value variable definition>

<optional binding variable definition list> ::= 
  <optional binding variable definition> [ { <comma> <optional binding variable definition> }... ]

<optional binding variable definition> ::= 
  <optional graph variable definition>
  | <optional binding table variable definition>
  | <optional value variable definition>

<parameter definition> ::= 
  <graph parameter definition>
  | <binding table parameter definition>
  | <value parameter definition>

```

Syntax Rules

- 1) If the <compact variable declaration> *CVD* is specified and *CVD* immediately contains the <value variable> *VV*, then *CVD* is effectively replaced by:

VALUE *VV*

10.5 Binding variable and parameter declarations and definitions

- 2) If the <compact variable definition> *CVD* is specified and *CVD* immediately contains the <compact value variable definition> *CVVD*, then *CVD* is effectively replaced by:

```
VALUE CVVD
```

General Rules

- 1) If the <compact variable declaration list> *CVDL* is specified, then the General Rules are applied to every <compact variable declaration> immediately contained in *CVDL* in the order of their occurrence from left to right in *CVDL*.
- 2) If the <compact variable definition list> *CVDL* is specified, then the General Rules are applied to every <compact variable definition> immediately contained in *CVDL* in the order of their occurrence from left to right in *CVDL*.
- 3) If the <compact value variable definition list> *CVVDL* is specified, then the General Rules are applied to every <compact value variable definition> immediately contained in *CVVDL* in the order of their occurrence from left to right in *CVVDL*.
- 4) If the <binding variable definition list> *BVDL* is specified, then the General Rules are applied to every <binding variable definition> immediately contained in *BVDL* in the order of their occurrence from left to right in *BVDL*.
- 5) If the <optional binding variable definition list> *OBVDL* is specified, then the General Rules are applied to every <optional binding variable definition> immediately contained in *OBVDL* in the order of their occurrence from left to right in *OBVDL*.

This document permits GQL-implementations to provide additional, implementation-defined types of binding variable declarations and definitions.

Conformance Rules

None.

10.6 Graph variable and parameter declaration and definition

Function

Declare or define a graph variable.

Format

```

<graph variable declaration> ::==
  [ PROPERTY ] GRAPH <graph variable> <of graph type>

<optional graph variable definition> ::==
  <graph variable definition>

<graph variable definition> ::==
  [ PROPERTY ] GRAPH <graph variable> <of graph type> <graph initializer>

<graph parameter definition> ::==
  [ PROPERTY ] GRAPH <parameter name> [ IF NOT EXISTS ] <of graph type> <graph initializer>

<graph variable> ::==
  <binding variable name>

<graph initializer> ::==
  <as or equals> <graph expression>
  | [ AS ] <nested graph query specification>
  | <colon> <catalog graph reference>

```

Syntax Rules

1) If the <graph initializer> *GI* is specified, then:

- a) If *GI* immediately contains the <nested graph query specification> *NGS*, then *GI* is effectively replaced by

AS PROPERTY GRAPH *NGS*

- b) If *GI* immediately contains the <catalog graph reference> *CGR*, then *GI* is effectively replaced by:

AS PROPERTY GRAPH *CGR*

General Rules

- 1) If the <graph variable declaration> *GVDECL* is specified, then *GVDECL* declares a variable whose name is the name of the <graph variable> immediately contained in *GVDECL* and whose declared type is the type of the <of graph type> immediately contained in *GVDECL*.
- 2) If the <optional graph variable definition> *OGVD* is specified, then:
 - a) Let *IGVD* be the inner <graph variable definition> immediately contained in *OGVD*.
 - b) Let *IGN* be the name of the <graph variable> immediately contained in *IGVD*.
 - c) If the current working record has no field with name *IGN*, then
 - i) Let *IGT* be the type specified by the <of graph type> immediately contained in *IGVD*.
 - ii) Let *ROIGI* be the result of the <graph initializer> immediately contained in *IGVD*.

10.6 Graph variable and parameter declaration and definition

- iii) If *ROIGI* is of type *IGT*, then a field with name *IGN* and value *ROIGI* is added to the current working record. Otherwise, an exception condition is raised:
- 3) If the <graph variable definition> *GVD* is specified, then:
- a) Let *GN* be the name of the <graph variable> immediately contained in *GVD*.
 - b) If the current working record has a field with name *GN*, an exception condition is raised:
 - c) Let *GT* be the type specified by the <of graph type> immediately contained in *GVD*.
 - d) Let *ROGI* be the result of the <graph initializer> immediately contained in *GVD*.
 - e) If *ROGI* is of type *GT*, then a field with name *GN* and value *GV* is added to the current working record. Otherwise, an exception condition is raised:
- 4) If the <graph parameter definition> *GPD* is specified, then:
- a) Let *PN* be the name of the <parameter> immediately contained in *GPD*.
 - b) Case:
 - i) If the current request context has a parameter with name *PN* or the current session context has a session parameter with name *PN* whose associated session parameter flag is FINAL, then:
 - 1) If IF NOT EXISTS is not specified, then an exception condition is raised:
 - ii) Otherwise:
 - 1) Let *PGT* be the type specified by the <of graph type> immediately contained in *GPD*.
 - 2) Let *ROPGI* be the result of the <graph initializer> immediately contained in *GPD*.
 - 3) If *ROPGI* is of type *PGT*, then a session parameter with name *PN* and value *ROPGI* is added to the current session context with session parameter flag MUTABLE. Otherwise, an exception condition is raised:
 - c) If the <graph variable> *GV* is specified, then the name of *GV* is the <binding variable name> immediately contained in *GV*.
 - d) If the <graph initializer> *GI* is specified, then the declared type of *GI* is the declared type of the <graph expression>, <nested graph query specification>, or <catalog graph reference> immediately contained in *GI*.
 - e) If the <graph initializer> *GI* is specified, then the result of *GI* is the result of the <graph expression>, <nested graph query specification>, or <catalog graph reference> immediately contained in *GI*.

Conformance Rules

None.

10.7 Binding table variable and parameter declaration and definition

Function

Declare or define a binding table variable.

Format

```

<binding table variable declaration> ::==
  [ BINDING ] TABLE <binding table variable> <of binding table type>

<optional binding table variable definition> ::==
  <binding table variable definition>

<binding table variable definition> ::==
  [ BINDING ] TABLE <binding table variable> <of binding table type> <binding table
  initializer>

<binding table parameter definition> ::==
  [ BINDING ] TABLE <parameter> [ IF NOT EXISTS ] <of binding table type> <binding table
  initializer>

<binding table variable> ::==
  <binding variable name>

<binding table initializer> ::==
  <as or equals> <binding table reference>
  | [ AS ] <nested query specification>
  | <colon> <catalog binding table reference>

```

Syntax Rules

None.

General Rules

- 1) If the <binding table variable declaration> *BTVDECL* is specified, then *BTVDECL* declares a variable whose name is the name of the <binding table variable> immediately contained in *BTVDECL* and whose declared type is the type of the <of binding table type> immediately contained in *BTVDECL*.
- 2) If the <optional binding table variable definition> *OBTVD* is specified, then:
 - a) Let *IBTVD* be the inner <binding table variable definition> immediately contained in *OBTVD*.
 - b) Let *IBTN* be the name of the <binding table variable> immediately contained in *IBTVD*.
 - c) If the current working record has no field with name *IBTN*, then
 - i) Let *IBTT* be the type specified by the <of binding table type> immediately contained in *IBTVD*.
 - ii) Let *ROIBTI* be the result of the <binding table initializer> immediately contained in *IBTVD*.
 - iii) If *ROIBTI* is of type *IBTT*, then a field with name *IBTN* and value *ROIBTI* is added to the current working record. Otherwise, an exception condition is raised:
- 3) If the <binding table variable definition> *BTVD* is specified, then:

10.7 Binding table variable and parameter declaration and definition

- a) Let BTN be the name of the <binding table variable> immediately contained in $BTVD$.
 - b) If the current working record has a field with name GN , an exception condition is raised.
 - c) Let BTT be the type specified by the <of binding table type> immediately contained in $BTVD$.
 - d) Let $ROBTI$ be the result of the <binding table initializer> immediately contained in $BTVD$.
 - e) If $ROBTI$ is of type BTT , then a field with name GN and value GV is added to the current working record. Otherwise, an exception condition is raised:
- 4) If the <binding table parameter definition> $BTPD$ is specified, then:
- a) Let PN be the name of the <parameter> immediately contained in $BTPD$.
 - b) Case:
 - i) If the current request context has a parameter with name PN or the current session context has a session parameter with name PN whose associated session parameter flag is FINAL, then:
 - 1) If IF NOT EXISTS is not specified, then an exception condition is raised:
 - ii) Otherwise:
 - 1) Let $PBTT$ be the type specified by the <of binding table type> immediately contained in $BTPD$.
 - 2) Let $ROPBTI$ be the result of the <binding table initializer> immediately contained in $BTPD$.
 - 3) If $ROPBTI$ is of type $PBTT$, then a session parameter with name PN and value $ROPBTI$ is added to the current session context with session parameter flag MUTABLE. Otherwise, an exception condition is raised:
 - c) If the <binding table variable> BTV is specified, then the name of BT is the <binding variable name> immediately contained in BT .
 - d) If the <binding table initializer> BTI is specified, then the declared type of BTI is the declared type of the <binding table reference>, <nested query specification>, or <catalog binding table reference> immediately contained in BTI .
 - e) If the <binding table initializer> BTI is specified, then the result of BTI is the result of the <binding table reference>, <nested query specification>, or <catalog binding table reference> immediately contained in BTI .

Conformance Rules

None.

10.8 Value variable and parameter declaration and definition

Function

Declare or define a value variable.

Format

```

<value variable declaration> ::==
  VALUE <value variable> [ <of value type> ]

<optional value variable definition> ::==
  <value variable definition>

<value variable definition> ::==
  VALUE <value variable> [ <of value type> ] <value initializer>

<value parameter definition> ::==
  VALUE <parameter> [ IF NOT EXISTS ] [ <of value type> ] <value initializer>

<value variable> ::==
  <binding variable name>

<value initializer> ::==
  <as or equals> <value expression>
  | [ AS ] <nested query specification>
  | <colon> <catalog object reference>

```

Syntax Rules

None.

General Rules

- 1) If the <value variable declaration> *VVDECL* is specified, then *VVDECL* declares a variable whose name is the name of the <value variable> immediately contained in *VVDECL* and whose declared type is the type of the <of value type> immediately contained in *VVDECL*.
- 2) If the <optional value variable definition> *OVVD* is specified, then:
 - a) Let *IVVD* be the inner <value variable definition> immediately contained in *OVVD*.
 - b) Let *IVN* be the name of the <value variable> immediately contained in *IVVD*.
 - c) If the current working record has no field with name *IVN*, then
 - i) Let *IVT* be the type specified by the <of value type> immediately contained in *IVVD*.
 - ii) Let *ROIVI* be the result of the <value initializer> immediately contained in *IVVD*.
 - iii) If *ROIVI* is of type *IVT*, then a field with name *IVN* and value *ROIVI* is added to the current working record. Otherwise, an exception condition is raised:
- 3) If the <value variable definition> *VVD* is specified, then:
 - a) Let *VN* be the name of the <value variable> immediately contained in *VVD*.
 - b) If the current working record has a field with name *VN*, an exception condition is raised:

10.8 Value variable and parameter declaration and definition

- c) Let VT be the type specified by the <of value type> immediately contained in VVD .
 - d) Let $ROVI$ be the result of the <value initializer> immediately contained in VVD .
 - e) If $ROVI$ is of type VT , then a field with name GN and value GV is added to the current working record. Otherwise, an exception condition is raised:
- 4) If the <value parameter definition> VPD is specified, then:
- a) Let PN be the name of the <parameter> immediately contained in VPD .
 - b) Case:
 - i) If the current request context has a parameter with name PN or the current session context has a session parameter with name PN whose associated session parameter flag is FINAL, then:
 - 1) If IF NOT EXISTS is not specified, then an exception condition is raised:
 - ii) Otherwise:
 - 1) Let PVT be the type specified by the <of value type> immediately contained in VPD .
 - 2) Let $ROPVI$ be the result of the <value initializer> immediately contained in VPD .
 - 3) If $ROPVI$ is of type PVT , then a session parameter with name PN and value $ROPVI$ is added to the current session context with session parameter flag MUTABLE. Otherwise, an exception condition is raised:
 - c) If the <value variable> VV is specified, then the name of VV is the <binding variable name> immediately contained in VV .
 - d) If the <value initializer> VI is specified, then the declared type of VI is the declared type of the <value expression>, <nested query specification>, or <catalog object reference> immediately contained in VI .
 - e) If the <value initializer> VI is specified, then the result of VI is the result of the <value expression>, <nested query specification>, or <catalog object reference> immediately contained in VI .

Conformance Rules

None.

11 Object expressions

11.1 Introduction to object expressions

This clause defines the expressions and related production rules for all primary objects.

11.2 <primary result object expression>

Function

Define a <primary result object expression>.

Format

```
<primary result object expression> ::=  
  <graph expression>  
  | <binding table reference>
```

Syntax Rules

- 1) Let OE be the specified <primary result object expression>.
- 2) The declared type of OE is the declared type of the immediately contained <graph expression> or <binding table reference>.

General Rules

- 1) The result of OE is the result of the immediately contained <graph expression> or <binding table reference>.

Conformance Rules

None.

11.3 <graph expression>

Function

Define a <graph expression>.

Format

```
<graph expression> ::=  
  <copy graph expression>  
  | <graph specification>  
  | <graph reference>  
  
<copy graph expression> ::=  
  COPY OF <graph expression>
```

Syntax Rules

None.

General Rules

- 1) Determine the graph G as follows.
 - a) Case:
 - a) If GE simply contains the <copy graph expression> CGE that immediately contains the inner <graph expression> IGE , then G is a copy of the graph that is the result of IGE .
 - b) If GE simply contains the <graph specification> GS , then G is the graph defined by GS .
 - c) If GE simply contains the <graph reference> GR , then G is the result of GR .
 - 2) The result of GE is G and the graph descriptor identified by GE is the graph descriptor of G .

Conformance Rules

None.

11.4 <graph type expression>

Function

Define a <graph type expression>.

Format

```

<graph type expression> ::=

  <copy graph type expression>
  | <like graph expression>
  | <graph type specification>
  | <graph type reference>

<as graph type> ::=

  <as or equals> <graph type expression>
  | <like graph expression shorthand>
  | [ AS ] <nested graph type specification>

<copy graph type expression> ::=
  COPY OF <graph type reference>

<like graph expression> ::=
  [ PROPERTY ] GRAPH TYPE <like graph expression shorthand>

<of graph type> ::=
  [ <of type prefix> ] <graph type expression>
  | <like graph expression shorthand>
  | [ <of type prefix> ] <nested graph type specification>

<like graph expression shorthand> ::=
  LIKE <graph expression>

```

Syntax Rules

- 1) If the <as graph type> *AGT* is specified and *AGT* immediately contains the <like graph expression shorthand> *LGES*, then *AGT* is effectively replaced by:

AS PROPERTY GRAPH TYPE *LGES*

- 2) If the <as graph type> *AGT* is specified and *AGT* immediately contains the <nested graph query specification> *NGQS*, then *AGT* is effectively replaced by:

AS PROPERTY GRAPH TYPE *NGQS*

- 3) If the <of graph type> *OGT* is specified and *OGT* immediately contains the <like graph expression shorthand> *LGES*, then *OGT* is effectively replaced by:

OF PROPERTY GRAPH TYPE *LGES*

- 4) If the <of graph type> *OGT* is specified and *OGT* immediately contains the <nested graph query specification> *NGQS*, then *OGT* is effectively replaced by:

OF PROPERTY GRAPH TYPE *NGQS*

General Rules

- 1) Let GTE be the <graph type expression>.
- 2) Determine the graph type GT as follows.

Case:

- a) If GTE simply contains the <copy graph type expression> that immediately contains the <graph type reference> GTR , then GT is a copy of the result of GTR .
 - b) If GTE simply contains the <like graph expression> that immediately contains the <like graph expression shorthand> that immediately contains the <graph expression> GE , then GT is the graph type of the result of GE .
 - c) If GTE simply contains the <graph type specification> GTS , then GT is the graph type defined by GTS .
 - d) If GTE simply contains the <graph type reference> GTR , then GT is the graph type of the result of GTR .
- 3) The result of GTE is GT and the graph type descriptor identified by GTE is the graph type descriptor of GT .

Conformance Rules

None.

11.5 <binding table type expression>

Function

Define a <binding table type expression>.

Format

```

<of binding table type> ::= 
  [ <of type prefix> ] <binding table type expression>
  | <like binding table shorthand>

<binding table type expression> ::= 
  <binding table type>
  | <like binding table type>

<binding table type> ::= 
  [ BINDING ] TABLE <record value type>

<like binding table type> ::= 
  [ BINDING ] TABLE <like binding table shorthand>

<like binding table shorthand> ::= 
  LIKE <binding table reference>

```

Syntax Rules

- 1) If the <of binding table type> *OBTT* is specified and *OBTT* immediately contains the <like binding table shorthand> *LBTS*, then *OBTT* is effectively replaced by:

OF BINDING TABLE TYPE *LBTS*

General Rules

- 1) If the <of binding table type> *OBTT* is specified, then the result of *OBTT* is the result of the immediately contained <binding table type expression>.
- 2) If the <binding table type expression> *BTE* is specified, then the result of *BTE* is the result of the immediately contained <binding table type> or <like binding table type>.
- 3) If the <binding table type> *BTT* is specified, then the result of *BTE* is the binding table type of records whose record type is the record type specified by the <record value type> immediately contained in *BTT*.
- 4) If the <like binding table type> *LBTT* is specified, then the result of *LBTT* is the binding table type of the result of the <binding table reference> simply contained in *LBTT*.

Conformance Rules

None.

12 Statements

12.1 <statement>

Function

Define a <statement>.

Format

```

<statement> ::= 
  [ <at schema clause> ] {
    <catalog-modifying statement>
  | <data-modifying statement>
  | <query statement>
  }

<catalog-modifying statement> ::= 
  <linear catalog-modifying statement>

<data-modifying statement> ::= 
  <conditional data-modifying statement>
  | <linear data-modifying statement>

<query statement> ::= 
  <composite query statement>
  | <conditional query statement>

```

Syntax Rules

None.

General Rules

- 1) If a <statement> *S* is specified, then *S* is executed by first applying the General Rules of Subclause 16.3, “<at schema clause>” to the immediately contained <at schema clause>, if any, followed by applying the General Rules to the immediately contained <catalog-modifying statement>, <data-modifying statement>, or <query statement>.

NOTE 91 — Not all statements set a material result; instead they may modify the current execution context. A material result is set by <primitive result statement>s and by a <simple data-modifying statement>s to determine the execution outcome and result of a procedure or a top-level <statement>.
- 2) If a <catalog-modifying statement> *CMS* is specified, then the General Rules are applied to the immediately contained <linear catalog-modifying statement> to execute *CMS*.
- 3) If a <data-modifying statement> *DMS* is specified, then the General Rules are applied to the immediately contained <conditional data-modifying statement> or <linear data-modifying statement> to execute *DMS*.
- 4) If a <query statement> *QS* is specified, then the General Rules are applied to the immediately contained <composite query statement>, or <conditional query statement> to execute *QS*.

Conformance Rules

None.

12.2 <call procedure statement>

Function

Define a <call procedure statement>.

Format

```
<call procedure statement> ::=  
  [ <statement mode> ] CALL <procedure call>  
  
<statement mode> ::=  
  OPTIONAL  
  | MANDATORY
```

Syntax Rules

- 1) Let *CPS* be the <call procedure statement>.
- 2) Let *PC* be the <procedure call> immediately contained in *CPS*.
- 3) Case:
 - a) If *CPS* immediately contains a <statement mode> *SM*, then

Case:

 - i) If *SM* is OPTIONAL, then *CPS* is effectively replaced by the <optional statement>:

$$\text{OPTIONAL } \{ \text{CALL } PC \text{ RETURN } * \}$$
 - ii) Otherwise, *CPS* is effectively replaced by the <mandatory statement>:

$$\text{MANDATORY } \{ \text{CALL } PC \text{ RETURN } * \}$$

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *NEW_TABLE* be a new binding table.
- 3) For each record *R* of *TABLE* in a new child execution context amended with *R*:
 - a) The General Rules of Subclause 16.13, “<procedure call>” are applied to *PC*; let *RESULT* be the binding table *RESULT* returned from the application of these General Rules.
 - b) Append the Cartesian Product of *R* and *RESULT* to *NEW_TABLE*.
- 4) Set the current working table to *NEW_TABLE*.
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

12.3 Statement classes

Function

Define various classes of statements.

Format

```

<simple catalog-modifying statement> ::=

    <primitive catalog-modifying statement>
    | <call catalog-modifying procedure statement>

<primitive catalog-modifying statement> ::=

    <create graph statement>
    | <create graph type statement>
    | <create procedure statement>
    | <create query statement>
    | <create function statement>
    | <drop graph statement>
    | <drop graph type statement>
    | <drop procedure statement>
    | <drop query statement>
    | <drop function statement>

<simple data-accessing statement> ::=

    <simple query statement>
    | <simple data-modifying statement>

<simple data-modifying statement> ::=

    <primitive data-modifying statement>
    | <do statement>
    | <call data-modifying procedure statement>

<primitive data-modifying statement> ::=

    <insert statement>
    | <merge statement>
    | <set statement>
    | <remove statement>
    | <delete statement>

<simple query statement> ::=

    <simple data-transforming statement>
    | <simple data-reading statement>

<simple data-reading statement> ::=

    <match statement>
    | <call query statement>

<simple data-transforming statement> ::=

    <primitive data-transforming statement>
    | <call function statement>

<primitive data-transforming statement> ::=

    <optional statement>
    | <mandatory statement>
    | <let statement>
    | <for statement>
    | <aggregate statement>
    | <filter statement>
    | <order by and page statement>

```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

13 Catalog-modifying statements

13.1 <linear catalog-modifying statement>

Function

Define a <linear catalog-modifying statement>.

Format

```
<linear catalog-modifying statement> ::=  
  <simple catalog-modifying statement>...
```

Syntax Rules

None.

General Rules

- 1) The <linear catalog-modifying statement> *LCMS* is executed by executing all <simple catalog-modifying statement>s immediately contained in *LCMS* in the order of their occurrence from left to right.

Conformance Rules

None.

13.2 <create schema statement>

Function

Create a schema.

Format

```
<create schema statement> ::=  
    CREATE SCHEMA <catalog schema parent and name> [ IF NOT EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CSPN* be the <catalog schema parent and name>.
- 2) Let *SN* be the <schema name> immediately contained in *CSPN*.
- 3) Let *PARENT* be determined as follows:
Case:
 - a) If *CSPN* does not immediately contain an <absolute url path>, then *PARENT* is the catalog root and shall be a GQL-directory.
 - b) Otherwise, *PARENT* is the result of the immediately contained <absolute url path> and shall be a GQL-directory.
- 4) If IF NOT EXISTS is not specified, then *SN* shall not identify an existing schema descriptor in *PARENT*.

General Rules

- 1) If IF NOT EXISTS is specified and *SN* identifies an existing schema descriptor in *PARENT*, then no further General Rules of this Subclause are applied.
- 2) Let *S* be the GQL-schema described by the GQL-schema descriptor containing
 - a) *SN* as the name of the GQL-schema.
 - b) The current authorization identifier as the owner of the GQL-schema.
- 3) Add the descriptor of *S* to the GQL-directory descriptor of *PARENT*.

Conformance Rules

None.

13.3 <drop schema statement>

Function

Destroy a schema.

Format

```
<drop schema statement> ::=  
  DROP SCHEMA <catalog schema parent and name> [ IF EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CSPN* be the <catalog schema parent and name>.
- 2) Let *SN* be the <schema name> immediately contained in *CSPN*.
- 3) Let *PARENT* be determined as follows:

Case:

 - a) If *CSPN* does not immediately contain an <absolute url path>, then *PARENT* is the catalog root and shall be a GQL-directory.
 - b) Otherwise, *PARENT* is the result of the immediately contained <absolute url path> and shall be a GQL-directory.
- 4) If IF EXISTS is not specified, then *SN* shall identify an existing schema descriptor in *PARENT*.
- 5) Let *SD* be the schema descriptor identified by *SN* in *PARENT*.
- 6) *SD* shall not contain any catalog object descriptors.

General Rules

- 1) If IF EXISTS is specified and *SN* does not identify an existing schema descriptor in *PARENT*, then no further General Rules of this Subclause are applied.
- 2) If the schema of *SD* is the current working schema, an exception condition is raised:
- 3) Destroy *SD* in *PARENT*.

Conformance Rules

None.

13.4 <create graph statement>

Function

Create a graph.

Format

```
<create graph statement> ::=  
  CREATE {  
    [ PROPERTY ] GRAPH <catalog graph parent and name> [ IF NOT EXISTS ]  
    | OR REPLACE [ PROPERTY ] GRAPH <catalog graph parent and name>  
    } [ <of graph type> ] [ <graph source> ]  
  
<graph source> ::=  
  AS <copy graph expression>
```

Syntax Rules

- 1) Let *CGS* be the <create graph statement>.
- 2) After applying all relevant syntactic transformations, let *CGPN* be the <catalog graph parent and name> immediately contained in *CGS*.
- 3) Let *GPS* be the implicit or explicit <graph parent specification> immediately contained in *CGPN*.
- 4) Let *GN* be the <graph name> immediately contained in *CGPN*.
- 5) If *CGS* does not immediately contain a <graph source>, then the following <graph source> is implicit:

$$\text{AS COPY OF EMPTY_GRAPH}$$
- 6) Let *GSR* be the implicit or explicit <graph source> immediately contained in *CGS*.
- 7) After applying all relevant syntactic transformations, let *CGE* be the <copy graph expression> immediately contained in *GSR* and let *IGE* be the inner <graph expression> immediately contained in *CGE*.
- 8) If *CGS* does not immediately contain an <of graph type>, then *CGS* is replaced with:

`CREATE PROPERTY GRAPH GPS GN LIKE IGE GSR`

NOTE 92 — This rule rewrites:

```
CREATE PROPERTY GRAPH Y AS COPY OF X  
as
```

```
CREATE PROPERTY GRAPH Y LIKE X AS COPY OF X  
i.e., it makes explicit that X provides the initial graph elements as well as the graph type for Y.
```

General Rules

- 1) Let *PARENT* be the descriptor of the result of *GPS*.
- 2) If *CGS* does not immediately contain IF NOT EXISTS or OR REPLACE, then *GN* shall not identify an existing catalog object descriptor in *PARENT*.

13.4 <create graph statement>

- 3) If GN identifies an existing catalog object descriptor in $PARENT$ then that catalog object descriptor shall be a graph descriptor.
- 4) If CGS immediately contains IF NOT EXISTS and GN identifies an existing graph descriptor in $PARENT$ then the completion condition *warning — graph already exists (01G01)* is raised and no further General Rules of this Subclause are applied.
- 5) Let OGT be the <of graph type> immediately contained in CGS , let GTE be the <graph type expression> immediately contained in OGT , and let GTD be the graph type descriptor identified by GTE .
- 6) Determine GTN as follows.

Case:

- a) If the graph type identified by GTD is not persistent, then:
 - i) Let $NGTN$ be a new system-generated name, that does not identify an existing catalog object descriptor in $PARENT$.
 - ii) The following <create graph type statement> is effectively executed:

```
CREATE PROPERTY GRAPH TYPE GPS NGTN LIKE GE
```

- iii) GTN is $NGTN$.

- b) Otherwise, GTN is the graph type name of GTD .

- 7) Let GT be the graph type identified by GTN .

- 8) Let G be a graph, with

- a) GN as its graph name,

- b) An empty set as its graph label set,

NOTE 93 — The Format for a graph label set definition is not yet defined.

- c) An empty set as its graph property set,

NOTE 94 — The Format for a graph property set definition is not yet defined.

- d) GTN as the name of the graph type.

- 9) G is populated according to $GSRC$ as follows.

- a) Let SG be the graph specified by $GSRC$.

- b) If any graph element of SG does not conform to GT , then the exception condition *graph type conformance error (G2000)* is raised.

- c) Copies of all nodes and edges in SG are inserted into G .

NOTE 95 — This rule probably need further refinement.

- 10) If CGS immediately contains OR REPLACE and GN identifies an existing graph descriptor EG in $PARENT$, then

- a) The following <drop graph statement> is effectively executed:

```
DROP PROPERTY GRAPH GPS GN
```

- 11) A graph descriptor is created that describes G and includes:

- a) The name G of the graph.

- b) The name GTN of the associated graph type.
- 12) G is created.

Conformance Rules

None.

13.5 <graph specification>

Function

Define a graph.

Format

```
<graph specification> ::=  
  [ PROPERTY ] GRAPH  
    { <nested graph query specification> | <nested ambient data-modifying procedure  
specification> }  
  
<nested graph query specification> ::=  
  <nested query specification>  
  
<nested ambient data-modifying procedure specification> ::=  
  <nested data-modifying procedure specification>
```

Syntax Rules

- 1) If the <nested ambient data-modifying procedure specification> *NADPS* is specified, then:
 - a) Let *NDPS* be the <nested data-modifying procedure specification> immediately contained in *NADPS*.
 - b) *NDPS* shall contain an <ambient linear data-modifying statement>.

General Rules

- 1) The graph defined by the <graph specification> *GS* is determined as follows.

Case:

- a) If *GS* immediately contains the <nested graph query specification> *NGQS*, then the graph defined by *GS* is the result of *NGQS*.
 - b) If *GS* immediately contains the <nested ambient data-modifying procedure specification> *NADPS*, then the graph defined by *GS* is the result of *NADPS*.
- 2) If the <nested graph query specification> *NGQS* is specified, then:
 - a) Let *NQS* be the <nested query specification> that is *NGQS*.
 - b) Let *RONQS* be the result of *NQS*.
 - c) If *RONQS* is a graph *G*, then the graph defined by *NGQS* is *G*. Otherwise, an exception condition is raised:
 - 3) If the <nested ambient data-modifying procedure specification> *NADPS* is specified, then:
 - a) Let *NDPS* be the <nested data-modifying procedure specification> immediately contained in *NADPS*.
 - b) The General Rules of Subclause 9.1, “<procedure specification>” for a <nested data-modifying procedure specification> are applied to *NADPS* in a new child execution context *CONTEXT* initialized with an empty working graph; *G'* be the current working graph available in *CONTEXT* after the application of these General Rules.

- c) Set the execution outcome to a successful outcome with result G' .

Conformance Rules

None.

13.6 <drop graph statement>

Function

Destroy a graph.

Format

```
<drop graph statement> ::=  
  DROP GRAPH <catalog graph parent and name> [ IF EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CGPN* be the <catalog graph parent and name>.
- 2) Let *GPS* be the implicit or explicit <graph parent specification> immediately contained in *CGPN*.
- 3) Let *GN* be the <graph name> immediately contained in *CGPN*.

General Rules

- 1) Let *PARENT* be the descriptor of the result of *GPS*.
- 2) If IF EXISTS is not specified, then *GN* shall identify a graph descriptor *GD* in *PARENT*.
- 3) If IF EXISTS is specified and *GN* does not identify an existing graph descriptor in *PARENT*, then the completion condition *warning — graph does not exist (01G03)* is raised and no further General Rules of this Subclause are applied.
- 4) Let *GTN* be the graph type name contained in *GD*.
- 5) Destroy the graph and graph descriptor identified by *GN* in *PARENT*.
- 6) If *GTN* is a system-generated name and is not contained in a graph type descriptor, then the following <drop graph type statement> is effectively executed:

```
DROP GRAPH TYPE GP GTN
```

Conformance Rules

None.

13.7 <create graph type statement>

Function

Create a graph type in the GQL-catalog.

Format

```
<create graph type statement> ::=  
  CREATE {  
    [ PROPERTY ] GRAPH TYPE <catalog graph type parent and name> [ IF NOT EXISTS ]  
    | OR REPLACE [ PROPERTY ] GRAPH TYPE <catalog graph type parent and name>  
  } <graph type initializer>  
  
<graph type initializer> ::=  
  <as graph type>  
  | <colon> <catalog graph type reference>
```

Syntax Rules

- 1) Let *CGTS* be the <create graph type statement> and let *CGTPN* be the <catalog graph type parent and name> immediately contained in *CGTS*.
- 2) Let *GTPS* be the implicit or explicit <graph type parent specification> immediately contained in *CGTPN*.
- 3) Let *GTN* be the <graph type name> immediately contained in *CGTPN*.
- 4) If the <graph type initializer> *GTI* is specified, then the result of *GTI* is the result of the <as graph type> immediately contained in *GTI*.
- 5) If the <graph type initializer> *GTI* is specified and *GTI* immediately contains the <catalog graph type reference> *CGTR*, then *GTI* is effectively replaced by:

AS PROPERTY GRAPH TYPE *CGTR*

- 6) Let *GTI* be the <graph type initializer> immediately contained in *CGTS*.
- 7) Let *GTE* be the <graph type expression> immediately contained in *GTI*.
- 8) *GTE* shall not simply contain a <graph reference>.

General Rules

- 1) Let *PARENT* be the descriptor of the result of *GTPS*.
- 2) Case:
 - a) If *CGTS* does not immediately contain IF NOT EXISTS or OR REPLACE, then *GTN* shall not identify an existing catalog object descriptor in *PARENT*.
 - b) If *GTN* identifies an existing catalog object descriptor in *PARENT*, then that catalog object descriptor shall be a graph type descriptor.
 - c) If *CGTS* immediately contains OR REPLACE, then *GTN* shall not identify an existing graph type descriptor in *PARENT* that is identified in any graph descriptor.

13.7 <create graph type statement>

- 3) If $CTGS$ immediately contains IF NOT EXISTS and GTN identifies an existing graph type descriptor in $PARENT$, then the completion condition *warning — graph type already exists (01G02)* is raised and no further General Rules of this Subclause are applied.
- 4) Let GTD be the graph type descriptor defined by GTE .
- 5) If $CGTS$ immediately contains OR REPLACE, then the following <drop graph type statement> is effectively executed:

```
DROP PROPERTY GRAPH TYPE GPS GTN
```

- 6) Create a graph type GTN in $PARENT$ whose graph type descriptor is GTD with the name GTN .

Conformance Rules

None.

13.8 <graph type specification>

Function

Define a graph type.

Format

```

<graph type specification> ::==
  [ PROPERTY ] GRAPH TYPE <nested graph type specification>

<nested graph type specification> ::==
  <left brace> <graph type specification body> <right brace>

<graph type specification body> ::==
  <element type definition list>

<element type definition list> ::==
  <element type definition> [ { <comma> <element type definition> }... ] 

<element type definition> ::==
  <node type definition>
  | <edge type definition>

```

Syntax Rules

None.

General Rules

- 1) The graph type defined by the <graph type specification> *GTS* is the graph type defined by the <graph type specification body> simply contained in *GTS*.
- 2) The graph type defined by the <nested graph type specification> *NGTS* is the graph type defined by the <graph type specification body> simply contained in *NGTS*.
- 3) The graph type defined by the <graph type specification body> *GTDB* is determined as follows:
 - a) Let *NTDS* be the set of <node type definition>s simply contained in *GTDB*.
 - b) Let *NTNS* be the set of <node type name>s simply contained in <node type definition>s contained in *NTDS*.
 - c) For every <node type definition> *NTD* contained in *NTDS* that simply contains a <node type name> *NTN*, let *NTDWN* (*NTN*) be *NTD*.
 - d) Let *ETDS* be the set of <edge type definition>s simply contained in *GTDB*.
 - e) Let *ETNS* be the set of <edge type name>s simply contained in <edge type definition>s contained in *ETDS*.
 - f) For every <edge type definition> *ETD* contained in *ETDS* that simply contains an <edge type name> *ETN*, let *ETDWN* (*ETN*) be *ETD*.
 - g) The graph type defined by *GTDB* is described by the graph type descriptor containing:
 - i) A graph type name that is empty.

13.8 <graph type specification>

NOTE 96 — The name will be supplied by the <create graph type statement> that contains it.

- ii) An empty set as the graph type label set.

NOTE 97 — The Format for a graph type label set definition is not yet defined.

- iii) An empty set as the graph type property type set.

NOTE 98 — The Format for a graph type property set definition is not yet defined.

- iv) The set of all node type descriptors defined by the <node type definition>s contained in *NTDS* as the node type set.

- v) The set of all edge type descriptors defined by the <edge type definition>s contained in *ETDS* as the edge type set.

- vi) The dictionary that maps every <node type name> *NTN* in *NTNS* to the node type defined by *NTDWN (NTN)* as the node type name dictionary.

- vii) The dictionary that maps every <node type name> *ETN* in *ETNS* to the edge type defined by *ETDWN (ETN)* as the edge type name dictionary.

Conformance Rules

None.

13.9 <node type definition>

Function

Define a node type.

Format

```

<node type definition> ::==
  <left paren> [ <node type name> ] [ <node type filler> ] <right paren>
  | <node synonym> [ TYPE ] <node type name> <node type filler>

<node type name> ::=
  !! Predicative production rule.
  <element type name>

<node type filler> ::=
  <node type label set definition>
  | <node type property type set definition>
  | <node type label set definition> <node type property type set definition>

<node type label set definition> ::=
  !! Predicative production rule.
  <label set definition>

<node type property type set definition> ::=
  !! Predicative production rule.
  <property type set definition>

```

Syntax Rules

- 1) Let NTD be the <node type definition>.
- 2) NTD is transformed as follows. If NTD contains a <node type name> NTN but does not contain a <node type label set definition>, then
 - a) If NTD contains a <node type property type set definition>, then let $NTPTSD$ be that <node type property type set definition>, otherwise let $NTPTSD$ be the zero-length character string.
 - b) The following <node type filler> is implicit and replaces the existing <node type filler>, if any:

: $NTN\ NTPTSD$

NOTE 99 — This is node label implication.

General Rules

- 1) If NTD contains a <node type label set definition> $NTLSD$, then let $NTLS$ the label set that is the result of the <label set definition> immediately contained in $NTLSD$. Otherwise let $NTLS$ be an empty label set.
- 2) If NTD contains a <node type property type set definition> $NTPTSD$, then let $NTPTS$ the property type set that is the result of the <property type set definition> immediately contained in $NTPTSD$. Otherwise let $NTPTS$ be an empty property type set.
- 3) The node type defined by NTD is described by a node type descriptor containing:
 - a) The node type name NTN , if specified.

- b) The node type label set $NTLS$.
- c) The node type property type set $NTPTS$.

Conformance Rules

None.

13.10 <edge type definition>

Function

Define an edge type.

Format

```

<edge type definition> ::==
  <full edge type pattern>
  | <abbreviated edge type pattern>
  | <edge kind> <edge synonym> [ TYPE ] <edge type name> <edge type filler> <endpoint
    definition>

<edge type name> ::=
  !! Predicative production rule.
  <element type name>

<edge type filler> ::=
  <edge type label set definition>
  | <edge type property type set definition>
  | <edge type label set definition> <edge type property type set definition>

<edge type label set definition> ::=
  !! Predicative production rule.
  <label set definition>

<edge type property type set definition> ::=
  !! Predicative production rule.
  <property type set definition>

<full edge type pattern> ::=
  <full edge type pattern pointing right>
  | <full edge type pattern pointing left>
  | <full edge type pattern any direction>

<full edge type pattern pointing right> ::=
  <source node type reference> <arc type pointing right> <destination node type reference>

<full edge type pattern pointing left> ::=
  <destination node type reference> <arc type pointing left> <source node type reference>

<full edge type pattern any direction> ::=
  <source node type reference> <arc type any direction> <destination node type reference>

<arc type pointing right> ::=
  <minus left bracket> <arc type filler> <bracket right arrow>

<arc type pointing left> ::=
  <left arrow bracket> <arc type filler> <right bracket minus>

<arc type any direction> ::=
  <tilde left bracket> <arc type filler> <right bracket tilde>

<arc type filler> ::=
  [ <edge type name> ] [ <edge type filler> ]

<abbreviated edge type pattern> ::=
  <abbreviated edge type pattern pointing right>
  | <abbreviated edge type pattern pointing left>
  | <abbreviated edge type pattern any direction>

```

13.10 <edge type definition>

```

<abbreviated edge type pattern pointing right> ::= 
  <source node type reference> <right arrow> <destination node type reference>

<abbreviated edge type pattern pointing left> ::= 
  <destination node type reference> <left arrow> <source node type reference>

<abbreviated edge type pattern any direction> ::= 
  <source node type reference> <tilde> <destination node type reference>

<source node type reference> ::= 
  <left paren> <source node type name> <right paren>
  | <left paren> [ <node type filler> ] <right paren>

<destination node type reference> ::= 
  <left paren> <destination node type name> <right paren>
  | <left paren> [ <node type filler> ] <right paren>

<edge kind> ::= 
  DIRECTED
  | UNDIRECTED

<endpoint definition> ::= 
  CONNECTING <endpoint pair definition>

<endpoint pair definition> ::= 
  <endpoint pair definition pointing right>
  | <endpoint pair definition pointing left>
  | <endpoint pair definition any direction>
  | <abbreviated edge type pattern>

<endpoint pair definition pointing right> ::= 
  <left paren> <source node type name> <connector pointing right> <destination node type name> <right paren>

<endpoint pair definition pointing left> ::= 
  <left paren> <destination node type name> <left arrow> <source node type name> <right paren>

<endpoint pair definition any direction> ::= 
  <left paren> <source node type name> <connector any direction> <destination node type name> <right paren>

<connector pointing right> ::= 
  TO
  | <right arrow>

<connector any direction> ::= 
  TO
  | <tilde>

<source node type name> ::= 
  !! Predicative production rule.
  <element type name>

<destination node type name> ::= 
  !! Predicative production rule.
  <element type name>

```

Syntax Rules

- 1) Let *ETD* be the <edge type definition>.
- 2) Let *GTDB* be the <graph type specification body> that simply contains *ETD*.

- 3) If *ETD* immediately contains an <edge type name>, then let *ETN* be that <edge type name>, otherwise let *ETN* be the zero-length character string.
- 4) Let *ETLSD* be the <edge type label set definition> simply contained in *ETD*.
- 5) If *ETD* contains an <edge type property type set definition>, then let *ETPTSD* be that <edge type property type set definition>, otherwise let *ETPTSD* be the zero-length character string.
- 6) If *ETD* simply contains an <endpoint pair definition> *EPD*, then

Case:

 - a) If *DIRECTED* is simply contained in *ETD*, then *EPD* shall not contain an <endpoint pair definition any direction> or an <abbreviated edge type pattern any direction>.
 - b) If *UNDIRECTED* is simply contained in *ETD*, then *EPD* shall contain an <endpoint pair definition any direction> or an <abbreviated edge type pattern any direction>.
- 7) *ETD* is transformed in the following steps:
 - a) If *ETD* simply contains an <endpoint pair definition> *EPD* and *EPD* does not contain an <abbreviated edge type pattern>, then
 - i) Let *SNTN* be the <source node type name> simply contained in *EPD*.
 - ii) Let *DNTN* be the <destination node type name> simply contained in *EPD*.
 - iii) Case:
 - 1) If *DIRECTED* is simply contained in *ETD*, then *EPD* is replaced by:

$$(\ SNTN \) \rightarrow (\ DNTN \)$$
 - 2) If *UNDIRECTED* is simply contained in *ETD*, then *EPD* is replaced by:

$$(\ SNTN \) \sim (\ DNTN \)$$

NOTE 100 — This rewrites an *ETD* containing an <endpoint pair definition> that is not an <abbreviated edge type pattern> into an *ETD* containing an <endpoint pair definition> that is an <abbreviated edge type pattern>.
 - b) If *ETD* simply contains an <abbreviated edge type pattern> *AETP*, then
 - i) Let *ASNTR* be the <source node type reference> simply contained in *AETP*.
 - ii) Let *ADNTR* be the <destination node type reference> simply contained in *AETP*.
 - iii) Case:
 - 1) If *AETP* does not contain an <abbreviated edge type pattern any direction>, then *ETD* is replaced by:

$$\text{ASNTR} - [\ ETN \ ETLSD \ ETPTSD \] \rightarrow \text{ADNTR}$$
 - 2) If *AETP* contains an <abbreviated edge type pattern any direction>, then *ETD* is replaced by

$$\text{ASNTR} \sim [\ ETN \ ETLSD \ ETPTSD \] \sim \text{ADNTR}$$

NOTE 101 — This rewrites an *ETD* containing an <abbreviated edge type pattern> into an *ETD* containing a <full edge type pattern>.
 - c) If *ETD* simply contains a <full edge type pattern pointing left> *FETPPL*, then

13.10 <edge type definition>

- i) Let $SNTR$ be the <source node type reference> simply contained in $FETPPL$.
- ii) Let $DNTR$ be the <destination node type reference> simply contained in $FETPPL$.
- iii) ETD is replaced by:

$SNTR - [ETN ETLSD ETPTSD] \rightarrow DNTR$

NOTE 102 — This rewrites an ETD containing a <full edge type pattern pointing left> into an ETD containing a <full edge type pattern pointing right>.

- d) If ETD simply contains an <edge type name> but does not simply contain an <edge type label set definition>, then the following <edge type filler> is implicit and replaces the existing <edge type filler>, if any:

: $ETN\ ETPTSD$

NOTE 103 — This is edge label implication.

- e) If ETD simply contains a <source node type name> $SNTN$, then

- i) $GTDB$ shall simply contain a <node type definition> NTD that simply contains a <node type name> that immediately contains $SNTN$.
- ii) If NTD contains a <node type filler>, then let NTF be that <node type filler>, otherwise let NTF be the zero-length character string.
- iii) $SNTN$ is replaced by:

NTF

NOTE 104 — This rewrites an ETD containing a <source node type name> into an ETD containing the appropriate <node type filler>.

- f) If ETD simply contains a <destination node type name> $DNTN$, then

- i) $GTDB$ shall simply contain a <node type definition> NTD that simply contains a <node type name> that immediately contains $DNTN$.
- ii) If NTD contains a <node type filler>, then let NTF be that <node type filler>, otherwise let NTF be the zero-length character string.
- iii) $DNTN$ is replaced by:

NTF

NOTE 105 — This rewrites an ETD containing a <destination node type name> into an ETD containing the appropriate <node type filler>.

General Rules

- 1) Let ETD be the <edge type definition> after the transformations in the Syntax Rules.
- 2) Let NTS be the set of all node types defined by the <node type definition>s simply contained in $GTDB$.
- 3) Let SNT be the node type defined by $SNTR$.
- 4) Let DNT be the node type defined by $DNTR$.
- 5) If SNT or DNT are not contained in NTS , then an exception condition *graph type definition error — endpoint node type not defined (G3001)* is raised.

- 6) If *ETD* contains an <edge type label set definition> *ETLSD*, then let *ETLS* the label set that is the result of the <label set definition> immediately contained in *ETLSD*. Otherwise let *ETLS* be an empty label set.
- 7) If *ETD* contains an <edge type property type set definition> *ETPTS*, then let *ETPTS* the property type set that is the result of the <property type set definition> immediately contained in *ETPTS*. Otherwise let *ETPTS* be an empty property type set.
- 8) The edge type defined by *ETD* is described by the edge type descriptor containing:
 - a) The edge type name *ETN*, if specified.
 - b) The edge type label set *ETLS*.
 - c) The edge type property type set *ETPTS*.
 - d) Case:
 - i) If *ETD* does not simply contain a <full edge type pattern any direction>, then:
 - 1) An indication that the edge type is directed.
 - 2) *SNT* as the source node type of *ETD*.
 - 3) *DNT* as the destination node type of *ETD*.
 - ii) Otherwise:
 - 1) An indication that the edge type is undirected.
 - 2) A set of endpoint node types containing *SNT* and *DNT*.

Conformance Rules

- 1) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain an <edge type definition> that simply contains an <edge kind> that is DIRECTED, an <endpoint pair definition> that is an <endpoint pair definition any direction>, or a <full edge type pattern> that is a <full edge type pattern any direction>.

13.11 <label set definition>

Function

Define a label set.

Format

```
<label set definition> ::=  
  LABEL <label>  
 | LABELS <label expression>  
 | <is label expression>
```

Syntax Rules

- 1) Let *LSD* be the <label set definition>.
- 2) *LSD* shall not contain a <label disjunction> nor a <label negation>.

General Rules

- 1) The label set defined by *LSD* is the set of <label>s contained in *LSD*.

Conformance Rules

None.

13.12 <property type set definition>

Function

Define a property type set.

Format

```

<property type set definition> ::= 
  <left brace> [ <property type definition list> ] <right brace>

<property type definition list> ::= 
  <property type definition> [ { <comma> <property type definition> }... ]

<property type definition> ::= 
  <property name> <type name>

```

Syntax Rules

- 1) Let $PTSD$ be the <property type set definition>.
- 2) No <property name> contained in $PTSD$ shall be equal to any other <property name> contained in $PTSD$.
- 3) Let $NPTD$ be the number of <property type definition>s immediately contained in the <property type definition list> immediately contained in $PTSD$.
- 4) For each k , $1 \leq k \leq NPTD$, let PTD_k be the k -th <property type definition>:
 - a) Let PN_k be the <property name> immediately contained in PTD_k .
 - b) Let TN_k be the <type name> immediately contained in PTD_k .
 - c) TN_k shall be the name of a property value type.
 - d) Let DT_k be the property value type with the name TN_k .

General Rules

- 1) For every k , $1 \leq k \leq NPTD$, let PT_k be a property type. The property type descriptor of PT_k comprises:
 - a) The property name PN_k .
 - b) The declared type DT_k .
- 2) The property type set defined by $PTSD$ is the set of all property type descriptors of PT_k for k , $1 \leq k \leq NPTD$.

Conformance Rules

None.

13.13 <drop graph type statement>

Function

Destroy a graph type.

Format

```
<drop graph type statement> ::=  
  DROP [ PROPERTY ] GRAPH TYPE <catalog graph type parent and name> [ IF EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CGTPN* be the <catalog graph type parent and name>.
- 2) Let *GTPS* be the implicit or explicit <graph type parent specification> immediately contained in *CGTPN*.
- 3) Let *GTN* be the <graph type name> immediately contained in *CGTPN*.

General Rules

- 1) Let *PARENT* be the descriptor of the result of *GTPS*.
- 2) If IF EXISTS is specified and *GTN* does not identify an existing graph type descriptor in *PARENT*, then the completion condition *warning — graph type does not exist (01G04)* is raised and no further General Rules of this Subclause are applied.
- 3) Destroy the graph type and graph type descriptor identified by *GTN* in *PARENT*.

Conformance Rules

None.

13.14<create procedure statement>

Function

Define a <create procedure statement>.

Format

```
<create procedure statement> ::=  
  CREATE {  
    PROCEDURE <catalog procedure parent and name> <of type signature> [ IF NOT EXISTS ]  
    | OR REPLACE PROCEDURE <catalog procedure parent and name> <of type signature>  
  } <procedure initializer>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

13.15 <drop procedure statement>

Function

Destroy a procedure.

Format

```
<drop procedure statement> ::=  
  DROP PROCEDURE <catalog procedure parent and name> [ IF EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CPPN* be the <catalog procedure parent and name>.
- 2) Let *PPS* be the implicit or explicit <procedure parent specification> immediately contained in *CPPN*.
- 3) Let *PN* be the <procedure name> immediately contained in *CPPN*.

General Rules

- 1) Let *PARENT* be the descriptor of the result of *PPS*.
- 2) If IF EXISTS is specified and *PN* does not identify an existing procedure descriptor in *PARENT*, then the completion condition *warning — procedure does not exist (01G06)* is raised and no further General Rules of this Subclause are applied.
- 3) Destroy the procedure descriptor identified by *PN* in *PARENT*.

Conformance Rules

None.

13.16 <create query statement>

Function

Define a <create query statement>.

Format

```
<create query statement> ::=  
  CREATE {  
    QUERY <catalog query parent and name> <of type signature> [ IF NOT EXISTS ]  
    | OR REPLACE QUERY <catalog query parent and name> <of type signature>  
  } <query initializer>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

13.17 <drop query statement>

Function

Destroy a query.

Format

```
<drop query statement> ::=  
  DROP QUERY <catalog query parent and name> [ IF EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CQPN* be the <catalog query parent and name>.
- 2) Let *QPS* be the implicit or explicit <query parent specification> immediately contained in *CQPN*.
- 3) Let *QN* be the <query name> immediately contained in *CQPN*.

General Rules

- 1) Let *PARENT* be the descriptor of the result of *QPS*.
- 2) If IF EXISTS is specified and *QN* does not identify an existing query descriptor in *PARENT*, then the completion condition *warning — query does not exist (01G07)* is raised and no further General Rules of this Subclause are applied.
- 3) Destroy the query descriptor identified by *QN* in *PARENT*.

Conformance Rules

None.

13.18<create function statement>

Function

Define a <create function statement>.

Format

```
<create function statement> ::=  
  CREATE {  
    FUNCTION <catalog function parent and name> <of type signature> [ IF NOT EXISTS ]  
    | OR REPLACE FUNCTION <catalog function parent and name> <of type signature>  
  } <function initializer>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

13.19 <drop function statement>

Function

Destroy a function.

Format

```
<drop function statement> ::=  
  DROP FUNCTION <catalog function parent and name> [ IF EXISTS ]
```

Syntax Rules

- 1) After applying all relevant syntactic transformations, let *CFPN* be the <catalog function parent and name>.
- 2) Let *FPS* be the implicit or explicit <function parent specification> immediately contained in *CFPN*.
- 3) Let *FN* be the <function name> immediately contained in *CFPN*.

General Rules

- 1) Let *PARENT* be the descriptor of the result of *FPS*.
- 2) If IF EXISTS is specified and *FN* does not identify an existing query descriptor in *PARENT*, then the completion condition *warning — function does not exist (01G08)* is raised and no further General Rules of this Subclause are applied.
- 3) Destroy the function descriptor identified by *FN* in *PARENT*.

Conformance Rules

None.

13.20 <call catalog-modifying procedure statement>

Function

Define a <call catalog-modifying procedure statement>.

Format

```
<call catalog-modifying procedure statement> ::=  
  <call procedure statement>
```

Syntax Rules

None.

General Rules

- 1) Let *CCPS* be the <call catalog-modifying procedure statement>, let *CPS* be the <call procedure statement> immediately contained in *CCPS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) Case:
 - a) If *PC* is the <inline procedure call> that immediately contains the <nested procedure specification> that does not immediately contain the <catalog-modifying procedure specification>, then an exception condition is raised:
 - b) If *PC* is the <named procedure call> that immediately contains the <procedure reference> *PR* and the result of *PR* is not a catalog-modifying procedure, then an exception condition is raised:
- 3) The General Rules for Subclause 12.2, “<call procedure statement>” are applied to *CPS*.

Conformance Rules

None.

14 Data-modifying statements

14.1 <linear data-modifying statement>

Function

Define a <linear data-modifying statement>.

Format

```

<linear data-modifying statement> ::= 
  <focused linear data-modifying statement>
  | <ambient linear data-modifying statement>

<focused linear data-modifying statement> ::= 
  <use graph clause> <focused linear data-modifying statement body>...

<focused linear data-modifying statement body> ::= 
  [ <simple linear query statement> ]
  [ { <use graph clause> <simple linear query statement> }... ]
  <simple data-modifying statement>
  [ <simple data-accessing statement>... ]
  [ { <use graph clause> <simple data-accessing statement> }... ]
  [ <primitive result statement> ]
  | <nested data-modifying procedure specification>

<ambient linear data-modifying statement> ::= 
  [ <simple linear query statement> ]
  <simple data-modifying statement>
  [ <simple data-accessing statement>... ]
  [ <primitive result statement> ]
  | <nested data-modifying procedure specification>

```

Syntax Rules

None.

General Rules

- 1) Let *LDMS* be the <linear data-modifying statement>.
- 2) The General Rules are applied to the <focused linear data-modifying statement> or <ambient linear data-modifying statement> immediately contained in *LDMS* to execute *LDMS*.
- 3) If a <focused linear data-modifying statement> *FLDMS* is specified, then *FLDMS* is executed by first applying the General Rules of Subclause 16.2, “<use graph clause>” to the immediately contained <use graph clause>, if any, followed by applying the General Rules to the immediately contained <focused linear data-modifying statement body>.
- 4) If a <focused linear data-modifying statement body> *FLDMSB* is specified, then *FLDMSB* is executed by applying the General Rules to all immediately contained <use graph clause>s, <simple linear query statement>s, <simple data-modifying statement>s, <simple data-accessing statement>s, <primitive

result statement>, and <nested data-modifying procedure specification> in the order of their occurrence from left to right.

- 5) If an <ambient linear data-modifying statement> *ALDMS* is specified, then *ALDMS* is executed by applying the General Rules to all immediately contained <simple linear query statement>s, <simple data-modifying statement>s, <simple data-accessing statement>s, <primitive result statement>, and <nested data-modifying procedure specification> in the order of their occurrence from left to right.
- 6) If a <query statement> *QS* is specified, then the General Rules are applied to the immediately contained <composite query statement>, <conditional query statement>, or the <select statement> to execute *QS*.

Conformance Rules

None.

14.2 <conditional data-modifying statement>

Function

Define a <conditional data-modifying statement>.

Format

```
<conditional data-modifying statement> ::=  
  <when then linear data-modifying statement branch>...  
  [ <else linear data-modifying statement branch> ]  
  
<when then linear data-modifying statement branch> ::=  
  <when clause> THEN <linear data-modifying statement>  
  | <when clause> <nested data-modifying procedure specification>  
  
<else linear data-modifying statement branch> ::=  
  ELSE <linear data-modifying statement>  
  
<when clause> ::=  
  WHEN <search condition>
```

Syntax Rules

- 1) Let *CDMS* be the <conditional data-modifying statement>.
- 2) If *CDM* immediately contains the <when then linear data-modifying statement branch> *WTLDMSB* such that *WTLDMSB* immediately contains only the <when clause> *WC* and the <nested data-modifying procedure specification> *NDMPS*, then *WTLDMSB* is effectively replaced by:

$$WC \text{ THEN } NDMPS$$
- 3) If *CDM* contains the <focused linear data-modifying statement> without an intervening <procedure body>, then *CDM* shall not contain an instance of <ambient linear data-modifying statement> without an intervening <procedure body>.
- 4) If *CDM* contains the <ambient linear data-modifying statement> without an intervening <procedure body>, then *CDM* shall not contain an instance of <focused linear data-modifying statement> without an intervening <procedure body>.
- 5) Let *BRANCHES* be the sequence of all <when then linear data-modifying statement branch>es and the (optional) <else linear data-modifying statement branch> that are immediately contained in *CDM* in the order of their occurrence in *CDM* from left to right, and let *NBRANCHES* be the number of elements of *BRANCHES*.
- 6) Let B_i be the *i*-the element of *BRANCHES* for $1 \leq i \leq NBRANCHES$.
- 7) B_i , $1 \leq i \leq NBRANCHES$ is an *executable branch* if it is either a <when then linear data-modifying statement branch> that immediately contains a <when clause> that immediately contains a <search condition> that is satisfied in the current execution context or if B_i is an <else linear data-modifying statement branch>.

General Rules

- 1) Let *NEW_TABLE* be a new empty binding table.

- 2) For each record R of the current working table in a new child execution context amended with R ,

Case:

- a) If there is a smallest integer $1 \leq k \leq NBRANCHES$ such that B_k is an executable branch, then
 - i) Let $LDMS$ be the <linear data-modifying statement> that is immediately contained in B_k .
 - ii) The General Rules of Subclause 12.1, “<statement>” for a <linear data-modifying statement> are applied to $LDMS$. Let $ROLDDBMS$ be the current working table available after the application of these General Rules.
 - iii) The Cartesian Product between R and $ROLDDBMS$ is appended to NEW_TABLE .
 - b) Otherwise, an exception condition is raised.
- 3) Set the current working table to NEW_TABLE .
- 4) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.3 <do statement>

Function

Execute a nested procedure without modifying the current working table.

Format

```
<do statement> ::=  
    DO <nested data-modifying procedure specification>
```

Syntax Rules

None.

General Rules

- 1) Let *DS* be the <do statement>.
- 2) Let *NDMPS* be the <nested data-modifying procedure specification> immediately contained in *DS*.
- 3) For each record *R* of the current working table in a new child execution context amended with *R*,
 - a) The General Rules of [Subclause 9.1, “<procedure specification>”](#) for a <nested data-modifying procedure specification> are applied to *NDMPS*.
- 4) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.4 <insert statement>

Function

Insert new nodes and edges into the current working graph.

Format

```
<insert statement> ::=  

  INSERT <simple graph pattern>  

  | OPTIONAL INSERT <simple graph pattern> [ <when clause> ]
```

Syntax Rules

- 1) Let *IS* be the <insert statement>.
- 2) Let *SGP* be the <simple graph pattern> immediately contained in *IS*.
- 3) If *IS* immediately contains OPTIONAL, then:
 - a) Case:
 - i) If *IS* immediately contains a <when clause> *WC*, then let *SC* be the <search condition> immediately contained in *WC*. Let *FS* be the <filter statement>:


```
FILTER SC
```

 - ii) Otherwise let *FS* be the zero-length character string.
 - b) *IS* is effectively replaced by the <optional statement>:


```
OPTIONAL {  
  FS  
  INSERT SGP  
  RETURN *  
}
```
 - 4) For each <element property specification> *EPS* simply contained in *SGP*:
 - a) Let *EPP* be the <element pattern predicate> that simply contains *EPS*.
 - b) Let *EPF* be the <element pattern filler> that simply contains *EPP*.
 - c) If *EPF* simply contains an <element variable declaration>, then:
 - i) Let *EVAR* be the <identifier> contained in the <element variable declaration> simply contained in *EPF*.
 - ii) Otherwise, let *EVAR* be an implementation-dependent <identifier> distinct from every element variable, subpath variable and path variable contained in *GP*.
 - d) Let *PECL* be the <property key value pair list> simply contained in *EPS*.
 - e) Let *NOPEC* be the number of <property key value pair>s simply contained in *PECL*.
 - f) Let *PEC₁*, ..., *PEC_{NOPEC}* be the <property key value pair>s simply contained in *PECL*.
 - g) For every *i*, $1 \leq i \leq NOPEC$:

14.4 <insert statement>

- i) Let $PROP_i$ be the <property name> simply contained in PEC_i .
- ii) Let VAL_i be the <value expression> simply contained in PEC_i .
- iii) Let $SETPEC_i$ be a <set item> formed as:

EVAR.PROP_i = VAL_i

- h) Let $EPSET$ be a <set item list> formed through the concatenation of <set item>s: $SETPEC_1$ AND ... AND ... $SETPEC_{NOPEC}$.
- i) EPP is effectively replaced by the zero-length character string and the following <set statement> is effectively inserted after *IS*:

SET EPSET

General Rules

- 1) Let $TABLE$ be the current working table.
- 2) Let NEW_TABLE be a new empty binding table.
- 3) For each record R of $TABLE$, in a new child execution context amended with R :
 - a) Insert new graph elements as specified by SGP and construct a new record NR that extends R with additional fields corresponding to the newly inserted graph elements.
 - b) Append NR to NEW_TABLE .
- 4) Set the current working table to NEW_TABLE .
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.5 <merge statement>

Function

Merge new nodes and edges into the current working graph.

Format

```
<merge statement> ::=  
  MERGE <simple graph pattern>
```

Syntax Rules

None.

General Rules

- 1) Let MS be the <merge statement>.
- 2) Let SGP be the <simple graph pattern> that is immediately contained in MS .
- 3) Let NEW_TABLE be a new empty binding table.
- 4) For each record R of the current working table in a new child execution context amended with R :
 - a) Merge graph elements as specified by SGP and append records whose fields bind those graph elements to the current working table.
 - b) Append the Cartesian product between R and the current working table to NEW_TABLE .
- 5) Set the current working table to NEW_TABLE .
- 6) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.6 <set statement>

Function

Set graph element properties and labels.

Format

```

<set statement> ::= 
    SET <set item list> [ <when clause> ]

<set item list> ::= 
    <set item> [ { <comma> <set item> }... ]

<set item> ::= 
    <set property item> | <set all properties item> | <set label item>

<set property item> ::= 
    <binding variable> <period> <property name> <equals operator> <value expression>

<set all properties item> ::= 
    <binding variable> <equals operator> <value expression>

<set label item> ::= 
    <label set expression>

<label set expression> ::= 
    & <label>... { & <label>... }
  
```

Syntax Rules

- 1) Let *SS* be the <set statement>.
- 2) Let *SIL* be the <set item list> immediately contained in *SS*.
- 3) If *SS* immediately contains a <when clause>, then let *SC* be the <search condition> immediately contained in *WC*. *SS* is effectively replaced by the <do statement>:

```

DO {
  FILTER SC
  SET SIL
}
  
```

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *n* be the number of records of *TABLE* and let *m* be the number of <set item>s immediately contained in *SIL*.
- 3) For each *i*-th record *R_i* of *TABLE*, $1 \leq i \leq n$ in a new child execution context amended with *R_i*:
 - a) For each *j*-th <set item> *SI_j* immediately contained in *SIL*, $1 \leq j \leq m$, determine the graph element *GE_{i,j}* as follows.

Case:

- i) If SI_j immediately contains a <set property item> SPI_j , let $GE_{i,j}$ be the result of evaluating the <binding variable> immediately contained in SPI_j .
 - ii) If SI_j immediately contains a <set all properties item> $SAPI_j$, let $GE_{i,j}$ be the result of evaluating the <binding variable> immediately contained in $SAPI_j$.
 - iii) If SI_j immediately contains a <set label item> SLI_j , let $GE_{i,j}$ be the result of evaluating the <binding variable> immediately contained in SLI_j .
- b) For each graph element $GE_{i,j}$, $1 \leq j \leq m$, if $GE_{i,j}$ is neither an existing graph element nor the null value then an exception condition is raised: *data exception — invalid graph element reference (22G09)*.
- c) For each j -th <set item> SI_j immediately contained in SIL , $1 \leq j \leq m$, determine the property value $PV_{i,j}$ as follows.
- Case:
- i) If SI_j immediately contains a <set property item> SPI_j , let $PV_{i,j}$ be the result of evaluating the <value expression> immediately contained in SPI_j .
 - ii) If SI_j immediately contains a <set all properties item> $SAPI_j$, let $PV_{i,j}$ be the result of evaluating the <value expression> immediately contained in $SAPI_j$.
 - iii) Otherwise, let $PV_{i,j}$ be undefined.
- d) For each j -th <set item> SI_j immediately contained in SIL , $1 \leq j \leq m$, if SI_j immediately contains a <set label item> SLI_j , let the label names $LN_{i,j}$ be the collection of all <label>s simply contained in SLI_j . Otherwise, let $LN_{i,j}$ be undefined.
- 4) For each i -th record R_i of $TABLE$, $1 \leq i \leq n$ (using the same order as [General Rule 3](#)), perform the following data-modifying operations in a new child execution context amended with R_i for each j -th <set item> SI_j immediately contained in SIL , $1 \leq j \leq m$,
- Case:
- a) If SI_j immediately contains a <set property item> SPI_j , then let PN_j be the <property name> immediately contained in SPI_j and if $GE_{i,j}$ is not the null value and $PV_{i,j}$ is not the null value, then set the property PN_j of $GE_{i,j}$ to $PV_{i,j}$, but if $GE_{i,j}$ is not the null value and $PV_{i,j}$ is the null value, then remove the property PN_j of $GE_{i,j}$.
 - b) If SI_j immediately contains a <set all properties item>, then if $GE_{i,j}$ is not the null value and if $PV_{i,j}$ is not the null value, then replace all properties of $GE_{i,j}$ with the fields of $PV_{i,j}$ but if $GE_{i,j}$ is not the null value and if $PV_{i,j}$ is the null value, then remove all properties of $GE_{i,j}$.
 - c) If SI_j immediately contains a <set label item> and $GE_{i,j}$ is not the null value, then set the labels $LN_{i,j}$ of $GE_{i,j}$.
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.7 <remove statement>

Function

Remove graph element properties and labels.

Format

```

<remove statement> ::= 
    REMOVE <remove item list> [ <when clause> ]

<remove item list> ::= 
    <remove item> [ { <comma> <remove item> }... ]

<remove item> ::= 
    <remove property item> | <remove label item>

<remove property item> ::= 
    <binding variable> <period> <property name>

<remove label item> ::= 
    <binding variable> <colon> <label set expression>

```

Syntax Rules

- 1) Let *RS* be the <remove statement>.
- 2) Let *RIL* be the <remove item list> immediately contained in *RS*.
- 3) If *RS* immediately contains a <when clause>, then let *SC* be the <search condition> immediately contained in *WC*. *RS* is effectively replaced by the <do statement>:

```

DO {
    FILTER SC
    REMOVE RIL
}

```

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *n* be the number of records of *TABLE* and let *m* be the number of <remove item>s immediately contained in *RIL*.
- 3) For each *i*-th record *R_i* of *TABLE*, $1 \leq i \leq n$ in a new child execution context amended with *R_i*:
 - a) For each *j*-th <remove item> *R_{i,j}* immediately contained in *RIL*, $1 \leq j \leq m$, determine the graph element *GE_{i,j}* as follows.

Case:

- i) If *R_{i,j}* immediately contains a <remove property item> *RPI_j*, let *GE_{i,j}* be the result of evaluating the <binding variable> immediately contained in *RPI_j*.
- ii) If *R_{i,j}* immediately contains a <remove label item> *RLI_j*, let *GE_{i,j}* be the result of evaluating the <binding variable> immediately contained in *RLI_j*.

- b) For each graph element GE_{ij} , $1 \leq j \leq m$, if GE_{ij} is neither an existing graph element nor the null value then an exception condition is raised: *data exception — invalid graph element reference (22G09)*.
 - c) For each j -th <remove item> RI_j immediately contained in RS , $1 \leq j \leq m$, if RI_j immediately contains a <remove label item> RLI_j , let the label names LN_{ij} be the collection of all <label>s simply contained in RLI_j . Otherwise, let LN_{ij} be undefined.
- 4) For each i -th record R_i of $TABLE$, $1 \leq i \leq n$ (using the same order as General Rule 3)), perform the following data-modifying operations in a new child execution context amended with R_i for each j -th <remove item> RI_j immediately contained in RIL , $1 \leq j \leq m$,
- Case:
- a) If RI_j immediately contains a <remove property item> RPI_j , then let PN_j be the <property name> immediately contained in RPI_j and if GE_{ij} is not the null value, then remove the property PN_j of GE_{ij} .
 - b) If RI_j immediately contains a <remove label item> and GE_{ij} is not the null value, then remove the labels LN_{ij} of GE_{ij} .
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.8 <delete statement>

Function

Delete graph elements.

Format

```
<delete statement> ::=  
  [ DETACH ] DELETE <delete item list> [ <when clause> ]  
  
<delete item list> ::=  
  <delete item> [ { <comma> <delete item> }... ]  
  
<delete item> ::=  
  <value expression>
```

Syntax Rules

- 1) Let *DS* be the <delete statement>.
- 2) If *DS* immediately contains DETACH, then let *PREFIX* be DETACH. Otherwise, let *PREFIX* be the zero-length character string.
- 3) Let *DIL* be the <delete item list> immediately contained in *DS*.
- 4) If *DS* immediately contains a <when clause>, then let *SC* be the <search condition> immediately contained in *WC*. *DS* is effectively replaced by the <do statement>:

```
DO {  
  FILTER SC  
  PREFIX DELETE DIL  
}
```

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *n* be the number of records of *TABLE* and let *m* be the number of <delete item>s immediately contained in *DIL*.
- 3) For each *i*-th record *R_i* of *TABLE*, $1 \leq i \leq n$ in a new child execution context amended with *R_i*:
 - a) For each *j*-th <delete item> *DI_j* immediately contained in *DIL*, $1 \leq j \leq m$, let the value *V_{i,j}* be the result of evaluating the <value expression> immediately contained in *DI_j*.
- 4) For each *i*-th record *R_i* of *TABLE*, $1 \leq i \leq n$ and for each *j*-th <delete item> *DI_j* immediately contained in *DIL*, $1 \leq j \leq m$ in a new child execution context amended with *R_i*:
 - a) If *V_{i,j}* is a reference value to a node *N* and DETACH is specified, then *N* and any edges connected to it are deleted.
 - b) If *V_{i,j}* is a reference value to a node *N* and DETACH is not specified, then

Case:

- i) If N has no edges, then N is deleted.
 - ii) Otherwise, an exception condition is raised: *dependent object error — edges still exist (G1001)*.
- c) If V_{ij} is a reference value to an edge E , then E is deleted.
- d) If V_{ij} is a path P , then perform all of the following steps completely unless an exception condition is raised:
- NOTE 106 — This specifies that P is to be deleted atomically with respect to following operations of the currently executing transaction, i.e., if any of its elements cannot be deleted, none of its constituting elements are.
- i) For each reference value E to an edge of P , delete its referent by applying [General Rule 4\)c\)](#) with V_{ij} defined as E .
 - ii) For each reference value N to a node of P , if DETACH is specified, then delete its referent by applying [General Rule 4\)a\)](#) with V_{ij} defined as N . Otherwise, delete its referent by applying [General Rule 4\)b\)](#) with V_{ij} defined as N .
- e) If V_{ij} is the null value, then do nothing.
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

14.9 <call data-modifying procedure statement>

Function

Define a <call data-modifying procedure statement>.

Format

```
<call data-modifying procedure statement> ::=  
  <call procedure statement>
```

Syntax Rules

None.

General Rules

- 1) Let *CDPS* be the <call data-modifying procedure statement>, let *CPS* be the <call procedure statement> immediately contained in *CDPS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) Case:
 - a) If *PC* is the <inline procedure call> that immediately contains the <nested procedure specification> that does not immediately contain the <data-modifying procedure specification>, then an exception condition is raised:
 - b) If *PC* is the <named procedure call> that immediately contains the <procedure reference> *PR* and the result of *PR* is not a data-modifying procedure, then an exception condition is raised:
- 3) The General Rules for Subclause 12.2, “<call procedure statement>” are applied to *CPS*.

Conformance Rules

None.

15 Query statements

15.1 <composite query statement>

Function

Define a <composite query statement>.

Format

```
<composite query statement> ::=  
  <composite query expression>
```

Syntax Rules

None.

General Rules

- 1) Let CQS be the <composite query statement> and let CQE be the <composite query expression> immediately contained in CQS .
- 2) Let $TABLE$ be the current working table.
- 3) The General Rules for Subclause 15.3, “<composite query expression>” are applied to CQE ; let NEW_TABLE be the result of the application of these General Rules.
- 4) Set the current working table to NEW_TABLE .
- 5) Set the current execution outcome to a successful outcome whose result table is NEW_TABLE .

Conformance Rules

None.

15.2 <conditional query statement>

Function

Define a <conditional query statement>.

Format

```

<conditional query statement> ::= 
  <when then linear query branch>... [ <else linear query branch> ]

<when then linear query branch> ::= 
  <when clause> THEN <linear query expression>
  | <when clause> <nested query specification>

<else linear query branch> ::= 
  ELSE <linear query expression>

```

Syntax Rules

- 1) Let CQS be the <conditional query statement>.
- 2) If CQE immediately contains the <when then linear query branch> $WTLQB$ such that $WTLQB$ immediately contains only the <when clause> WC and the <nested query specification> NQS , then $WTLQB$ is effectively replaced by:
 $WC \text{ THEN } NQS$
- 3) If CQS contains a <focused linear query statement> without an intervening <procedure body>, then CQS shall not contain an <ambient linear query statement> without an intervening <procedure body>.
- 4) If CQS contains an <ambient linear query statement> without an intervening <procedure body>, then CQS shall not contain a <focused linear query statement> without an intervening <procedure body>.
- 5) Let $BRANCHES$ be the sequence of all <when then linear query branch>es and the (optional) <else linear query branch> that are immediately contained in CQS in the order of their occurrence in CQS from left to right, and let $NBRANCHES$ be the number of elements of $BRANCHES$.
- 6) Let B_i be the i -the element of $BRANCHES$ for $1 \leq i \leq NBRANCHES$.
- 7) B_i ($1 \leq i \leq NBRANCHES$) is an *executable branch* if it is either a <when then linear query branch> that immediately contains a <when clause> that immediately contains a <search condition> that is satisfied in the current execution context or if B_i is an <else linear query branch>.

General Rules

- 1) Let NEW_TABLE be a new empty binding table.
- 2) For each record R of the current working table in a new child execution context amended with R ,
 Case:
 - a) If there is a smallest integer $1 \leq k \leq NBRANCHES$ such that B_k is an executable branch, then
 - i) Let LQE be the <linear query expression> that is immediately contained in B_k .

- ii) Let $ROLQE$ be the result of evaluating LQE .
 - iii) The Cartesian Product between R and $ROLQE$ is appended to NEW_TABLE .
- b) Otherwise, an exception condition is raised.
- 3) Set the current working table to NEW_TABLE .
- 4) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

15.3 <composite query expression>

Function

Define a <composite query expression>.

Format

```

<composite query expression> ::==
  <composite query expression> <query conjunction> <linear query expression>
  | <linear query expression>

<query conjunction> ::==
  <set operator>
  | OTHERWISE

<set operator> ::==
  UNION [ <set quantifier> ]
  | EXCEPT [ <set quantifier> ]
  | INTERSECT [ <set quantifier> ]

```

Syntax Rules

- 1) If <set operator> is specified and <set quantifier> is not specified, then DISTINCT is implicit.
- 2) Let *CQE* be the <composite query expression>.
- 3) If a <query conjunction> *QC* is immediately contained in *CQE*, then all <query conjunction>s contained in *CQE* without an intervening <procedure body> shall be *QC*.
- 4) If *CQE* contains a <focused linear query statement> without an intervening <procedure body> then *CQE* shall not contain an <ambient linear query statement> without an intervening <procedure body>.
- 5) If *CQE* contains an <ambient linear query statement> without an intervening <procedure body> then *CQE* shall not contain a <focused linear query statement> without an intervening <procedure body>.

General Rules

- 1) Case:
 - a) If <query conjunction> is specified, then:
 - i) Let *ICQE* be the <composite query expression> immediately contained in *CQE* and let *LQE* be the <linear query expression> immediately contained in *CQE*.
 - ii) Let *ICQER* be a new binding table set to the result of evaluating *ICQE*.
 - iii) Let *LQER* be a new binding table set to the result of evaluating *LQE*.
 - iv) Let *FCQER* be a new binding table.
 - v) If <set operator> is specified, then
 - 1) If *ICQER* and *LQER* do not have identical sets of column names, then an exception condition is raised: *syntax error or access rule violation — invalid reference (42002)*.
 - 2) *FCQER* contains the following records:

- A) Let R be a record that is a duplicate of some record in $ICQER$ or of some record in $LQER$ or both. Let m be the number of duplicates of R in $ICQER$ and let n be the number of duplicates of R in $LQER$, where $m \geq 0$ (zero) and $n \geq 0$ (zero).
- B) If $DISTINCT$ is specified or implicit, then
 - Case:
 - I) If $UNION$ is specified, then
 - Case:
 - 1) If $m > 0$ (zero) or $n > 0$ (zero), then $FCQER$ contains exactly one duplicate of R .
 - 2) Otherwise, $FCQER$ contains no duplicate of R .
 - II) If $EXCEPT$ is specified, then
 - Case:
 - 1) If $m > 0$ (zero) and $n = 0$ (zero), then $FCQER$ contains exactly one duplicate of R .
 - 2) Otherwise, $FCQER$ contains no duplicate of R .
 - III) If $INTERSECT$ is specified, then
 - Case:
 - 1) If $m > 0$ (zero) and $n > 0$ (zero), then $FCQER$ contains exactly one duplicate of R .
 - 2) Otherwise, $FCQER$ contains no duplicates of R .
 - C) If ALL is specified, then
 - Case:
 - I) If $UNION$ is specified, then the number of duplicates of R that $FCQER$ contains is $(m + n)$.
 - II) If $EXCEPT$ is specified, then the number of duplicates of R that $FCQER$ contains is the maximum of $(m - n)$ and 0 (zero).
 - III) If $INTERSECT$ is specified, then the number of duplicates of R that $FCQER$ contains is the minimum of m and n .
 - vi) If $OTHERWISE$ is specified, then
 - Case:
 - 1) If $ICQER$ contains at least one record, then let $FCQER$ be $ICQER$.
 - 2) Otherwise, let $FCQER$ be $LQER$.
 - vii) The result of the application of this Subclause is $FCQER$.
 - b) Otherwise, the result of the application of this Subclause is the result of the <linear query expression>.

Conformance Rules

None.

15.4 <linear query expression>

Function

Define a <linear query expression>.

Format

```
<linear query expression> ::=  
  <linear query statement>
```

Syntax Rules

None.

General Rules

- 1) Let LQE be the <linear query expression> and let LQS be the <linear query statement> immediately contained in LQE .
- 2) Let $TABLE$ be the current working table.
- 3) The General Rules for Subclause 15.5, “<linear query statement>” are applied to LQS in a new child execution context $CONTEXT$ with $TABLE$ as its working table; let $RESULT$ be the result available in $CONTEXT$ after the application of these General Rules.

NOTE 107 — Re-using the current working table in $CONTEXT$ ensures that all operands of a <composite query expression> are evaluated on it.

- 4) The result of the application of this Subclause is $RESULT$.

Conformance Rules

None.

15.5 <linear query statement>

Function

Define a <linear query statement>.

Format

```

<linear query statement> ::==
  <focused linear query statement>
  | <ambient linear query statement>

<focused linear query statement> ::==
  <from graph clause> <focused linear query statement body>
  | <select statement>

<focused linear query statement body> ::==
  [ <simple linear query statement>
    [ { <from graph clause> <simple linear query statement> }... ] ]
  <primitive result statement>
  | <nested query specification>

<ambient linear query statement> ::==
  [ <simple linear query statement> ] <primitive result statement>
  | <nested query specification>

<simple linear query statement> ::==
  <simple query statement>...

```

Syntax Rules

None.

General Rules

- 1) Let LQS be the <linear query statement>.
- 2) Let $ELTS$ be the sequence of <from graph clause>s, <simple query statement>s, <nested query specification>s, <primitive result statement>s, and <select statement>s contained in LQS without an intervening <procedure body> and in the order of their occurrence in LQS from left to right.
- 3) Let n be the number of elements of $ELTS$.
- 4) For every element ELT_i from $ELTS$, $1 \leq i \leq n$:

Case:

- a) If ELT_i is a <from graph clause>, then the General Rules of Subclause 16.1, “<from graph clause>” are applied to ELT_i .
- b) If ELT_i is a <simple query statement>, then the General Rules of Subclause 12.3, “Statement classes” for <simple query statement>s are applied to ELT_i .
- c) If ELT_i is a <nested query specification>, then the General Rules of Subclause 9.1, “<procedure specification>” for <nested query specification>s are applied to ELT_i .

- d) If ELT_i is a <primitive result statement>, then the General Rules of Subclause 15.8.1, “<primitive result statement>” are applied to ELT_i .
- e) If ELT_i is a <select statement>, then the General Rules of Subclause 15.8.3, “<select statement>” are applied to ELT_i .

Conformance Rules

None.

15.6 Data-reading statements

15.6.1 <match statement>

Function

Expand the current working table with matches from a graph pattern.

Format

```
<match statement> ::=  
  [ <statement mode> ] MATCH <graph pattern>
```

Syntax Rules

- 1) Let *MS* be the <match statement>.
- 2) Let *GP* be the <graph pattern> that is immediately contained in *MS*.
- 3) If *MS* immediately contains a <statement mode> *SM*, then

Case:

- a) If *SM* is OPTIONAL, then *MS* is effectively replaced by the <optional statement>:

```
OPTIONAL { MATCH GP RETURN * }
```

- b) Otherwise, *MS* is effectively replaced by the <mandatory statement>:

```
MANDATORY { MATCH GP RETURN * }
```

- 4) For each <element property specification> *EPS* simply contained in *GP*:
 - a) Let *EPP* be the <element pattern predicate> that simply contains *EPS*.
 - b) Let *EPF* be the <element pattern filler> that simply contains *EPP*.
 - c) If *EPF* simply contains an <element variable declaration>, then:
 - i) Let *EVAR* be the <identifier> contained in the <element variable declaration> simply contained in *EPF*.
 - ii) Otherwise, let *EVAR* be an implementation-dependent <identifier> distinct from every element variable, subpath variable and path variable contained in *GP*.
 - d) Let *PECL* be the <property key value pair list> simply contained in *EPS*.
 - e) Let *NOPEC* be the number of <property key value pair>s simply contained in *PECL*.
 - f) Let *PEC₁*, ..., *PEC_{NOPEC}* be the <property key value pair>s simply contained in *PECL*.
 - g) For every *i*, $1 \leq i \leq NOPEC$:
 - i) Let *PROP_i* be the <property name> simply contained in *PEC_i*.
 - ii) Let *VAL_i* be the <value expression> simply contained in *PEC_i*.
 - iii) Let *RPEC_i* be a <comparison predicate> formed as:

EVAR.PROP_i = VAL_i

- h) Let *EPSC* be a <boolean value expression> formed through the concatenation of <boolean factor>s: *RPEC₁* AND ... AND ... *RPEC_{NOPEC}*.
- i) *EPP* is effectively replaced by:

WHERE *EPSC*

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *NEW_TABLE* be a new empty binding table.
- 3) For each record *R* of *TABLE* in a new child execution context amended with *R*:
 - a) The General Rules of Subclause 16.6, “<graph pattern>” are applied for *GP*; let *M* be the *MATCHES* returned from the application of these General Rules.
 - b) Append the Cartesian product of *R* and *M* to *NEW_TABLE*.
- 4) Set the current working table to *NEW_TABLE*.
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

15.6.2 <call query statement>

Function

Define a <call query statement>.

Format

```
<call query statement> ::=  
  <call procedure statement>
```

Syntax Rules

None.

General Rules

- 1) Let *CQS* be the <call query statement>, let *CPS* be the <call procedure statement> immediately contained in *CQS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) Case:
 - a) If *PC* is the <inline procedure call> that immediately contains the <nested procedure specification> that does not immediately contain the <query specification>, then an exception condition is raised:
 - b) If *PC* is the <named procedure call> that immediately contains the <procedure reference> *PR* and the result of *PR* is not a query, then an exception condition is raised:
- 3) The General Rules for Subclause 12.2, “<call procedure statement>” are applied to *CPS*.

Conformance Rules

None.

15.7 Data-transforming statements

15.7.1 <mandatory statement>

Function

Raise an exception condition if the specified <procedure body> sets the current working table to an empty binding table.

Format

```
<mandatory statement> ::=  
    MANDATORY <procedure call>
```

Syntax Rules

- 1) Let *MS* be the <mandatory statement>.
- 2) Let *PC* be the <procedure call> immediately contained in *MS*.
- 3) The declared type of *PC* shall be a binding table type.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Let *NEW_TABLE* be a new empty binding table.
- 3) For each record *R* of *TABLE*, in a new child execution context amended with *R*:
 - a) The General Rules of Subclause 16.13, “<procedure call>” are applied to *PC*; let *RESULT* be the binding table *RESULT* obtained from the application of these General Rules.
 - b) Case:
 - i) If *RESULT* is an empty result table, then an exception condition is raised: *data exception — empty binding table returned (22G0D)*.
 - ii) Otherwise, the Cartesian product of *R* and *RESULT* is appended to *NEW_TABLE*.
- 4) Set the current working table to *NEW_TABLE*.
- 5) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

15.7.2 <optional statement>

Function

Replace each record of the current working table with a set of new records with additional fields unless that set is empty in which case the record is extended with fields containing the null value instead.

Format

```
<optional statement> ::=  
    OPTIONAL <procedure call>
```

Syntax Rules

- 1) Let OS be the <optional statement>.
- 2) Let PC be the <procedure call> immediately contained in OS .
- 3) Let $DTPC$ be the declared type of PC .
- 4) $DTPC$ shall be a binding table type.

General Rules

- 1) Let $TABLE$ be the current working table.
- 2) Let NEW_TABLE be a new empty binding table.
- 3) Let $FTSET$ be the set of field types of the record type of $DTPC$.
- 4) For each record R of $TABLE$ in a new child execution context amended with R :
 - a) The General Rules of Subclause 16.13, “<procedure call>” are applied to PC ; let $RESULT$ be the binding table $RESULT$ obtained from the application of these General Rules.
 - b) Case:
 - i) If $RESULT$ is an empty result table, then a new result table comprising a single record NR that extends R with additional fields that are completely derived from $FTSET$ such that for each field type FT from $FTSET$ there is a field F in NR whose field name is the field name of FT , whose field type is the field type of FT , and whose field value is the null value is appended to NEW_TABLE .
 - ii) Otherwise, the Cartesian product of R and $RESULT$ is appended to NEW_TABLE .
- 5) Set the current working table to NEW_TABLE .
- 6) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

15.7.3 <filter statement>

Function

Select a subset of the records of the current working table.

Format

```
<filter statement> ::=  
  FILTER { <where clause> | <search condition> }
```

Syntax Rules

- 1) Let *FS* be the <filter statement>.
- 2) If *FS* immediately contains the <search condition> *SC*, then it effectively is replaced by the <filter statement>:

```
FILTER WHERE SC
```

General Rules

- 1) Let *WC* be the <where clause> that is immediately contained in *FS*.
- 2) The General Rules for Subclause 16.12, “<where clause>” are applied to *WC*; let *NEW_TABLE* be the binding table *WHERE* returned as the result of the application of these General Rules.
- 3) Set the current working table to *NEW_TABLE*.

Conformance Rules

None.

15.7.4 <let statement>

Function

Add columns to the current working table.

Format

```
<let statement> ::=  
    LET <compact variable definition list>  
  | <statement mode> LET <compact variable definition list> <where clause>
```

Syntax Rules

- 1) Let LS be the <let statement> and let $CVDL$ be the <compact variable definition list> immediately contained in LS .
- 2) Let $CVDSEQ$ be the sequence of <compact variable definition>s immediately contained in $CVDL$ in the order of their occurrence in $CVDL$ from left to right and let n be the number of elements of $CVDSEQ$.
- 3) Let CVD_i be the i -th element of $CVDSEQ$, for $1 \leq i \leq n$.
- 4) Case:
 - a) If LS immediately contains the <statement mode> OPTIONAL, then let WC be the <where clause> immediately contained in LS , let SC be the <search condition> immediately contained in WC . LS is effectively replaced by:

```
OPTIONAL {  
  FILTER SC  
  LET CVDL  
  RETURN *  
}
```

- b) If LS immediately contains the <statement mode> MANDATORY, then let WC be the <where clause> immediately contained in LS , let SC be the <search condition> immediately contained in WC . LS is effectively replaced by:

```
MANDATORY {  
  FILTER SC  
  LET CVDL  
  RETURN *  
}
```

- c) Otherwise, let the new <binding variable definition block> $DBLK$ be the <newline>-separated list of all CVD_i for $1 \leq i \leq n$. LS is effectively replaced by:

```
CALL {  
  DBLK  
  RETURN *  
}
```

General Rules

None.

Conformance Rules

None.

15.7.5 <aggregate statement>

Function

Aggregate over the current working table.

Format

```
<aggregate statement> ::=  
    AGGREGATE <compact value variable definition list> <where clause>
```

Syntax Rules

- 1) Let *AS* be the <aggregate statement>.
- 2) Let *CVVSEQ* be the sequence of every <compact value variable definition> in the <compact value variable definition list> immediately contained in *AS* in the order of their occurrence from left to right and let *NCVVSEQ* be the number of elements of *CVVSEQ*.
- 3) Let *CVVN_i* be the <value variable> immediately contained in the *i*-the element of *CVVSEQ* and let *CVVE_i* be the <value expression> immediately contained in the *i*-the element of *CVVSEQ*, for 1 (one) $\leq i \leq NCVVSEQ$.
- 4) *CVVN_i* shall not be the name of a binding variable that is in scope, for 1 (one) $\leq i \leq NCVVSEQ$.
- 5) *CVVE_i* shall be an <aggregate function>, for 1 (one) $\leq i \leq NCVVSEQ$.
- 6) *CVVN_i* is a new fixed variable in scope after *AS* whose declared type is the declared type of *CVVE_i*, for 1 (one) $\leq i \leq NCVVSEQ$.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) If *AS* immediately contains the <where clause> *WC*, then the General Rules for Subclause 16.12, “<where clause>” are applied to *WC*; let *FILTERED_TABLE* be the result of the application of these General Rules. Otherwise let *FILTERED_TABLE* be a copy of *TABLE*.
- 3) Set the current working table to *FILTERED_TABLE*.
- 4) Let *R* be a new record with fields *F_i* such that the field name of *F_i* is the name of *CVVN_i* and the field value of *F_i* is the result of evaluating *CVVE_i*, for 1 (one) $\leq i \leq NCVVSEQ$.
- 5) Let *RESULT_TABLE* be the Cartesian Product between *TABLE* and *R*.
- 6) Set the current working table to *RESULT_TABLE*.
- 7) Set the current execution outcome to a successful outcome with an omitted result.

Conformance Rules

None.

15.7.6 <for statement>

Function

Provide iteration over (unnesting of) collections by expanding the current working table.

Format

```

<for statement> ::=

  [ <statement mode> ] FOR <for item list> [ <for ordinality or index> ] [ <where clause> ]
  ]

<for item list> ::=
  <for item> [ { AND <for item> }... ]

<for item> ::=
  <for item alias> <collection value expression>

<for item alias> ::=
  <identifier> IN

<for ordinality or index> ::=
  WITH { ORDINALITY | INDEX } [ <identifier> ]

```

Syntax Rules

- 1) Let *FS* be the <for statement>.
- 2) Let *FIL* be the <for item list> that is immediately contained in *FS* and let *NFIL* be the number of elements of *FIL*.
- 3) Let *FOI* be the <for ordinality or index> if specified. Otherwise, let *FOI* be the zero-length character string.
- 4) Let *WC* be the <where clause> if specified. Otherwise, let *WC* be the zero-length character string.
- 5) Case:
 - a) If *FS* immediately contains a <statement mode> *SM*, then

Case:

 - i) If *SM* is OPTIONAL, then *FS* is effectively replaced by the <optional statement>:

```
OPTIONAL { FOR FIL FOI WC RETURN * }
```

 - ii) If *SM* is MANDATORY, then *FS* is effectively replaced by the <mandatory statement>:

```
MANDATORY { FOR FIL FOI WC RETURN * }
```
 - b) Otherwise:
 - i) If *WC* immediately contains the <search condition> *SC*, then *FS* is effectively replaced by:


```
FILTER WHERE SC
FOR FIL FOI
```
 - ii) If *FOI* immediately specifies WITH ORDINALITY but does not immediately contain an <identifier>, then *FS* is effectively replaced by the <for statement>:

FOR *FIL* WITH ORDINALITY *ordinality*

- iii) If *FOI* immediately specifies WITH INDEX but does not immediately contain an <identifier>, then *FS* is effectively replaced by the <for statement>:

FOR *FIL* WITH INDEX *index*

General Rules

- 1) Let *NEW_TABLE* be a new binding table.
- 2) Let *FI_i* be the *i*-th <for item> of *FIL*, let *FCVE_i* be the <collection value expression> immediately contained in *FI_i*, and let *FIA_i* be the <for item alias> immediately contained in *FI_i*, for $1 \leq i \leq NFIL$.
- 3) For each record *R* of the current working table in a new child execution context:
 - a) Set the current working record to *R*.
 - b) Let *FCVER_i* be the result of evaluating *FCVE_i* in the current execution context, for $1 \leq i \leq NFIL$.
 - c) Let the list *LV_i* be determined as follows, for $1 \leq i \leq NFIL$,

Case:

 - i) If *FCVER_i* is a list, then *LV_i* is *FIVER_i*.
 - ii) If *FCVER_i* is a set or a multiset, then *LV_i* is an implementation-dependent permutation of all elements of *FIVER_i*.
 - iii) If *FCVER_i* is the null value, then *LV_i* is the empty list.
 - iv) Otherwise, an exception condition is raised: *data exception — invalid value type (22G03)*.
 - d) Let *LVMAXLEN* be the largest length of any list *LV_i*, $1 \leq i \leq NFIL$.
 - e) For each *j*, $1 \leq j \leq LVMAXLEN$:
 - i) For each *i*, $1 \leq i \leq NVIL$, if *j* is less than or equal to the length of *LV_i*, then let *LVE_{i,j}* be the *j*-th element of *LV_i*. Otherwise let *LVE_{i,j}* be the null value.
 - ii) Let *LR_j* be the record with the fields *LRF_{i,j}*, $1 \leq i \leq NFIL$ such that the field name of each field *LRF_{i,j}* is the <identifier> immediately contained in *FIA_i* and the field value of each field *LRF_{i,j}* is *LVE_{i,j}*.
 - iii) If the <for ordinality or index> is specified, then

Case:

 - 1) If *FOI* immediately contains WITH ORDINALITY, then let *LRO_j* be the record obtained by adding a field to *LR_j* whose field name is the <identifier> that is immediately contained in *FOI* and whose field value is *j* represented as a value of an implementation-defined exact numeric type with scale 0 (zero).
 - 2) If *FOI* immediately contains WITH INDEX, then let *LRO_j* be the record obtained by adding a field to *LR_j* whose field name is the <identifier> that is immediately contained

in FOI and whose field value is $j-1$ represented as a value of an implementation-defined exact numeric type with scale 0 (zero).

- 3) Otherwise, let LRO_j be LR_j .
 - iv) Append LRO_j to the current working table.
 - f) Append the Cartesian product of R and the current working table to NEW_TABLE .
- 4) Set the current working table to NEW_TABLE .

Conformance Rules

None.

15.7.7 <order by and page statement>

Function

Order the records of the current working table according to a provided sort specification and optionally retain only a specified number of records.

Format

```
<order by and page statement> ::=  
  <order by clause> [ <offset clause> ] [ <limit clause> ]  
  | <offset clause> [ <limit clause> ]  
  | <limit clause>
```

Syntax Rules

None.

General Rules

- 1) Let *OPS* be the <order by and page statement>.
- 2) Let *TABLE* be the current working table.
- 3) Let *ORDERED* be the binding table determined as follows.

Case:

- a) If *OPS* immediately contains the <order by clause> *OBC*, then the General Rules of Subclause 16.20, “<order by clause>” are applied to *OBC* in a new child execution context with *TABLE* as the working table; let *ORDERED* be the binding table *ORDER_BY* returned from the application of these General Rules.
- b) Otherwise, let *ORDERED* be *TABLE*.

- 4) Let *OFFSETED* be the binding table determined as follows.

Case:

- a) If *OPS* immediately contains the <offset clause> *OC*, then the General Rules of Subclause 16.24, “<offset clause>” are applied to *OC* in a new child execution context with *ORDERED* as the working table; let *OFFSETED* be the binding table *OFFSET* returned from the application of these General Rules.
- b) Otherwise, let *OFFSETED* be *ORDERED*.

- 5) Let *LIMITED* be the binding table determined as follows.

Case:

- a) If *OPS* immediately contains the <limit clause> *LC*, then the General Rules of Subclause 16.23, “<limit clause>” are applied to *LC* in a new child execution context with *OFFSETED* as the working table; let *LIMITED* be the binding table *LIMIT* returned from the application of these General Rules.
- b) Otherwise, let *LIMITED* be *OFFSETED*.

- 6) Set the current working table to *LIMITED*.

- 7) Set the current execution outcome to a successful outcome with an unknown result.

Conformance Rules

None.

15.7.8 <call function statement>

Function

Define a <call function statement>.

Format

```
<call function statement> ::=  
  <call procedure statement>
```

Syntax Rules

None.

General Rules

- 1) Let *CFS* be the <call function statement>, let *CPS* be the <call procedure statement> immediately contained in *CFS*, and let *PC* be the <procedure call> immediately contained in *CPS*.
- 2) Case:
 - a) If *PC* is the <inline procedure call> that immediately contains the <nested procedure specification> that does not immediately contain the <function specification>, then an exception condition is raised:
 - b) If *PC* is the <named procedure call> that immediately contains the <procedure reference> *PR* and the result of *PR* is not a function, then an exception condition is raised:
- 3) The General Rules for Subclause 12.2, “<call procedure statement>” are applied to *CFS*.

Conformance Rules

None.

15.8 Result projection statements

15.8.1 <primitive result statement>

Function

Define what to include in a query result set.

Format

```
<primitive result statement> ::=  
    <return statement> [ <order by and page statement> ]  
  | <project statement>  
  | END
```

Syntax Rules

None.

General Rules

- 1) If the <return statement> *RS* is specified:
 - a) The General Rules of Subclause 15.8.2, “<return statement>” are applied to *RS*.
 - b) If the <order by and page statement> *OPS* is specified, then the General Rules of Subclause 15.7.7, “<order by and page statement>” are applied to *OPS*.
- 2) If the <project statement> *PS* is specified, then the General Rules of Subclause 15.8.4, “<project statement>” are applied to *PS*.
- 3) If END is specified, then the current execution outcome is set to a successful outcome with an omitted result.

Conformance Rules

None.

15.8.2 <return statement>

Function

Define a <return statement> for determining the result table.

Format

```

<return statement> ::= 
    RETURN <return statement body>

<return statement body> ::= 
    [ <set quantifier> ] { <asterisk> | <return item list> }
    [ <group by clause> ]

<return item list> ::= 
    <return item> [ { <comma> <return item> }... ]

<return item> ::= 
    <value expression> [ <return item alias> ]

<return item alias> ::= 
    AS <identifier>

```

Syntax Rules

- 1) Let *RS* be the <return statement>.
- 2) Let *RSB* be the <return statement body> immediately contained in *RS*.
- 3) If a <set quantifier> is not immediately contained in *RSB*, then ALL is the implicit <set quantifier> of *RSB*.
- 4) Let *SQ* be the explicit or implicit <set quantifier> of *RSB*.
- 5) If *RSB* immediately contains a <group by clause>, then let *GBC* be that <group by clause>. Otherwise, let *GBC* be the zero-length character string.
- 6) If *RSB* immediately contains an <asterisk>, then:
 - a) *RSB* shall not immediately contain a <group by clause>.
 - b) Let *BVSEQ* be the permutation of all binding variables in scope for *RS*, ordered in standard sort order, let *NBVSEQ* be the number of such binding variables, and let *BVi* be the *i*-th such binding variable in *BVSEQ*, for $1 \leq i \leq NBVSEQ$.
 - c) For $1 \leq i \leq NBVSEQ$, let the new <return item list> *NRIL* be a comma-separated list of <return item>s:

BVi AS BVi
 - d) *RS* is effectively replaced by the <return statement>:

RETURN *SQ NRIL GBC*
- 7) Let *RIL* be the <return item list> immediately contained in *RSB*, let *NRIL* be the number of elements of *RIL*, and let *RIi* be the *i*-th element of *RIL*, for $1 \leq i \leq NRIL$.

- 8) For any <return item> RI , let the *expression of RI* be the <value expression> immediately contained in RI and if RI immediately contains a <return item alias> RIA , then let the *alias name* of RI be the <identifier> that is immediately contained in RIA .
- 9) For each <return item> RI_i from RIL , $1 \leq i \leq NRIL$,

Case:

- a) If the expression of RI_i immediately contains a <binding variable> $RIBV_i$ but RI_i does not have an alias name, then RI_i is effectively replaced by the <return item>:

$RIBV_i$ AS $RIBV_i$

- b) Otherwise, RI_i shall have an alias name.

- 10) Let the set of preserved column names $PCNSET$ be the set of all binding variables in scope for RS that correspond to a column name of the current working table but that are different from the alias name of any <return item> in RIL and let $NPCN$ be the number of elements of $PCNSET$.

- 11) Case:

- a) If GBC is the zero-length character string, then:

- i) Let $GRISET$ be the set of *grouping* <return item>s contained in RIL whose expression contains no <aggregate function> and let $NGRI$ be the number of such <return item>s in $GRISET$.
- ii) Let $ARISET$ be the set of *aggregating* <return item>s contained in RIL whose expression contains an <aggregate function> and let $NARI$ be the number of such <return item>s in $ARISET$.
- iii) If $NARI$ is greater than 0 (zero), then:
 - 1) For each j between 1 (one) and $NGRI$, let GRI_j be an enumeration of the <return item>s of $GRISET$ and let $GRIAI_j$ be an enumeration of the alias names of GRI_j .
 - 2) Let GEL be the comma-separated list of <grouping element>s:

$GRIAI_1, \dots, GRIAI_{NGRI}$

- 3) The following <group by clause> is implicit:

Case:

- A) If $NGRI$ is greater than 0 (zero), then:

GROUP BY GEL

- B) Otherwise:

GROUP BY ()

- b) Otherwise:

- i) Let $GRISET$ be the set of grouping <return item>s contained in RIL whose alias name is simply contained in $FGBC$ and let $NGRI$ be the number of such <return item>s in $GRISET$.
- ii) Let $ARISET$ be the set of aggregating <return item>s contained in RIL whose alias name is not simply contained in $FGBC$ and let $NARI$ be the number of such <return item>s in $ARISET$.

15.8 Result projection statements

- 12) Let $XRISET$ be the set of all <return item>s contained in RIL that are not contained in $GRISET$ and that are not contained in $ARISET$ and let $NXRI$ be the number of such <return item>s in $XRISET$.
- 13) If a <binding variable> contained in the expression RIE of some <return item> RI of RIL is not either a binding variable in the scope for RS or reference a fixed variable that is defined in RIE , then an exception condition is raised: *syntax error or access rule violation — invalid reference (42002)*.

General Rules

- 1) Let $TABLE$ be the current working table, let n be the number of records of $TABLE$, and let R_i be the i -th record of $TABLE$ in an order determined by iteration over $TABLE$, for $1 \text{ (one)} \leq i \leq n$.
- 2) Let $RETURN_TABLE$ be a new empty binding table.
- 3) Let $FGBC$ be the implicit or explicit *group by clause* that is immediately contained in RS .
- 4) Case:
 - a) If $FGBC$ is the zero-length character string, then for each record R of $TABLE$ in a new child execution context amended with R :
 - i) Let S be a new record with fields F_i such that the field name of F_i is the alias name of RI_i and the field value of F_i is the result of evaluating the expression of RI_i for each <return item> RI_i from RIL , $1 \text{ (one)} \leq i \leq n$.
 - ii) If SQ is ALL, then let SP be a new record comprising all fields of S and additional fields F_i such that the field name of F_i is PCN_i and the field value of F_i is the field value of the field with field name PCN_i in R for each preserved column name PCN_i from $PCNSET$, $1 \text{ (one)} \leq i \leq NPCN$. Otherwise, let SP be S .
 - iii) Add SP to $RETURN_TABLE$.
 - b) Otherwise:
 - i) Let the *grouping record* GR_i of a record R_i of $TABLE$ be a new record with fields F_k such that the field name of F_k is the alias name of GRI_k and the field value of F_k is the result of evaluating the expression of GRI_k in a new child execution context amended with R_i , for $1 \text{ (one)} \leq i \leq n$ and $1 \text{ (one)} \leq k \leq NGRI$.
 - ii) Let GR_TABLE be a new empty binding table of all grouping records GR_i of all records R_i of $TABLE$, for $1 \text{ (one)} \leq i \leq n$.
 - iii) The General Rules of Subclause 16.17, “<group by clause>” are applied to GBC in a new child execution context with working table GR_TABLE ; let $GROUP_BY$ be the binding table $GROUP_BY$ returned from the application of these General Rules.
 - iv) Let $KEYS$ be a permutation of $GROUP_BY$ in an implementation-dependent order and let $NKEYS$ be the number of elements of $KEYS$.
 - v) For each i -th element of $KEYS$ K_i , $1 \text{ (one)} \leq i \leq NKEYS$:
 - 1) Let $PART_i$ be a new binding table comprising only the records R_j from $TABLE$, $1 \text{ (one)} \leq j \leq n$ for which the grouping record GR_j for R_j is equivalent to K_i .
 - 2) Let $NPART_i$ be the number of records of $PART_i$.
 - 3) For each record $P_{i,j}$, $1 \text{ (one)} \leq j \leq NPART_i$ in a new child execution context:

- A) Set the current working record to $P_{i,j}$.
 - B) Set the current working table to $PART_i$.

NOTE 108 — This is used to determine the result of evaluating <aggregate function>s.
 - C) Let $NR_{i,j}$ be a new record comprising
 - I) all fields of K_i ,
 - II) additional fields AF_k such that the field name of AF_k is the alias name of ARI_k and the field value of AF_k is the result of evaluating the expression of ARI_k , for $1 \leq k \leq NARI$, and
 - III) additional fields XF_k such that the field name of XF_k is the alias name of XRI_k and the field value of XF_k is the result of evaluating the expression of XRI_k , for $1 \leq k \leq NXRI$.
 - D) Append $NR_{i,j}$ to $RETURN_TABLE$.
- 5) If SQ is DISTINCT, then set the current working table to a duplicate-free binding table obtained from the set of all records of $RETURN_TABLE$. Otherwise, set the current working table to $RETURN_TABLE$.
 - 6) If SQ is ALL and $FGBC$ is the zero-length character string, then let $SANITIZED_TABLE$ be the binding table obtained from the current working table by discarding all columns whose column name is in $PCNSET$ and set the current working table to $SANITIZED_TABLE$.
 - 7) Let $FINAL_TABLE$ be a new binding table obtained from the current working table by determining the preferred column sequence to be the sequence of alias names of all <return item>s RI_i from RIL , $1 \leq i \leq NRIL$ in the order of their occurrence in RIL .
 - 8) Set the current working table to $FINAL_TABLE$.
 - 9) Set the current execution outcome to a successful outcome whose result table is $FINAL_TABLE$.

Conformance Rules

None.

15.8.3 <select statement>

Function

Provide an SQL-style query over graph data to produce a tabular query result set.

Format

```

<select statement> ::=

  SELECT [ <set quantifier> ] <select item list>
  <select statement body>
  [ <where clause> ]
  [ <group by clause> ]
  [ <having clause> ]
  [ <order by clause> ]
  [ <offset clause> ] [ <limit clause> ]

<select item list> ::=
  <select item> [ { <comma> <select item> }... ]

<select item> ::=
  <value expression> [ <select item alias> ]

<select item alias> ::=
  AS <identifier>

<having clause> ::=
  HAVING <search condition>

<select statement body> ::=
  FROM <select graph match list>
  | <select query specification>

<select graph match list> ::=
  <select graph match> [ { <comma> <select graph match> }... ]

<select graph match> ::=
  <graph expression> <match statement>

<select query specification> ::=
  FROM <nested query specification>
  | <from graph clause> <nested query specification>

```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

15.8.4 <project statement>

Function

Define a <project statement> for projecting a result.

Format

```
<project statement> ::=  
    PROJECT <value expression>
```

Syntax Rules

- 1) Let *VE* be the specified <value expression>.
- 2) If *VE* contains a <binding variable> *BV* without an intervening instance of <procedure body>, then one of the following conditions shall be true:
 - a) *BV* identifies an iterated variable and *BV* is either contained in a <dependent value expression> that is contained in *VE* or *BV* is contained in a <value expression> that is immediately contained in a <general set function> that is contained in *VE*.
 - b) *BV* identifies a fixed variable.

General Rules

- 1) Let *ROVE* be the result of evaluating *VE*.
- 2) If the *ROVE* is a reference value, then the object type of the identified object shall be a valid result object type, as specified by Subclause 4.9.4, “Execution outcomes”. Otherwise, an exception condition is raised:
- 3) Set the current execution outcome to a successful outcome whose result is *ROVE*.

Conformance Rules

None.

16 Common elements

16.1 <from graph clause>

Function

Set the current working graph in a <focused linear query statement> or <select statement>.

Format

```
<from graph clause> ::=  
    FROM <graph expression>
```

Syntax Rules

None.

General Rules

- 1) Let *FGC* be the <from graph clause>.
- 2) Let *GE* be the <graph expression> immediately contained in *FGC*.
- 3) Set the current working graph to the result of *GE*.

Conformance Rules

None.

16.2 <use graph clause>

Function

Set the current working graph in a <focused linear data-modifying statement>.

Format

```
<use graph clause> ::=  
    USE <graph expression>
```

Syntax Rules

None.

General Rules

- 1) Let UGC be the <use graph clause>.
- 2) Let GE be the <graph expression> immediately contained in UGC .
- 3) Set the current working graph to the result of GE .

Conformance Rules

None.

16.3 <at schema clause>

Function

Set the current working schema in a <statement>.

Format

```
<at schema clause> ::=  
    AT <schema reference>
```

Syntax Rules

- 1) Let S be the schema that is the result of the <schema reference> that is immediately contained in the <at schema clause>.

General Rules

- 1) Set the current working schema to S .

Conformance Rules

None.

16.4 Named elements

Function

Specify named elements.

Format

```
<static variable> ::=  
  <static variable name>  
  
<binding variable> ::=  
  <binding variable name>  
  
<label> ::=  
  <label name>  
  
<parameter> ::=  
  <parameter name>
```

Syntax Rules

- 1) If the <static variable> *SV* with name *SN* is specified, then

Case:

- a) If a static variable with name *SN* is defined in scope of *SV*, then *SV* shall identify the innermost such defined static variable.
- b) Otherwise, *SV* shall identify a catalog object with name *SN* in the current working schema that is a supported result.

- 2) The name of a <parameter> *P* is the <parameter name> immediately contained in *P*.

General Rules

- 1) If a <binding variable> *BV* is specified, then

a) Case:

- i) If the current working record has a field *F* whose field name is *BV*, then the result of evaluating *BV* is the field value of *F*.
- ii) If the current working schema contains a visible catalog object *SO* whose name is *BV*, then the result of evaluating *BV* is *SO*.
- iii) Otherwise, an exception condition is raised: *syntax error or access rule violation — invalid reference (42002)*.

- 2) If a <parameter> *P* is specified, then

a) Let *PN* be the <parameter name> contained in *P*.

b) Case:

- i) If there is a context parameter *CP* with parameter name *PN* in the current request context, then the result of evaluating *P* is the parameter value of *CP*.

- ii) If there is a context parameter CP with parameter name PN in the current session context, then the result of evaluating P is the parameter value of CP .
- iii) Otherwise, the result of evaluating P is the null value.

Conformance Rules

None.

16.5 <type signature>

Function

Specifies a type signature.

Format

```

<of type signature> ::= 
  [ <of type prefix> ] <type signature>

<type signature> ::= 
  <parenthesized formal parameter list> [ <of type prefix> ] <procedure result type>

<parenthesized formal parameter list> ::= 
  <left paren> [ <formal parameter list> ] <right paren>

<formal parameter list> ::= 
  <mandatory formal parameter list>
    [ <comma> <optional formal parameter list> ]
  | <optional formal parameter list>

<mandatory formal parameter list> ::= 
  <formal parameter declaration list>

<optional formal parameter list> ::= 
  OPTIONAL <formal parameter definition list>

<formal parameter declaration list> ::= 
  <formal parameter declaration> [ { <comma> <formal parameter declaration> }... ] 

<formal parameter definition list> ::= 
  <formal parameter definition> [ { <comma> <formal parameter definition> }... ] 

<formal parameter declaration> ::= 
  <parameter cardinality> <compact variable declaration>

<formal parameter definition> ::= 
  <parameter cardinality> <compact variable definition>

<optional parameter cardinality> ::= 
  [ <parameter cardinality> ]

<parameter cardinality> ::= 
  SINGLE | MULTI | MULTIPLE

<procedure result type> ::= 
  <value type>

```

Syntax Rules

None.

General Rules

- 1) If the <of type signature> *OTS* is specified, then the result of *OTS* is the result *ROTS* of the <type signature> immediately contained in *OTS* and the type signature descriptor identified by *OTS* is the type signature descriptor of *ROTS*.

- 2) Let TS be the <type signature>.
- 3) Let $MFPL$ be the <mandatory formal parameter list> simply contained in TS .
- 4) Let $OFPL$ be the <optional formal parameter list> simply contained in TS .
- 5) Let RT be the <procedure result type> simply contained in TS .
- 6) The result of TS is a new type signature with a type signature descriptor comprising:
 - a) The required mandatory parameter declarations $MFPL$.
 - b) The allowed optional parameter definitions $OFPL$.
 - c) The result type RT .

Conformance Rules

None.

16.6 <graph pattern>

Function

Specify a pattern to be matched in a graph.

Format

```

<graph pattern> ::= 
  <path pattern list>
    [ <keep clause> ]
    [ <graph pattern where clause> ]
    [ <yield clause> ]

<path pattern list> ::= 
  <path pattern> [ { <comma> <path pattern> }... ]

<path pattern> ::= 
  [ <path variable> <equals operator> ] [ <path pattern prefix> ] <path pattern expression>

<keep clause> ::= 
  KEEP <path pattern prefix>

<graph pattern where clause> ::= 
  WHERE <search condition>

```

Syntax Rules

- 1) Let *GP* be the <graph pattern>.
- 2) If *BNF1* and *BNF2* are instances of two BNF non-terminals, both contained in *GP* without an intervening <graph pattern>, then *BNF1* and *BNF2* are said to be *at the same depth of graph pattern matching*.
 NOTE 109 — *BNF1* may contain *BNF2* while being at the same depth of graph pattern matching.
- 3) In a <path pattern list>, if two <path pattern>s expose an element variable *EV*, then both shall expose *EV* as an unconditional singleton variable.
 NOTE 110 — This case expresses an implicit join on *EV*. Implicit joins between conditional singleton variables or group variables are forbidden.
- 4) A node variable shall not be equivalent to an edge variable declared at the same depth of graph pattern matching.
- 5) If <keep clause> *KP* is specified, then:
 - a) Let *PSP* be the <path pattern prefix> simply contained in *KP*.
 - b) For each <path pattern> *PP* simply contained in *GP*:
 - i) *PP* shall not contain a <path search prefix>.
 - ii) Case:
 - 1) If *PP* specifies a <path variable> *PV*, let *PVDECL* be
 $PV =$
 - 2) Otherwise, let *PVDECL* be the zero-length character string.
 - iii) Case:

16.6 <graph pattern>

1) If PP specifies a <path mode prefix>, let PMP be that <path mode prefix>.

2) Otherwise, let PMP be the zero-length character string.

iv) Let PPE be the <path pattern expression> simply contained in PP .

v) PP is replaced by

$PVDECL PSP [PMP PPE]$

c) The <keep clause> is removed from the <graph pattern>.

6) After the preceding transformations, for every <path pattern> PP , if PP contains a <path pattern prefix> PPP that specifies a <path mode> PM , then

a) Case:

i) If PP specifies a <path variable> PV , let $PVDECL$ be

$PV =$

ii) Otherwise, let $PVDECL$ be the zero-length character string.

b) Let PPE be the <path pattern expression> simply contained in PP .

c) PP is replaced by

$PVDECL PPP [PM PPE]$

NOTE 111 — One effect of the preceding transforms is that every <path mode> expressed outside a <parenthesized path pattern expression> is also expressed within a <parenthesized path pattern expression>. For example

ALL SHORTEST TRAIL GROUP <path pattern expression>
is rewritten as

ALL SHORTEST TRAIL GROUP [TRAIL <path pattern expression>]

The TRAIL specified outside the parentheses is now redundant. The benefit is that the definition of a consistent path binding in Subclause 22.2, “Machinery for graph pattern matching” only needs to consider <path mode>s declared in <parenthesized path pattern expression>s.

7) After the preceding transformations, every <quantified path primary> QPP contained in the <graph pattern> shall satisfy at least one of the following conditions:

a) The <graph pattern quantifier> of QPP is bounded.

b) QPP is contained in a restrictive <parenthesized path pattern expression>.

c) QPP is contained in a selective <path pattern>.

8) If a <yield clause> is not specified, then:

a) Let $EVIGS$ be the set of the element variables in the scope of GP .

b) A <path variable> that is simply contained in GP is a *path variable in the global scope* of GP . Let $PVIGS$ be the set of the path variables in the global scope of GP .

c) Let $GPVIGS$ be a permutation of $EVIGS \cup PVIGS$ in the order of their first occurrence as part of being immediately specified by an <element variable declaration> simply contained in GP or by a <path pattern> simply contained in GP respectively.

d) Let n be the number of graph pattern variables in $GPVIGS$ and let $GPVIGS_i$ be the i -th element of $GPVIGS$, for $1 \leq i \leq n$.

- e) Let the <yield item list> YIL be a comma-separated list of <yield item>s $GPVIGS_i$ AS $GPVIGS_i$, for $1 \leq i \leq n$.
- f) YIELD YIL is the implicit <yield clause> of GP .

General Rules

- 1) Let PG be the current working graph.
- 2) Let GP be the <graph pattern> after the transformations in the Syntax Rules of this and other Sub-clauses. Let PPL be the <path pattern list> simply contained in GP .
- 3) The General Rules of Subclause 22.2, “Machinery for graph pattern matching”, are applied with PG as **PROPERTY GRAPH** and PPL as **PATH PATTERN LIST**; let $MACH$ be the **MACHINERY** returned from the application of those General Rules
- 4) The following components of $MACH$ are identified:
 - a) ABC , the alphabet, formed as the disjoint union of the following:
 - i) SNV , the set of node variables.
 - ii) SEV , the set of edge variables.
 - iii) SPS , the set of subpath symbols.
 - iv) SAS , the set of anonymous symbols.
 - v) SBS , the set of bracket symbols.
 - b) $REDUCE$, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 5) Let NP be the number of <path pattern>s simply contained in PPL . Let PP_1, \dots, PP_{NP} be the <path pattern>s simply contained in PPL after the transformations in the Syntax Rules.
- 6) For every i , $1 \leq i \leq NP$:
 - a) Let PPE be the <parenthesized path pattern expression> simply contained in PP_i .
 - b) The General Rules of Subclause 22.3, “Evaluation of a <path pattern expression>”, are applied with PG as **PROPERTY GRAPH**, PPL as **PATH PATTERN LIST**, $MACH$ as **MACHINERY**, PPE as **SPECIFIC BNF INSTANCE**, and the empty set as **PARTIAL MATCH**; let $SMTEMP_i$ be the **SET OF MATCHES** returned from the application of those General Rules.
 - c) Case:
 - i) If PP_i is a selective <path pattern>, then the General Rules of Subclause 22.4, “Evaluation of a selective <path pattern>”, are applied with PG as **PROPERTY GRAPH**, PPL as **PATH PATTERN LIST**, $MACH$ as **MACHINERY**, the empty set as **PARTIAL MATCH**, PP_i as **SELECTOR**, and $SMTEMP_i$ as **INPUT SET OF LOCAL MATCHES**; let SM_i be the **OUTPUT SET OF LOCAL MATCHES** returned from the application of those General Rules.
 - ii) Otherwise, let SM_i be $SMTEMP_i$.
- 7) Let $CROSS$ be the cross product $SM_1 \times \dots \times SM_{NP}$.
- 8) Let $INNER$ be the set of multi-path bindings MPB in $CROSS$ such that, for every unconditional singleton variable USV exposed by PPL , USV is bound to a unique graph element by MPB .

16.6 <graph pattern>

NOTE 112 — Anonymous symbols are not element variables; there is no requirement that two anonymous symbols bind to the same graph element.

- 9) A *match* of *GP* is a multi-path binding $M = \{ PB_1, \dots PB_{NP} \}$ of *NP* path bindings in *INNER*, such that the following are true:

- a) For every j , $1 \leq j \leq NP$, and for or every <parenthesized path pattern expression> *PPPE* contained in PP_j , let i be the bracket index of *PPPE*, and let ' $[_i$ ' and ' $]_i$ ' be the bracket symbols associated with *PPPE*. A *binding* of *PPPE* is a substring of PB_j that begins with the bracket binding ($[_i$, ' $[_i$ ') and ends with the next bracket binding ($]_i$, ' $]_i$ ').

For every binding *BPPPE* of *PPPE* contained in PB_j , the following are true:

- i) For every element variable *EV* that is exposed as an unconditional singleton by *PPPE*, *EV* is bound to a unique graph element by the element variable bindings contained in *BPPPE*.

NOTE 113 — Anonymous symbols are not element variables; there is no requirement that two anonymous symbols bind to the same graph element.

- ii) If *PPPE* contains a <parenthesized path pattern where clause> *PPPWC*, then the value of *V1* is *True* when the General Rules of Subclause 22.5, “*Applying bindings to evaluate an expression*”, are applied with *GP* as *GRAPH PATTERN*, the <search condition> simply contained in *PPPWC* as *EXPRESSION*, *MACH* as *MACHINERY*, *M* as *MULTI-PATH BINDING*, and a reference to *BPPPE* as *REFERENCE TO LOCAL CONTEXT*; let *V1* be the *VALUE* returned from the application of those General Rules.

- b) If *GP* contains a <graph pattern where clause> *GPWC*, then the value of *V2* is *True* when the General Rules of Subclause 22.5, “*Applying bindings to evaluate an expression*”, are applied with *GP* as *GRAPH PATTERN*, *GPWC* as *EXPRESSION*, *MACH* as *MACHINERY*, *M* as *MULTI-PATH BINDING*, and a reference to *M* as *REFERENCE TO LOCAL CONTEXT*; let *V2* be the *VALUE* returned from the application of those General Rules.

- 10) A *reduced match* $RM = \{ RPB_1, \dots RPB_{NP} \}$ is obtained from a match $M = \{ PB_1, \dots PB_{NP} \}$ by replacing every path binding PB_i from M with the reduced path binding RPB_i obtained from PB_i for $1 \leq i \leq NP$.

NOTE 114 — Set-theoretic deduplication may occur here. That is, two or more matches may reduce to the same reduced match; this scenario is regarded as contributing only a single reduced match to the result set.

- 11) The match records $MATCHES_{RM}$ are obtained for a reduced match $RM = \{ RPB_1, \dots RPB_{NP} \}$ of *GP* by applying the General Rules of Subclause 16.16, “<*yield clause*>” for *YC* when the reduced path bindings of *RM* are applied in the scope of *GP* in a new child execution context, as specified by General Rule 15) of Subclause 22.2, “*Machinery for graph pattern matching*”; $MATCHES_{RM}$ is the *YIELD* returned from the application of these General Rules.
- 12) Let *MATCHES* be a new binding table comprising the match records obtained for each reduced match of *GP* in an implementation-dependent order.
- 13) The result of the application of the General Rules of this Subclause is *MATCHES*.

Conformance Rules

- 1) Without Feature G010, “*Graph Pattern Keep*”, conforming GQL language shall not contain a <keep clause>.
- 2) Without Feature G011, “*Graph Pattern Where*”, conforming GQL language shall not contain a <graph pattern where clause>.

16.7 <path pattern expression>

Function

Specify a pattern to match a single path in a property graph.

Format

```

<path pattern expression> ::=

  <path term>
  | <path multiset alternation>
  | <path pattern union>

<path multiset alternation> ::=

  <path term> <multiset alternation operator> <path term>
  [ { <multiset alternation operator> <path term> }... ]

<path pattern union> ::=

  <path term> <vertical bar> <path term> [ { <vertical bar> <path term> }... ]

<path term> ::=

  <path factor>
  | <path concatenation>

<path concatenation> ::=

  <path term> <path factor>
```

```

<path factor> ::=

  <path primary>
  | <quantified path primary>
  | <questioned path primary>

<quantified path primary> ::=

  <path primary> <graph pattern quantifier>

<questioned path primary> ::=

  <path primary> <question mark>
```

NOTE 115 — Unlike most regular expression languages, <question mark> is not equivalent to the quantifier {0,1}: the quantifier {0,1} exposes variables as group, whereas <question mark> does not change the singleton variables that it exposes to group. However, <question mark> does expose any singleton variables as conditional singletons.

```

<path primary> ::=

  <element pattern>
  | <parenthesized path pattern expression>
  | <simplified path pattern expression>

<element pattern> ::=

  <node pattern>
  | <edge pattern>

<node pattern> ::=

  <left paren> <element pattern filler> <right paren>

<element pattern filler> ::=

  [ <element variable declaration> ]
  [ <is label expression> ]
  [ <element pattern predicate> ]
  [ <element pattern cost clause> ]

<element variable declaration> ::=

  <element variable>
```

16.7 <path pattern expression>

```

<is label expression> ::= 
  <is or colon> <label expression>

<is or colon> ::= 
  IS
  | <colon>

<element pattern predicate> ::= 
  <element pattern where clause>
  | <element property specification>

<element pattern where clause> ::= 
  WHERE <search condition>

<element property specification> ::= 
  <left brace> <property key value pair list> <right brace>

<property key value pair list> ::= 
  <property key value pair> [ { <comma> <property key value pair> }... ]

<property key value pair> ::= 
  <property name> <colon> <value expression>

<element pattern cost clause> ::= 
  <cost clause>

<cost clause> ::= 
  COST <value expression> [ DEFAULT <value expression> ]

<edge pattern> ::= 
  <full edge pattern>
  | <abbreviated edge pattern>

<full edge pattern> ::= 
  <full edge pointing left>
  | <full edge undirected>
  | <full edge pointing right>
  | <full edge left or undirected>
  | <full edge undirected or right>
  | <full edge left or right>
  | <full edge any direction>

<full edge pointing left> ::= 
  <left arrow bracket> <element pattern filler> <right bracket minus>

<full edge undirected> ::= 
  <tilde left bracket> <element pattern filler> <right bracket tilde>

<full edge pointing right> ::= 
  <minus left bracket> <element pattern filler> <bracket right arrow>

<full edge left or undirected> ::= 
  <left arrow tilde bracket> <element pattern filler> <right bracket tilde>

<full edge undirected or right> ::= 
  <tilde left bracket> <element pattern filler> <bracket tilde right arrow>

<full edge left or right> ::= 
  <left arrow bracket> <element pattern filler> <bracket right arrow>

<full edge any direction> ::= 
  <minus left bracket> <element pattern filler> <right bracket minus>

<abbreviated edge pattern> ::= 
  <left arrow>

```

```

| <tilde>
| <right arrow>
| <left arrow tilde>
| <tilde right arrow>
| <left minus right>
| <minus sign>

<graph pattern quantifier> ::==
  <asterisk>
  | <plus sign>
  | <fixed quantifier>
  | <general quantifier>

<fixed quantifier> ::=
  <left brace> <unsigned integer> <right brace>

<general quantifier> ::=
  <left brace> [ <lower bound> ] <comma> [ <upper bound> ] <right brace>

<lower bound> ::=
  <unsigned integer>

<upper bound> ::=
  <unsigned integer>

<parenthesized path pattern expression> ::=
  <left paren>
    [ <subpath variable declaration> ]
    [ <path mode prefix> ]
    <path pattern expression>
    [ <parenthesized path pattern where clause> ]
    [ <parenthesized path pattern cost clause> ]
  <right paren>
  | <left bracket>
    [ <subpath variable declaration> ]
    [ <path mode prefix> ]
    <path pattern expression>
    [ <parenthesized path pattern where clause> ]
    [ <parenthesized path pattern cost clause> ]
  <right bracket>

<subpath variable declaration> ::=
  <subpath variable> <equals operator>

<parenthesized path pattern where clause> ::=
  WHERE <search condition>

<parenthesized path pattern cost clause> ::=
  <cost clause>

```

Syntax Rules

- 1) Every <graph pattern quantifier> is normalized, as follows:

- a) <asterisk> is equivalent to

{0,}

- b) <plus sign> is equivalent to

{1,}

16.7 <path pattern expression>

- c) If <fixed quantifier> FQ is specified, then let UI be the <unsigned integer> contained in FQ . FQ is equivalent to

 $\{UI, UI\}$

- d) If <general quantifier> GQ is specified, and if <lower bound> is not specified, then the <unsigned integer> 0 (zero) is supplied as the <lower bound>.

- 2) If <general quantifier> GQ is specified or implied by the preceding normalizations, then:

Case:

- a) If <upper bound> is specified, then

- i) The value of <upper bound> VUP shall be greater than 0 (zero).
- ii) The value of <lower bound> LUP shall be less than or equal to VUP .
- iii) If LUP equals VUP , then GQ is a *fixed quantifier*.
- iv) GQ is a *bounded quantifier*.

- b) Otherwise, GQ is an *unbounded quantifier*.

- 3) A <graph pattern quantifier> that is not a fixed quantifier is a *variable quantifier*.

- 4) A <path pattern expression> that contains at the same depth of graph pattern matching a variable quantifier, a <questioned path primary>, a <path multiset alternation>, or a <path pattern union> is a *possibly variable length path pattern*.

- 5) A <path pattern expression> that is not a possibly variable length path pattern is a *fixed length path pattern*.

- 6) Two successive <element pattern>s contained in a <path concatenation> at the same depth of graph pattern matching shall not be <node pattern>s.

- 7) The *minimum path length* of certain BNF non-terminals defined in this Subclause is defined recursively as follows:

- a) The minimum path length of a <node pattern> is 0 (zero).
- b) The minimum path length of an <edge pattern> is 1 (one).
- c) The minimum path length of a <path concatenation> is the sum of the minimum path lengths of its operands.
- d) The minimum path length of a <path pattern union> or <path multiset alternation> is the minimum of the minimum path length of its operands.
- e) The minimum path length of a <quantified path primary> is the product of the minimum path length of the simply contained <path primary> and the value of the <lower bound>.
- f) The minimum path length of a <questioned path primary> is 0 (zero).
- g) The minimum path length of a <parenthesized path pattern expression> is the minimum path length of the simply contained <path pattern expression>.
- h) If $BNT1$ and $BNT2$ are two BNF non-terminals such that $BNT1 ::= BNT2$ and the minimum path length of $BNT2$ is defined, then the minimum path length of $BNT1$ is also defined and is the same as the minimum path length of $BNT2$.

- 8) The <path primary> immediately contained in a <quantified path primary> or <questioned path primary> shall have minimum path length that is greater than 0 (zero).

9) The <path primary> simply contained in a <quantified path primary> shall not contain a <quantified path primary> at the same depth of graph pattern matching.

10) Let PMA be a <path multiset alternation>.

- a) A <path term> simply contained in PMA is a *multiset alternation operand* of PMA .
- b) Let $NOPMA$ be the number of multiset alternation operands of PMA . Let $OPMA_1, \dots OPMA_{NOPMA}$ be an enumeration of the operands of PMA . Let $SOPMA_1, \dots SOPMA_{NOPMA}$ be implementation-dependent <identifier>s that are mutually distinct and distinct from every element variable, subpath variable and path variable contained in GP .
- c) For every k , $1 \leq k \leq NOPMA$,

Case:

- i) If $OPMA_1$ is a <parenthesized path pattern expression> that simply contains a <subpath variable declaration>, then let $OPMAX_i$ be $OPMA_i$.
- ii) Otherwise, let $OPMAX_i$ be the <parenthesized path pattern expression>

$(SOPMA_i = OPMA_i)$

- d) PMA is equivalent to

$OPMAX_1 | \dots | OPMAX_{NOPMA}$

11) A <path term> simply contained in a <path pattern union> PSD is a *path pattern union operand* of PSD .

An element variable GPV that is declared in a path pattern union operand $PPUO1$ of PSD shall not be referenced in a <parenthesized path pattern where clause> contained in a different path pattern union operand $PPUO2$ of PSD unless GPV is also declared in $PPUO2$.

12) An <element pattern> EP that contains an <element pattern where clause> $EPWC$ is transformed as follows:

- a) Let EPF be the <element pattern filler> simply contained in EP .
- b) Let $PREFIX$ be the <delimiter token> contained in EP before EPF and let $SUFFIX$ be the <delimiter token> contained in EP after EPF .
- c) Let EV be the <element variable> simply contained in EPF . Let ILE be the <is label expression> contained in EPF , if any, otherwise let ILE be the zero-length string. Let $EPCC$ be the <element pattern cost clause> contained in EPF , if any; otherwise, let $EPCC$ be the zero-length string.
- d) EP is replaced by

$[PREFIX EV ILE EPCC SUFFIX EPWC]$

13) An <element pattern> that does not contain an <element variable declaration>, an <is label expression>, or an <element pattern predicate> is said to be *empty*.

14) Each <path pattern expression> is transformed in the following steps:

- a) If the <path primary> immediately contained in a <quantified path primary> or <questioned path primary> is an <edge pattern> EP , then EP is replaced by

(EP)

NOTE 116 — For example:

16.7 <path pattern expression>

 $\rightarrow *$

becomes:

 $(\rightarrow) \{0,\}$

which in later transformations becomes:

 $((\rightarrow)) \{0,\}$

- b) If two successive <element pattern>s contained in a <path concatenation> at the same depth of graph pattern matching are <edge pattern>s, then an implicit empty <node pattern> is inserted between them.
- c) If an edge pattern EP contained in a <path term> PST at the same depth of graph pattern matching is not preceded by a <node pattern> contained in PST at the same depth of graph pattern matching, then an implicit empty <node pattern> VP is inserted in PST immediately prior to EP .
- d) If an edge pattern EP contained in a <path term> PST at the same depth of graph pattern matching is not followed by a <node pattern> contained in PST at the same depth of graph pattern matching, than an implicit empty <node pattern> VP is inserted in PST immediately after EP .

NOTE 117 — As a result of the preceding transformations, a fixed length path pattern has an odd number of <element pattern>s, beginning and ending with <node pattern>s, and alternating between <node pattern>s and <edge pattern>s.

- e) Every <abbreviated edge pattern> AEP is replaced with an empty <full edge pattern> as follows:

Case:

- i) If AEP is <left arrow>, then AEP is replaced by

 $\leftarrow [\] \rightarrow$

- ii) If AEP is <tilde>, then AEP is replaced by

 $\sim [\] \sim$

- iii) If AEP is <right arrow>, then AEP is replaced by

 $\leftarrow [\] \rightarrow$

- iv) If AEP is <left arrow tilde>, then AEP is replaced by

 $\leftarrow \sim [\] \sim$

- v) If AEP is <tilde right arrow>, then AEP is replaced by

 $\sim [\] \sim \rightarrow$

- vi) If AEP is <left minus right>, then AEP is replaced by

 $\leftarrow [\] \rightarrow$

- vii) If AEP is <minus sign>, then AEP is replaced by

 $\leftarrow [\] \rightarrow$

- 15) The *minimum node count* of certain BNF non-terminals defined in this Subclause is defined recursively as follows:

- a) The minimum node count of a <node pattern> is 1 (one).

- b) The minimum node count of an <edge pattern> is 0 (zero).
- c) The minimum node count of a <path concatenation> is the sum of the minimum node counts of its operands.
- d) The minimum node count of a <path pattern union> or <path multiset alternation> is the minimum of the minimum node count of its operands.
- e) The minimum node count of a <quantified path primary> is the product of the minimum node count of the simply contained <path primary> and the value of the <lower bound> of the simply contained <graph pattern quantifier>.
- f) The minimum node count of a <questioned path primary> is 0 (zero).
- g) The minimum node count of a <parenthesized path pattern expression> is the minimum node count of the simply contained <path pattern expression>.
- h) If $BNT1$ and $BNT2$ are two BNF non-terminals such that $BNF1 ::= BNF2$ and the minimum node count of $BNF2$ is defined, then the minimum node count of $BNF1$ is also defined and is the same as the minimum node count of $BNF2$.

16) The <path pattern expression> simply contained in a <path pattern> shall have a minimum node count that is greater than 0 (zero).

NOTE 118 — The minimum node count is to be computed after the syntactic transform that adds implicit node patterns. Thus a single <edge pattern> is a permitted <path pattern> because it implies two <node pattern>s.

- 17) An <element variable> contained in an <element variable declaration> $GPVD$ is said to be *declared* by $GPVD$, and by the <element pattern> that simply contains $GPVD$.
- 18) An <element variable> that is declared by a <node pattern> is a *node variable*. An <element variable> that is declared by an <edge pattern> is an *edge variable*.
- 19) The scope of an <element variable> that is declared by an <element pattern> EP includes the following:

- a) The <element pattern where clause> of EP , if any.
- b) The <graph pattern where clause> of GP , if any.
- c) The <parenthesized path pattern where clause> of a <parenthesized path pattern expression> that contains EP at the same depth of graph pattern matching.
- d) The explicit or implicit <yield clause> of GP .

20) Let EP be an <element pattern>.

- a) If EP simply contains an <element variable declaration> EVD , then EP *declares* the element variable EV simply contained in EVD . In addition, a BNF non-terminal that contains EP at the same depth of graph pattern matching *declares* EV .
- b) If EP contains an <element pattern where clause> $EPWC$, then
 - i) EP shall simply contain an <element variable declaration> $GPVD$. Let GPV be the element variable simply contained in $GPVD$.
 - ii) $EPWC$ is an <element pattern where clause> that is *associated* with the element variable GPV declared by EP .
 - iii) If every <property reference> contained in $EPWC$ at the same depth of graph pattern matching does not reference GPV , then an exception condition is raised: *syntax error or access rule violation — invalid reference (42002)*.

16.7 <path pattern expression>

- c) If EP declares element variable GPV and simply contains an <is label expression> $ISLE$, then the <label expression> simply contained in $ISLE$ is *associated* with GPV .
- 21) If EV is an element variable, and BNT is an instance of a BNF non-terminal, then the terminology “ BNT exposes EV ” is defined as follows. The full terminology is one of the following: “ BNT exposes EV as an unconditional singleton variable”, “ BNT exposes EV as a conditional singleton variable”, “ BNT exposes EV as an effectively bounded group variable” or “ BNT exposes EV as an effectively unbounded group variable”. The terms “unconditional singleton variable”, “conditional singleton variable”, “effectively bounded group variable”, and “effectively unbounded group variable” are called the *degree of exposure*.
- a) An <element pattern> EP that declares an element variable EV exposes EV as an unconditional singleton.
 - b) If a <path concatenation> PPC declares EV then let PT be the <path term> and let PF be the <path factor> simply contained in PPC .
- Case:
- i) If EV is exposed as an unconditional singleton by both PT and PF , then EV is exposed as an unconditional singleton by PPC .
- NOTE 119 — This case expresses an implicit join on EV within PPC . Implicit joins between conditional singleton variables or group variables are forbidden.
- ii) Otherwise, EV shall only be exposed by one of PT or PF . In this case EV is exposed by PPC in the same degree that it is exposed by PT or PF .
- c) If a <path pattern union> or <path multiset alternation> PA declares EV , then
- Case:
- i) If every operand of PA exposes EV as an unconditional singleton variable, then PA exposes EV as an unconditional singleton variable.
 - ii) If at least one operand of PA exposes EV as an effectively unbounded group variable, then PA exposes EV as an effectively unbounded group variable.
 - iii) If at least one operand of PA exposes EV as an effectively bounded group variable, then PA exposes EV as an effectively bounded group variable.
 - iv) Otherwise, PA exposes EV as a conditional singleton variable.
- d) If a <quantified path primary> QPP declares EV , then
- Case:
- i) If QPP contains a <graph pattern quantifier> that is a <fixed quantifier> or a <general quantifier> that contains an <upper bound>, then QPP exposes EV as an effectively bounded group variable.
 - ii) If QPP is contained at the same depth of graph pattern matching in a restrictive <parenthesized path pattern expression>, then QPP exposes EV as an effectively bounded group variable.
- NOTE 120 — The preceding definition is to be applied after the syntactic transformation to insure that every <path mode prefix> is at the head of a <parenthesized path pattern expression>.
- iii) Otherwise, QPP exposes EV as an effectively unbounded group variable.
- e) If a <questioned path primary> $QUPP$ declares EV , then let PP be the <path primary> simply contained in $QUPP$.

Case:

- i) If PP exposes EV as a group variable, then $QUPP$ exposes EV as a group variable with the same degree of exposure.
- ii) Otherwise, $QUPP$ exposes EV as a conditional singleton variable.
- f) A <parenthesized path pattern expression> exposes the same variables as the simply contained <path pattern expression>, in the same degree.

NOTE 121 — A restrictive <path mode> declared by a <parenthesized path pattern expression> makes variables effectively bounded, but it does so even for proper subexpressions within the scope of the <path mode> and has already been handled by the rules for <quantified path primary>.

- g) If a <path pattern> PP declares EV , then let PPE be the simply contained <path pattern expression>.

Case:

- i) If PPE exposes EV as an unconditional singleton, a conditional singleton, or an effectively bounded group variable, then PP exposes EV with the same degree of exposure.
- ii) Otherwise, PP exposes EV as an effectively bounded group variable.

NOTE 122 — That is, even if PPE exposes EV as an effectively unbounded group variable, PP still exposes EV as effectively bounded, because in this case PP is required to be a selective <path pattern>.

- h) If $BNT1$ and $BNT2$ are two BNF non-terminals such that $BNT1 ::= BNT2$ and $BNT2$ exposes EV , then $BNT1$ exposes EV to the same degree of exposure as $BNT2$.

- 22) A <parenthesized path pattern where clause> $PPPWC$ simply contained in a <parenthesized path pattern expression> $PPPE$ shall not reference a path variable, or subpath variable that is not declared in $PPPE$.

General Rules

None.

NOTE 123 — The evaluation of a <path pattern expression> is performed by the General Rules of Subclause 22.3, “Evaluation of a <path pattern expression>”.

Conformance Rules

- 1) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain <full edge undirected>, <full edge left or undirected>, <full edge undirected or right>, or <full edge left or right>.
- 2) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain an <abbreviated edge pattern> that is <tilde>, <left arrow tilde>, <tilde right arrow> or <left minus right>.
- 3) Without Feature G020, “Path Multiset Alternation”, conforming GQL language shall not contain a <path multiset alternation>.
- 4) Without Feature G021, “Path Pattern Union”, conforming GQL language shall not contain a <path pattern union>.
- 5) Without Feature G030, “Quantified Paths”, conforming GQL language shall not contain a <quantified path primary>.
- 6) Without Feature G031, “Quantified Edges”, conforming GQL language shall not contain a <quantified path primary> that immediately contains a <path primary> that is an <edge pattern>.

16.7 <path pattern expression>

- 7) Without Feature G040, “Questioned Paths”, conforming GQL language shall not contain a <questioned path primary>.
- 8) Without Feature G050, “Simplified Path Pattern Expression”, conforming GQL language shall not contain a <simplified path pattern expression>.
- 9) Without Feature G060, “Non-local element pattern predicates”, in conforming GQL language, the <element pattern where clause> of an <element pattern> *EP* shall only reference the <element variable> declared in *EP*.
- 10) Without Feature G070, “Abbreviated Edge Patterns”, conforming GQL language shall not contain an <abbreviated edge pattern>.
- 11) Without Feature G081, “Parenthesized Path Pattern Where”, conforming GQL language shall not contain a <parenthesized path pattern where clause>.
- 12) Without Feature G082, “Non-local predicates”, in conforming GQL language, a <parenthesized path pattern where clause> simply contained in a <parenthesized path pattern expression> *PPPE* shall not reference an <element variable> that is not declared in *PPPE*.
- 13) Without Feature G083, “Parenthesized Path Pattern Cost”, conforming GQL language shall not contain a <parenthesized path pattern cost clause>.

16.8 <path pattern prefix>

Function

Specify a path-finding operation and a path mode.

Format

```

<path pattern prefix> ::==
  <path mode prefix>
  | <path search prefix>

<path mode prefix> ::==
  <path mode> [ <path or paths> ]

<path mode> ::==
  WALK
  | TRAIL
  | SIMPLE
  | ACYCLIC

<path search prefix> ::==
  <all path search>
  | <any path search>
  | <shortest path search>

<all path search> ::==
  ALL [ <path mode> ] [ <path or paths> ]

<path or paths> ::==
  PATH | PATHS

<any path search> ::==
  ANY [ <number of paths> ] [ <path mode> ] [ <path or paths> ]

<number of paths> ::==
  <unsigned integer specification>

<shortest path search> ::==
  <all shortest path search>
  | <any shortest path search>
  | <counted shortest path search>
  | <counted shortest group search>

<all shortest path search> ::==
  ALL SHORTEST [ <path mode> ] [ <path or paths> ]

<any shortest path search> ::==
  ANY SHORTEST [ <path mode> ] [ <path or paths> ]

<counted shortest path search> ::==
  SHORTEST <number of paths> [ <path mode> ] [ <path or paths> ]

<counted shortest group search> ::==
  SHORTEST <number of groups> [ <path mode> ] [ <path or paths> ] { GROUP | GROUPS }

<number of groups> ::==
  <unsigned integer specification>

```

Syntax Rules

- 1) If a <parenthesized path pattern expression> does not specify a <path mode prefix>, then WALK PATHS is implicit.
- 2) If a <path pattern prefix> *PPP* does not specify <all path search>, then:
 - a) Case:
 - i) If *PPP* does not simply contain a <path mode>, then let *PM* be WALK.
 - ii) Otherwise, let *PM* be the <path mode> simply contained in *PPP*.
 - b) Case:
 - i) If *PPP* does not simply contain a <number of paths> or <number of groups>, then let *N* be an <unsigned integer> whose value is 1 (one).
 - ii) Otherwise let *N* be the <number of paths> or <number of groups> simply contained in *PPP*. The declared type of *N* shall be exact numeric with scale 0 (zero). If *N* is a <literal>, then the value of *N* shall be positive.
 - c) Case:
 - i) If *PPP* is an <any path search>, then *PPP* is equivalent to
ANY *N PM PATHS*
 - ii) If *PPP* is a <shortest path search>, then

Case:

 - 1) If *PPP* is <all shortest path search>, then *PPP* is equivalent to
SHORTEST 1 *PM GROUP*
 - 2) If *PPP* is <any shortest path search>, then *PPP* is equivalent to
SHORTEST 1 *PM PATH*
 - 3) If *PPP* is <counted shortest path search>, then *PPP* is equivalent to
SHORTEST *N PM PATHS*
 - 4) If *PPP* is <counted shortest group search>, then *PPP* is equivalent to
SHORTEST *N PM GROUPS*
- 3) A <path pattern prefix> that specifies a <path mode> other than WALK is *restrictive*. A <parenthesized path pattern expression> that immediately contains a restrictive <path mode prefix> is *restrictive*.
- 4) A <path search prefix> other than <all path search> is *selective*. A <path pattern> that simply contains a selective <path search prefix> is *selective*.
- 5) Let *PPPE* be a selective <path pattern>.
 Then
 - a) An element variable exposed by *PPPE* is an *interior variable* of *PPPE*.
 - b) A node variable *LVV* is the *left boundary variable* of *PPPE* if the following conditions are true:

- i) *PPPE* exposes *LVV* as an unconditional singleton variable.
 - ii) *LVV* is declared in the first implicit or explicit *<node pattern>* *LVP* contained in *PPPE*.
 - iii) *LVP* is not contained in a *<path pattern union>* or *<path multiset alternation>* that is contained in *PPPE*.
- c) A node variable *RVV* is the *right boundary variable* of *PPPE* if the following conditions are true:
- i) *PPPE* exposes *RVV* as an unconditional singleton variable.
 - ii) *RVV* is declared in the last implicit or explicit *<node pattern>* *RVP* contained in *PPPE*.
 - iii) *RVP* is not contained in a *<path pattern union>* or *<path multiset alternation>* that is contained in *PPPE*.
- d) An element variable that is exposed by *PPPE* that is neither a left boundary variable of *PPPE* nor a right boundary variable of *PPPE* is a *strict interior variable* of *PPPE*.
- 6) An element variable that is not declared in a selective *<path pattern>* is an *exterior variable*.
- 7) A strict interior variable of one selective *<path pattern>* shall not be equivalent to an exterior variable, nor to an interior variable of another selective *<path pattern>*.

NOTE 124 — Implicit joins of boundary variables of selectors with exterior variables or boundary variables of other selectors are permitted.

General Rules

None.

NOTE 125 — Restrictive *<path mode>*s are enforced as part of the check for consistent path bindings in the generation of the set of local matches in Subclause 16.7, “*<path pattern expression>*”. Selective *<parenthesized path pattern expression>*s are evaluated by Subclause 22.4, “Evaluation of a selective *<path pattern>*”.

Conformance Rules

- 1) Without Feature G091, “Shortest Path Search”, conforming GQL language shall not contain a *<shortest path search>*.
- 2) Without Feature G092, “Advanced Path Modes: TRAIL”, conforming GQL language shall not contain *<path mode>* that specifies TRAIL.
- 3) Without Feature G093, “Advanced Path Modes: SIMPLE”, conforming GQL language shall not contain *<path mode>* that specifies SIMPLE.
- 4) Without Feature G094, “Advanced Path Modes: ACYCLIC”, conforming GQL language shall not contain *<path mode>* that specifies ACYCLIC.

16.9 <simple graph pattern>

Function

Define a <simple graph pattern>.

Format

```
<simple graph pattern> ::=  
  <simple path pattern list>  
  
<simple path pattern list> ::=  
  <simple path pattern> [ { <comma> <simple path pattern> }... ]  
  
<simple path pattern> ::=  
  !! Predicative production rule.  
  <path pattern expression>
```

Syntax Rules

- 1) A <simple path pattern> shall be a <path pattern> that after the application of the Syntax Rules for <path pattern> is either a <node pattern> or a concatenation of <node pattern> <edge pattern> <node pattern>.
- 2) Let *ECSPP* be a <simple path pattern> that simply contains an <edge pattern>.
 - a) Every <node pattern> simply contained in *ECSPP* shall simply contain an <element variable declaration>.
 - b) Every <node pattern> simply contained in *ECSPP* shall not simply contain an <is label expression>, an <element pattern predicate> or an <element pattern cost clause>.
 - c) If any <identifier> simply contained in an <element variable declaration> that is simply contained in a <node pattern> simply contained in *ECSPP* references a variable that is not in scope, then an exception condition is raised: *syntax error or access rule violation — invalid reference (42002)*.
- 3) No <element pattern> simply contained in a <simple path pattern> shall simply contain an <element pattern where clause>.
- 4) Every <full edge pattern> simply contained in a <simple path pattern> after the application of the Syntax Rules for <abbreviated edge pattern>, shall simply contain one of <full edge pointing left>, <full edge undirected>, or <full edge pointing right>.

NOTE 126 — The Syntax Rules for <abbreviated edge pattern> rewrite an <abbreviated edge pattern> to a <full edge pattern>. See [Syntax Rule 14\)e](#)).

- 5) No <is label expression> simply contained in a <simple path pattern> shall contain a <label disjunction> or a <label negation>.
- 6) No <value expression> simply contained in a <simple path pattern> *SPP* shall contain a <binding variable> that references a graph element that is introduced by an <element variable> simply contained in *SPP*.

General Rules

None.

NOTE 127 — <simple path pattern> provides restrictions for path pattern syntax used in other subclauses. The subclauses that use <simple path pattern> perform the evaluation of the resulting path pattern and so no general rules are needed in this subclause.

Conformance Rules

None.

16.10 <label expression>

Function

Specify an expression that matches one or more labels of a graph.

Format

```

<label expression> ::=

  <label term>
  | <label disjunction>

<label disjunction> ::=

  <label expression> <vertical bar> <label term>

<label term> ::=

  <label factor>
  | <label conjunction>

<label conjunction> ::=

  <label term> <ampersand> <label factor>

<label factor> ::=

  <label primary>
  | <label negation>

<label negation> ::=

  <exclamation mark> <label primary>

<label primary> ::=

  <label>
  | <wildcard label>
  | <parenthesized label expression>

<wildcard label> ::=

  <percent>

<parenthesized label expression> ::=

  <left paren> <label expression> <right paren>
  | <left bracket> <label expression> <right bracket>

```

Syntax Rules

- 1) Let *LE* be the <label expression>.
- 2) Let *GP* be the <graph pattern> that simply contains *LE*. Let *GC* be the innermost <use graph clause> or the <from graph clause> in scope for *GP*. Let *GR* be the <graph reference> that is simply contained in *GC*. If such a clause exists, let *PG* be the graph that is identified by *GR*. Otherwise, let *PG* be the current working graph.
- 3) Every <label> contained in *LE* shall identify a label of *PG*.
- 4) Case:
 - a) If *LE* is simply contained in a <node pattern>, then *LE* is a *node <label expression>*. Every <label> contained in *LE* shall identify a node label of *PG*.
 - b) Otherwise, *LE* is an *edge <label expression>*. Every <label> contained in *LE* shall identify an edge label of *PG*.

General Rules

None.

NOTE 128 — The <label expression> is enforced by the General Rules of Subclause 22.3, “Evaluation of a <path pattern expression>”, which in turn invokes Subclause 22.1, “Satisfaction of a <label expression> by a label set”.

Conformance Rules

None.

16.11 <simplified path pattern expression>

Function

Express a path pattern as a regular expression of edge labels.

Format

```

<simplified path pattern expression> ::=

  <simplified defaulting left>
  | <simplified defaulting undirected>
  | <simplified defaulting right>
  | <simplified defaulting left or undirected>
  | <simplified defaulting undirected or right>
  | <simplified defaulting left or right>
  | <simplified defaulting any direction>

<simplified defaulting left> ::=
  <left minus slash> <simplified contents> <slash minus>

<simplified defaulting undirected> ::=
  <tilde slash> <simplified contents> <slash tilde>

<simplified defaulting right> ::=
  <minus slash> <simplified contents> <slash minus right>

<simplified defaulting left or undirected> ::=
  <left tilde slash> <simplified contents> <slash tilde>

<simplified defaulting undirected or right> ::=
  <tilde slash> <simplified contents> <slash tilde right>

<simplified defaulting left or right> ::=
  <left minus slash> <simplified contents> <slash minus right>

<simplified defaulting any direction> ::=
  <minus slash> <simplified contents> <slash minus>

<simplified contents> ::=
  <simplified term>
  | <simplified path union>
  | <simplified multiset alternation>

<simplified path union> ::=
  <simplified term> <vertical bar> <simplified term>
  [ { <vertical bar> <simplified term> }... ]

<simplified multiset alternation> ::=
  <simplified term> <multiset alternation operator> <simplified term> [ { <multiset
  alternation operator> <simplified term> }... ]

<simplified term> ::=
  <simplified factor low>
  | <simplified concatenation>

<simplified concatenation> ::=
  <simplified term> <simplified factor low>

<simplified factor low> ::=
  <simplified factor high>
  | <simplified conjunction>

```

16.11 <simplified path pattern expression>

```

<simplified conjunction> ::==
  <simplified factor low> & <simplified factor high>

<simplified factor high> ::=
  <simplified tertiary>
  | <simplified quantified>
  | <simplified questioned>

<simplified quantified> ::=
  <simplified tertiary> <graph pattern quantifier>

<simplified questioned> ::=
  <simplified tertiary> ?<question mark>

<simplified tertiary> ::=
  <simplified direction override>
  | <simplified secondary>

<simplified direction override> ::=
  <simplified override left>
  | <simplified override undirected>
  | <simplified override right>
  | <simplified override left or undirected>
  | <simplified override undirected or right>
  | <simplified override left or right>
  | <simplified override any direction>

<simplified override left> ::=
  <left angle bracket> <simplified secondary>

<simplified override undirected> ::=
  <tilde> <simplified secondary>

<simplified override right> ::=
  <simplified secondary> <right angle bracket>

<simplified override left or undirected> ::=
  <left arrow tilde> <simplified secondary>

<simplified override undirected or right> ::=
  <tilde> <simplified secondary> <right angle bracket>

<simplified override left or right> ::=
  <left angle bracket> <simplified secondary> <right angle bracket>

<simplified override any direction> ::=
  <minus sign> <simplified secondary>

<simplified secondary> ::=
  <simplified primary>
  | <simplified negation>

<simplified negation> ::=
  !<exclamation mark> <simplified primary>

<simplified primary> ::=
  <label>
  | <left paren> <simplified contents> <right paren>
  | <left bracket> <simplified contents> <right bracket>

```

Syntax Rules

- 1) A <simplified negation> shall not contain a <simplified concatenation>, <simplified quantified>, <simplified questioned>, or <simplified multiset alternation>.
- 2) A <simplified direction override> shall not contain another <simplified direction override>.
- 3) A <simplified direction override> shall not contain <simplified concatenation>, <simplified quantified>, <simplified questioned>, or <simplified multiset alternation>.
- 4) A <simplified conjunction> shall not contain a <simplified concatenation>, <simplified quantified>, <simplified questioned>, or <simplified multiset alternation>.
- 5) A <simplified path pattern expression> *SPPE* is replaced by:

(*SPPE*)

NOTE 129 — This is done once for each <simplified path pattern expression> prior to the following recursive transformation and not with each iteration of the transformation.

- 6) The following rules are recursively applied until no <simplified path pattern expression>s remain.

NOTE 130 — The rules work from the root of the parse tree of a <simplified path pattern expression>. At each step, the coarsest analysis of a <simplified path pattern expression> is replaced, eliminating at least one level of the parse tree, measured from the root. Note that each replacement may create more <simplified path pattern expression>s than before, but these replacements have less depth. Eventually the recursion replaces <simplified path pattern expression> with <edge pattern>.

- a) Let *SPPE* be a <simplified path pattern expression>.
 - i) Let *SC* be the <simplified contents> contained in *SPPE*.
 - ii) Let *PREFIX* be the <minus slash>, <left minus slash>, <tilde slash>, <left tilde slash>, or <left minus slash> contained in *SPPE*.
 - iii) Let *SUFFIX* be the <slash minus right>, <slash minus>, <slash tilde>, or <slash tilde right> contained in *SPPE*.
 - iv) Let *EDGEPRE* and *EDGESUF* be determined by [Table 2, “Conversion of simplified syntax delimiters to default edge delimiters”](#) from the row containing the values of *PREFIX* and *SUFFIX*.

Table 2 — Conversion of simplified syntax delimiters to default edge delimiters

<i>PREFIX</i>	<i>SUFFIX</i>	<i>EDGEPRE</i>	<i>EDGESUF</i>
-/	/->	-[]->
<-/	/-	<-[]-
~/	/~	~[]~
~/	/~>	~[]~>
<~/	/~	<~[]~
<-/	/->	<-[]->
-/	/-	-[]-

16.11 <simplified path pattern expression>

b) Case:

- i) If *SC* is a <simplified path union> *SPU*, let *N* be the number of <simplified term>s simply contained in *SPU*, and let *ST*₁, ... *ST*_{*N*} be those <simplified term>s. Then *SPPE* is replaced by:

*PREFIX ST*₁ *SUFFIX* | *PREFIX ST*₂ *SUFFIX* | ... | *PREFIX ST*_{*N*} *SUFFIX*

- ii) If *SC* is a <simplified multiset alternation> *SMA*, let *N* be the number of <simplified term>s simply contained in *SMA*, and let *ST*₁, ... *ST*_{*N*} be those <simplified term>s. Then *SPPE* is replaced by:

*PREFIX ST*₁ *SUFFIX* |+| *PREFIX ST*₂ *SUFFIX* |+| ... |+| *PREFIX ST*_{*N*} *SUFFIX*

- iii) If *SC* is a <simplified concatenation> *SCAT*, let *ST* be the <simplified term> and let *SFL* be the <simplified factor low> simply contained in *SCAT*. Then *SPPE* is replaced by:

PREFIX ST SUFFIX PREFIX SFL SUFFIX

- iv) If *SC* is a <simplified conjunction> *SAND*, then *SPPE* is replaced by:

EDGEPRE IS SAND EDGESUF

NOTE 131 — As a result, *SAND* is now interpreted as a <label expression> within an <edge pattern>. By earlier Syntax Rules, there are no operators allowed in *SAND* that cannot be interpreted as operators of a <label expression>.

- v) If *SC* is a <simplified quantified> *SQ*, let *ST* be the <simplified tertiary> simply contained in *SC* and let *GPQ* be the <graph pattern quantifier> simply contained in *SQ*. Then *SPPE* is replaced by:

(*PREFIX ST SUFFIX*) *GPQ*

- vi) If *SC* is a <simplified questioned> *SQU*, let *ST* be the <simplified tertiary> simply contained in *SC*. Then *SPPE* is replaced by:

(*PREFIX ST SUFFIX*) ?

- vii) If *SC* is a <simplified direction override> *SDO*, let *SS* be the <simplified secondary> simply contained in *SDO*. Then

Case:

- 1) If *SDO* is <simplified override left>, then *SPPE* is replaced by:

<-[IS SS]-

- 2) If *SDO* is <simplified override undirected>, then *SPPE* is replaced by:

~[IS SS]~

- 3) If *SDO* is <simplified override left or undirected>, then *SPPE* is replaced by:

<~[IS SS]~

- 4) If *SDO* is <simplified override undirected or right>, then *SPPE* is replaced by:

~[IS SS]~>

- 5) If *SDO* is <simplified override left or right>, then *SPPE* is replaced by:

16.11 <simplified path pattern expression>

<- [IS SS] ->

- 6) If *SDO* is <simplified override any direction>, then *SPPE* is replaced by:

- [IS SS] -

- viii) If *SC* is a <simplified negation> *SN*, then *SPPE* is replaced by:

EDGEPR IS SN EDGESUF

NOTE 132 — As a result, *SN* is now interpreted as a <label expression> within an <edge pattern>. By earlier Syntax Rules, there are no operators allowed in *SN* that cannot be interpreted as operators of a <label expression>.

- ix) If *SC* is a <simplified primary> *SP*, then

Case:

- 1) If *SP* is a <label>, then *SPPE* is replaced by:

EDGEPR IS SP EDGESUF

- 2) Otherwise, let *INNER* be the <simplified contents> simply contained in *SC*. Then *SPPE* is replaced by:

(PREFIX INNER SUFFIX)

General Rules

None.

Conformance Rules

- 1) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain <simplified defaulting undirected>, <simplified defaulting left or undirected>, <simplified defaulting undirected or right>, or <simplified defaulting left or right>.
- 2) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain <simplified override undirected>, <simplified override left or undirected>, <simplified override undirected or right>, or <simplified override left or right>.

16.12 <where clause>

Function

Compute a new binding table by selecting records from the current working table fulfilling the specified <search condition>.

Format

```
<where clause> ::=  
    WHERE <search condition>
```

Syntax Rules

None.

General Rules

- 1) Let WC be the <where clause> and let SC be the <search condition> immediately contained in WC .
- 2) Let $WHERE$ be a new empty binding table.
- 3) For each record R of the current working table:
 - a) Let $INCLUDE$ be the result of evaluating SC in a new child execution context amended with R .
 - b) If $INCLUDE$ is True, then add R to $WHERE$.
- 4) The result of the application of this Subclause is $WHERE$.

Conformance Rules

None.

16.13 <procedure call>

Function

Define a <procedure call>.

Format

```
<procedure call> ::=  
  <inline procedure call>  
  | <named procedure call>
```

Syntax Rules

- 1) Let *PC* be the <procedure call>.
- 2) Case:
 - a) If the <named procedure call> *NPC* is specified, then the declared type of *PC* is the declared type of *NPC*.
 - b) Otherwise, the <inline procedure call> *IPC* is specified, and the declared type of *PC* is the declared type of *IPC*.

General Rules

- 1) Case:
 - a) If the <named procedure call> *NPC* is specified, then the General Rules of Subclause 16.15, “<named procedure call>” are applied to *NPC*; let *OUTCOME* be the *OUTCOME* returned from the application of these General Rules but let *RESULT* be the *RESULT* returned from the application of these General Rules.
 - b) Otherwise, the <inline procedure call> *IPC* is specified, and the General Rules of Subclause 16.14, “<inline procedure call>” are applied to *IPC*; let *OUTCOME* be the *OUTCOME* returned from the application of these General Rules but let *RESULT* be the *RESULT* returned from the application of these General Rules.
- 2) The outcome of the application of these General Rules is *OUTCOME* and the result of the application of these General Rules is *RESULT*.

Conformance Rules

None.

16.14<inline procedure call>

Function

Define an <inline procedure call>.

Format

```
<inline procedure call> ::=  
  <nested procedure specification>
```

Syntax Rules

- 1) Let *IPC* be the <inline procedure call>, and let *PROC* be the <nested procedure specification> that is immediately contained in *IPC*.
- 2) Let *PROCSIG* be the procedure signature of *PROC*.
- 3) The declared type of *IPC* is the expected result type of *PROCSIG*.

General Rules

- 1) Let *CONTEXT* be the current execution context.
- 2) The following steps are performed in a new child execution context with the working table of *CONTEXT* as its working table:
 - a) Execute *PROC* with no arguments.
 - b) Let *OUTCOME* be the current execution outcome and let *RESULT* be the current execution result.
- 3) The outcome of the application of these General Rules is *OUTCOME* and the result of the application of these General Rules is *RESULT*.

Conformance Rules

None.

16.15 <named procedure call>

Function

Define a <named procedure call>.

Format

```

<named procedure call> ::==
  <procedure reference> <left paren> [ <procedure argument list> ] <right paren>
  [ <yield clause> ]

<procedure argument list> ::==
  <procedure argument> [ { <comma> <procedure argument> }... ]

<procedure argument> ::==
  <value expression>

```

Syntax Rules

- 1) Let NPC be the <named procedure call> and let $PROC$ be the procedure referenced by the <procedure reference> that is immediately contained in NPC .
- 2) Let $PROCSIG$ be the procedure signature of $PROC$.
- 3) Let $ARGEXPS$ be the sequence of all <value expression>s that are simply contained in NPC in the order of their occurrence in NPC from left to right and let $NUMARGS$ be the number of such elements in $ARGEXPS$.
- 4) If $NUMARGS$ is less than the number of arguments required by $PROCSIG$, then an exception condition is raised: *procedure call error — incorrect number of arguments (G0001)*.
- 5) If $NUMARGS$ is greater than the number of arguments allowed by $PROCSIG$, then an exception condition is raised: *procedure call error — incorrect number of arguments (G0001)*.
- 6) Let $ARGEXP_i$ be the i -th element of $ARGEXPS$, for $1 \leq i \leq NUMARGS$.
- 7) If there is an $ARGEXP_i$ whose declared type is known but is not a subtype of the required procedure parameter type of the i -th procedure parameter of $PROCSIG$ for $1 \leq i \leq NUMARGS$, then an exception condition is raised: *procedure call error — invalid parameter type (G0002)*.
- 8) If NPC immediately contains a <yield clause> and the expected result type of $PROCSIG$ is not a binding table type, then an exception condition is raised: *procedure call error — invalid result type (G0003)*.
- 9) The declared type of NPC is the expected result type of $PROCSIG$.

General Rules

- 1) Let $ARGV_i$ be the result of evaluating $ARGEXP_i$, for $1 \leq i \leq NARGS$.
- 2) If there is an $ARGV_i$ such that $ARGV_i$ is not of the required procedure parameter type of the i -th procedure parameter of $PROCSIG$ for $1 \leq i \leq NUMARGS$, then an exception condition is raised: *procedure call error — invalid parameter type (G0002)*.
- 3) Let $CONTEXT$ be the current execution context.

- 4) Let R be a new record comprising fields F_i such that the field name of F_i is the procedure parameter name of i -th procedure parameter of $PROCSIG$ and the field value of F_i is $ARGV_i$, for $1 \leq i \leq NUMARGS$.
- 5) The following steps are performed in a new child execution context with R as its working record:
 - a) If $PROC$ is defined in the GQL-catalog, then:
 - i) Set the current working schema to the working schema of $PROC$.
 - ii) Set the current working graph to the working graph of $PROC$.
 - b) Execute $PROC$.
 - c) If a <yield clause> YC is specified, then:
 - i) If the current execution outcome is a successful outcome with a result table, then set the current working table to the result table of the current execution outcome. Otherwise, an exception condition is raised:
 - ii) The General Rules for Subclause 16.16, “<yield clause>” are applied to YC ; let $YIELD$ be the $YIELD$ returned from the application of these General Rules.
 - iii) Set the current execution outcome to a successful outcome with a result table of $YIELD$.
- 6) The outcome of the application of these General Rules is $OUTCOME$ and the result of the application of these General Rules is $RESULT$.

Conformance Rules

None.

16.16 <yield clause>

Function

Restructure a binding table.

Format

```
<yield clause> ::=  
    YIELD <yield item list>  
  
<yield item list> ::=  
    <yield item> [ { <comma> <yield item> }... ]  
  
<yield item> ::=  
    { <yield item name> [ <yield item alias> ] }  
  
<yield item name> ::=  
    <identifier>  
  
<yield item alias> ::=  
    AS <variable name>
```

Syntax Rules

- 1) Let YC be the <yield clause>.
- 2) Let YIL be the collection of <yield item>s simply contained in YC .
- 3) Let n be the number of <yield item>s in YIL .
- 4) For each <yield item> YI_i from YIL , $1 \leq i \leq n$:
 - a) Let YIN_i be the <identifier> immediately contained in the <yield item name> specified by YI_i .
 - b) If YI_i does not immediately contain a <yield item alias>, then YI_i is effectively replaced by YIN_i AS YIN_i .

General Rules

- 1) Let $YIELD$ be a new empty binding table.
- 2) For each record R of the current working table in a new child execution context amended with R :
 - a) Let T be a new empty record.
 - b) For each <yield item> YI_i from YIL , $1 \leq i \leq n$:
 - i) Let YIN_i be the <yield item name> specified by YI_i .
 - ii) Let F_i be the field of the current working record whose field name is YIN_i .
 - iii) Let YIV_i be the field value of F_i .

NOTE 133 — As opposed to the General Rules for <binding variable>, <yield item>s only consider the current working record and ignore the working records of any parent execution contexts of the current execution context in the current execution stack.

- iv) Let YIA_i be the <variable name> contained in the <yield item alias> specified by YI_i .
 - v) Add a new field with field name YIA_i and with field value YIV_i to T .
- c) Add T to $YIELD$.
- 3) The result of the application of this Subclause is $YIELD$.

16.17 <group by clause>

Function

Define a <group by clause> for specifying the set of grouping keys to be used during grouping.

Format

```

<group by clause> ::= 
    GROUP BY <grouping element list>

<grouping element list> ::= 
    <grouping element> [ { <comma> <grouping element> } ]
    | <empty grouping set>

<grouping element> ::= 
    <binding variable>

<empty grouping set> ::= 
    <left paren> <right paren>

```

Syntax Rules

- 1) Every <binding variable> shall unambiguously reference a column of the current working table.

General Rules

- 1) Let *GBC* be the <group by clause> and let *GEL* be the <grouping element list>.
 - 2) Case:
 - a) If *GEL* is the <empty grouping set>, then let *GROUP_BY* be a duplicate-free binding table comprising the unit value.

NOTE 134 — The unit value is a **value** of the **record type** that has zero **fields** in this document.
 - b) Otherwise:
 - i) Let *GESEQ* be the sequence of <grouping element>s immediately contained in *GEL* in the order of their occurrence from left to right and let *NGESEQ* the number of such <grouping element>s in *GESEQ*.
 - ii) Let GE_i be *i*-th element of *GESEQ* and let *GEBV_i* be the <binding variable> simply contained in *GEBV_i*, for $1 \leq i \leq NGESEQ$.
 - iii) Let *GROUP_BY* be a new empty duplicate-free binding table.
 - iv) For each record *R* of the current working table in a new child execution context amended with *R*:
 - 1) Let *T* be a new record comprising fields *F_i* such that the field name of *F_i* is *GEBV_i* and the field value of *F_i* is the result of evaluating *GEBV_i*, for $1 \leq i \leq NGESEQ$.
 - 2) If *GROUP_BY* does not contain *T*, then *T* is added.
 - 3) The result of the application of this Subclause is the duplicate-free binding table *GROUP_BY*.

Conformance Rules

None.

16.18 <order by clause>

Function

Define an <order by clause> for obtaining an ordered binding table from the current working table.

Format

```
<order by clause> ::=  
    ORDER BY <sort specification list>
```

Syntax Rules

None.

General Rules

- 1) Let *OBC* be the <order by clause>. Let *SSL* be the <sort specification list> immediately contained in *OBC*.
- 2) Let *ORDER_BY* be a new ordered binding table created from a collection of all records of the current working table sorted by applying the General Rules of Subclause 16.20, “<sort specification list>” to *SSL*.
- 3) The result of the application of this Subclause is the binding table *ORDER_BY*.

Conformance Rules

None.

16.19 <aggregate function>

Function

Specify a value computed from a collection of rows.

Format

```

<aggregate function> ::==
  COUNT <left paren> <asterisk> <right paren>
  | <general set function>
  | <binary set function>

<general set function> ::=
  <general set function type>
    <left paren> <set quantifier> <value expression> <right paren>

<binary set function> ::=
  <binary set function type>
    <left paren> <dependent value expression> <comma> <independent value expression>
  <right paren>

<general set function type> ::=
  AVG
  | COUNT
  | MAX
  | MIN
  | SUM
  | PRODUCT
  | COLLECT
  | stDev
  | stDevP

<set quantifier> ::=
  DISTINCT
  | ALL

<binary set function type> ::=
  percentileCont
  | percentileDist

<dependent value expression> ::=
  [ <set quantifier> ] <numeric value expression>

<independent value expression> ::=
  <numeric value expression>

```

Syntax Rules

- 1) Let AF be the <aggregate function>.
- 2) If <general set function> is specified and <set quantifier> is not specified, then ALL is implicit.
- 3) If <dependent value expression> is specified and <set quantifier> is not specified, then ALL is implicit.
- 4) AF shall not contain a <procedure body>.
- 5) A <general set function> shall not contain an <aggregate function>.
- 6) A <dependent value expression> shall not contain an <aggregate function>.

- 7) If COUNT is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

General Rules

- 1) Let *TABLE* be the current working table.
- 2) Case:
 - a) If *AF* immediately contains the <general set function> *GSF*, then let *EXP* be the <value expression> immediately contained in *GSF* and let *SQ* be the <set quantifier> immediately contained in *GSF*.
 - b) Otherwise, *AF* immediately contains the <binary set function> *BSF*. Let *EXP* be the <dependent value expression> immediately contained in *BSF* and let *SQ* be the <set quantifier> immediately contained in *EXP*.
- 3) Let *VALUES* be a new empty collection.
- 4) For each record *R* of *TABLE* in a new child execution context amended with *R*:
 - a) Let *EXPRE* be the result of evaluating *EXP*.
 - b) If *SQ* is DISTINCT and *EXPRE* is not in *VALUES*, then add *EXPRE* to *VALUES*. Otherwise, add *EXPRE* to *VALUES*.
- 5) Case:
 - a) If *AF* immediately contains the <general set function> *GSF*, then let *RESULT* be the result of evaluating *GSF* on *VALUES*.
 - b) Otherwise, *AF* immediately contains the <binary set function> *BSF*. Let *IVE* be the <independent value expression> immediately contained in *BSF*, let *IVERE* be the result of evaluating *IVE*, and let *RESULT* be the result of evaluating *BSF* on *VALUES* and *IVERE*.
- 6) The result of evaluating *AF* is *RESULT*.

Conformance Rules

None.

16.20 <sort specification list>

Function

Specify a sort order.

Format

```

<sort specification list> ::= 
  <sort specification> [ { <comma> <sort specification> }... ]

<sort specification> ::= 
  <sort key> [ <ordering specification> ] [ <null ordering> ]

<sort key> ::= 
  <value expression>

<ordering specification> ::= 
  ASC
  | DESC

<null ordering> ::= 
  NULLS FIRST
  | NULLS LAST

```

Syntax Rules

- 1) Each <value expression> immediately contained in the <sort key> contained in a <sort specification> is an operand of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 22.7, “Ordering operations”, apply.
- 2) Let *SSL* be the <sort specification list>
- 3) Let *NSS* be the number of <sort specification>s immediately contained in *SSL*.
- 4) Let *SS_i*, $1 \leq i \leq NSS$, be the *i*-th <sort specification> immediately contained in *SSL*.
- 5) For each *SS_i*, $1 \leq i \leq NSS$:
 - a) Let *SK_i* be the <sort key> immediately contained in *SS_i*.
 - b) If *SS_i* does not immediately contain an <ordering specification>, then *SS_i* is effectively replaced by:
 $SK_i \text{ ASC}$
- 6) If <null ordering> is not specified, then an implementation-defined <null ordering> is implicit. The implementation-defined default for <null ordering> shall not depend on the context outside of <sort specification list>.

General Rules

- 1) A <sort specification list> defines an ordering of rows, as follows:
 - a) Let *N* be the number of <sort specification>s.

16.20 <sort specification list>

- b) Let K_i , $1 \leq i \leq N$, be the <sort key> contained in the i -th <sort specification>.
- c) Each <sort specification> specifies the *sort direction* for the corresponding sort key K_i . If DESC is not specified in the i -th <sort specification>, then the sort direction for K_i is ascending and the applicable <comp op> is the <less than operator>. Otherwise, the sort direction for K_i is descending and the applicable <comp op> is the <greater than operator>.
- d) Let P be any record of the collection of rows to be ordered, and let Q be any other record of the same collection of rows.
- e) Let PV_i and QV_i be the values of K_i in P and Q , respectively. The relative position of rows P and Q in the result is determined by comparing PV_i and QV_i as follows:
 - i) The comparison is performed according to the General Rules of Case:
 - 1) If the declared type of K_i is a record type, then this Subclause, applied recursively.
 - 2) Otherwise, Subclause 19.3, “<comparison predicate>”, where the <comp op> is the applicable <comp op> for K_i .
 - ii) The comparison is performed with the following special treatment of null values.
 - Case:
 - 1) If PV_i and QV_i are both the null value, then they are considered equal to each other.
 - 2) If PV_i is the null value and QV_i is not the null value, then
 - Case:
 - A) If NULLS FIRST is specified or implied, then $PV_i <\text{comp op}> QV_i$ is considered to be True.
 - B) If NULLS LAST is specified or implied, then $PV_i <\text{comp op}> QV_i$ is considered to be False.
 - 3) If PV_i is not the null value and QV_i is the null value, then
 - Case:
 - A) If NULLS FIRST is specified or implied, then $PV_i <\text{comp op}> QV_i$ is considered to be False.
 - B) If NULLS LAST is specified or implied, then $PV_i <\text{comp op}> QV_i$ is considered to be True.
 - f) PV_i is said to *precede* QV_i if the value of the <comparison predicate> “ $PV_i <\text{comp op}> QV_i$ ” is True for the applicable <comp op>.
 - g) If PV_i and QV_i are not the null value and the result of “ $PV_i <\text{comp op}> QV_i$ ” is Unknown, then the relative ordering of PV_i and QV_i is implementation-dependent.
 - h) The relative position of record P is before record Q if PV_n precedes QV_n for some n , $1 \leq n \leq N$, and PV_i is not distinct from QV_i for all $i < n$.
 - i) Two rows that are not distinct with respect to the <sort specification>s are said to be *peers* of each other. The relative ordering of peers is implementation-dependent.

Conformance Rules

None.

16.21 <limit clause>

Function

Define a <limit clause> for obtaining a new binding table that retains only a limited number of records of the current working table.

Format

```
<limit clause> ::=  
    LIMIT <unsigned integer specification>
```

Syntax Rules

None.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) If *TABLE* is not ordered, then let *ORDERED_TABLE* be a new ordered binding table created from the result of sorting the collection of all records of *TABLE* according to an implementation-dependent order. Otherwise, let *ORDERED_TABLE* be *TABLE*.
- 3) Let *LIMIT* be a new ordered binding table obtained by selecting only the first *V* records of *ORDERED_TABLE* and discarding all subsequent records.
- 4) The result of the application of this Subclause is the binding table *LIMIT*.

Conformance Rules

None.

16.22 <offset clause>

Function

Define an <offset clause> for obtaining a new binding table that retains all records of the current working table except for some discarded initial records.

Format

```
<offset clause> ::=  
  <offset synonym> <unsigned integer specification>  
  
<offset synonym> ::=  
  OFFSET | SKIP
```

Syntax Rules

None.

General Rules

- 1) Let *TABLE* be the current working table.
- 2) If *TABLE* is not ordered, then let *ORDERED_TABLE* be a new ordered binding table created from the result of sorting the collection of all records of *TABLE* according to an implementation-dependent order. Otherwise, let *ORDERED_TABLE* be *TABLE*.
- 3) Let *OFFSET* be a new ordered binding table obtained from all but the first *V* records of *ORDERED_TABLE*.
- 4) The result of the application of this Subclause is the binding table *OFFSET*.

Conformance Rules

None.

17 Object references

17.1 Schema references

Function

Define a schema reference.

Format

```
<schema reference> ::=  
  <predefined schema parameter>  
  | <catalog schema parent and name>  
  | <external object reference>  
  
<catalog schema parent and name> ::=  
  [ <absolute url path> ] <solidus> <schema name>  
  | <url path parameter>
```

Syntax Rules

- 1) If the <schema reference> *SR* is specified, then the result of *SR* is the result of the immediately contained <predefined schema parameter>, <catalog schema parent and name>, or <external object reference>.
- 2) If the <catalog schema parent and name> *CSPN* is specified, then
 - a) After applying all relevant syntactic transformations, let *PARENT* be determined as follows:

Case:

 - i) If *CSPN* does not immediately contain and <absolute url path>, then *PARENT* is the catalog root and shall be a GQL-directory.
 - ii) Otherwise, *PARENT* is the result of the immediately contained <absolute url path> and shall be a GQL-directory.
 - b) The result of *CSPN* is determined as follows:
 - i) Let *SN* be the <schema name> immediately contained in *CSPN*.
 - ii) *SN* shall identify an existing schema descriptor in *PARENT*.
 - iii) The result of *CSPN* is the GQL-schema identified by *SN* in *PARENT*.

General Rules

None.

Conformance Rules

None.

17.2 Graph references

Function

Specify a graph reference.

Format

```

<graph reference> ::= 
    <graph resolution expression>
  | <local graph reference>

<graph resolution expression> ::= 
  [ PROPERTY ] GRAPH <catalog graph reference>

<catalog graph reference> ::= 
    <catalog graph parent and name>
  | <predefined graph parameter>
  | <external object reference>

<catalog graph parent and name> ::= 
    <graph parent specification> <graph name>
  | <curl path parameter>

<graph parent specification> ::= 
  [ <parent catalog object reference> ] [ <qualified object name> <period> ]

<local graph reference> ::= 
    <qualified graph name>

<qualified graph name> ::= 
  [ <qualified object name> <period> ] <graph name>

```

Syntax Rules

- 1) If the <graph parent specification> *GPS* does not immediately contain a <parent catalog object reference>, then a <parent catalog object reference> of ./ is implicit.

General Rules

- 1) If the <graph reference> *GR* is specified, then the result of *GR* is the result of the immediately contained <graph resolution expression> or <local graph reference>.
- 2) If the <graph resolution expression> *GRE* is specified, then the result of *GRE* is the result of the immediately contained <catalog graph reference>.
- 3) If the <catalog graph reference> *CGR* is specified, then:
 - a) Let *ROCGR* be determined as follows.

Case:

- i) If *CGR* simply contains the <graph parent specification> *GPS* and the <graph name> *GN*, then *ROCGR* is the result of resolving *GN* with the sequence comprising only the result of *GPS* as the start object candidate sequence, as specified by [General Rule 7](#).
- ii) If *CGR* immediately contains the <predefined graph parameter> *PGP*, then *ROCGR* is the result of *PGP*.

- iii) If CGR immediately contains an <external object reference> EOR , then the result of $ROCGR$ is the result of EOR .
 - b) If $ROCGR$ is a graph, then the result of CGR is $ROCGR$. Otherwise, an exception condition is raised:
- 4) If the <graph parent specification> GPS is specified, then let $ROPCOR$ be the result of the <parent catalog object reference> immediately contained in GPS and determine the result of GPS as follows.
- Case:
- a) If GPS immediately contains the <qualified object name> QON , then the result of GPS is result of alias-resolving QON with the sequence comprising only $ROPCOR$ as the start object candidate sequence, as specified by General Rule 3) of Subclause 17.9, “<qualified object name>”.
 - b) Otherwise, the result of GPS is $ROPCOR$.
- 5) If the <local graph reference> LGR is specified, then the result of LGR is the result of resolving the <qualified graph name> immediately contained in LGR with the current working record with appended current working schema as the start object candidate sequence, as specified by General Rule 6).
- 6) If the <qualified graph name> QGN is specified, then QGN is resolved with the start object candidate sequence $SOCSEQ$ as follows:
- a) If QGN immediately contains the <qualified object name> QON , then $SOCSEQ'$ is the sequence comprising the result of alias-resolving QON with $SOCSEQ$ as the start object candidate sequence, as specified by General Rule 3) of Subclause 17.9, “<qualified object name>”. Otherwise, $SOCSEQ'$ is $SOCSEQ$.
 - b) Let GN be the <graph name> immediately contained in QGN and let RES be the result of resolving GN with $SOCSEQ'$ as the start object candidate sequence, as specified by General Rule 7).
 - c) The result of resolving QGN with $SOCSEQ$ as the candidate start object sequence is RES .
- 7) If the <graph name> GN is specified, then GN is resolved with the start object candidate sequence $SOCSEQ$ as follows:
- a) Let $NSOCSEQ$ be the number of elements of $SOCSEQ$, let SOC_i be the i -th element of $SOCSEQ$, and let RES_i be determined as follows, for $1 \leq i \leq NSOCSEQ$.
- Case:
- i) If SOC_i is a GQL-object with named components that contains a graph G with name GN , then RES_i is G .
 - ii) If SOC_i is a GQL-object with named components that contains an alias with name GN that resolves to a graph G , then RES_i is G .
 - iii) If SOC_i is a GQL-object with named components that contains a simple view SV with name GN that returns a graph, then RES_i is the graph obtained by calling SV with no arguments.
 - iv) If SOC_i is a GQL-object with named components that contains an alias with name GN that resolves to a simple view SV that returns a graph, then RES_i is the graph obtained by calling SV with no arguments.
 - v) Otherwise, RES_i is the null value.

- b) If there is $1 \leq k \leq NSOCSEQ$ such that RES_k is not the null value and such that for all $1 \leq j < k$ it holds that RES_j is the null value, then the result of resolving GN with the start object candidate sequence $SOCSEQ$ is RES_k . Otherwise, an exception condition is raised:

Conformance Rules

None.

17.3 Graph type references

Function

Specify a graph type reference.

Format

```

<graph type reference> ::==
  <graph type resolution expression>
  | <local graph type reference>

<graph type resolution expression> ::==
  [ PROPERTY ] GRAPH TYPE <catalog graph type reference>

<catalog graph type reference> ::==
  <catalog graph type parent and name>
  | <external object reference>

<catalog graph type parent and name> ::==
  <graph type parent specification> <graph type name>
  | <curl path parameter>

<graph type parent specification> ::==
  [ <parent catalog object reference> ] [ <qualified object name> <period> ]

<local graph type reference> ::==
  <qualified graph type name>

<qualified graph type name> ::==
  [ <qualified object name> <period> ] <graph type name>

```

Syntax Rules

- 1) If the <graph type parent specification> *GTPS* does not immediately contain a <parent catalog object reference>, then a <parent catalog object reference> of ./ is implicit.

General Rules

- 1) If the <graph type reference> *GTR* is specified, then the result of *GTR* is the result of the immediately contained <graph type resolution expression> or <local graph type reference>.
- 2) If the <graph type resolution expression> *GTRE* is specified, then the result of *GTRE* is the result of the immediately contained <catalog graph type reference>.
- 3) If the <catalog graph type reference> *CGTR* is specified, then:
 - a) Let *ROCGTR* be determined as follows.

Case:

- i) If *CGTR* simply contains the <graph type parent specification> *GTPS* and the <graph type name> *GTN*, then *ROCGTR* is the result of resolving *GTN* with the sequence comprising only the result of *GTPS* as the start object candidate sequence, as specified by [General Rule 7](#).
- ii) If *CGTR* immediately contains an <external object reference> *EOR*, then the result of *ROCGTR* is the result of *EOR*.

17.3 Graph type references

- b) If $ROCGTR$ is a graph type, then the result of $CGTR$ is $ROCGTR$. Otherwise, an exception condition is raised:
- 4) If the <graph type parent specification> $GTPS$ is specified, then let $ROPCOR$ be the result of the <parent catalog object reference> immediately contained in GPS and determine the result of $GTPS$ as follows.
- Case:
- a) If $GTPS$ immediately contains the <qualified object name> QON , then the result of $GTPS$ is result of alias-resolving QON with the sequence comprising only $ROPCOR$ as the start object candidate sequence, as specified by [General Rule 3](#) of Subclause 17.9, “<qualified object name>”.
 - b) Otherwise, the result of $GTPS$ is $ROPCOR$.
- 5) If the <local graph type reference> $LGTR$ is specified, then the result of $LGTR$ is the result of resolving the <qualified graph type name> immediately contained in $LGTR$ with the current working record with appended current working schema as the start object candidate sequence, as specified by [General Rule 6](#).
- 6) If the <qualified graph type name> $QGTN$ is specified, then $QGTN$ is resolved with the start object candidate sequence $SOCSEQ$ as follows:
- a) If $QGTN$ immediately contains the <qualified object name> QON , then $SOCSEQ'$ is the sequence comprising the result of alias-resolving QON with $SOCSEQ$ as the start object candidate sequence, as specified by [General Rule 3](#) of Subclause 17.9, “<qualified object name>”. Otherwise, $SOCSEQ'$ is $SOCSEQ$.
 - b) Let GTN be the <graph type name> immediately contained in $QGTN$ and let RES be the result of resolving GTN with $SOCSEQ'$ as the start object candidate sequence, as specified by [General Rule 7](#).
 - c) The result of resolving $QGTN$ with $SOCSEQ$ as the candidate start object sequence is RES .

- 7) If the <graph type name> GTN is specified, then GTN is resolved with the start object candidate sequence $SOCSEQ$ as follows:

- a) Let $NSOCSEQ$ be the number of elements of $SOCSEQ$, let SOC_i be the i -th element of $SOCSEQ$, and let RES_i be determined as follows, for $1 \leq i \leq NSOCSEQ$.

Case:

- i) If SOC_i is a GQL-object with named components that contains a graph type GT with name GTN , then RES_i is GT .
 - ii) If SOC_i is a GQL-object with named components that contains an alias with name GTN that resolves to a graph type GT , then RES_i is GT .
 - iii) Otherwise, RES_i is the null value.
- b) If there is $1 \leq k \leq NSOCSEQ$ such that RES_k is not the null value and such that for all $1 \leq j < k$ it holds that RES_j is the null value, then the result of resolving GTN with the start object candidate sequence $SOCSEQ$ is RES_k . Otherwise, an exception condition is raised:

Conformance Rules

None.

17.4 Binding table references

Function

Specify a binding table reference.

Format

```

<binding table reference> ::==
  <binding table resolution expression>
  | <local binding table reference>

<binding table resolution expression> ::==
  [ BINDING ] TABLE <catalog binding table reference>

<catalog binding table reference> ::==
  <catalog binding table parent and name>
  | <predefined table parameter>
  | <external object reference>

<catalog binding table parent and name> ::==
  <binding table parent specification> <binding table name>
  | <curl path parameter>

<binding table parent specification> ::==
  [ <parent catalog object reference> ] [ <qualified object name> <period> ]

<local binding table reference> ::==
  <qualified binding table name>

<qualified binding table name> ::==
  [ <qualified object name> <period> ] <binding table name>

```

Syntax Rules

- 1) If the <binding table parent specification> *BTPS* does not immediately contain a <parent catalog object reference>, then a <parent catalog object reference> of . / is implicit.

General Rules

- 1) If the <binding table reference> *BTR* is specified, then the result of *BTR* is the result of the immediately contained <binding table resolution expression> or <local binding table reference>.
- 2) If the <binding table resolution expression> *BTRE* is specified, then the result of *BTRE* is the result of the immediately contained <catalog binding table reference>.
- 3) If the <catalog binding table reference> *CBTR* is specified, then:
 - a) Let *ROCBTR* be determined as follows.

Case:

- i) If *CBTR* simply contains the implicit or explicit <binding table parent specification> *BTPS* and the <binding table name> *BTN*, then *ROCBTR* is the result of resolving *BTN* with the sequence comprising only the result of *BTPS* as the start object candidate sequence, as specified by [General Rule 7](#).

17.4 Binding table references

- ii) If $CBTR$ immediately contains the <predefined table parameter> PTP , then $ROCBTR$ is the result of PTP .
 - iii) If $CBTR$ immediately contains an <external object reference> EOR , then the result of $ROCBTR$ is the result of EOR .
- b) If $ROCBTR$ is a binding table, then the result of $CBTR$ is $ROCBTR$. Otherwise, an exception condition is raised:
- 4) If the implicit or explicit <binding table parent specification> $BTPS$ is specified, then let $ROPCOR$ be the result of the <parent catalog object reference> immediately contained in $BTPS$ and determine the result of $BTPS$ as follows.
- Case:
- a) If $BTPS$ immediately contains the <qualified object name> QON , then the result of $BTPS$ is result of alias-resolving QON with the sequence comprising only $ROPCOR$ as the start object candidate sequence, as specified by [General Rule 3](#) of [Subclause 17.9](#), “<qualified object name>”.
 - b) Otherwise, the result of $BTPS$ is $ROPCOR$.
- 5) If the <local binding table reference> $LBTR$ is specified, then the result of $LBTR$ is the result of resolving the <qualified binding table name> immediately contained in $LBTR$ with the current working record with appended current working schema as the start object candidate sequence, as specified by [General Rule 6](#).
- 6) If the <qualified binding table name> $QBTN$ is specified, it is resolved with the start object candidate sequence $SOCSEQ$ as follows:
- a) If $QBTN$ immediately contains the <qualified object name> QON , then $SOCSEQ'$ is the sequence comprising the result of alias-resolving QON with $SOCSEQ$ as the start object candidate sequence, as specified by [General Rule 3](#) of [Subclause 17.9](#), “<qualified object name>”. Otherwise, $SOCSEQ'$ is $SOCSEQ$.
 - b) Let BTN be the <binding table name> immediately contained in $QBTN$ and let RES be the result of resolving BTN with $SOCSEQ'$ as the start object candidate sequence, as specified by [General Rule 7](#).
 - c) The result of resolving $QBTN$ with $SOCSEQ$ as the candidate start object sequence is RES .
- 7) If the <binding table name> BTN is specified, BTN is resolved with the start object candidate sequence $SOCSEQ$ as follows:

- a) Let $NSOCSEQ$ be the number of elements of $SOCSEQ$, let SOC_i be the i -th element of $SOCSEQ$, and let RES_i be determined as follows, for $1 \leq i \leq NSOCSEQ$.

Case:

- i) If SOC_i is a GQL-object with named components that contains a binding table BT with name BTN , then RES_i is BT .
- ii) If SOC_i is a GQL-object with named components that contains an alias with name BTN that resolves to a binding table BT , then RES_i is BT .
- iii) If SOC_i is a GQL-object with named components that contains a query Q with name BTN that returns a binding table, then RES_i is the binding table obtained by calling Q with no arguments.

- iv) If SOC_i is a GQL-object with named components that contains an alias with name BTN that resolves to a query Q that returns a binding table, then RES_i is the binding table obtained by calling Q with no arguments.
 - v) Otherwise, RES_i is the null value.
- b) If there is $1 \leq k \leq NSOCSEQ$ such that RES_k is not the null value and such that for all $1 \leq j < k$ it holds that RES_j is the null value, then the result of resolving BTN with the start object candidate sequence $SOCSEQ$ is RES_k . Otherwise, an exception condition is raised:

Conformance Rules

None.

17.5 Procedure references

Function

Specify a procedure reference.

Format

```

<procedure reference> ::= 
    <procedure resolution expression>
  | <local procedure reference>

<procedure resolution expression> ::= 
  PROCEDURE <catalog procedure reference>

<catalog procedure reference> ::= 
    <catalog procedure parent and name>
  | <external object reference>

<catalog procedure parent and name> ::= 
    <procedure parent specification> <procedure name>
  | <curl path parameter>

<procedure parent specification> ::= 
  [ <parent catalog object reference> ] [ <qualified object name> <period> ]

<local procedure reference> ::= 
  <qualified procedure name>

<qualified procedure name> ::= 
  [ <qualified object name> <period> ] <procedure name>

```

Syntax Rules

- 1) If the <procedure parent specification> *PPS* does not immediately contain a <parent catalog object reference>, then a <parent catalog object reference> of *./* is implicit.

General Rules

- 1) If the <procedure reference> *PR* is specified, then the result of *PR* is the result of the immediately contained <procedure resolution expression> or <local procedure reference>.
- 2) If the <procedure resolution expression> *PRE* is specified, then the result of *PRE* is the result of the immediately contained <catalog procedure reference>.
- 3) If the <catalog procedure reference> *CPR* is specified, then:
 - a) Let *ROCPR* be determined as follows.
 - i) If *CPR* simply contains the implicit or explicit <procedure parent specification> *PPS* and the <procedure name> *PN*, then *ROCPR* is the result of resolving *PN* with the sequence comprising only the result of *PPS* as the start object candidate sequence, as specified by [General Rule 7](#).
 - ii) If *CPR* immediately contains an <external object reference> *EOR*, then the result of *ROCPR* is the result of *EOR*.

- b) If $ROCPR$ is a procedure, then the result of CPR is $ROCPR$. Otherwise, an exception condition is raised:
 - 4) If the implicit or explicit <procedure parent specification> PPS is specified, then let $ROPCOR$ be the result of the <parent catalog object reference> immediately contained in PPS and determine the result of PPS as follows.
- Case:
- a) If PPS immediately contains the <qualified object name> QON , then the result of PPS is result of alias-resolving QON with the sequence comprising only $ROPCOR$ as the start object candidate sequence, as specified by [General Rule 3](#) of Subclause 17.9, “<qualified object name>”.
 - b) Otherwise, the result of PPS is $ROPCOR$.

- 5) If the <local procedure reference> LPR is specified, then the result of LPR is the result of resolving the <qualified procedure name> immediately contained in LPR with the current working record with appended current working schema as the start object candidate sequence, as specified by [General Rule 6](#).
- 6) If the <qualified procedure name> QPN is specified, then QPN is resolved with the start object candidate sequence $SOCSEQ$ as follows:
 - a) If QPN immediately contains the <qualified object name> QON , then $SOCSEQ'$ is the sequence comprising the result of alias-resolving QON with $SOCSEQ$ as the start object candidate sequence, as specified by [General Rule 3](#) of Subclause 17.9, “<qualified object name>”. Otherwise, $SOCSEQ'$ is $SOCSEQ$.
 - b) Let PN be the <procedure name> immediately contained in QPN and let RES be the result of resolving PN with $SOCSEQ'$ as the start object candidate sequence, as specified by [General Rule 7](#).
 - c) The result of resolving QPN with $SOCSEQ$ as the candidate start object sequence is RES .

- 7) If the <procedure name> PN is specified, then PN is resolved with the start object candidate sequence $SOCSEQ$ as follows:
 - a) Let $NSOCSEQ$ be the number of elements of $SOCSEQ$, let SOC_i be the i -th element of $SOCSEQ$, and let RES_i be determined as follows, for $1 \leq i \leq NSOCSEQ$.
 - Case:
 - i) If SOC_i is a GQL-object with named components that contains a procedure P with name PN , then RES_i is P .
 - ii) If SOC_i is a GQL-object with named components that contains an alias with name PN that resolves to a procedure P , then RES_i is P .
 - iii) Otherwise, RES_i is the null value.
 - b) If there is $1 \leq k \leq NSOCSEQ$ such that RES_k is not the null value and such that for all $1 \leq j < k$ it holds that RES_j is the null value, then the result of resolving PN with the start object candidate sequence $SOCSEQ$ is RES_k . Otherwise, an exception condition is raised:

Conformance Rules

None.

17.6 Query references

Function

Specify a query reference.

Format

```

<query reference> ::= 
  <query resolution expression>
  | <local query reference>

<query resolution expression> ::= 
  QUERY <catalog query reference>

<catalog query reference> ::= 
  <catalog query parent and name>
  | <external object reference>

<catalog query parent and name> ::= 
  <query parent specification> <query name>
  | <curl path parameter>

<query parent specification> ::= 
  [ <parent catalog object reference> ] [ <qualified object name> <period> ]

<local query reference> ::= 
  <qualified query name>

<qualified query name> ::= 
  [ <qualified object name> <period> ] <query name>

```

Syntax Rules

- 1) If the <query parent specification> *QPS* does not immediately contain a <parent catalog object reference>, then a <parent catalog object reference> of ./ is implicit.

General Rules

- 1) If the <query reference> *QR* is specified, then the result of *QR* is the result of the immediately contained <query resolution expression> or <local query reference>.
- 2) If the <query resolution expression> *QRE* is specified, then the result of *QRE* is the result of the immediately contained <catalog query reference>.
- 3) If the <catalog query reference> *CQR* is specified, then:
 - a) Let *ROCQR* be determined as follows.

Case:

 - i) If *CQR* simply contains the implicit or explicit <query parent specification> *QPS* and the <query name> *QN*, then *ROCQR* is the result of resolving *QN* with the sequence comprising only the result of *QPS* as the start object candidate sequence, as specified by [General Rule 7](#).
 - ii) If *CQR* immediately contains an <external object reference> *EOR*, then the result of *ROCQR* is the result of *EOR*.

- b) If $ROCQR$ is a query, then the result of CQR is $ROCQR$. Otherwise, an exception condition is raised:
- 4) If the implicit or explicit <query parent specification> QPS is specified, then let $ROPCOR$ be the result of the <parent catalog object reference> immediately contained in QPS and determine the result of QPS as follows.

Case:

- a) If QPS immediately contains the <qualified object name> QON , then the result of QPS is result of alias-resolving QON with the sequence comprising only $ROPCOR$ as the start object candidate sequence, as specified by [General Rule 3](#) of [Subclause 17.9, “<qualified object name>”](#).
 - b) Otherwise, the result of QPS is $ROPCOR$.
- 5) If the <local query reference> LQR is specified, then the result of LQR is the result of resolving the <qualified query name> immediately contained in LQR with the current working record with appended current working schema as the start object candidate sequence, as specified by [General Rule 6](#).
- 6) If the <qualified query name> QQN is specified, then QQN is resolved with the start object candidate sequence $SOCSEQ$ as follows:
- a) If QQN immediately contains the <qualified object name> QON , then $SOCSEQ'$ is the sequence comprising the result of alias-resolving QON with $SOCSEQ$ as the start object candidate sequence, as specified by [General Rule 3](#) of [Subclause 17.9, “<qualified object name>”](#). Otherwise, $SOCSEQ'$ is $SOCSEQ$.
 - b) Let QN be the <query name> immediately contained in QQN and let RES be the result of resolving QN with $SOCSEQ'$ as the start object candidate sequence, as specified by [General Rule 7](#).
 - c) The result of resolving QQN with $SOCSEQ$ as the candidate start object sequence is RES .
- 7) If the <query name> QN is specified, then QN is resolved with the start object candidate sequence $SOCSEQ$ as follows:
- a) Let $NSOCSEQ$ be the number of elements of $SOCSEQ$, let SOC_i be the i -th element of $SOCSEQ$, and let RES_i be determined as follows, for $1 \leq i \leq NSOCSEQ$.
- Case:
- i) If SOC_i is a GQL-object with named components that contains a query Q with name QN , then RES_i is Q .
 - ii) If SOC_i is a GQL-object with named components that contains an alias with name QN that resolves to a query Q , then RES_i is Q .
 - iii) Otherwise, RES_i is the null value.
- b) If there is $1 \leq k \leq NSOCSEQ$ such that RES_k is not the null value and such that for all $1 \leq j < k$ it holds that RES_j is the null value, then the result of resolving QN with the start object candidate sequence $SOCSEQ$ is RES_k . Otherwise, an exception condition is raised:

Conformance Rules

None.

17.7 Function references

Function

Specify a function reference.

Format

```

<function reference> ::==
  <function resolution expression>
  | <local function reference>

<function resolution expression> ::==
  FUNCTION <catalog function reference>

<catalog function reference> ::==
  <catalog function parent and name>
  | <external object reference>

<catalog function parent and name> ::==
  <function parent specification> <function name>
  | <curl path parameter>

<function parent specification> ::==
  [ <parent catalog object reference> ] [ <qualified object name> <period> ]

<local function reference> ::==
  <qualified function name>

<qualified function name> ::==
  [ <qualified object name> <period> ] <function name>

```

Syntax Rules

- 1) If the implicit or explicit <function parent specification> *FPS* does not immediately contain a <parent catalog object reference>, then a <parent catalog object reference> of ./ is implicit.

General Rules

- 1) If the <function reference> *FR* is specified, then the result of *FR* is the result of the immediately contained <function resolution expression> or <local function reference>.
- 2) If the <function resolution expression> *FRE* is specified, then the result of *FRE* is the result of the immediately contained <catalog function reference>.
- 3) If the <catalog function reference> *CFR* is specified, then:
 - a) Let *ROCFR* be determined as follows.

Case:

- i) If *CFR* simply contains the implicit or explicit <function parent specification> *FPS* and the <function name> *FN*, then *ROCFR* is the result of resolving *FN* with the sequence comprising only the result of *FPS* as the start object candidate sequence, as specified by [General Rule 7](#).
- ii) If *CFR* immediately contains an <external object reference> *EOR*, then the result of *ROCFR* is the result of *EOR*.

- b) If *ROCFR* is a function, then the result of *CFR* is *ROCFR*. Otherwise, an exception condition is raised:
- 4) If the implicit or explicit <function parent specification> *FPS* is specified, then let *ROPCOR* be the result of the <parent catalog object reference> immediately contained in *FPS* and determine the result of *FPS* as follows.

Case:

- a) If *FPS* immediately contains the <qualified object name> *QON*, then the result of *FPS* is result of alias-resolving *QON* with the sequence comprising only *ROPCOR* as the start object candidate sequence, as specified by [General Rule 3](#) of Subclause 17.9, “<qualified object name>”.
- b) Otherwise, the result of *FPS* is *ROPCOR*.
- 5) If the <local function reference> *LFR* is specified, then the result of *LFR* is the result of resolving the <qualified function name> immediately contained in *LFR* with the current working record with appended current working schema as the start object candidate sequence, as specified by [General Rule 6](#).
- 6) If the <qualified function name> *QFN* is specified, then *QFN* is resolved with the start object candidate sequence *SOCSEQ* as follows:
 - a) If *QFN* immediately contains the <qualified object name> *QON*, then *SOCSEQ'* is the sequence comprising the result of alias-resolving *QON* with *SOCSEQ* as the start object candidate sequence, as specified by [General Rule 3](#) of Subclause 17.9, “<qualified object name>”. Otherwise, *SOCSEQ'* is *SOCSEQ*.
 - b) Let *FN* be the <function name> immediately contained in *QFN* and let *RES* be the result of resolving *FN* with *SOCSEQ'* as the start object candidate sequence, as specified by [General Rule 7](#).
 - c) The result of resolving *QFN* with *SOCSEQ* as the candidate start object sequence is *RES*.
- 7) If the <function name> *FN* is specified, then *FN* is resolved with the start object candidate sequence *SOCSEQ* as follows:
 - a) Let *NSOCSEQ* be the number of elements of *SOCSEQ*, let *SOC_i* be the *i*-th element of *SOCSEQ*, and let *RES_i* be determined as follows, for $1 \leq i \leq NSOCSEQ$.
 - i) If *SOC_i* is a GQL-object with named components that contains a function *F* with name *FN*, then *RES_i* is *F*.
 - ii) If *SOC_i* is a GQL-object with named components that contains an alias with name *FN* that resolves to a function *F*, then *RES_i* is *F*.
 - iii) Otherwise, *RES_i* is the null value.
 - b) If there is $1 \leq k \leq NSOCSEQ$ such that *RES_k* is not the null value and such that for all $1 \leq j < k$ it holds that *RES_j* is the null value, then the result of resolving *FN* with the start object candidate sequence *SOCSEQ* is *RES_k*. Otherwise, an exception condition is raised:

Conformance Rules

None.

17.8 <catalog object reference>

Function

Define a <catalog object reference>.

Format

```

<catalog object reference> ::==
  <catalog url path>

<parent catalog object reference> ::=
  <catalog object reference> [ <solidus> ]

<catalog url path> ::=
  <absolute url path>
  | <relative url path>
  | <parameterized url path>

<absolute url path> ::=
  <solidus> [ <simple url path> ]

<relative url path> ::=
  <parent object relative url path>
  | <simple relative url path>
  | <period>

<parent object relative url path> ::=
  <predefined parent object parameter> [ <solidus> <simple url path> ]

<simple relative url path> ::=
  <double period> [ { <solidus> <double period> }... ] [ <solidus> <simple url path> ]
  | <simple url path>

<parameterized url path> ::=
  <url path parameter> [ <solidus> <simple url path> ]

<simple url path> ::=
  <url segment> [ { <solidus> <url segment> }... ]

<url segment> ::=
  <identifier>

```

Syntax Rules

- 1) If the <parent catalog object reference> *PCOR* is specified, then *PCOR* shall either only simply contain an <absolute url path> that is <solidus> or *PCOR* shall immediately contain the <solidus>.
- 2) If the <relative url path> *RUP* is specified and *RUP* is <period>, then *RUP* is effectively replaced by CURRENT_SCHEMA.

General Rules

- 1) If the <catalog object reference> *COR* is specified, then the result of *COR* is the result of the <catalog url path> immediately contained in *COR*.
- 2) If the <catalog url path> *CUP* is specified, then its result is determined as follows.

Case:

NOTE 135 — The <parameterized url path> is always effectively replaced by the Syntax Rules of Subclause 17.10, “<url path parameter>”.

- a) If CUP is the <absolute url path> AUP , then the result of CUP is the result of AUP .
 - b) If CUP is the <relative url path> RUP , then the result of CUP is the result of RUP .
- 3) If the <absolute url path> AUP is specified, then

Case:

- a) If AUP immediately contains the <simple url path> SUP , then the result of AUP is the result of resolving SUP with the GQL-catalog root as the resolution start object as specified by [General Rule 8](#).
 - b) Otherwise, the result of AUP is the GQL-catalog root.
- 4) If the <relative url path> RUP is specified, then:
- a) If RUP immediately contains the <parent object relative url path> $PORUP$, then the result of RUP is $PORUP$.
 - b) If RUP immediately contains the <simple relative url path> $SRUP$, then the result of RUP is $SRUP$.
- 5) If the <parent object relative url path> $PORUP$ is specified, then
- a) Let $ROPPOP$ be the result of the <predefined parent object parameter> immediately contained in $PORUP$.
 - b) If $PORUP$ immediately contains the <simple url path> SUP , then the result of $PORUP$ is the result of resolving SUP with $ROPPOP$ as the resolution start object, as specified by [General Rule 8](#). Otherwise, the result of $PORUP$ is $ROPPOP$.
- 6) If the <simple relative url path> $SRUP$ is specified, the result of $SRUP$ is the result of resolving $SRUP$ with the current working schema as the resolution start object, as specified by [General Rule 7](#).
- 7) The result of resolving the <simple relative url path> $SRUP$ with the resolution start object RSO is determined as follows:
- a) Let NDS be the number of <double period> immediately contained in $RUPT$.
 - b) If NDS is larger than the number of parent GQL-objects of RSO , then an exception condition is raised: *syntax error or access rule violation — invalid reference (42002)*.
 - c) If NDS is 0 (zero), then let RSO' be RSO . Otherwise, let RSO' be the NDS -th parent GQL-object of RSO .
 - d) Case:
 - i) If $SRUP$ immediately contains the <simple url path> SUP , then the result of $SRUP$ is the result of resolving SUP with RSO' as the resolution start object, as specified by [General Rule 7](#).
 - ii) Otherwise, the result of $SRUP$ is RSO' .
- 8) The result of resolving the <simple url path> SUP with the resolution start object RO_0 is determined as follows:
- a) Let $USSEQ$ be the sequence of all <url segment>s that are immediately contained in SUP in the order of their occurrence from left to right, let $NUSSEQ$ be the number of elements of $USSEQ$, and let US_i be the i -th element of $USSEQ$, for $1 \leq i \leq NUSSEQ$.

- b) Let the resolved object RO_i be determined as follows, for $1 \leq i \leq NUSSEQ$.

Case:

- i) If RO_{i-1} is a GQL-object with named components and RO_{i-1} contains a GQL-object X with name US_i , then RO_i is X .
 - ii) Otherwise, an exception condition is raised:
- c) The result of resolving SUP with the resolution start object RO_0 is RO_{NUSSEQ} .

Conformance Rules

None.

17.9 <qualified object name>

Function

Define a <qualified object name>.

Format

```
<qualified object name> ::=  
  <qualified name prefix> <object name>  
  
<qualified name prefix> ::=  
  [ { <object name> <period> }... ]
```

Syntax Rules

None.

General Rules

- 1) Let QON be the <qualified object name>.
- 2) QON is object-resolved with the start object candidate sequence $SOCSEQ$ as follows:

NOTE 136 — Object resolution of a <qualified object name> whose final component is an alias does not resolve that alias.

- a) Let $NSOCSEQ$ be the number of elements of $SOCSEQ$ and let SOC_i be the i -th element of $SOCSEQ$, for $1 \leq i \leq NSOCSEQ$.
- b) Let $ONSEQ$ be the sequence of all <object name>s that are simply contained in QON in the order of their occurrence from left to right, let $NONSEQ$ be the number of elements of $ONSEQ$, and let ON_j be the j -th element of $ONSEQ$, for $1 \leq j \leq NONSEQ$.
- c) Let the resolution start object be a GQL-object GO_0 that is determined as follows:
 - i) If there is a smallest integer k with $1 \leq k \leq NSOCSEQ$ such that SOC_k is a GQL-object with named components that contains a GQL-object with name ON_1 , then GO_0 is SOC_k .
 - ii) Otherwise, an exception condition is raised:
- d) Let the GQL-object GO_j be determined as follows, for $1 \leq j \leq NONSEQ$.

Case:

- i) If $j < NONSEQ$ and GO_{j-1} is a GQL-object with named components but not an alias and GO_{j-1} contains a GQL-object X with name ON_j , then GO_j is X .
- ii) If $j < NONSEQ$ and GO_{j-1} is an alias that successfully resolves to a GQL-object with named components that in turn contains a GQL-object X with name ON_j , then GO_j is X .
- iii) If $j = NONSEQ$ and GO_{j-1} is a GQL-object with named components and GO_{j-1} contains a GQL-object X with name ON_j , then ON_j is X .
- iv) Otherwise, an exception condition is raised:

- e) The result of resolving QON with $SOCSEQ$ as the start object candidate sequence is GO_{NONSEQ} .
- 3) QON is alias-resolved with the start object candidate sequence $SOSSEQ$ as follows:
 - NOTE 137 — Alias resolution of a <qualified object name> whose final component is an alias does resolve that alias.
 - a) Let $SIMPLE$ be the result of object-resolving QON with $SOCSEQ$ as the start object candidate sequence, as specified by [General Rule 2](#).
 - b) If $SIMPLE$ is an alias then, then RES is the result of resolving $SIMPLE$ to determine its alias target. Otherwise, RES is $SIMPLE$.
 - c) The result of object-resolving QON with $SOCSEQ$ as the start object candidate sequence is RES .

Conformance Rules

None.

17.10<url path parameter>

Function

Define a <url path parameter>.

Format

```
<url path parameter> ::=  
    <parameter>
```

Syntax Rules

- 1) Let UPP be the <url path parameter>.
- 2) Let P be the <parameter> immediately contained in UPP .
- 3) Let ROP be the result of P .
NOTE 138 — The result of P is determined from the GQL-request before regular execution and available during the evaluation of Syntax Rules.
- 4) Determine UPP' as follows.

Case:

- a) If ROP is a character string that conforms to the Format and Syntax Rules for either an <absolute url path> or a <relative url path>, then UPP' is ROP .
- b) If ROP is an array whose elements are either character strings or the null value, then
 - i) If ROP is an empty array, then UPP' is . (<period>).
 - ii) Otherwise:
 - 1) Let $NROP$ be the number of elements of ROP .
 - 2) For $1 \leq i \leq NROP$, let ROP_i be the i -th element of ROP .
 - 3) For $1 \leq i \leq NROP$,

Case:

- A) If ROP_i is the null value, then let ROP'_i be <double period>.
- B) Otherwise, let ROP'_i be a <delimited identifier> whose representative form is ROP_i .

NOTE 139 — The representative form of a <delimited identifier> is defined in [Syntax Rule 5](#)) of [Subclause 21.4](#), “<token> and <separator>”.

- 4) Let UPP' be the /-separated concatenation of all ROP'_i , for $1 \leq i \leq NROP$.
- c) Otherwise, an exception condition is raised: *data exception (22000)*
- 5) UPP is effectively replaced by UPP' .

General Rules

None.

Conformance Rules

None.

17.11 <external object reference>

Function

Define an <external object reference>.

Format

```
<external object reference> ::=  
  <external object url>  
  
<external object url> ::=  
  !! See the Syntax Rules.
```

Syntax Rules

- 1) Let *EOR* be the <external object reference>.
- 2) Let *EOU* be the <external object url> immediately contained in *EOR*
- 3) *EOU* shall either be an absolute-URL character string or an absolute-URL-with-fragment character string as specified by [URL](#) or it alternatively shall be a URI with a mandatory scheme as specified by [RFC 3986](#) and [RFC 3978](#).
- 4) *EOU* shall not conform to the Format for a <catalog url path>.

General Rules

- 1) The result of *EOR* is the result of resolving *EOU* as specified by the [General Rule 2](#)).
- 2) The result of resolving *EOU* is implementation-defined. Determining the result of resolving *EOU* shall either result in a GQL-object or raise an implementation-defined exception.

Conformance Rules

None.

17.12 <element reference>

Function

Reference a graph element or list of graph elements.

Format

```
<element reference> ::=  
  <element variable>
```

Syntax Rules

- 1) Let *EV* be the <element variable> simply contained in the <element reference> *ER*. Let *MS* be the innermost <match statement> that declares *EV* and that contains *ER*. Let *GP* be the <graph pattern> simply contained in *MS*. The *degree of reference* of *ER* is defined as follows.

Case:

- a) If *ER* is simply contained in the <graph pattern where clause> of *GT*, then the degree of reference of *ER* is the degree of exposure of *EV* by *GP*.
- b) If *ER* is contained in a <parenthesized path pattern where clause> *PPWC* simply contained in *GP*, then let *PPPE* be the <parenthesized path pattern expression> that simply contains *PPWC*.

NOTE 140 — This rule is to be applied after the syntactic transform that converts any <element pattern where clause> to a <parenthesized path pattern where clause>.

Case:

- i) If *EV* is declared by *PPPE*, then the degree of reference of *ER* is the degree of exposure of *EV* by *PPPE*.
- ii) Otherwise, let *PP* be the innermost <graph pattern> or <parenthesized path pattern expression> that contains *PPWC* and that declares *EV*. The degree of reference of *ER* is the degree of exposure of *EV* by *PP*. The degree of exposure of *ER* shall be singleton.

Access Rules

None.

General Rules

None.

NOTE 141 — Every <element reference> is evaluated in the General Rules of Subclause 22.5, “Applying bindings to evaluate an expression”.

Conformance Rules

None.

18 Functions

19 Predicates

19.1 <search condition>

Function

Specify a condition that is *True*, *False*, or *Unknown*, depending on the value of a <boolean value expression>.

Format

```
<search condition> ::=  
  <boolean value expression>
```

Syntax Rules

None.

General Rules

- 1) The result of the <search condition> is the result of the <boolean value expression>.
- 2) A <search condition> is said to be *satisfied* if and only if the result of the <boolean value expression> is *True*.

Conformance Rules

None.

19.2 <predicate>

Function

Specify a condition that can be evaluated to give a Boolean value.

Format

```
<predicate> ::=  
  <comparison predicate>  
  | <exists predicate>  
  | <null predicate>  
  | <normalized predicate>  
  | <directed predicate>  
  | <labeled predicate>  
  | <source/destination predicate>  
  | <all_different predicate>  
  | <same predicate>
```

Syntax Rules

None.

General Rules

- 1) The result of a <predicate> is the truth value of the immediately contained <comparison predicate>, <exists predicate>, <null predicate> <normalized predicate>, <directed predicate>, <labeled predicate>, <source/destination predicate>, <all_different predicate>, or <same predicate>.

Conformance Rules

None.

19.3 <comparison predicate>

Function

Specify a comparison of two values.

Format

```

<comparison predicate> ::==
  <non-parenthesized value expression primary> <comparison predicate part 2>

<comparison predicate part 2> ::==
  <comp op> <non-parenthesized value expression primary>

<comp op> ::==
  <equals operator>
  | <not equals operator>
  | <less than operator>
  | <greater than operator>
  | <less than or equals operator>
  | <greater than or equals operator>

```

Syntax Rules

- 1) Let L and R respectively denote the first and second <non-parenthesized value expression primary>s.
- 2) Case:
 - a) If <comp op> is <equals operator> or <not equals operator>, then L and R are operands of an equality operation. The Syntax Rules and Conformance Rules of Subclause 22.6, "Equality operations", apply.
 - b) Otherwise, L and R are operands of an ordering operation. The Syntax Rules and Conformance Rules of Subclause 22.7, "Ordering operations", apply.
- 3) Let CP be the <comparison predicate> " $L <comp op> R$ ". The following syntactic transformations are applied.

Case:

 - a) If the <comp op> is <not equals operator>, then CP is equivalent to:
$$\text{NOT } (L = R)$$
 - b) If the <comp op> is <greater than operator>, then CP is equivalent to:
$$(R < L)$$
 - c) If the <comp op> is <less than or equals operator>, then CP is equivalent to:
$$(L < R \text{ OR } R = R)$$
 - d) If the <comp op> is <greater than or equals operator>, then CP is equivalent to:
$$(R < L \text{ OR } R = R)$$

OR
 $R = L$)

General Rules

- 1) If the most specific types of the two <non-parenthesized value expression primary>s are not comparable, then an exception condition is raised: *data exception — values not comparable (22G04)*.
- 2) Let XV and YV be two values represented by <value expression>s L and R , respectively. The result of:

$X <\text{comp op}> Y$

is determined as follows.

Case:

- a) If at least one of XV and YV is the null value, then

$X <\text{comp op}> Y$

is *Unknown*.

- b) Otherwise, the result of

$X <\text{comp op}> Y$

is *True* or *False* as follows:

- i) $X = Y$

is *True* if and only if XV and YV are equal.

- ii) $X < Y$

is *True* if and only if XV is less than YV .

- iii) $X <\text{comp op}> Y$

is *False* if and only if

$X <\text{comp op}> Y$

is not *True*

- 3) Numbers are compared with respect to their algebraic value.

- 4) The comparison of two character strings is determined as follows:

- a) Both character strings are normalized using Normalization Form D (NFD) of [The Unicode® Standard](#).
- b) If the length in characters of X is not equal to the length in characters of Y , then the shorter character string is effectively replaced, for the purposes of comparison, with a copy of itself that has been extended to the length of the longer character string by concatenation on the right of one or more <space> characters.
- c) The result of the comparison of X and Y is determined by applying the Unicode Collation Algorithm with the Default Unicode Collation Element Table, as specified in [Unicode Technical Standard #10](#).

- 5) The comparison of two datetimes is determined according to the duration resulting from their subtraction.
- 6) The comparison of two durations is determined by the comparison of their corresponding values after conversion to a duration expressed in seconds. In the conversion a year is taken to be 365 days and a month 30 days.
- 7) In comparisons of Boolean values, *True* is greater than *False*

Conformance Rules

None.

NOTE 142 — If <comp op> is <equals operator> or <not equals operator>, then the Conformance Rules of Subclause 22.6, “Equality operations” apply. Otherwise, the Conformance Rules of Subclause 22.7, “Ordering operations” apply.

19.4 <exists predicate>

Function

Specify an existential subquery.

Format

```
<exists predicate> ::=  
  EXISTS {  
    <left paren> <graph pattern> <right paren>  
  | <nested query specification>  
 }
```

Syntax Rules

- 1) Let *EP* be the <exists predicate>.
- 2) If *EP* immediately contains the <graph pattern> *GP*, *EP* is effectively replaced by:

```
EXISTS { MATCH GP RETURN * }
```

General Rules

- 1) Let *NQS* be the <nested query specification> immediately contained in *EP*.
- 2) In a new child execution context:
 - a) The General Rules of Subclause 9.1, “<procedure specification>” for a <nested query specification> are applied to *NQS*.
 - b) Case:
 - i) If the current execution result is a non-empty graph, the result of *EP* is *True*.
 - ii) If the current execution result is an empty graph, the result of *EP* is *False*.
 - iii) If the current execution result is a non-empty table, the result of *EP* is *True*.
 - iv) If the current execution result is an empty table, the result of *EP* is *False*.
 - v) If the current execution result is a non-empty collection value, the result of *EP* is *True*.
 - vi) If the current execution result is an empty collection value, the result of *EP* is *False*.
 - vii) If the current execution result is a non-empty map or struct, the result of *EP* is *True*.
 - viii) If the current execution result is an empty map or struct, the result of *EP* is *False*.
 - ix) If the current execution result is *True*, the result of *EP* is *True*.
 - x) If the current execution result is *False*, the result of *EP* is *False*.
 - xi) If the current execution result is the null value, the result of *EP* is *False*.
 - xii) Otherwise, an exception condition is raised:

Conformance Rules

None.

19.5 <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::=  
  <value expression primary> <null predicate part 2>  
  
<null predicate part 2> ::=  
  IS [ NOT ] NULL
```

Syntax Rules

None.

General Rules

- 1) Let R be the <value expression primary> and let V be the value of R .
- 2) Case:
 - a) If V is the null value, then " R IS NULL" is True and the value of " R IS NOT NULL" is False.
 - b) Otherwise " R IS NULL" is False and the value of " R IS NOT NULL" is True.

Conformance Rules

None.

19.6 <normalized predicate>

Function

Determine whether a character string value is normalized.

Format

```
<normalized predicate> ::=  
  <string value expression> <normalized predicate part 2>  
  
<normalized predicate part 2> ::=  
  IS [ NOT ] [ <normal form> ] NORMALIZED
```

Syntax Rules

- 1) Let <string value expression> be *SVE*.
- 2) Case:
 - a) If <normal form> is specified, then let *NF* be <normal form>.
 - b) Otherwise, let *NF* be NFC.
- 3) The expression

SVE IS NOT NF NORMALIZED

is equivalent to

NOT (*SVE IS NF NORMALIZED*)

General Rules

- 1) The result of *SVE IS NF NORMALIZED* is

Case:

- a) If the value of *SVE* is the null value, then *Unknown*.
- b) If the value of *SVE* is in the normalization form specified by *NF*, as defined by [Unicode Standard Annex #15](#), then *True*.
- c) Otherwise, *False*.

Conformance Rules

None.

19.7 <directed predicate>

Function

Determine whether an edge variable is bound to a directed edge.

Format

```
<directed predicate> ::=  
  <element reference> <directed predicate part 2>  
  
<directed predicate part 2> ::=  
  IS [ NOT ] DIRECTED
```

Syntax Rules

- 1) Let ER be the <element reference>. ER shall have singleton degree of reference, and shall reference an edge variable.
- 2) If NOT is specified, then the <directed predicate> is equivalent to

NOT (ER IS DIRECTED)

Access Rules

None.

General Rules

- 1) Let LOE be the list of elements that are bound to ER .
- 2) The value of the <directed predicate>

ER IS DIRECTED

is determined as follows:

Case:

- a) If LOE is empty, then Unknown.
- b) If the sole graph element in LOE is a directed edge, then True.
- c) Otherwise, False.

Conformance Rules

- 1) Without Feature G101, “IS DIRECTED predicate”, conforming SQL language shall not contain an <directed predicate>.

19.8 <labeled predicate>

Function

Determine whether a graph element satisfies a <label expression>.

Format

```
<labeled predicate> ::=  
  <element reference> <labeled predicate part 2>  
  
<labeled predicate part 2> ::=  
  IS [ NOT ] LABELED <label expression>
```

Syntax Rules

- 1) Let *ER* be the <element reference>. *ER* shall have singleton degree of reference.
- 2) Let *LE* be the <label expression>.
- 3) If NOT is specified, then the <labeled predicate> is equivalent to

NOT (*ER* IS LABELED *LE*)

Access Rules

None.

General Rules

- 1) Let *LOE* be the list of elements that are bound to *ER*.
- 2) The value of the <labeled predicate> *LP*

ER IS LABELED *LE*

is determined as follows:

Case:

- a) If *LOE* is empty, then the value of *LP* is *Unknown*.
- b) Otherwise, let *E* be the sole graph element in *LOE*. Let *LS* be the label set of *E*. The Syntax Rules of Subclause 22.1, “Satisfaction of a <label expression> by a label set”, are applied with *LE* as *LABEL EXPRESSION* and *LS* as *LABEL SET*; let *TV* be the *TRUTH VALUE* returned from the application of those Syntax Rules. The value of *LP* is *TV*.

Conformance Rules

- 1) Without Feature G102, “IS LABELED predicate”, conforming GQL language shall not contain an <labeled predicate>.

19.9 <source/destination predicate>

Function

Determine whether a node is the source or destination of an edge.

Format

```

<source/destination predicate> ::= 
  <node reference> <source predicate part 2>
  | <node reference> <destination predicate part 2>

<node reference> ::= 
  <element reference>

<source predicate part 2> ::= 
  IS [ NOT ] SOURCE [ OF ] <edge reference>

<destination predicate part 2> ::= 
  IS [ NOT ] DESTINATION [ OF ] <edge reference>

<edge reference> ::= 
  <element reference>

```

Syntax Rules

- 1) Let *NR* be the <node reference>. *NR* shall have singleton degree of reference, and shall reference a node variable.
- 2) Let *ER* be the <edge reference>. *ER* shall have singleton degree of reference, and shall reference an edge variable.
- 3) Let *SOD* be the <key word> SOURCE or DESTINATION simply contained in the <source/destination predicate>.
- 4) If NOT is specified, then the <source/destination predicate> is equivalent to

NOT (VR IS SOD OF ER)

Access Rules

None.

General Rules

- 1) Let *LOV* be the list of elements that are bound to *VR* and let *LOE* be the list of elements that are bound to *ER*.
- 2) The value of the <source/destination predicate> *SDP*

VR IS SOD OF ER

is determined as follows:

Case:

19.9 <source/destination predicate>

- a) If LOV is empty or if LOE is empty, then the value of SDP is Unknown.
- b) Otherwise, let V be the sole graph element in LOV , and let E be the sole graph element in LOE .

Case:

- i) If E is an undirected edge, then the value of SDP is False.
- ii) If SOD is SOURCE and V is the source node of E , then the value of SDP is True.
- iii) If SOD is DESTINATION and V is the destination node of E , then the value of SDP is True.
- iv) Otherwise the value of SDP is False.

Conformance Rules

- 1) Without Feature G103, “IS SOURCE and IS DESTINATION predicate”, conforming GQL language shall not contain a <source/destination predicate>.

19.10 <all_different predicate>

Function

Determine whether all graph elements bound to a list of element references are pairwise different from one another.

Format

```
<all_different predicate> ::=  
  ALL_DIFFERENT <left paren> <element reference> <comma> <element reference>  
    [ { <comma> <element reference> }... ]  
  <right paren>
```

Syntax Rules

- 1) Each <element reference> simply contained in <all_different predicate> *ADP* shall have unconditional singleton degree of reference.

Access Rules

None.

General Rules

- 1) Let N be the number of <element reference>s simply contained in *ADP*, and let ER_1, \dots, ER_N be an enumeration of those <element reference>s.
- 2) For all i , $1 \leq i \leq N$, let LOE_i be the list of graph elements of ER_i , and let GE_i be the sole graph element in LOE_i .
- 3) The value of *ADP* is

Case:

- a) If there exist j, k such that $j < k$ and GE_j is the same graph element as GE_k , then *False*.
- b) Otherwise, *True*.

Conformance Rules

- 1) Without Feature G104, “ALL_DIFFERENT predicate”, conforming GQL language shall not contain an <all_different predicate>.

19.11 <same predicate>

Function

Determine whether all element references in a list of element references bind to the same graph element.

Format

```
<same predicate> ::=  
  SAME <left paren> <element reference> <comma> <element reference>  
    [ { <comma> <element reference> }... ]  
<right paren>
```

Syntax Rules

- 1) Each <element reference> simply contained in <same predicate> SP shall have unconditional singleton degree of reference.

Access Rules

None.

General Rules

- 1) Let N be the number of <element reference>s simply contained in SP , and let ER_1, \dots, ER_N be an enumeration of those <element reference>s.
- 2) For all i , $1 \leq i \leq N$, let LOE_i be the list of graph elements of ER_i , and let GE_i be the sole graph element in LOE_i .
- 3) The value of SP is

Case:

- a) If every GE_i is the same graph element, then True.
- b) Otherwise, False.

Conformance Rules

- 1) Without Feature G105, “SAME predicate”, conforming GQL language shall not contain a <same predicate>.

20 Value expressions

20.1 <value specification>

Function

Specify one or more values or parameters.

Format

```

<value specification> ::==
  <literal>
  | <parameter value specification>

<unsigned value specification> ::==
  <unsigned literal>
  | <parameter value specification>

<unsigned integer specification> ::==
  <unsigned integer>
  | <parameter>

<parameter value specification> ::==
  <parameter>
  | <predefined parameter>

<predefined parameter> ::==
  <predefined parent object parameter>
  | <predefined table parameter>
  | CURRENT_USER

<predefined parent object parameter> ::==
  <predefined schema parameter>
  | <predefined graph parameter>

<predefined schema parameter> ::==
  HOME_SCHEMA
  | CURRENT_SCHEMA

<predefined graph parameter> ::==
  EMPTY_PROPERTY_GRAPH
  | EMPTY_GRAPH
  | HOME_PROPERTY_GRAPH
  | HOME_GRAPH
  | CURRENT_PROPERTY_GRAPH
  | CURRENT_GRAPH

<predefined table parameter> ::==
  EMPTY_BINDING_TABLE
  | EMPTY_TABLE
  | UNIT_BINDING_TABLE
  | UNIT_TABLE

```

Syntax Rules

- 1) The declared type of CURRENT_USER is character string.
- 2) The declared type of CURRENT_SCHEMA and HOME_SCHEMA is the schema type.
- 3) The declared type of EMPTY_GRAPH, EMPTY_PROPERTY_GRAPH, CURRENT_GRAPH, CURRENT_PROPERTY_GRAPH, HOME_GRAPH, and HOME_PROPERTY_GRAPH is a specific graph type to which the specified graph conforms. This graph type shall be a graph type supported by the implementation and is either the graph type specified explicitly by the user when creating the graph or in the absence of having been specified by the user inferred in an implementation-defined way.
- 4) The declared type of EMPTY_TABLE, EMPTY_BINDING_TABLE, UNIT_TABLE, and UNIT_BINDING_TABLE is the most specific binding type to which the specified table conforms. This binding table type shall be a binding table type supported by the implementation and is inferred in an implementation-defined way.
- 5) The declared type of <unsigned integer specification> shall be an implementation-defined integer type.

General Rules

- 1) A <value specification> or <unsigned value specification> specifies a value that is not selected from a graph.
- 2) A <parameter name> identifies a session parameter or a request parameter.
- 3) The value specified by a <literal> is the value represented by that <literal>.
- 4) The object specified by EMPTY_BINDING_TABLE and EMPTY_TABLE is a table with no rows and no columns.
- 5) The object specified by EMPTY_GRAPH and EMPTY_PROPERTY_GRAPH is a graph that has no nodes, edges, labels, and properties.
- 6) The value specified by CURRENT_USER is

Case:

 - a) If there is a current authorization identifier, then the value of that current authorization identifier.
 - b) Otherwise, the null value.
- 7) The value specified by HOME_SCHEMA is the current home schema.
- 8) The value specified by HOME_PROPERTY_GRAPH or by HOME_GRAPH is the current home graph.
- 9) The value specified by CURRENT_SCHEMA is the current working schema.
- 10) The value specified by CURRENT_PROPERTY_GRAPH or by CURRENT_GRAPH is the current working graph.
- 11) The object specified by UNIT_BINDING_TABLE and UNIT_TABLE is a table with exactly one record and no columns.
- 12) If a <value specification> evaluates to the null value, then an exception condition is raised: *data exception — null value not allowed (22004)*.
- 13) If the most specific type of <unsigned integer specification> is not an exact numeric with scale 0 (zero) then an exception condition is raised: *data exception — invalid value type (22G03)*.

14) Let V be the result of <unsigned integer specification>.

Case:

- a) If V is the null value, then an exception condition is raised: *data exception — null value not allowed* (22004).
- b) If V is negative, then an exception condition is raised: *data exception — negative limit value* (22G02).

Conformance Rules

None.

20.2 <value expression>

Function

Specify a constant or a value.

Format

```

<value expression> ::= 
  <untyped value expression> [ <of value type> ]

<untyped value expression> ::= 
  <common value expression>
  | <boolean value expression>

<common value expression> ::= 
  <numeric value expression>
  | <string value expression>
  | <datetime value expression>
  | <duration value expression>
  | <collection value expression>
  | <map value expression>
  | <record value expression>
  | <reference value expression>

<reference value expression> ::= 
  <primary result object expression>
  | <graph element value expression>

<collection value expression> ::= 
  <list value expression>
  | <multiset value expression>
  | <set value expression>
  | <ordered set value expression>

<set value expression> ::= 
  <value expression primary>

<ordered set value expression> ::= 
  <value expression primary>

<map value expression> ::= 
  <value expression primary>

<record value expression> ::= 
  <value expression primary>

```

Syntax Rules

None.

General Rules

- 1) The declared type of a <value expression> is the value type specified by the immediately contained <value type> if present, or otherwise the declared type of the immediately contained <untyped value expression>.

- 2) The declared type of an <untyped value expression> is the declared type of the immediately contained <common value expression> or <boolean value expression>.
- 3) The declared type of a <common value expression> is the declared type of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <duration value expression>, <reference value expression>, <collection value expression>, <map value expression>, or <record value expression> respectively.
- 4) The declared type of a <reference value expression> is the declared type of the immediately contained <primary result object expression> or <graph element value expression> respectively.
- 5) The declared type of a <collection value expression> is the declared type of the immediately contained <list value expression>, <multiset value expression>, <set value expression>, or <ordered set value expression>.
- 6) The declared type of a <set value expression> is the declared type of the immediately contained <value expression primary>.
- 7) If the most specific type of a <value expression primary> immediately contained in a <set value expression> is not a set value type, then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 8) The declared type of an <ordered set value expression> is the declared type of the immediately contained <value expression primary>.
- 9) If the most specific type of a <value expression primary> immediately contained in an <ordered set value expression> is not an ordered set value type, then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 10) The declared type of a <map value expression> is the declared type of the immediately contained <value expression primary>.
- 11) If the most specific type of a <value expression primary> immediately contained in a <map value expression> is not a map type, then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 12) The declared type of a <record value expression> is the declared type of the immediately contained <value expression primary>.
- 13) If the most specific type of a <value expression primary> immediately contained in a <record value expression> is not a record type, then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 14) The value of a <value expression> is the value of the immediately contained <untyped value expression>.
- 15) The value of an <untyped value expression> is the value of the immediately contained <common value expression>, <boolean value expression>.
- 16) The value of a <common value expression> is the value of the immediately contained <numeric value expression>, <string value expression>, <datetime value expression>, <duration value expression>, <graph element value expression>, or <collection value expression> respectively.

Conformance Rules

None.

20.3 <boolean value expression>

Function

Specify a Boolean value.

Format

```

<boolean value expression> ::=

  <boolean term>
  | <boolean value expression> OR <boolean term>
  | <boolean value expression> XOR <boolean term>

<boolean term> ::=

  <boolean factor>
  | <boolean term> AND <boolean factor>

<boolean factor> ::=

  [ NOT ] <boolean test>

<boolean test> ::=

  <boolean primary> [ {
    IS [ NOT ]
  | <equals operator>
  | <not equals operator>
  } <truth value> ]

<truth value> ::=

  TRUE
  | FALSE
  | UNKNOWN
  | NULL

<boolean primary> ::=

  <predicate>
  | <boolean predicand>

<boolean predicand> ::=

  <parenthesized Boolean value expression>
  | <non-parenthesized value expression primary>

<parenthesized Boolean value expression> ::=

  <left paren> <boolean value expression> <right paren>

```

Syntax Rules

- 1) $X \text{ XOR } Y$ is equivalent to the following <boolean value expression>:

$(X \text{ OR } Y) \text{ AND NOT } (X \text{ AND } Y)$

- 2) In <truth value> the <key word> NULL is equivalent to UNKNOWN.
- 3) If NOT is specified in a <boolean test>, then let BP be the contained <boolean primary> and let TV be the contained <truth value>. The <boolean test> is equivalent to:

$(\text{NOT} (BP \text{ IS } TV))$

General Rules

- 1) If the most specific type of a <non-parenthesized value expression primary> is not of Boolean type then an exception condition is raised: *data exception — invalid value type* (22G03).
- 2) The result is derived by the application of the specified Boolean operators (“AND”, “OR”, “NOT”, and “IS”) to the results derived from each <boolean primary>. If Boolean operators are not specified, then the result of the <boolean value expression> is the result of the specified <boolean primary>.
- 3) NOT (*True*) is *False*, NOT (*False*) is *True*, and NOT (*Unknown*) is *Unknown*.
- 4) Table 3, “Truth table for the AND Boolean operator”, Table 4, “Truth table for the OR Boolean operator”, and Table 5, “Truth table for the IS Boolean operator” specify the semantics of AND, OR, and IS, respectively.

Table 3 — Truth table for the AND Boolean operator

AND	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>True</i>	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>False</i>	<i>False</i>	<i>False</i>	<i>False</i>
<i>Unknown</i>	<i>Unknown</i>	<i>False</i>	<i>Unknown</i>

Table 4 — Truth table for the OR Boolean operator

OR	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>True</i>	<i>True</i>	<i>True</i>	<i>True</i>
<i>False</i>	<i>True</i>	<i>False</i>	<i>Unknown</i>
<i>Unknown</i>	<i>True</i>	<i>Unknown</i>	<i>Unknown</i>

Table 5 — Truth table for the IS Boolean operator

IS	TRUE	FALSE	UNKNOWN
<i>True</i>	<i>True</i>	<i>False</i>	<i>False</i>
<i>False</i>	<i>False</i>	<i>True</i>	<i>False</i>
<i>Unknown</i>	<i>False</i>	<i>False</i>	<i>True</i>

Conformance Rules

None.

20.4 <numeric value expression>

Function

Specify a numeric value.

Format

```

<numeric value expression> ::==
  <term>
  | <numeric value expression> <plus sign> <term>
  | <numeric value expression> <minus sign> <term>

<term> ::==
  <factor>
  | <term> <asterisk> <factor>
  | <term> <solidus> <factor>

<factor> ::==
  [ <sign> ] <numeric primary>

<numeric primary> ::=
  <value expression primary>
  | <numeric value function>

```

Syntax Rules

- 1) If a <numeric value expression> immediately contains a <minus sign> *NMS* and immediately contains a <term> that is a <factor> that immediately contains a <sign> that is a <minus sign> *FMS*, then there shall be a <separator> between *NMS* and *FMS*.

General Rules

- 1) The declared type of a <factor> is that of the immediately contained <numeric primary>.
- 2) If the most specific type of a <numeric primary> is not numeric then an exception is raised: *data exception — invalid value type (22G03)*.
- 3) Case:
 - a) If the most specific type of either operand of a dyadic arithmetic operator is approximate numeric, then the declared type of the result is an implementation-defined approximate numeric type.
 - b) Otherwise, the declared type of both operands of a dyadic arithmetic operator is exact numeric and the declared type of the result is an implementation-defined exact numeric type, with precision and scale determined as follows:
 - i) Let $S1$ and $S2$ be the scale of the first and second operands respectively.
 - ii) The precision of the result of addition and subtraction is implementation-defined, and the scale is the maximum of $S1$ and $S2$.
 - iii) The precision of the result of multiplication is implementation-defined, and the scale is $S1 + S2$.
 - iv) The precision and scale of the result of division are implementation-defined.

- 4) If the value of any <numeric primary> simply contained in a <numeric value expression> is the null value, then the result of the <numeric value expression> is the null value.
- 5) If the <numeric value expression> contains only a <numeric primary>, then the result of the <numeric value expression> is the value of the specified <numeric primary>.
- 6) The monadic arithmetic operators <plus sign> and <minus sign> (+ and -, respectively) specify monadic plus and monadic minus, respectively. Monadic plus does not change its operand. Monadic minus reverses the sign of its operand.
- 7) The dyadic arithmetic operators <plus sign>, <minus sign>, <asterisk>, and <solidus> (+, -, *, and /, respectively) specify addition, subtraction, multiplication, and division, respectively. If the value of a divisor is zero, then an exception condition is raised: *data exception — division by zero* (22012).
- 8) If the most specific type of the result of an arithmetic operation is exact numeric, then

Case:

- a) If the operator is not division and the mathematical result of the operation is not exactly representable with the precision and scale of the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range* (22003).
 - b) If the operator is division and the approximate mathematical result of the operation represented with the precision and scale of the declared type of the result loses one or more leading significant digits after rounding or truncating if necessary, then an exception condition is raised: *data exception — numeric value out of range* (22003). The choice of whether to round or truncate is implementation-defined.
- 9) If the most specific type of the result of an arithmetic operation is approximate numeric and the exponent of the approximate mathematical result of the operation is not within the implementation-defined exponent range for the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range* (22003).

Conformance Rules

None.

20.5 <value expression primary>

Function

Specify a value that is syntactically self-delimited.

Format

```

<value expression primary> ::= 
  <parenthesized value expression>
  | <non-parenthesized value expression primary>

<parenthesized value expression> ::= 
  <left paren> <value expression> <right paren>

<non-parenthesized value expression primary> ::= 
  <property reference>
  | <binding variable>
  | <parameter value specification>
  | <unsigned value specification>
  | <aggregate function>
  | <collection value constructor>
  | <value query expression>
  | <case expression>
  | <cast specification>
  | <element_id function>

```

Syntax Rules

None.

General Rules

- 1) The declared type of a <value expression primary> is the declared type of the simply contained <value expression>, <unsigned value specification>, <collection value constructor>, <value query expression>, <property reference>, <element_id function>, <parameter value specification>, <case expression>, <cast specification>, or <binding variable>.
- 2) The value of a <value expression primary> is the value of the simply contained <value expression>, <unsigned value specification>, <collection value constructor>, <value query expression>, <property reference>, <element_id function>, <parameter value specification>, <case expression>, <cast specification>, or <binding variable>.

Conformance Rules

None.

20.6 <numeric value function>

Function

Specify a function yielding a value of type numeric.

Format

```

<numeric value function> ::=

    <length expression>
  | <absolute value expression>
  | <modulus expression>
  | <trigonometric function>
  | <general logarithm function>
  | <common logarithm>
  | <natural logarithm>
  | <exponential function>
  | <power function>
  | <square root>
  | <floor function>
  | <ceiling function>
  | <inDegree function>
  | <outDegree function>

<length expression> ::=
    <char length expression>
  | <byte length expression>
  | <path length expression>

<char length expression> ::=
  CHARACTER_LENGTH <left paren> <character string value expression> <right paren>

<byte length expression> ::=
{
  BYTE_LENGTH
  | OCTET_LENGTH
} <left paren> <string value expression> <right paren>

<path length expression> ::=
  LENGTH <left paren> <binding variable> <right paren>

<absolute value expression> ::=
  ABS <left paren> <numeric value expression> <right paren>

<modulus expression> ::=
  MOD <left paren> <numeric value expression dividend> <comma>
    <numeric value expression divisor> <right paren>

<numeric value expression dividend> ::=
  <numeric value expression>

<numeric value expression divisor> ::=
  <numeric value expression>

<trigonometric function> ::=
  <trigonometric function name> <left paren> <numeric value expression> <right paren>

<trigonometric function name> ::=
  SIN | COS | TAN | COT | SINH | COSH | TANH | ASIN | ACOS | ATAN | DEGREES | RADIANS

<general logarithm function> ::=

```

20.6 <numeric value function>

```

LOG <left paren> <general logarithm base> <comma>
      <general logarithm argument> <right paren>

<general logarithm base> ::= 
    <numeric value expression>

<general logarithm argument> ::= 
    <numeric value expression>

<common logarithm> ::= 
    LOG10 <left paren> <numeric value expression> <right paren>

<natural logarithm> ::= 
    LN <left paren> <numeric value expression> <right paren>

<exponential function> ::= 
    EXP <left paren> <numeric value expression> <right paren>

<power function> ::= 
    POWER <left paren> <numeric value expression base> <comma>
        <numeric value expression exponent> <right paren>

<numeric value expression base> ::= 
    <numeric value expression>

<numeric value expression exponent> ::= 
    <numeric value expression>

<square root> ::= 
    SQRT <left paren> <numeric value expression> <right paren>

<floor function> ::= 
    FLOOR <left paren> <numeric value expression> <right paren>

<ceiling function> ::= 
    { CEIL | CEILING } <left paren> <numeric value expression> <right paren>

<inDegree function> ::= 
    inDegree <left paren> <binding variable> <right paren>

<outDegree function> ::= 
    outDegree <left paren> <binding variable> <right paren>

```

Syntax Rules

- 1) If <common logarithm> is specified, then let *NVE* be the simply contained <numeric value expression>. The <common logarithm> is equivalent to

LOG (10, *NVE*)

- 2) If <square root> is specified, then let *NVE* be the simply contained <numeric value expression>. The <square root> is equivalent to

POWER (*NVE*, 0.5)

General Rules

- 1) If a <length expression> is specified, then the declared type of the result is an implementation-defined exact numeric type with scale 0 (zero).

- 2) If <path length expression> is specified and the immediately contained <binding variable> does not identify a path then an exception is raised: *data exception — invalid value type (22G03)*.
- 3) If <absolute value expression> is specified, then the declared type of the result is the declared type of the immediately contained <numeric value expression>.
- 4) If <trigonometric function> is specified, then the declared type of the result is an implementation-defined approximate numeric type.
- 5) The declared type of the result of <general logarithm function> is an implementation-defined approximate numeric type.
- 6) The declared type of the result of <natural logarithm> is an implementation-defined approximate numeric type.
- 7) The declared type of the result of <exponential function> is an implementation-defined approximate numeric type.
- 8) The declared type of the result of <power function> is an implementation-defined approximate numeric type.
- 9) If <floor function> or <ceiling function> is specified, then

Case:

- a) If the declared type of the immediately contained <numeric value expression> NVE is exact numeric, then the declared type of the result is exact numeric with implementation-defined precision, with the radix of NVE , and with scale 0 (zero).
 - b) Otherwise, an approximate numeric with implementation-defined precision.
- 10) If <absolute value expression> is specified, then let N be the value of the immediately contained <numeric value expression>.
- Case:
- a) If N is the null value, then the result is the null value.
 - b) If $N \geq 0$, then the result is N .
 - c) Otherwise, the result is $-1 * N$. If $-1 * N$ is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- 11) If <modulus expression> is specified, then let N be the value of the immediately contained <numeric value expression dividend> and let M be the value of the immediately contained <numeric value expression divisor>.
- Case:
- a) If at least one of N and M is the null value, then the result is the null value.
 - b) If M is zero, then an exception condition is raised: *data exception — division by zero (22012)*.
 - c) Otherwise, the result is the unique exact numeric value R with scale 0 (zero) such that all of the following are true:
 - i) R has the same sign as N .
 - ii) The absolute value of R is less than the absolute value of M .
 - iii) $N = M * K + R$ for some exact numeric value K with scale 0 (zero).

- 12) If <trigonometric function> is specified, then let V be the value of the immediately contained <numeric value expression> that represents an angle expressed in radians.

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise, let OP be the <trigonometric function name>.

Case:

- i) If OP is ACOS, then

Case:

- 1) If V is less than -1 (negative one) or V is greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- 2) Otherwise, the result of the <trigonometric function> is the inverse cosine of V .

- ii) If OP is ASIN, then

Case:

- 1) If V is less than -1 (negative one) or V is greater than 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- 2) Otherwise, the result of the <trigonometric function> is the inverse sine of V .

- iii) If OP is ATAN, then the result is the inverse tangent of V .

- iv) If OP is COS, then the result is the cosine of V .

- v) If OP is COSH, then the result is the hyperbolic cosine of V .

- vi) If OP is SIN, then the result is the sine of V .

- vii) If OP is SINH, then the result is the hyperbolic sine of V .

- viii) If OP is TAN, then the result is the tangent of V .

- ix) If OP is TANH, then the result is the hyperbolic tangent of V .

- x) If OP is COT, then the result is the cotangent of V .

- xi) If OP is DEGREES, then the result is the value of V , taken to be in radians, expressed as degrees.

- xii) If OP is RADIANS, then the result is the value of V , taken to be in degrees, expressed as radians.

- 13) If <general logarithm function> is specified, then let VB be the value of the <general logarithm base> and let VA be the value of the <general logarithm argument>.

Case:

- a) If at least one of VA and VB is the null value, then the result is the null value.
- b) If VA is negative or 0 (zero), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- c) If VB is negative, 0 (zero), or 1 (one), then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- d) Otherwise, the result is the logarithm with base VB of VA .

14) If <natural logarithm> is specified, then let V be the value of the immediately contained <numeric value expression>.

Case:

- a) If V is the null value, then the result is the null value.
- b) If V is 0 (zero) or negative, then an exception condition is raised: *data exception — invalid argument for natural logarithm (2201E)*.
- c) Otherwise, the result is the natural logarithm of V .

15) If <exponential function> is specified, then let V be the value of the immediately contained <numeric value expression>.

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise, the result is e (the base of natural logarithms) raised to the power V . If the result is not representable in the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

16) If <power function> is specified, then let $NVEB$ be the <numeric value expression base>, then let VB be the value of $NVEB$, let $NVEE$ be the <numeric value expression exponent>, and let VE be the value of $NVEE$.

Case:

- a) If at least one of VB and VE is the null value, then the result is the null value.
- b) If VB is 0 (zero) and VE is negative, then an exception condition is raised: *data exception — invalid argument for power function (2201F)*.
- c) If VB is 0 (zero) and VE is 0 (zero), then the result is 1 (one).
- d) If VB is 0 (zero) and VE is positive, then the result is 0 (zero).
- e) If VB is negative and VE is not equal to an exact numeric value with scale 0 (zero), then an exception condition is raised: *data exception — invalid argument for power function (2201F)*.
- f) If VB is negative and VE is equal to an exact numeric value with scale 0 (zero) that is an even number, then the result is the result of

$\text{EXP}(NVEE * \text{LN}(-NVEB))$

- g) If VB is negative and VE is equal to an exact numeric value with scale 0 (zero) that is an odd number, then the result is the result of

$-\text{EXP}(NVEE * \text{LN}(-NVEB))$

- h) Otherwise, the result is the result of

$\text{EXP}(NVEE * \text{LN}(NVEB))$

17) If <floor function> is specified, then let V be the value of the immediately contained <numeric value expression> NVE .

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise,

Case:

- i) If the most specific type of NVE is exact numeric, then the result is the greatest exact numeric value with scale 0 (zero) that is less than or equal to V . If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- ii) Otherwise, the result is the greatest whole number that is less than or equal to V . If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

18) If <ceiling function> is specified, then let V be the value of the immediately contained <numeric value expression> NVE .

Case:

- a) If V is the null value, then the result is the null value.
- b) Otherwise,

Case:

- i) If the most specific type of NVE is exact numeric, then the result is the least exact numeric value with scale 0 (zero) that is greater than or equal to V . If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.
- ii) Otherwise, the result is the least whole number that is greater than or equal to V . If this result is not representable by the declared type of the result, then an exception condition is raised: *data exception — numeric value out of range (22003)*.

Conformance Rules

None.

20.7 <string value expression>

Function

Specify a character string value or a byte string value.

Format

```
<string value expression> ::=  
    <character string value expression>  
  | <byte string value expression>  
  
<character string value expression> ::=  
    <character string concatenation>  
  | <character string factor>  
  
<character string concatenation> ::=  
    <character string value expression> <concatenation operator> <character string factor>  
  
<character string factor> ::=  
    <character string primary>  
  
<character string primary> ::=  
    <value expression primary>  
  | <string value function>  
  
<byte string value expression> ::=  
    <byte string concatenation>  
  | <byte string factor>  
  
<byte string factor> ::=  
    <byte string primary>  
  
<byte string primary> ::=  
    <value expression primary>  
  | <string value function>  
  
<byte string concatenation> ::=  
    <byte string value expression> <concatenation operator> <byte string factor>
```

Syntax Rules

None.

General Rules

- 1) The declared type of a <string value expression> is the declared type of the immediately contained <character string value expression> or <byte string value expression>.
- 2) The declared type of a <character string value expression> is the declared type of the immediately contained <character string concatenation> or <character string factor>.
- 3) The declared type of a <character string concatenation> is character string.
- 4) The declared type of a <character string factor> is the declared type of the immediately contained <character string primary>.

20.7 <string value expression>

- 5) If the most specific type of a <character string primary> is not character string then an exception is raised: *data exception — invalid value type (22G03)*.
- 6) The declared type of a <byte string value expression> is the declared type of the immediately contained <byte string concatenation> or <byte string factor>.
- 7) The declared type of a <byte string concatenation> is a byte string type.
- 8) The declared type of a <byte string factor> is the declared type of the immediately contained <byte string primary>.
- 9) If the most specific type of a <byte string primary> is not a byte string type then an exception is raised: *data exception — invalid value type (22G03)*.
- 10) If the value of any <character string primary> simply contained in a <string value expression> is the null value, then the result of the <string value expression> is the null value.
- 11) If <character string concatenation> is specified, then:
 - a) In the remainder of this General Rule, the term “length” is taken to mean “length in characters”.
 - b) Let S_1 and S_2 be the result of the <character string value expression> and <character string factor>, respectively.

Case:

 - i) If at least one of S_1 and S_2 is the null value, then the result of the <character string concatenation> is the null value.
 - ii) Otherwise:
 - 1) Let S be the character string comprising S_1 followed by S_2 and let M be the length of S .
 - 2) S is replaced by

Case:

 - A) If the <search condition> $S_1 \text{ IS NORMALIZED AND } S_2 \text{ IS NORMALIZED}$ evaluates to *True*, then

NORMALIZE (S)

 - B) Otherwise, an implementation-defined character string.
 - 3) Case:
 - A) Let VL be the implementation-defined maximum length of character strings.

Case:

 - I) If M is less than or equal to VL , then the result of the <character string concatenation> is S with length M .
 - II) If M is greater than VL and the right-most $M-VL$ characters of S are all the <whitespace> characters, then the result of the <character string concatenation> is the first VL characters of S with length VL .
 - III) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.
- 12) If <byte string concatenation> is specified, then let S_1 and S_2 be the result of the <byte string value expression> and <byte string factor>, respectively.

Case:

- a) If at least one of S_1 and S_2 is the null value, then the result of the <byte string concatenation> is the null value.
- b) Otherwise, let S be the byte string comprising S_1 followed by S_2 and let M be the length in bytes of S .

Case:

- i) Let VL be the implementation-defined maximum length of byte strings.

Case:

- 1) If M is less than or equal to VL , then the result of the <byte string concatenation> is S with length M .
- 2) If M is greater than VL and the right-most $M-VL$ bytes of S are all X'00', then the result of the <byte string concatenation> is the first VL bytes of S with length VL .
- 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

Conformance Rules

None.

20.8 <string value function>

Function

Specify a function yielding a character string value or a byte string value.

Format

```

<string value function> ::=

  <character string function>
  | <byte string function>

<character string function> ::=

  <substring function>
  | <fold>
  | <trim function>
  | <normalize function>

<substring function> ::=
  SUBSTRING <left paren> <character string value expression> <comma> <start position>
    [ <comma> <string length> ] <right paren>
  | LEFT <left paren> <character string value expression> <comma> <string length> <right
    paren>
  | RIGHT <left paren> <character string value expression> <comma> <string length> <right
    paren>

<fold> ::=
  { UPPER | toUpper | LOWER | toLower } <left paren> <character string value expression>
  <right paren>

<trim function> ::=
  TRIM <left paren> <trim source> [ <comma> <trim specification> [ <trim character string>
    ] ] <right paren>
  | lTrim <left paren> <trim source> <right paren>
  | rTrim <left paren> <trim source> <right paren>

<trim source> ::=
  <character string value expression>

<trim specification> ::=
  LEADING
  | TRAILING
  | BOTH

<trim character string> ::=
  <character string value expression>

<normalize function> ::=
  NORMALIZE <left paren> <character string value expression>
    [ <comma> <normal form> ] <right paren>

<normal form> ::=
  NFC
  | NFD
  | NFKC
  | NFKD

<byte string function> ::=
  <byte substring function>
  | <byte string trim function>

```

```

<byte substring function> ::= 
    SUBSTRING <left paren> <byte string value expression> <comma> <start position>
        [ <comma> <string length> ] <right paren>
    | LEFT <left paren> <byte string value expression> <comma> <string length> <right paren>
    | RIGHT <left paren> <byte string value expression> <comma> <string length> <right paren>

<byte string trim function> ::= 
    TRIM <left paren> <byte string trim source> [ <comma> <trim specification> [ <trim byte
        string> ] ] <right paren>
    | lTrim <left paren> <byte string trim source> <right paren>
    | rTrim <left paren> <byte string trim source> <right paren>

<byte string trim source> ::= 
    <byte string value expression>

<trim byte string> ::= 
    <byte string value expression>

<start position> ::= 
    <numeric value expression>

<string length> ::= 
    <numeric value expression>

```

Syntax Rules

- 1) If <substring function> *CSF* is specified, then

Case:

- a) If LEFT is specified, then let *SRC* be <character string value expression>. *CSF* is equivalent to

```
SUBSTRING ( SRC <comma> LEADING )
```

- b) If RIGHT is specified, then let *SRC* be <character string value expression>. *CSF* is equivalent to

```
SUBSTRING ( SRC <comma> TRAILING )
```

- 2) If <trim function> is specified, then

Case:

- a) If lTrim is specified, then let *SRC* be <trim source>. <trim function> is equivalent to

```
TRIM ( SRC <comma> LEADING )
```

- b) If rTrim is specified, then let *SRC* be <trim source>. <trim function> is equivalent to

```
TRIM ( SRC <comma> TRAILING )
```

- 3) If <normalize function> is specified, then:

Case:

- a) If <normal form> is specified, then let *NF* be <normal form>.

- b) Otherwise, let *NF* be NFC.

- 4) If <byte substring function> *BSF* is specified, then

Case:

20.8 <string value function>

- a) If LEFT is specified, then let BRC be <byte string value expression>. BSF is equivalent to

```
SUBSTRING (  $BRC$  <comma> LEADING )
```

- b) If RIGHT is specified, then let BRC be <byte string value expression>. BSF is equivalent to

```
SUBSTRING (  $BRC$  <comma> TRAILING )
```

- 5) If <byte string trim function> is specified, then:

Case:

- a) If lTrim is specified, then let SRC be <byte string trim source>. <byte string trim function> is equivalent to

```
TRIM (  $SRC$  <comma> LEADING )
```

- b) If rTrim is specified, then let SRC be <byte string trim source>. <byte string trim function> is equivalent to

```
TRIM (  $SRC$  <comma> TRAILING )
```

General Rules

- 1) The declared type of <string value function> is the declared type of the immediately contained <character string function>, or <byte string function>.
- 2) The declared type of <character string function> is the declared type of the immediately contained <substring function>, <fold>, <trim function>, or <normalize function>.
- 3) if the most specific type of either a <start position> or a <string length> is not exact numeric with scale 0 (zero) then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 4) The result of <string value function> is the result of the immediately contained <character string function> or <byte string function>.
- 5) If <substring function> CSF is specified, then the declared type of CSF is character string with maximum length equal to the maximum length of the <character string value expression>.
- 6) The result of <character string function> is the result of the immediately contained <substring function>, <fold>, <trim function>, or <normalize function>.
- 7) If <fold> is specified, then the declared type of the result of <fold> is that of the <character string value expression>.
- 8) If <trim function> is specified, then
 - a) The declared type of the <trim function> is character string with maximum length equal to the maximum length of the <trim source>.
 - b) If a <trim character string> is specified, then if <trim character string> and <trim source> are not comparable then an exception condition is raised: *data exception — values not comparable (22G04)*.
- 9) The declared type of the <normalize function> is character string.
- 10) The declared type of <byte string function> is the declared type of the immediately contained <byte substring function>, or <byte string trim function>.

- 11) If <byte substring function> BSF is specified, then the declared type of BSF is the byte string type with maximum length equal to the maximum length of the <byte string value expression>.
- 12) The declared type of the <trim function> is the character string type with maximum length equal to the maximum length of the <trim source>.
- 13) If <substring function> is specified, then:
 - a) Let C be the value of the <character string value expression>, let LC be the length in characters of C , and let S be the value of the <start position>.
 - b) If <string length> is specified, then let L be the value of <string length> and let E be $S+L$. Otherwise, let E be the larger of $LC + 1$ (one) and S .
 - c) If at least one of C , S , and L is the null value, then the result of the <substring function> is the null value.
 - d) If E is less than S , then an exception condition is raised: *data exception — substring error (22011)*.
 - e) Case:
 - i) If S is greater than LC or if E is less than 1 (one), then the result of the <substring function> is the zero-length character string.
 - ii) Otherwise,
 - 1) Let $S1$ be the larger of S and 1 (one). Let $E1$ be the smaller of E and $LC+1$. Let $L1$ be $E1-S1$.
 - 2) The result of the <substring function> is a character string containing the $L1$ characters of C starting at character number $S1$ in the same order that the characters appear in C .
- 14) If <normalize function> is specified, then:
 - a) Let S be the value of <character string value expression>.
 - b) If S is the null value, then the result of the <normalize function> is the null value.
 - c) Let NR be S in the normalized form specified by NF in accordance with [Unicode Standard Annex #15](#).
 - d) If the length in characters of NR is less than or equal to implementation-defined maximum length of a character string, then the result of the <normalize function> is NR .
 - e) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.
- 15) If <fold> is specified, then:
 - a) Let S be the value of the <character string value expression>.
 - b) If S is the null value, then the result of the <fold> is the null value.
 - c) Case:
 - i) If $UPPER$ is specified, then let FR be a copy of S in which every lower-case character that has a corresponding upper-case character or characters and every title case character that has a corresponding upper-case character or characters is replaced by that upper-case character or characters.
 - ii) If $LOWER$ is specified, then let FR be a copy of S in which every upper-case character that has a corresponding lower-case character or characters and every title case character

20.8 <string value function>

that has a corresponding lower-case character or characters is replaced by that lower-case character or characters.

- d) FR is replaced by

Case:

- i) If the <search condition> S IS NORMALIZED evaluated to *True*, then

NORMALIZE (FR)

- ii) Otherwise, FR .

- e) Let FRL be the length in characters of FR and let $FRML$ be the implementation-defined maximum length of a character string.

- f) Case:

- i) If FRL is less than or equal to $FRML$, then the result of the <fold> is FR .

- ii) Otherwise the result of the <fold> is the first $FRML$ characters of FR . If any of the right-most ($FRL - FRML$) characters of FR are not <whitespace> characters, then a completion condition is raised: *data exception — string data, right truncation (22001)*.

- 16) If <trim function> is specified, then:

- a) Let S be the value of the <trim source>.

- b) If <trim character string> is specified, then let SC be the value of <trim character string>; otherwise, let SC be the zero-length character string.

- c) If at least one of S and SC is the null value, then the result of the <trim function> is the null value.

- d) If the length in characters of SC is not 0 (zero) or 1 (one), then an exception condition is raised: *data exception — trim error (22027)*.

- e) Case:

- i) If BOTH is specified or if no <trim specification> is specified, then the result of the <trim function> is the value of S with

Case:

- 1) If the length in characters of SC is 1 (one), then any leading or trailing characters equal to SC removed.

- 2) Otherwise, any leading or trailing <whitespace> characters removed.

- ii) If TRAILING is specified, then the result of the <trim function> is the value of S with

Case:

- 1) If the length in characters of SC is 1 (one), then any trailing characters equal to SC removed.

- 2) Otherwise, any trailing <whitespace> characters removed.

- iii) If LEADING is specified, then the result of the <trim function> is the value of S with

Case:

- 1) If the length in characters of SC is 1 (one), then any leading characters equal to SC removed.

- 2) Otherwise, any leading <whitespace> characters removed.
- 17) The result of <byte string function> is the result of the immediately contained <byte substring function>, or <byte string trim function>.
- 18) If <byte substring function> is specified, then
- Let B be the value of the <byte string value expression>, let LB be the length in bytes of B , and let S be the value of the <start position>.
 - If <string length> is specified, then let L be the value of <string length> and let E be $S+L$. Otherwise, let E be the larger of $LB+1$ and S .
 - If at least one of B , S , and L is the null value, then the result of the <byte substring function> is the null value.
 - If E is less than S , then an exception condition is raised: *data exception — substring error (22011)*.
 - Case:
 - If S is greater than LB or if E is less than 1 (one), then the result of the <byte substring function> is the zero-length byte string.
 - Otherwise:
 - Let $S1$ be the larger of S and 1 (one). Let $E1$ be the smaller of E and $LB+1$. Let $L1$ be $E1-S1$.
 - The result of the <byte substring function> is a byte string containing $L1$ bytes of B starting at byte number $S1$ in the same order that the bytes appear in B .
- 19) If <byte string trim function> is specified, then
- Let S be the value of the <byte string trim source>.
 - Let SO be the value of <trim byte string>.
 - If at least one of S and SO is the null value, then the result of the <byte string trim function> is the null value.
 - If the length in bytes of SO is not 1 (one), then an exception condition is raised: *data exception — trim error (22027)*.
 - Case:
 - If BOTH is specified or if no <trim specification> is specified, then the result of the <byte string trim function> is the value of S with any leading or trailing bytes equal to SO removed.
 - If TRAILING is specified, then the result of the <byte string trim function> is the value of S with any trailing bytes equal to SO removed.
 - If LEADING is specified, then the result of the <byte string trim function> is the value of S with any leading bytes equal to SO removed.

Conformance Rules

None.

20.9 <datetime value expression>

Function

Specify a datetime value.

Format

```

<datetime value expression> ::=

  <datetime term>
  | <duration value expression> <plus sign> <datetime term>
  | <datetime value expression> <plus sign> <duration term>
  | <datetime value expression> <minus sign> <duration term>

<datetime term> ::=

  <datetime factor>

<datetime factor> ::=

  <datetime primary>

<datetime primary> ::=

  <value expression primary>
  | <datetime value function>

```

Syntax Rules

None.

General Rules

- 1) If the most specific type of a <datetime primary> is not datetime then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 2) If the value of any <datetime primary>, <duration value expression>, <datetime value expression>, or <duration term> simply contained in a <datetime value expression> is the null value, then the result of the <datetime value expression> is the null value.
- 3) The value of a <datetime primary> is the value of the immediately contained <value expression primary> or <datetime value function>.
- 4) If a <datetime value expression> immediately contains the operator <plus sign> or <minus sign>, then the result is evaluated as specified in clause 14 “Date and time arithmetic” of ISO 8601-2:2019.

Conformance Rules

None.

20.10 <datetime value function>

Function

Specify a function yielding a value of type datetime.

Format

```

<datetime value function> ::=

  <date function>
  | <time function>
  | <datetime function>
  | <local time function>
  | <local datetime function>

<date function> ::=

  CURRENT_DATE
  | DATE <left paren> [ <date function parameters> ] <right paren>

<time function> ::=

  CURRENT_TIME
  | TIME <left paren> [ <time function parameters> ] <right paren>

<local time function> ::=

  LOCALTIME
  | LOCALTIME <left paren> [ <time function parameters> ] <right paren>

<datetime function> ::=

  CURRENT_TIMESTAMP
  | DATETIME <left paren> [ <datetime function parameters> ] <right paren>

<local datetime function> ::=

  LOCALTIMESTAMP
  | LOCALDATETIME <left paren> [ <datetime function parameters> ] <right paren>

<date function parameters> ::=

  <date string>
  | <map value constructor>

<time function parameters> ::=

  <time string>
  | <map value constructor>

<datetime function parameters> ::=

  <datetime string>
  | <map value constructor>

```

Syntax Rules

- 1) CURRENT_DATE is equivalent to:

date()

- 2) CURRENT_TIME is equivalent to:

time()

- 3) LOCALTIME is equivalent to:

`localtime()`

- 4) CURRENT_TIMESTAMP is equivalent to:

`datetime()`

- 5) LOCALTIMESTAMP is equivalent to:

`localdatetime()`

General Rules

- 1) The declared type of a <date function> is DATE. The declared type of a <time function> is TIME. The declared type of a <datetime function> is DATETIME. The declared type of a <local time function> is LOCALTIME. The declared type of a <local datetime function> is LOCALDATETIME.

- 2) If the set of <map key>s contained in a <date function parameters> is not one of:

- 'year'
- 'year' and 'month'
- 'year', 'month', and 'day'
- 'year' and 'week'
- 'year', 'week', and 'dayOfWeek'
- 'year' and 'ordinalDay'

then an exception is raised: *data exception — invalid datetime function map key (22G05)*.

- 3) If the set of <map key>s contained in a <time function parameters> is not one of:

- 'hour'
- 'hour' and 'minute'
- 'hour', 'minute', and 'second'
- 'hour', 'minute', and 'second'
- 'hour', 'minute', and 'second'
- 'hour', 'minute', 'second' and one or more of 'millisecond', 'microsecond', and 'nanosecond'.
- In addition, if <time function parameters> is contained in a <time function>, then, the set of permitted <map key>s also includes 'timezone'.

then an exception is raised: *data exception — invalid datetime function map key (22G05)*.

- 4) If the set of <map key>s contained in a <datetime function parameters> is not one of:

- 'year', 'month', 'day', and any of the set of <map key>s permitted in a <time function parameters>.
- 'year', 'week', 'dayOfWeek', and any of the set of <map key>s permitted in a <time function parameters>.
- 'year', 'ordinalDay', and any of the set of <map key>s permitted in a <time function parameters>.

then an exception is raised: *data exception — invalid datetime function map key (22G05)*.

- 5) If a <datetime function parameters> *DFP* is contained in <local datetime function> and *DFP* contains the <map key> 'timezone' then an exception is raised: *data exception — invalid datetime function map key (22G05)*.
- 6) If the value of each <map value> associated with the <map key> 'millisecond' is not between 0 (zero) and 999, or the value associated with 'microsecond', not between 0 (zero) and 999999 and the value for 'nanosecond' not between 0 (zero) and 999999999, or, if more than one of 'millisecond', 'microsecond', and 'nanosecond' is specified, any value exceeds 999 then an exception is raised: *data exception — invalid datetime function map value (22G06)*. If the values associated with other <map key>s do not conform to the range of values specified in clause 4.3 "Time scale components and units" of ISO 8601-1:2019 with, for the 'timezone' key, the optional addition of an IANA Time Zone Database time zone designator enclosed between a <left bracket> and a <right bracket> or, if both an IANA Time Zone Database time zone designator and a ISO 8601-1:2019 time shift specification are present in a 'timezone' key and the IANA Time Zone Database time zone designator does not resolve to the same value as the ISO 8601-1:2019 time shift specification then an exception is raised: *data exception — invalid datetime function map value (22G06)*.
- 7) If <date function parameters>, <time function parameters>, or <datetime function parameters> are specified, the map is transformed, respectively into an equivalent <date string>, <time string>, or <datetime string>.
- 8) Case:
 - a) If the <datetime value function>s date(), time(), datetime(), localtime(), and localdatetime() have no parameters then they respectively return the current date, time, datetime, localtime and localdatetime.

For time() the returned value has a displacement equal to the current time zone displacement of the time zone identified by the current time zone identifier of the session context.

For datetime() the returned value has a displacement equal to the current time zone displacement of the time zone identified by the current time zone identifier of the session context and a time zone identifier equal to the current time zone identifier of the session context.
 - b) Otherwise, they return respectively the date, time, datetime, localtime and localdatetime values associated with the representation as defined by ISO 8601-1:2019 and ISO 8601-2:2019.

Conformance Rules

None.

20.11 <duration value expression>

Function

Specify a duration value.

Format

```

<duration value expression> ::=

  <duration term>
  | <duration value expression 1> <plus sign> <duration term 1>
  | <duration value expression 1> <minus sign> <duration term 1>
  | <left paren> <datetime value expression> <minus sign> <datetime term> <right paren>

<duration term> ::=

  <duration factor>
  | <duration term 2> <asterisk> <factor>
  | <duration term 2> <solidus> <factor>
  | <term> <asterisk> <duration factor>

<duration factor> ::=

  [ <sign> ] <duration primary>

<duration primary> ::=

  <value expression primary>
  | <duration value function>

<duration value expression 1> ::=
  <duration value expression>

<duration term 1> ::=
  <duration term>

<duration term 2> ::=
  <duration term>

```

Syntax Rules

- 1) If <duration term>, DT immediately contains <solidus>, then let $DT2$ be the <duration term 2> immediately contained in DT and F be the <factor> immediately contained in DT . DT is effectively replaced by:

$DT2 * (1 / F)$

General Rules

- 1) The declared type of a <duration value expression> is duration.
- 2) If the most specific type of a <value expression primary> immediately contained in a <duration primary> is not duration then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 3) If <datetime value expression> is specified, then if <datetime value expression> and <datetime term> are not comparable then an exception condition is raised: *data exception — values not comparable (22G04)*.

- 4) If <duration term> immediately contains <asterisk>, then let V be the value of the immediately contained <term> or <factor>. The result of the <duration term> is the duration specified by clause 14.3, "Multiplication" of ISO 8601-2:2019 with V as the coefficient, and <duration term 2> or <duration factor> as durationA.
- 5) The result of <duration value expression> DVE is

Case:

- a) If DVE immediately contains <duration term> then the value of DVE is the value of <duration term>.
- b) If DVE immediately contains <duration value expression 1> then the result is as specified by clause 14, "Date and time arithmetic" of ISO 8601-2:2019.
- c) Otherwise, the value of DVE is the duration that if added, as specified by clause 14, "Date and time arithmetic" of ISO 8601-2:2019, to the <datetime term> would yield the same value as <datetime value expression>.

Conformance Rules

None.

20.12 <duration value function>

Function

Specify a function yielding a value of type duration.

Format

```
<duration value function> ::=  
  <duration function>  
  | <duration absolute value function>  
  
<duration function> ::=  
  DURATION <left paren> <duration function parameters> <right paren>  
  
<duration function parameters> ::=  
  <duration string>  
  | <map value constructor>  
  
<duration absolute value function> ::=  
  ABS <left paren> <duration value expression> <right paren>
```

Syntax Rules

None.

General Rules

- 1) The declared type of a <duration function> is DURATION.
- 2) If a <map key>s contained in a <duration function parameters> is not one of 'years', 'months', 'days', 'weeks', 'days', 'hours', 'minutes', 'seconds', 'milliseconds', 'microseconds', or 'nanoseconds' then an exception condition is raised: *data exception — invalid duration function map key (22G07)*.
- 3) If <duration function parameters> is specified, the map is transformed into an equivalent <duration string>.
- 4) The result of a <duration function> is the duration specified for the <duration string> defined in clause 5.5.2 "Duration" of ISO 8601-1:2019 as extended by clauses 4.3 "Additional explicit forms" and 4.4 "Numerical extensions" of ISO 8601-2:2019.
- 5) If <duration absolute value function> is specified, then let N be the value of the <duration value expression>.

Case:

- a) If N is the null value, then the result is the null value.
- b) If $N \geq 0$ (zero), then the result is N .
- c) Otherwise, the result is $-1 * N$.

Conformance Rules

None.

20.13 <graph element value expression>

Function

Specify a graph element value.

Format

```
<graph element value expression> ::=  
  <graph element primary>  
  
<graph element primary> ::=  
  <graph element function>  
  | <value expression primary>
```

Syntax Rules

None.

General Rules

- 1) The declared type of a <graph element value expression> is the value type of the <graph element primary>.
- 2) The declared type of a <graph element primary> is the value type of the <graph element function> or the <value expression primary>.
- 3) The value of the <graph element value expression> is the result of the <graph element primary>.
- 4) The result of a <graph element primary> is the result of the <graph element function> or the <value expression primary>.

Conformance Rules

None.

20.14<graph element function>

Function

Specify a function yielding a value of graph element type.

Format

```

<graph element function> ::==
  <start node function>
  | <end node function>

<start node function> ::==
  startNode <left paren> <binding variable> <right paren>

<end node function> ::==
  endNode <left paren> <binding variable> <right paren>

```

Syntax Rules

None.

General Rules

- 1) If the most specific type of a <binding variable> is not an edge type then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 2) The declared type of a <graph element function> is node.
- 3) The value of the <graph element function> is the value of the immediately contained <start node function> or <end node function>.
- 4) The result of a <start node function> is the node that is the source node of the edge identified by <binding variable>.
- 5) The result of an <end node function> is the node that is the destination node of the edge identified by <binding variable>.

Conformance Rules

None.

20.15<collection value constructor>

Function

Define a <collection value constructor>.

Format

```
<collection value constructor> ::=  
  <list value constructor>  
  | <multiset value constructor>  
  | <set value constructor>  
  | <ordered set value constructor>  
  | <map value constructor>  
  | <record value constructor>
```

Syntax Rules

None.

General Rules

- 1) The declared type of a <collection value constructor> is the declared type of the <list value constructor>, <multiset value constructor>, <set value constructor>, <ordered set value constructor>, <map value constructor>, or <record value constructor> that it immediately contains.
- 2) The value of a <collection value constructor> is the value of the <list value constructor>, <multiset value constructor>, <set value constructor>, <ordered set value constructor>, <map value constructor>, or <record value constructor> that it immediately contains.

Conformance Rules

None.

20.16 <list value expression>

Function

Specify a list value.

Format

```

<list value expression> ::= 
  <list concatenation>
  | <list primary>

<list concatenation> ::= 
  <list value expression 1> <concatenation operator> <list primary>

<list value expression 1> ::= 
  <list value expression>

<list primary> ::= 
  <list value function>
  | <value expression primary>

```

Syntax Rules

None.

General Rules

- 1) The declared type of the <list value expression> is the declared type of the immediately contained <list concatenation> or <list primary>.
 - 2) The declared type of <list primary> is the declared type of the immediately contained <list value function> or <value expression primary>.
 - 3) If the most specific type of <value expression primary> is not a list type then an exception condition is raised: *data exception — invalid value type (22G03)*.
 - 4) If <list concatenation> is specified, then:
 - a) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the most specific types of <list value expression 1> and <list primary> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those General Rules.
 - b) Let *IDMC* be the implementation-defined maximum cardinality of a list type.
 - c) The declared type of the result of <list concatenation> is a list type whose element type is the element type of *DT*.
 - d) The value of the result of <list value expression> is the value of the immediately contained <list concatenation> or <list primary>.
 - e) If <list concatenation> is specified, then let *LV1* be the value of <list value expression 1> and let *LV2* be the value of <list primary>.
- Case:

- i) If at least one of $LV1$ and $LV2$ is the null value, then the result of the <list concatenation> is the null value.
- ii) If the sum of the cardinality of $LV1$ and the cardinality of $LV2$ is greater than $IDMC$, then an exception condition is raised: *data exception — list data, right truncation (22G0B)*.
- iii) Otherwise, the result is the list comprising every element of $LV1$ followed by every element of $LV2$.

Conformance Rules

None.

20.17 <list value function>

Function

Specify a function yielding a value of a list type.

Format

```
<list value function> ::=  
    <tail list function>  
  | <trim list function>  
  
<tail list function> ::=  
    tail <left paren> <list value expression> <right paren>  
  
<trim list function> ::=  
    TRIM <left paren> <list value expression> <comma> <numeric value expression> <right paren>
```

Syntax Rules

None.

General Rules

- 1) The declared type of the <list value function> is the declared type of the immediately contained <tail list function>, or <trim list function>.
- 2) If <tail list function> is specified, then the declared type of the <tail list function> is the declared type of the immediately contained <list value expression>.
- 3) If <trim list function> is specified, then:
 - a) if the most specific type of the <numeric value expression> is not exact numeric type with scale 0 (zero) then an exception condition is raised: *data exception — invalid value type (22G03)*.
 - b) The declared type of the <trim list function> is the declared type of the immediately contained <list value expression>.
- 4) The value of the <list value function> is the value of the immediately contained <tail list function>, or <trim list function>.
- 5) The result of <tail list function> is determined as follows:
 - a) Let *AV* be the value of the <list value expression>.
 - b) If *AV* is the null value, then the result is the null value and no further General Rules of this Sub-clause are applied.
 - c) Let *AC* be the cardinality of *AV*.
 - d) If *AC* is 0 (zero), then an exception condition is raised: *data exception — list element error (22G0C)*.
 - e) Let *N* be *AC* – 1 (one).
 - f) Case:
 - i) If *N* = 0 (zero), then the result is a list whose cardinality is 0 (zero).

- ii) Otherwise, the result is a list of N elements such that for every i , $2 \leq i \leq AC$, the value of the i -th element of the result is the value of the i -th element of AV .
- 6) The result of <trim list function> is determined as follows:
- a) Let NV be the value of the <numeric value expression>.
 - b) If NV is the null value, then the result is the null value and no further General Rules of this Sub-clause are applied.
 - c) If NV is less than 0 (zero), then an exception condition is raised: *data exception — list element error (22GOC)*.
 - d) Let AV be the value of the <list value expression>.
 - e) If AV is the null value, then the result is the null value and no further General Rules of this Sub-clause are applied.
 - f) Let AC be the cardinality of AV .
 - g) If NV is greater than AC , then an exception condition is raised: *data exception — list element error (22GOC)*.
 - h) Let N be $AC - NV$.
 - i) Case:
 - i) If $N = 0$ (zero), then the result is a list whose cardinality is 0 (zero).
 - ii) Otherwise, the result is a list of N elements such that for every i , $1 \text{ (one)} \leq i \leq N$, the value of the i -th element of the result is the value of the i -th element of AV .

Conformance Rules

None.

20.18 <list value constructor>

Function

Specify construction of a list.

Format

```

<list value constructor> ::= 
    <list value constructor by enumeration>

<list value constructor by enumeration> ::= 
    <list value type name> <left bracket> <list element list> <right bracket>

<list element list> ::= 
    <list element> [ { <comma> <list element> }... ]

<list element> ::= 
    <value expression>
  
```

Syntax Rules

None.

General Rules

- 1) The declared type of <list value constructor> is the declared type of the immediately contained <list value constructor by enumeration>.
- 2) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the declared types of the <list element>s immediately contained in the <list element list> of the <list value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those General Rules. The declared type of the <list value constructor by enumeration> is a list type with element type *DT*. If the number of <list element>s in the <list element list> of the <list value constructor by enumeration> is greater than the implementation-defined maximum cardinality for list types whose element type is *DT* then an exception condition is raised: *data exception — maximum list cardinality exceeded (22G08)*.
- 3) The value of <list value constructor> is the value of the immediately contained <list value constructor by enumeration>.
- 4) The result of <list value constructor by enumeration> is a list whose *i*-th element is the value of the *i*-th <list element> immediately contained in the <list element list>, cast as the value type of *DT*.

Conformance Rules

None.

20.19 <multiset value expression>

Function

Specify a multiset value.

Format

```

<multiset value expression> ::= 
  <multiset term>
  | <multiset value expression> MULTISET UNION [ ALL | DISTINCT ] <multiset term>
  | <multiset value expression> MULTISET EXCEPT [ ALL | DISTINCT ] <multiset term>

<multiset term> ::= 
  <multiset primary>
  | <multiset term> MULTISET INTERSECT [ ALL | DISTINCT ] <multiset primary>

<multiset primary> ::= 
  <multiset value function>
  | <value expression primary>

```

Syntax Rules

- 1) If a <multiset term> *MI* immediately contains MULTISET INTERSECT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>). *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 22.9, “Bag element grouping operations”, apply.
- 2) If *MU* is a <multiset value expression> that immediately contains MULTISET UNION, then let *OP1* be the first operand (the <multiset value expression>) and let *OP2* be the second operand (the <multiset term>). If DISTINCT is specified, then *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 22.9, “Bag element grouping operations”, apply.
- 3) If *ME* is a <multiset value expression> that immediately contains MULTISET EXCEPT, then let *OP1* be the first operand (the <multiset value expression>) and let *OP2* be the second operand (the <multiset term>). *OP1* and *OP2* are multiset operands of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Table 1, “Symbols used in BNF”, apply.

General Rules

- 1) The declared type of a <multiset primary> is the declared type of the immediately contained <multiset value function> or <value expression primary>.
- 2) If the most specific type of a <multiset value function> or <value expression primary> immediately contained in a <multiset primary> is not a multiset type then an exception is raised: *data exception — invalid value type (22G03)*.
- 3) If a <multiset term> *MI* immediately contains MULTISET INTERSECT, then let *OP1* be the first operand (the <multiset term>) and let *OP2* be the second operand (the <multiset primary>).
 - a) Let *ET1* be the element type of *OP1* and let *ET2* be the element type of *OP2*. The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with *ET1* and *ET2* as *DTSET*; let *ET* be the *RESTYPE* returned from the application of those General Rules. The result type of the MULTISET INTERSECT operation is multiset with element type *ET*.

20.19 <multiset value expression>

- b) If DISTINCT is specified, then let SQ be DISTINCT. Otherwise, let SQ be ALL.
- c) MI is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
      ELSE MULTISET ( SELECT T1.V
                       FROM UNNEST (OP1) AS T1(V)
                       INTERSECT SQ
                       SELECT T2.V
                       FROM UNNEST (OP2) AS T2(V)
                   )
      END )
```

- 4) If a <multiset value expression> MU that immediately contains MULTISET UNION, then let $OP1$ be the first operand (the <multiset value expression>) and let $OP2$ be the second operand (the <multiset term>).

- a) Let $ET1$ be the element type of $OP1$ and let $ET2$ be the element type of $OP2$. The General Rules of Subclause 22.8, "Result of value type combinations", are applied with $ET1$ and $ET2$ as $DTSET$; let ET be the $RESTYPE$ returned from the application of those General Rules. The result type of the MULTISET UNION operation is multiset with element type ET .
- b) If DISTINCT is specified, then let SQ be DISTINCT. Otherwise, let SQ be ALL.
- c) MU is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
      ELSE MULTISET ( SELECT T1.V
                       FROM UNNEST (OP1) AS T1(V)
                       UNION SQ
                       SELECT T2.V
                       FROM UNNEST (OP2) AS T2(V)
                   )
      END )
```

- 5) If a <multiset value expression> ME that immediately contains MULTISET EXCEPT, then let $OP1$ be the first operand (the <multiset value expression>) and let $OP2$ be the second operand (the <multiset term>).

- a) Let $ET1$ be the element type of $OP1$ and let $ET2$ be the element type of $OP2$. The General Rules of Subclause 22.8, "Result of value type combinations", are applied with $ET1$ and $ET2$ as $DTSET$; let ET be the $RESTYPE$ returned from the application of those General Rules. The result type of the MULTISET EXCEPT operation is multiset with element type ET .
- b) If DISTINCT is specified, then let SQ be DISTINCT. Otherwise, let SQ be ALL.
- c) ME is equivalent to

```
( CASE WHEN OP1 IS NULL OR OP2 IS NULL THEN NULL
      ELSE MULTISET ( SELECT T1.V
                       FROM UNNEST (OP1) AS T1(V)
                       EXCEPT SQ
                       SELECT T2.V
                       FROM UNNEST (OP2) AS T2(V)
                   )
      END )
```

- 6) The value of a <multiset primary> is the value of the immediately contained <multiset value function> or <value expression primary>.
- 7) The value of a <multiset term> that is a <multiset primary> is the value of the <multiset primary>.

- 8) The value of a <multiset value expression> that is a <multiset term> is the value of <multiset term>.

Conformance Rules

None.

20.20 <multiset value function>

Function

Specify a function yielding a value of a multiset type.

Format

```
<multiset value function> ::=  
  <multiset set function>  
  
<multiset set function> ::=  
  SET <left paren> <multiset value expression> <right paren>
```

Syntax Rules

- 1) Let *MVE* be the <multiset value expression> immediately contained in <multiset set function>. *MVE* is a multiset operand of a multiset element grouping operation. The Syntax Rules and Conformance Rules of Subclause 22.9, “Bag element grouping operations”, apply.
- 2) The <multiset set function> is equivalent to

```
( CASE WHEN MVE IS NULL THEN NULL  
      ELSE MULTISET ( SELECT DISTINCT M.E  
                       FROM UNNEST (MVE) AS M(E) )  
      END )
```

General Rules

None.

Conformance Rules

None.

20.21 <multiset value constructor>

Function

Specify construction of a multiset.

Format

```
<multiset value constructor> ::=  
  <multiset value constructor by enumeration>  
  
<multiset value constructor by enumeration> ::=  
  MULTISET <left brace> <multiset element list> <right brace>  
  
<multiset element list> ::=  
  <multiset element> [ { <comma> <multiset element> }... ]  
  
<multiset element> ::=  
  <value expression>
```

Syntax Rules

None.

General Rules

- 1) The declared type of <multiset value constructor> is the declared type of the immediately contained <multiset value constructor by enumeration>.
- 2) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the declared types of the <multiset element>s immediately contained in the <multiset element list> of the <multiset value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those General Rules. The declared type of the <multiset value constructor by enumeration> is a multiset type with element type *DT*.
- 3) The value of <multiset value constructor> is the value of the immediately contained <multiset value constructor by enumeration>.
- 4) The result of <multiset value constructor by enumeration> is a multiset whose elements are the values of the <multiset element>s immediately contained in the <multiset element list>, cast as the value type of *DT*.

Conformance Rules

None.

20.22 <set value constructor>

Function

Specify construction of a set.

Format

```

<set value constructor> ::= 
  <set value constructor by enumeration>

<set value constructor by enumeration> ::= 
  SET <left brace> <set element list> <right brace>

<set element list> ::= 
  <set element> [ { <comma> <set element> }... ]

<set element> ::= 
  <value expression>

```

Syntax Rules

None.

General Rules

- 1) The declared type of <set value constructor> is the declared type of the immediately contained <set value constructor by enumeration>.
- 2) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the declared types of the <set element>s immediately contained in the <set element list> of the <set value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those General Rules. The declared type of the <set value constructor by enumeration> is a set type with element type *DT*.
- 3) The value of <set value constructor> is the value of the immediately contained <set value constructor by enumeration>.
- 4) The result of <set value constructor by enumeration> is a set whose elements are the values of the <set element>s immediately contained in the <set element list>, cast as the value type of *DT*.

Conformance Rules

None.

20.23 <ordered set value constructor>

Function

Specify construction of an ordered set.

Format

```

<ordered set value constructor> ::= 
  <ordered set value constructor by enumeration>

<ordered set value constructor by enumeration> ::= 
  ORDERED SET {
    <left brace> <ordered set element list> <right brace>
  | <left bracket> <ordered set element list> <right bracket>
  }

<ordered set element list> ::= 
  <ordered set element> [ { <comma> <ordered set element> }... ]

<ordered set element> ::= 
  <value expression>

```

Syntax Rules

None.

General Rules

- 1) The declared type of <ordered set value constructor> is the declared type of the immediately contained <ordered set value constructor by enumeration>.
- 2) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the declared types of the <ordered set element>s immediately contained in the <ordered set element list> of the <ordered set value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those General Rules. The declared type of the <ordered set value constructor by enumeration> is an ordered set type with element type *DT*.
- 3) The value of <ordered set value constructor> is the result of the immediately contained <ordered set value constructor by enumeration>.
- 4) The result of <ordered set value constructor by enumeration> is an ordered set whose elements are the values of the <ordered set element>s immediately contained in the <ordered set element list> in the order of their occurrence from left to right and cast as the value type of *DT*.

Conformance Rules

None.

20.24 <map value constructor>

Function

Specify construction of a map.

Format

```

<map value constructor> ::=

    <map value constructor by enumeration>

<map value constructor by enumeration> ::=

    MAP <left brace> <map element list> <right brace>

<map element list> ::=

    <map element> [ { <comma> <map element> }... ]

<map element> ::=

    <map key> <map value>

<map key> ::=

    <value expression> <colon>

<map value> ::=

    <value expression>

```

Syntax Rules

None.

General Rules

- 1) The declared type of <map value constructor> is the declared type of the immediately contained <map value constructor by enumeration>.
- 2) If the declared type of a <value expression> immediately contained in <map key> is not a <predefined type> then an exception condition is raised: *data exception — invalid value type (22G03)*.
- 3) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the declared types of the <map key>s immediately contained in the <map element list> of the <map value constructor by enumeration> as *DTSET*; let *KT* be the *RESTYPE* returned from the application of those General Rules.
- 4) The General Rules of Subclause 22.8, “Result of value type combinations”, are applied with the declared types of the <map value>s immediately contained in the <map element list> of the <map value constructor by enumeration> as *DTSET*; let *DT* be the *RESTYPE* returned from the application of those General Rules.
- 5) The declared type of the <map value constructor by enumeration> is a map type with key element type *KT* and value element type *DT*.
- 6) The value of <map value constructor> is the value of the immediately contained <map value constructor by enumeration>.
- 7) The result of <map value constructor by enumeration> is a map whose elements are the values of the <map element>s immediately contained in the <map element list>.

- 8) The value of <map element> is a (key, value) pair where the value of the “key” is the value of the <map key>s immediately contained in the <map element>, cast as the value type of KT and the value of the “value” is the value of the <map value> immediately contained in the <map element>, cast as the value type of DT .

Conformance Rules

None.

20.25 <record value constructor>

Function

Specify construction of a record.

Format

```

<record value constructor> ::==
  <record value constructor by enumeration>
  | UNIT

<record value constructor by enumeration> ::==
  [ RECORD ] <left brace> <field list> <right brace>

<field list> ::==
  <field> [ { <comma> <field> }... ]

<field> ::==
  <field name> <field value>

<field value> ::==
  <value expression>

```

Syntax Rules

- 1) If a <record value constructor> *RVC* is specified that is UNIT, then *RVC* is effectively replaced by:
 $\{ \}$
- 2) If a <field list> *FL* is specified, then it shall not simply contain two <field>s *F1* and *F2* such that the <field name> immediately contained in *F1* is equal to the <field name> immediately contained in *F2*.

General Rules

- 1) The declared type of <record value constructor> is the declared type of the immediately contained <record value constructor by enumeration>.
- 2) The declared type of the <record value constructor by enumeration> is a record type whose field types are the field types of the <field>s immediately contained in the <field list>.
- 3) The field type of <field> is a (field name, field value type) pair where the “field name” is the value of the <field name> immediately contained in the <field> and the “field value type” is the declared type of the <field value> immediately contained in the <field>.
- 4) The value of <record value constructor> is the result of the immediately contained <record value constructor by enumeration>.
- 5) The result of <record value constructor by enumeration> is a record whose elements are the values of the <field>s immediately contained in the <field list>.
- 6) The value of <field> is a (field name, field value) pair where the “field name” is the <field name> immediately contained in the <field> and the value of the “field value” is the value of the <field value> immediately contained in the <field>.

Conformance Rules

None.

20.26 <property reference>

Function

Reference a property of a graph element.

Format

```
<property reference> ::=  
  <graph element primary> <period> <property name>
```

Syntax Rules

None.

General Rules

- 1) Let PR be the <property reference>.
- 2) Let GE be the result of evaluating the <graph element primary>. If GE is not a graph element, raise an exception condition.
- 3) If GE has a property P whose property name is the <property name>, then the result of evaluating PR is the property value of P . Otherwise, the result of evaluating PR is the null value.

Conformance Rules

None.

20.27 <value query expression>

Function

Define a <value query expression>.

Format

```
<value query expression> ::=  
    VALUE <nested query specification>
```

Syntax Rules

None.

General Rules

None.

Conformance Rules

None.

20.28<case expression>

Function

Specify a conditional value.

Format

```

<case expression> ::=

  <case abbreviation>
  | <case specification>

<case abbreviation> ::=

  NULLIF <left paren> <value expression> <comma> <value expression> <right paren>
  | COALESCE <left paren> <value expression>
    { <comma> <value expression> }... <right paren>

<case specification> ::=

  <simple case>
  | <searched case>

<simple case> ::=

  CASE <case operand> <simple when clause>... [ <else clause> ] END

<searched case> ::=

  CASE <searched when clause>... [ <else clause> ] END

<simple when clause> ::=

  WHEN <when operand list> THEN <result>

<searched when clause> ::=

  WHEN <search condition> THEN <result>

<else clause> ::=

  ELSE <result>

<case operand> ::=

  <non-parenthesized value expression primary>
  | <element reference>

<when operand list> ::=

  <when operand> [ { <comma> <when operand> }... ]

<when operand> ::=

  <non-parenthesized value expression primary>
  | <comparison predicate part 2>
  | <null predicate part 2>
  | <directed predicate part 2>
  | <labeled predicate part 2>
  | <source predicate part 2>
  | <destination predicate part 2>

<result> ::=

  <result expression>
  | NULL

<result expression> ::=

  <value expression>

```

Syntax Rules

- 1) If a <case expression> specifies a <case abbreviation>, then:

- a) `NULLIF (V1, V2)` is equivalent to the following <case specification>:

```
CASE WHEN
V1=V2 THEN
NULL ELSE V1
END
```

- b) `COALESCE (V1, V2)` is equivalent to the following <case specification>:

```
CASE
WHEN NOT V1 IS NULL THEN V1
ELSE V2
END
```

- c) `COALESCE (V1, V2, ..., Vn)`, for $n \geq 3$, is equivalent to the following <case specification>:

```
CASE
WHEN NOT V1 IS NULL THEN V1
ELSE COALESCE (V2, ..., Vn)
END
```

- 2) If a <case specification> specifies a <simple case>, then let *CO* be the <case operand>.

- a) If any <when operand> is <directed predicate part 2>, <labeled predicate part 2>, <source predicate part 2> or <destination predicate part 2>, then *CO* shall be <element reference> and every <when operand> shall be <directed predicate part 2>, <labeled predicate part 2>, <source predicate part 2> or <destination predicate part 2>; otherwise, *CO* shall not be <element reference> and no <when operand> shall be <directed predicate part 2>, <labeled predicate part 2>, <source predicate part 2> or <destination predicate part 2>.

- b) Let *N* be the number of <simple when clause>s.

- c) For each *i* between 1 (one) and *N*, let *WOL_i* be the <when operand list> of the *i*-th <simple when clause>. Let *M(i)* be the number of <when operand>s simply contained in *WOL_i*. For each *j* between 1 (one) and *M(i)*, let *WO_{i,j}* be the *j*-th <when operand> simply contained in *WOL_i*.

- d) For each *i* between 1 (one) and *N*, and for each *j* between 1 (one) and *M(i)*,

Case:

- i) If *WO_{i,j}* is a <non-parenthesized value expression primary>, then let *EWO_{i,j}* be

$= WO_{i,j}$

- ii) Otherwise, let *EWO_{i,j}* be *WO_{i,j}*.

- e) Let *R_i* be the <result> of the *i*-th <simple when clause>.

- f) If <else clause> is specified, then let *CEEC* be that <else clause>; otherwise, let *CEEC* be the zero-length character string.

- g) The <simple case> is equivalent to a <searched case> in which the *i*-th <searched when clause> takes the form:

`WHEN (CO EWOi,1) OR`

```

    ... OR
    ( CO EWOi,M(i) )
    THEN Ri

```

- h) The <else clause> of the equivalent <searched case> takes the form:

CEEC

- i) The Conformance Rules of the Subclauses of [Clause 19, "Predicates"](#), are applied to the result of this syntactic transformation.

NOTE 143 — The specific Subclauses of [Clause 19, "Predicates"](#), are determined by the predicates that are created as a result of the syntactic transformation.

- 3) At least one <result> in a <case specification> shall specify a <result expression>.
 4) If an <else clause> is not specified, then ELSE NULL is implicit.

General Rules

- 1) The General Rules of [Subclause 22.8, "Result of value type combinations"](#), are applied with the set of declared types of all <result expression>s in the <case specification> as *DTSET*; let *RT* be the *RESTYPE* returned from the application of those General Rules. The declared type of the <case specification> is *RT*.
- 2) Case:
- If a <result> specifies NULL, then its value is the null value.
 - If a <result> specifies a <value expression>, then its value is the value of that <value expression>.
- 3) Case:
- If the value of the <search condition> of some <searched when clause> in a <case specification> is *True*, then the value of the <case expression> is the value of the <result> of the first (leftmost) <searched when clause> whose <search condition> evaluates to *True*, cast as the declared type of the <case specification>.
 - If no <search condition> in a <case specification> evaluates to *True*, then the value of the <case expression> is the value of the <result> of the explicit or implicit <else clause>, cast as the declared type of the <case specification>.

20.29 <cast specification>

Function

Specify a data conversion.

Format

```
<cast specification> ::=  
    CAST <left paren> <cast operand> AS <cast target> <right paren>  
  
<cast operand> ::=  
    <value expression>  
    | <null literal>  
  
<cast target> ::=  
    <predefined type>
```

Syntax Rules

- 1) Let *TD* be the data type identified by <predefined type>.
- 2) The declared type of the result of the <cast specification> is *TD*.
- 3) If the <cast operand> is a <value expression>, then let *SD* be the declared type of the <value expression>.
- 4) If the <cast operand> is a <value expression>, then the valid combinations of *TD* and *SD* in a <cast specification> are given by the following table. “Y” indicates that the combination is syntactically valid without restriction; “M” indicates that the combination is valid subject to other Syntax Rules in this Subclause being satisfied; and “N” indicates that the combination is not valid.

<i>SD</i>	<i>TD</i>									
	EN	UN	AN	C	D	T	DT	DU	BO	B
EN	Y	Y	Y	Y	N	N	N	N	N	N
UN	Y	Y	Y	Y	N	N	N	N	N	N
AN	Y	Y	Y	Y	N	N	N	N	N	N
C	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
D	N	N	N	Y	Y	N	Y	N	N	N
T	N	N	N	Y	N	Y	Y	N	N	N
DT	N	N	N	Y	Y	Y	Y	N	N	N
DU	N	N	N	Y	N	N	N	Y	N	N
BO	N	N	N	Y	N	N	N	N	Y	N
B	N	N	N	N	N	N	N	N	N	Y

Where:

EN = Signed Exact Numeric
 UN = Unsigned Exact Numeric
 AN = Approximate Numeric
 C = Character String
 D = Date
 T = Time & Localtime
 DT = Datetime & Localdatetime
 DU = Duration
 BO = Boolean
 B = Byte String

General Rules

- 1) Let CS be the <cast specification>. If the <cast operand> is a <value expression> VE , then let SV be the value of VE .
- 2) Case:
 - a) If the <cast operand> specifies NULL, then the result of CS is the null value and no further General Rules of this Subclause are applied.
 - b) If SV is the null value, then the result of CS is the null value and no further General Rules of this Subclause are applied.
- 3) If TD is a signed exact numeric type, then

Case:

 - a) If SD is a numeric type, then

Case:

 - i) If there is a representation of SV in the value type TD that does not lose any leading significant digits after rounding or truncating if necessary, then TV is that representation. The choice of whether to round or truncate is implementation-defined.
 - ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.
 - b) If SD is character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.

Case:

 - i) If SV does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 21.1, “<literal>”, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.
 - ii) Otherwise, let LT be that <signed numeric literal>. The <cast specification> is equivalent to $\text{CAST} (LT \text{ AS } TD)$.
- 4) If TD is an unsigned exact numeric type, then

Case:

 - a) If SD is a numeric type, then

Case:

 - i) If there is a representation of SV in the value type TD that does not lose any leading significant digits or the sign after rounding or truncating if necessary, then TV is that representation. The choice of whether to round or truncate is implementation-defined.
 - ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.
 - b) If SD is character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.

Case:

- i) If SV does not comprise an <unsigned numeric literal> as defined by the rules for <literal> in Subclause 21.1, “<literal>”, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.
 - ii) Otherwise, let LT be that <signed numeric literal>. The <cast specification> is equivalent to $\text{CAST} (LT \text{ AS } TD)$.
- 5) If TD is an approximate numeric type, then
- Case:
- a) If SD is a numeric type, then
- Case:
- i) If there is a representation of SV in the value type TD that does not lose any leading significant digits after rounding or truncating if necessary, then TV is that representation. The choice of whether to round or truncate is implementation-defined.
 - ii) Otherwise, an exception condition is raised: *data exception — numeric value out of range (22003)*.
- b) If SD is a character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.
- Case:
- i) If SV does not comprise a <signed numeric literal> as defined by the rules for <literal> in Subclause 21.1, “<literal>”, then an exception condition is raised: *data exception — invalid character value for cast (22018)*.
 - ii) Otherwise, let LT be that <signed numeric literal>. The <cast specification> is equivalent to $\text{CAST} (LT \text{ AS } TD)$.
- 6) If TD is a fixed-length character string type, then let LTD be the length in characters of TD .
- Case:
- a) If SD is an exact numeric type, then:
 - i) Let YP be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 21.1, “<literal>”, whose scale is the same as the scale of SD , whose interpreted value is the absolute value of SV , and that is not an <unsigned hexadecimal integer>.
 - ii) Case:
 - 1) If SV is less than 0 (zero), then let Y be the result of ' $-$ ' $|$ YP .
 - 2) Otherwise, let Y be YP .
 - iii) Case:
 - 1) If Y contains a code point that is a noncharacter in the character repertoire of UCS then an exception condition is raised: *data exception — non-character in character string (22029)*.
 - 2) If the length in characters LY of Y is equal to LTD , then TV is Y .
 - 3) If the length in characters LY of Y is less than LTD , then TV is Y extended on the right by $LTD-LY$ <space>s.

- 4) Otherwise, an exception condition is raised: *data exception — string data, right truncation* (22001).
- b) If *SD* is an approximate numeric type, then:
- i) Let *YP* be a character string as follows.
- Case:
- 1) If *SV* equals 0 (zero), then *YP* is '0E0'.
 - 2) Otherwise, *YP* is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 21.1, “<literal>”, whose interpreted value is equal to the absolute value of *SV* and whose <mantissa> consists of a single <digit> that is not '0' (zero), followed by a <period> and an <unsigned integer>.
- ii) Case:
- 1) If *SV* is less than 0 (zero), then let *Y* be the result of ' - ' | *YP*.
 - 2) Otherwise, let *Y* be *YP*.
- iii) Case:
- 1) If *Y* contains a code point that is a noncharacter in the character repertoire of UCS then an exception condition is raised: *data exception — non-character in character string* (22029).
 - 2) If the length in characters *LY* of *Y* is equal to *LTD*, then *TV* is *Y*.
 - 3) If the length in characters *LY* of *Y* is less than *LTD*, then *TV* is *Y* extended on the right by *LTD-LY* <space>s.
 - 4) Otherwise, an exception condition is raised: *data exception — string data, right truncation* (22001).
- c) If *SD* is a fixed-length character string type or a variable-length character string type, then
- Case:
- i) If the length in characters of *SV* is equal to *LTD*, then *TV* is *SV*.
 - ii) If the length in characters of *SV* is larger than *LTD*, then *TV* is the first *LTD* characters of *SV*. If any of the remaining characters of *SV* are non-<whitespace> characters, then a completion condition is raised: *warning — string data, right truncation* (01004).
 - iii) If the length in characters *M* of *SV* is smaller than *LTD*, then *TV* is *SV* extended on the right by *LTD-M* <space>s.
- d) If *SD* is a temporal instant type or a duration type, then let *YS* be the shortest character string that conforms to the definition of <literal> in Subclause 21.1, “<literal>”, and such that the interpreted value of *YS* is *SV*, and such that
- Case:
- i) If *SD* is a date type, the character string includes the time scale components: [year], [month] and [day].
 - ii) If *SD* is a localtime type, the character string includes the time scale components: [hour], [min] and [sec], but does not include the 'T' time designator.
 - iii) If *SD* is a time type, the character string includes the time scale components: [hour], [min], [sec], and [shift], but does not include the 'T' time designator.

- iv) If SD is a localdatetime type, the character string includes the time scale components: [year], [month], [day], [hour], [min] and [sec].
- v) If SD is a datetime type, the character string includes the time scale components: [year], [month], [day], [hour], [min], [sec], and [shift].

Case:

- i) If Y contains a code point that is a noncharacter in the character repertoire of UCS then an exception condition is raised: *data exception — non-character in character string (22029)*.
- ii) If the length in characters LY of Y is equal to LTD , then TV is Y .
- iii) If the length in characters LY of Y is less than LTD , then TV is Y extended on the right by $LTD-LY$ <space>s.
- iv) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

- e) If SD is a Boolean type, then

Case:

- i) If SV is *True* and LTD is not less than 4, then TV is 'TRUE' extended on the right by $LTD-4$ <space>s.
- ii) If SV is *False* and LTD is not less than 5, then TV is 'FALSE' extended on the right by $LTD-5$ <space>s.
- iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast (22018)*.

- 7) If TD is a variable-length character string type, then let $MLTD$ be the maximum length in characters of TD .

Case:

- a) If SD is an exact numeric type, then:

- i) Let YP be the shortest character string that conforms to the definition of <exact numeric literal> in Subclause 21.1, “<literal>”, whose scale is the same as the scale of SD , whose interpreted value is the absolute value of SV , and that is not an <unsigned hexadecimal integer>.
- ii) Case:
 - 1) If SV is less than 0 (zero), then let Y be the result of ' - ' | YP .
 - 2) Otherwise, let Y be YP .
- iii) Case:
 - 1) If Y contains code point that is a noncharacter in the character repertoire of UCS then an exception condition is raised: *data exception — non-character in character string (22029)*.
 - 2) If the length in characters LY of Y is less than or equal to $MLTD$, then TV is Y .
 - 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

- b) If SD is an approximate numeric type, then

- i) Let YP be a character string as follows.

Case:

- 1) If SV equals 0 (zero), then YP is '0E0'.
- 2) Otherwise, YP is the shortest character string that conforms to the definition of <approximate numeric literal> in Subclause 21.1, "<literal>", whose interpreted value is equal to the absolute value of SV and whose <mantissa> consists of a single <digit> that is not '0', followed by a <period> and an <unsigned integer>.

- ii) Case:

- 1) If SV is less than 0 (zero), then let Y be the result of ' $-$ ' || YP .
- 2) Otherwise, let Y be YP .

- iii) Case:

- 1) If Y contains code point that is a noncharacter in the character repertoire of UCS then an exception condition is raised: *data exception — non-character in character string (22029)*.
- 2) If the length in characters LY of Y is less than or equal to $MLTD$, then TV is Y .
- 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation (22001)*.

- c) If SD is a fixed-length character string type, or a variable-length character string type, then

Case:

- i) If the length in characters of SV is less than or equal to $MLTD$, then TV is SV .
- ii) If the length in characters of SV is larger than $MLTD$, then TV is the first $MLTD$ characters of SV . If any of the remaining characters of SV are non-<whitespace> characters, then a completion condition is raised: *warning — string data, right truncation (01004)*.

- d) If SD is a temporal instant type or a duration type, then

- i) If SD is a temporal instant type or a duration type, then let YS be the shortest character string that conforms to the definition of <literal> in Subclause 21.1, "<literal>", and such that the interpreted value of YS is SV , and such that

Case:

- 1) If SD is a date type, the character string includes the time scale components: [year], [month] and [day].
- 2) If SD is a localtime type, the character string includes the time scale components: [hour], [min] and [sec], but does not include the 'T' time designator.
- 3) If SD is a time type, the character string includes the time scale components: [hour], [min], [sec], and [shift], but does not include the 'T' time designator.
- 4) If SD is a localdatetime type, the character string includes the time scale components: [year], [month], [day], [hour], [min] and [sec].
- 5) If SD is a datetime type, the character string includes the time scale components: [year], [month], [day], [hour], [min], [sec], and [shift].

- ii) Case:

- 1) If Y contains a code point that is a noncharacter in the character repertoire of UCS then an exception condition is raised: *data exception — non-character in character string* (22029).
 - 2) If the length in characters LY of Y is less than or equal to $MLTD$, then TV is Y .
 - 3) Otherwise, an exception condition is raised: *data exception — string data, right truncation* (22001).
- e) If SD is a Boolean type, then
- Case:
- i) If SV is *True* and $MLTD$ is not less than 4, then TV is 'TRUE'.
 - ii) If SV is *False* and $MLTD$ is not less than 5, then TV is 'FALSE'.
 - iii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast* (22018).
- 8) If TD is the date type, then
- Case:
- a) If SD is character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.
- Case:
- i) If the rules for <literal> in Subclause 21.1, “<literal>”, can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
 - ii) Otherwise, an exception condition is raised: *data exception — invalid datetime format* (22007).
- b) If SD is the date type, then TV is SV .
 - c) If SD is the datetime type or localdatetime type, then let D be the result of $\text{SUBSTRING}(\text{CAST}(VE \text{ AS } STRING), 1, 8)$. TV is the result of $\text{DATE}(DS)$.
- 9) Let STZ be the time zone identifier of the current session context. If STZ is 'Z', then let $STZD$ be "DURATION ('P0000')"; otherwise let $STZD$ be "DURATION ('P' || $\text{SUBSTRING}(STZ, 1, 3)$ || 'H' || $\text{SUBSTRING}(STZ, 4, 2)$ || 'M')".
- 10) If TD is the localtime type, then
- Case:
- a) If SD is a character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.
- Case:
- i) If the rules for <literal> in Subclause 21.1, “<literal>”, can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
 - ii) Otherwise, an exception condition is raised: *data exception — invalid datetime format* (22007).
- b) If SD is a localtime type, then TV is SV .
 - c) If SD is a time type, then:

20.29 <cast specification>

- i) Let $TIME$ be the result of `CAST (VE AS STRING)`. Let TL be the result of `CHARACTER_LENGTH (TIME)`. If `SUBSTRING (TIME, TL - 1) = 'Z'`, then $TIME$ is replaced by `SUBSTRING (TIME, 1, TL-1) || '+0000'`. Let TZ be the result of `SUBSTRING (TIME, TL - 4)`. Let TZD be "`DURATION ('P' || SUBSTRING (TZ, 1, 3) || 'H' || SUBSTRING (TZ, 4, 2) || 'M')`". Let T be the result of `SUBSTRING (TIME, 1, TL - 5)`.
- ii) TV is the result of `LOCALTIME (T) + TZD`.
- d) If SD is `localdatetime` type, then
 - i) Let T be the result of `SUBSTRING (CAST (VE AS STRING), 10)`.
 - ii) TV is the result of `LOCALTIME (T)`.
- e) If SD is `datetime` type, then
 - i) Let $TIME$ be the result of `SUBSTRING (CAST (VE AS STRING), 10)`. Let TL be the result of `CHARACTER_LENGTH (TIME)`. If `SUBSTRING (TIME, TL - 1) = 'Z'`, then $TIME$ is replaced by `SUBSTRING (TIME, 1, TL-1) || '+0000'`. Let TZ be the result of `SUBSTRING (TIME, TL - 4)`. Let TZD be "`DURATION ('P' || SUBSTRING (TZ, 1, 3) || 'H' || SUBSTRING (TZ, 4, 2) || 'M')`". Let T be the result of `SUBSTRING (TIME, 1, TL - 5)`.
 - ii) TV is the result of `LOCALTIME (T) + TZD`.

11) If TD is the `time` type, then

Case:

- a) If SD is a character string type, then SV is replaced by SV with any leading or trailing `<whitespace>` removed.

Case:

- i) If the rules for `<literal>` in Subclause 21.1, "`<literal>`", can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
- ii) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.
- b) If SD is a `time` type, then TV is SV .
- c) If SD is a `localtime` type, then:
 - i) Let T be the result of `CAST (VE AS STRING)`.
 - ii) TV is the result of `TIME (T || STZ)`.
- d) If SD is `datetime` type, then
 - i) Let T be the result of `SUBSTRING (CAST (VE AS STRING), 10)`.
 - ii) TV is the result of `TIME (T)`.
- e) If SD is `localdatetime` type, then
 - i) Let T be the result of `SUBSTRING (CAST (VE AS STRING), 10)`.
 - ii) TV is the result of `TIME (T || STZ)`.

12) If TD is the `localdatetime` type, then

Case:

- a) If SD is a character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.

Case:

- i) If the rules for <literal> in Subclause 21.1, “<literal>”, can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
- ii) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.
- b) If SD is a date type, then let D be the result of $\text{CAST} (VE \text{ AS STRING })$. TV is the result of $\text{LOC-ALDATETIME} (D \parallel 'T000000')$.
- c) If SD is a localtime type, then
 - i) Let CD be the result of $\text{CAST} (\text{DATE}() \text{ AS STRING })$.
 - ii) Let T be the result of $\text{CAST} (VE \text{ AS STRING })$.
 - iii) TV is the result of $\text{LOCALDATETIME} (CD \parallel 'T' \parallel T)$.
- d) If SD is a time type, then:
 - i) Let TZ be the result of $\text{SUBSTRING} (\text{CAST} (VE \text{ AS STRING }), 6)$. If TZ is ‘Z’, then let TZD be “ $\text{DURATION} ('P0000')$; otherwise let TZD be “ $\text{DURATION} ('P' \parallel \text{SUBSTRING} (TZ, 1, 3) \parallel 'H' \parallel \text{SUBSTRING} (TZ, 4, 2) \parallel 'M')$ ”.
 - ii) Let T be the result of $\text{SUBSTRING} (\text{CAST} (VE \text{ AS STRING }), 1, 6)$.
 - iii) Let CD be the result of $\text{CAST} (\text{DATE}() \text{ AS STRING })$.
 - iv) Let TA be the result of $\text{CAST} (\text{LOCALTIME} (T) + TZD \text{ AS STRING })$.
 - v) TV is the result of $\text{LOCALDATETIME} (CD \parallel 'T' \parallel TA)$.
- e) If SD is a localdatetime type, then TV is SV .
- f) If SD is datetime type, then
 - i) Let TZ be the result of $\text{SUBSTRING} (\text{CAST} (VE \text{ AS STRING }), 16)$. If TZ is ‘Z’, then let TZD be $\text{DURATION} ('P0000')$; otherwise let TZD be $\text{DURATION} ('P' \parallel \text{SUBSTRING} (TZ, 1, 3) \parallel 'H' \parallel \text{SUBSTRING} (TZ, 4, 2) \parallel 'M')$.
 - ii) Let DT be the result of $\text{SUBSTRING} (\text{CAST} (VE \text{ AS STRING }), 1, 15)$.
 - iii) TV is the result of $\text{LOCALDATETIME} (DT) + TZD$.

- 13) If TD is the datetime type, then

Case:

- a) If SD is a character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.

Case:

- i) If the rules for <literal> in Subclause 21.1, “<literal>”, can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
- ii) Otherwise, an exception condition is raised: *data exception — invalid datetime format (22007)*.

20.29 <cast specification>

- b) If SD is a date type, then let D be the result of $\text{CAST} (VE \text{ AS } \text{STRING})$. TV is the result of $\text{DATETIME} (D \mid\mid 'T000000' \mid\mid STZ)$.
- c) If SD is a localtime type, then
 - i) Let CD be the result of $\text{CAST} (\text{DATE}() \text{ AS } \text{STRING})$.
 - ii) Let T be the result of $\text{CAST} (VE \text{ AS } \text{STRING})$.
 - iii) TV is the result of $\text{DATETIME} (CD \mid\mid 'T' \mid\mid T \mid\mid STZ)$.
- d) If SD is a time type, then:
 - i) Let CD be the result of $\text{CAST} (\text{DATE}() \text{ AS } \text{STRING})$.
 - ii) Let T be the result of $\text{CAST} (VE \text{ AS } \text{STRING})$.
 - iii) TV is the result of $\text{DATETIME} (CD \mid\mid 'T' \mid\mid T)$.
- e) If SD is localdatetime type, then
 - i) Let DT be the result of $\text{CAST} (VE \text{ AS } \text{STRING})$.
 - ii) TV is the result of $\text{DATETIME} (DT \mid\mid STZ)$.
- f) If SD is a datetime type, then TV is SV .

14) If TD is a duration type, then

Case:

- a) If SD is a character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.

Case:

- i) If the rules for <literal> in Subclause 21.1, “<literal>”, can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
- ii) Otherwise, an exception condition is raised: *data exception — invalid duration format (22G0H)*.

- b) If SD is a duration type, then TV is SV .

15) If TD is Boolean type, then

Case:

- a) If SD is a character string type, then SV is replaced by SV with any leading or trailing <whitespace> removed.

Case:

- i) If the rules for <literal> in Subclause 21.1, “<literal>”, can be applied to SV to determine a valid value of the value type TD , then let TV be that value.
- ii) Otherwise, an exception condition is raised: *data exception — invalid character value for cast (22018)*.

- b) If SD is a Boolean type, then TV is SV .

16) If TD and SD are byte string types, then

- a) If TD is a byte string type, then let MINLTD be the minimum length in bytes of TD and let MAXLTD be the maximum length in bytes of TD .

b) Case:

- i) If the length in bytes of SV is equal to LTD , then TV is SV .
- ii) If the length in bytes of SV is larger than $MAXLTD$, then TV is the first $MAXLTD$ bytes of SV and a completion condition is raised: *warning — string data, right truncation (01004)*.
- iii) If the length in bytes M of SV is smaller than $MINLTD$, then TV is SV extended on the right by $MINLTD-M$ X'00's.

17) The result of CS is TV .

Conformance Rules

None.

20.30 <element_id function>

Function

Generate a unique identifier for a graph element.

Format

```
<element_id function> ::=  
ELEMENT_ID <left paren> <element reference> <right paren>
```

Syntax Rules

- 1) The <element reference> shall have singleton degree of reference.
- 2) The declared type of <element_id function> is an implementation-defined type that is permitted as the declared type of an operand of an equality operation according to the Syntax Rules of Subclause 22.6, “Equality operations”.

Access Rules

- 1) Access Rules for <element_id function> are implementation-defined.

General Rules

- 1) Let LOE be the list of graph elements bound to the <element reference>.
- 2) Case:
 - a) If LOE is empty, then the value of <element_id function> is the null value.
 - b) Otherwise, let GE be the sole graph element in LOE . The value of <element_id function> is an implementation-dependent value that encapsulates the identity of GE in the graph that contains GE for the duration of the innermost executing statement.

Conformance Rules

- 1) Without Feature G100, “ELEMENT_ID function”, conforming GQL language shall not contain an <element_id function>.

21 Lexical elements

21.1 <literal>

Function

Specify a value.

Format

```

<literal> ::= 
    <signed numeric literal>
  | <general literal>

<general literal> ::= 
    <predefined type literal>
  | <list literal>
  | <set literal>
  | <multiset literal>
  | <ordered set literal>
  | <map literal>
  | <record literal>

<predefined type literal> ::= 
    <boolean literal>
  | <character string literal>
  | <byte string literal>
  | <temporal literal>
  | <duration literal>
  | <null literal>

<unsigned literal> ::= 
    <unsigned numeric literal>
  | <general literal>

<boolean literal> ::= 
    TRUE | FALSE | UNKNOWN

<character string literal> ::= 
    <single quoted character sequence>
  | <double quoted character sequence>

<unbroken character string literal> ::= 
    <unbroken single quoted character sequence>
  | <unbroken double quoted character sequence>

<single quoted character sequence> ::= 
    <unbroken single quoted character sequence>
      [ { <separator> <unbroken single quoted character sequence> }... ] 

<double quoted character sequence> ::= 
    <unbroken double quoted character sequence>
      [ { <separator> <unbroken double quoted character sequence> }... ] 

<unbroken single quoted character sequence> ::= 
    <quote> [ <single quoted character representation>... ] <quote>
  
```

21.1 <literal>

```

<unbroken double quoted character sequence> ::= 
  <double quote> [ <double quoted character representation>... ] <double quote>

<unbroken accent quoted character sequence> ::= 
  <grave accent> [ <accent quoted character representation>... ] <grave accent>

<single quoted character representation> ::= 
  <character representation>
  !! See the Syntax Rules.

<double quoted character representation> ::= 
  <character representation>
  !! See the Syntax Rules.

<accent quoted character representation> ::= 
  <character representation>
  !! See the Syntax Rules.

<character representation> ::= 
  <string literal character>
  | <escaped character>

<string literal character> ::= 
  !! See the Syntax Rules.

<escaped character> ::= 
  <escaped reverse solidus>
  | <escaped quote>
  | <escaped double quote>
  | <escaped tab>
  | <escaped backspace>
  | <escaped newline>
  | <escaped carriage return>
  | <escaped form feed>
  | <unicode escape value>

<escaped reverse solidus> ::= 
  <reverse solidus> <reverse solidus>

<escaped quote> ::= 
  <reverse solidus> <quote>

<escaped double quote> ::= 
  <reverse solidus> <double quote>

<escaped tab> ::= 
  <reverse solidus> t

<escaped backspace> ::= 
  <reverse solidus> b

<escaped newline> ::= 
  <reverse solidus> n

<escaped carriage return> ::= 
  <reverse solidus> r

<escaped form feed> ::= 
  <reverse solidus> f

<unicode escape value> ::= 
  <unicode 4 digit escape value>
  | <unicode 6 digit escape value>

<unicode 4 digit escape value> ::= 

```

```
<reverse solidus> u <hex digit> <hex digit> <hex digit> <hex digit>
<unicode 6 digit escape value> ::= 
  <reverse solidus> U <hex digit> <hex digit> <hex digit> <hex digit> 
  <hex digit>

<byte string literal> ::=
  X <quote> [ <space>... ] [ { <hex digit> [ <space>... ] <hex digit> [ <space>... ] }...
    ] <quote>
    [ { <separator> <quote> [ <space>... ] [ { <hex digit> [ <space>... ]
      <hex digit> [ <space>... ] }... ] <quote> }... ]

<numeric literal> ::=
  <signed numeric literal>
  | <unsigned numeric literal>

<signed numeric literal> ::=
  [ <sign> ] <unsigned numeric literal>

<unsigned numeric literal> ::=
  <exact numeric literal>
  | <approximate numeric literal>

<exact numeric literal> ::=
  <unsigned integer>
  | <unsigned decimal integer> [ <period> [ <unsigned decimal integer> ] ]
  | <period> <unsigned decimal integer>

<sign> ::=
  <plus sign>
  | <minus sign>

<unsigned integer> ::=
  <unsigned decimal integer>
  | <unsigned hexadecimal integer>
  | <unsigned octal integer>
  | <unsigned binary integer>

<unsigned decimal integer> ::=
  <digit> [ { [ <underscore> ] <digit> }... ]

<unsigned hexadecimal integer> ::=
  0x { [ <underscore> ] <hex digit> }...

<unsigned octal integer> ::=
  0o { [ <underscore> ] <octal digit> }...

<unsigned binary integer> ::=
  0b { [ <underscore> ] <binary digit> }...

<signed decimal integer> ::=
  [ <sign> ] <unsigned decimal integer>

<approximate numeric literal> ::=
  <mantissa> E <exponent>

<mantissa> ::=
  <exact numeric literal>

<exponent> ::=
  <signed decimal integer>

<temporal literal> ::=
  <date literal>
  | <time literal>
```

21.1 <literal>

```

| <datetime literal>

<date literal> ::= DATE <date string>

<time literal> ::= TIME <time string>

<datetime literal> ::= { DATETIME | TIMESTAMP } <datetime string>

<date string> ::= <unbroken character string literal>

<time string> ::= <unbroken character string literal>

<datetime string> ::= <unbroken character string literal>

<duration literal> ::= DURATION <duration string>
| <SQL-interval literal>

<duration string> ::= <unbroken character string literal>

<SQL-interval literal> ::= !! See the Syntax Rules.

<null literal> ::= NULL

<list literal> ::= <list value constructor by enumeration>

<set literal> ::= <set value constructor by enumeration>

<mset literal> ::= <mset value constructor by enumeration>

<ordered set literal> ::= <ordered set value constructor by enumeration>

<map literal> ::= <map value constructor by enumeration>

<record literal> ::= <record value constructor by enumeration>

```

Syntax Rules

- 1) An <unsigned decimal integer> that immediately contains <underscore>s is equivalent to the same <unsigned decimal integer> with every <underscore> removed.
- 2) An <unsigned hexadecimal integer> that immediately contains <underscore>s is equivalent to the same <unsigned hexadecimal integer> with every <underscore> removed.
- 3) An <unsigned octal integer> that immediately contains <underscore>s is equivalent to the same <unsigned octal integer> with every <underscore> removed.

- 4) An <unsigned binary integer> that immediately contains <underscore>s is equivalent to the same <unsigned binary integer> with every <underscore> removed.
- 5) An <unsigned hexadecimal integer> is equivalent to an <unsigned decimal integer> containing a series of <digit>s that represent the same integer value as the series of <hex digit>s.
- 6) An <unsigned octal integer> is equivalent to an <unsigned decimal integer> containing a series of <digit>s that represent the same integer value as the series of <octal digit>s.
- 7) An <unsigned binary integer> is equivalent to an <unsigned decimal integer> containing a series of <digit>s that represent the same integer value as the series of <binary digit>s.
- 8) A <mantissa> shall not contain an <unsigned integer> that is not an <unsigned decimal integer>.
- 9) The maximum number of <digit>s immediately contained in an <unsigned integer> is implementation-defined but shall not be less than 9.
- 10) An <exact numeric literal> without a <period> has an implicit <period> following the last <digit>.
- 11) The declared type of an <exact numeric literal> *ENL* is an exact numeric type whose scale is the number of <digit>s to the right of the <period>. There shall be an exact numeric type capable of representing the value of *ENL* exactly.
- 12) The declared type of an <approximate numeric literal> *ANL* is an implementation-defined approximate numeric type. The value of *ANL* shall not be greater than the maximum value nor less than the minimum value that can be represented by the approximate numeric type.
- 13) The declared type of a <boolean literal> is the Boolean type.
- 14) The <character string literal> specifies the character string specified by the <single quoted character sequence> or the <double quoted character sequence> that it contains.
- 15) The <unbroken character string literal> specifies the character string specified by the <unbroken single quoted character sequence> or the <unbroken double quoted character sequence> that it contains.
- 16) The <single quoted character sequence> specifies the character string comprising the quote-separated concatenation of the character strings specified by the <unbroken single quoted character sequence>s that it contains.
- 17) The <double quoted character sequence> specifies the character string comprising the double quote-separated concatenation of the character strings specified by the <unbroken double quoted character sequence>s that it contains.
- 18) The <unbroken single quoted character sequence> specifies the character string comprising the sequence of characters defined by the <single quoted character representation>s that it contains.
- 19) The <unbroken double quoted character sequence> specifies the character string comprising the sequence of characters defined by the <double quoted character representation>s that it contains.
- 20) The <unbroken accent quoted character sequence> specifies the character string comprising the sequence of characters defined by the <accent quoted character representation>s that it contains.
- 21) The <single quoted character representation> shall not be a <quote> or a <reverse solidus> unless either of these occurs as part of an <escaped character>.
- 22) The <double quoted character representation> shall not be a <double quote> or a <reverse solidus> unless either of these occurs as part of an <escaped character>.
- 23) The <accent quoted character representation> shall not be a <grave accent> or a <reverse solidus> unless either of these occurs as part of an <escaped character>.

21.1 <literal>

- 24) A <string literal character> is any character except for those occurring as part of an <escaped character>.
- 25) In an <escaped character> each <escaped reverse solidus> represents a <reverse solidus> character.
- 26) In an <escaped character> each <escaped quote> represents a <quote> character.
- 27) In an <escaped character> each <escaped double quote> represents a <double quote> character.
- 28) In an <escaped character> each <escaped tab> represents the Unicode character identified by the code point \u0009.
- 29) In an <escaped character> each <escaped backspace> represents the Unicode character identified by the code point \u0008.
- 30) In an <escaped character> each <escaped newline> represents the Unicode character identified by the code point u\000A.
- 31) In an <escaped character> each <escaped carriage return> represents the Unicode character identified by the code point u\000D.
- 32) In an <escaped character> each <escaped form feed> represents the Unicode character identified by the code point u\000C.
- 33) In an <escaped character> each <unicode escape value> represents the Unicode character identified by the code point.
- 34) The declared type of a <character string literal> is character string.
- 35) In a <byte string literal>, the sequence

```
<quote> [ <space>... ] { <hex digit> [ <space>... ]
<hex digit> [ <space>... ] }... <quote>
```

is equivalent to the sequence

```
<quote> { <hex digit> <hex digit> }... <quote>
```

NOTE 144 — The <hex digit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <byte string literal>.

- 36) In a <byte string literal>, the sequence

```
<quote> { <hex digit> <hex digit> }... <quote> <separator>
<quote> { <hex digit> <hex digit> }... <quote>
```

is equivalent to the sequence

```
<quote> { <hex digit> <hex digit> }... { <hex digit> <hex digit> }... <quote>
```

NOTE 145 — The <hex digit>s in the equivalent sequence are in the same sequence and relative sequence as in the original <byte string literal>.

- 37) In a <byte string literal>, the introductory 'X' may be represented either in upper-case (as 'X') or in lower-case (as 'x').
- 38) In a <character string literal>, or <byte string literal>, a <separator> shall contain a <newline>.
- 39) The declared type of a <byte string literal> is a byte string type. Each <hex digit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hex digit>s a, b, c, d, e, and f have respectively the same values as the <hex digit>s A, B, C, D, E, and F.

- 40) The character string specified by the <unbroken character string literal> immediately contained in <date string> shall conform to the representation specified in clause 5.2 “Date” of ISO 8601-1:2019 as extended by clauses 4.3 “Additional explicit forms” and 4.4 “Numerical extensions” of ISO 8601-2:2019.
- 41) The declared type of <date literal> is DATE.
- 42) The character string specified by the <unbroken character string literal> immediately contained in <time string> shall conform to the representation specified in clause 5.3 “Time of day” of ISO 8601-1:2019 as extended by clauses 4.3 “Additional explicit forms” and 4.4 “Numerical extensions” of ISO 8601-2:2019.
- 43) If the <time string> does not contain a representation of a time shift then the declared type of <time literal> is LOCALTIME, otherwise the declared type of <time literal> is TIME.
- 44) The character string specified by the <unbroken character string literal> immediately contained in <datetime string> shall conform to the representation specified in clause 5.4 “Date and time of day” of ISO 8601-1:2019 as extended by clauses 4.3 “Additional explicit forms” and 4.4 “Numerical extensions” of ISO 8601-2:2019 with the optional addition of a IANA Time Zone Database time zone designator enclosed between a <left bracket> and a <right bracket>. If both an IANA Time Zone Database time zone designator and a ISO 8601-1:2019 time shift specification are present then the IANA Time Zone Database time zone designator shall resolve to the same value as the ISO 8601-1:2019 time shift specification.
- 45) If the <datetime string> does not contain a representation of a time shift then the declared type of <datetime literal> is LOCALDATETIME, otherwise the declared type of <datetime literal> is DATETIME.
- 46) The declared type of <duration literal> is DURATION.
- 47) The declared type of <null literal> is NULL.
- 48) The <unbroken character string literal> immediately contained in <duration string> shall conform to the representation specified in clause 5.5.2 “Duration” of ISO 8601-1:2019 as extended by clauses 4.3 “Additional explicit forms” and 4.4 “Numerical extensions” of ISO 8601-2:2019.
- 49) <SQL-interval literal> shall conform to the Syntax Rules of <interval literal> in ISO/IEC 9075-2:202x.
- 50) If <SQL-interval literal> is specified, then:
 - If <SQL-interval literal> contains a <years value> y then let Y be 'Yy', otherwise let Y be the zero-length character string.
 - If <SQL-interval literal> contains a <months value> m then let M be 'Mm', otherwise let M be the zero-length character string.
 - If <SQL-interval literal> contains a <days value> d then let D be 'Dd', otherwise let D be the zero-length character string.
 - If <SQL-interval literal> contains an <hours value> h then let H be 'Hh', otherwise let H be the zero-length character string.
 - If <SQL-interval literal> contains a <minutes value> mn then let MN be 'Mmn', otherwise let MN be the zero-length character string.
 - If <SQL-interval literal> contains a <seconds value> s then let S be 'Ss', otherwise let S be the zero-length character string.
 - If <SQL-interval literal> contains an <hours value>, <minutes value>, or <seconds value> then let T be 'T', otherwise let T be the zero-length character string.

21.1 <literal>

- If <SQL-interval literal> contains at most one <minus sign>, then let SN be '-', otherwise let SN be the zero-length character string.

<SQL-interval literal> it is equivalent to the <duration literal>:

DURATION 'SNPYMDTHMNS'

- 51) Every <value expression> contained in a <list element> of the <list element list> of the <list value constructor by enumeration> contained in a <list literal> shall be a <literal>.
- 52) The declared type of <list literal> is the declared type of the immediately contained <list value constructor by enumeration>.
- 53) Every <value expression> contained in a <set element> of the <set element list> of the <set value constructor> contained in a <set literal> shall be a <literal>.
- 54) The declared type of <set literal> is the declared type of the immediately contained <set value constructor by enumeration>.
- 55) Every <value expression> contained in a <multiset element> of the <multiset element list> of the <multiset value constructor by enumeration> contained in a <multiset literal> shall be a <literal>.
- 56) The declared type of <multiset literal> is the declared type of the immediately contained <multiset value constructor by enumeration>.
- 57) Every <value expression> contained in an <ordered set element> of the <ordered set element list> of the <ordered set value constructor by enumeration> contained in an <ordered set literal> shall be a <literal>.
- 58) The declared type of <ordered set literal> is the declared type of the immediately contained <ordered set value constructor by enumeration>.
- 59) Every <value expression> contained in a <map element> of the <map element list> of the <map value constructor by enumeration> contained in a <map literal> shall be a <literal>.
- 60) The declared type of <map literal> is the declared type of the immediately contained <map value constructor by enumeration>.
- 61) Every <value expression> contained in a <field> of the <field list> of the <record value constructor by enumeration> contained in a <record literal> shall be a <literal>.
- 62) The declared type of <record literal> is the declared type of the immediately contained <record value constructor by enumeration>.

General Rules

- 1) Except when it is contained in an <exact numeric literal>, the value of an <unsigned integer> is the numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the series of <digit>s that constitutes the <unsigned integer>.
- 2) The value of an <exact numeric literal> is the numeric value determined by application of the normal mathematical interpretation of positional decimal notation to the source characters that constitute the <exact numeric literal>.
- 3) Let ANL be an <approximate numeric literal>. Let $ANDT$ be the declared type of ANL . Let ANV be the product of the exact numeric value represented by the <mantissa> of ANL and the number obtained by raising the number 10 to the power of the exact numeric value represented by the <exponent> of ANL . If ANV is a value of $ANDT$, then the value of ANL is ANV ; otherwise, the value of ANL is a value of $ANDT$ obtained from ANV by rounding or truncation. The choice of whether to round or truncate is implementation-defined.

- 4) The <sign> in a <signed numeric literal> is a monadic arithmetic operator. The monadic arithmetic operators + and – specify monadic plus and monadic minus, respectively. If neither monadic plus nor monadic minus are specified in a <signed numeric literal>, or if monadic plus is specified, then the literal is positive. If monadic minus is specified in a <signed numeric literal>, then the literal is negative.
- 5) The truth value of a <boolean literal> is *True* if TRUE is specified, is *False* if FALSE is specified, and is *Unknown* if UNKNOWN is specified.
- 6) The value of a <character string literal> is the character string that it specifies.
- 7) The value of an <unbroken character string literal> is the character string that it specifies.
- 8) The value of a <byte string literal> is the byte string comprising sequence of bits defined by the <hex digit>s that it contains. Each <hex digit> appearing in the literal is equivalent to a quartet of bits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are interpreted as 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111, respectively. The <hex digit>s a, b, c, d, e, and f have respectively the same values as the <hex digit>s A, B, C, D, E, and F.
- 9) If <date string> contains a representation with reduced precision then the lowest valid value for each of the omitted lower order time scale components is implicit.
- 10) The value of a <date literal> is a calendar date in the Gregorian calendar.
- 11) If <time string> contains a representation with reduced precision then value for each of the omitted lower order time scale components is 0 (zero).
- 12) The value of a <time literal> is a time of day. If the <time string> contained a representation of a time shift then the time shift information is preserved.
- 13) If <datetime string> contains a representation with reduced precision then value for each of the omitted lower order time scale components is 0 (zero).
- 14) The value of a <datetime literal> is a time. If the <datetime string> contained a representation of a time shift then the time shift information is preserved.
- 15) The value of a <duration literal> is a duration or a negative duration.

NOTE 146 — See clause 4.4.1.9 Duration of ISO 8601-2:2019 for the definition of a negative duration.

- 16) The value of a <null literal> is the null value.
- 17) The value of <list literal> is the value of the immediately contained <list value constructor by enumeration>.
- 18) The value of <set literal> is the value of the immediately contained <set value constructor by enumeration>.
- 19) The value of <multipset literal> is the value of the immediately contained <multipset value constructor by enumeration>.
- 20) The value of <ordered set literal> is the value of the immediately contained <ordered set value constructor by enumeration>.
- 21) The value of <map literal> is the value of the immediately contained <map value constructor by enumeration>.
- 22) The value of <record literal> is the value of the immediately contained <record value constructor by enumeration>.

Conformance Rules

None.

21.2 <value type>

Function

Specify a value type.

Format

```
<value type> ::=  
    ANY  
    | <predefined type>  
    | <graph element type>  
    | <collection type>  
    | <map value type>  
    | <record value type>  
    | <graph type expression>  
    | <binding table type expression>  
    | NOTHING  
  
<of value type> ::=  
    [ <of type prefix> ] <value type>  
  
<of type prefix> ::=  
    <double colon> | OF  
  
<predefined type> ::=  
    <boolean type>  
    | <character string type>  
    | <byte string type>  
    | <numeric type>  
    | <temporal type>  
  
<boolean type> ::=  
    BOOL | BOOLEAN  
  
<character string type> ::=  
    { STRING | VARCHAR } [ <left paren> <max length> <right paren> ]  
  
<byte string type> ::=  
    BYTES [ <left paren> [ <min length> <comma> ] <max length> <right paren> ]  
    | BINARY [ <fixed length> ]  
    | VARBINARY [ <max length> ]  
  
<min length> ::=  
    <unsigned decimal integer>  
  
<max length> ::=  
    <unsigned decimal integer>  
  
<fixed length> ::=  
    <unsigned decimal integer>  
  
<numeric type> ::=  
    <exact numeric type>  
    | <approximate numeric type>  
  
<exact numeric type> ::=  
    <binary exact numeric type>  
    | <decimal exact numeric type>  
  
<binary exact numeric type> ::=
```

21.2 <value type>

```

<binary exact signed numeric type>
| <binary exact unsigned numeric type>

<binary exact signed numeric type> ::=

    INT8
    | INT16
    | INT32
    | INT64
    | INT128
    | INT256
    | SMALLINT
    | INT [ <left paren> <precision> <right paren> ]
    | BIGINT
    | [ SIGNED ] <verbose binary exact numeric type>

<binary exact unsigned numeric type> ::=

    UINT8
    | UINT16
    | UINT32
    | UINT64
    | UINT128
    | UINT256
    | UINT [ <left paren> <precision> <right paren> ]
    | UNSIGNED <verbose binary exact numeric type>

<verbose binary exact numeric type> ::=

    INTEGER8
    | INTEGER16
    | INTEGER32
    | INTEGER64
    | INTEGER128
    | INTEGER256
    | INTEGER [ <left paren> <precision> <right paren> ]

<decimal exact numeric type> ::=
{ DECIMAL | DEC } <left paren> <precision> [ <comma> <scale> ] <right paren>

<precision> ::=
<unsigned decimal integer>

<scale> ::=
<unsigned decimal integer>

<approximate numeric type> ::=

    FLOAT16
    | FLOAT32
    | FLOAT64
    | FLOAT128
    | FLOAT128
    | FLOAT [ <left paren> <precision> [ <comma> <scale> ] <right paren> ]
    | REAL
    | DOUBLE [ PRECISION ]

<temporal type> ::=
    DATETIME
    | LOCALDATETIME
    | DATE
    | TIME
    | LOCALTIME
    | DURATION

<graph element type> ::=
    NODE

```

```

| VERTEX
| EDGE
| RELATIONSHIP

<collection type> ::= 
    <list value type>
    | <multiset value type>
    | <set value type>
    | <ordered set value type>

<list value type> ::= 
    <value type> <list value type name>

<list value type name> ::= 
    LIST | ARRAY

<multiset value type> ::= 
    <value type> MULTISET

<set value type> ::= 
    <value type> SET

<ordered set value type> ::= 
    <value type> ORDERED SET

<map value type> ::= 
    MAP <left angle bracket> <map key type> <comma> <value type> <right angle bracket>

<map key type> ::= 
    <predefined type>

<record value type> ::= 
    [ RECORD ] <left brace> [ <field type list> ] <right brace>

<field type list> ::= 
    <field type> [ { <comma> <field type> }... ]

<field type> ::= 
    <field name> [ <of type prefix> ] <value type>

```

Syntax Rules

- 1) The base type of Boolean types is named BOOLEAN DATA. The preferred name of Boolean types is implementation-defined as either BOOLEAN or BOOL.
- 2) Every <boolean type> specifies a *Boolean type*.
- 3) For each Boolean type *BT*, there is a Boolean type *BNF(BT)*, known as the *normal form* of *BT* (which may be *BT* itself), such that the declared name of *BNF(BT)* is the preferred name of Boolean types.
- 4) The value of every <max length> shall be greater than or equal to 1 (one).
- 5) The base type of character string types is named STRING DATA. The preferred name of character string types is implementation-defined as either VARCHAR or STRING.
- 6) Every <character string type> *CST* specifies a *character string type*.
- 7) If the <max length> *CSMAXL* is specified in a <character string type> *CST*, then the maximum length of the character string type specified by *CST* is the value of *CSMAXL*. Otherwise, the maximum length of the character string type specified by *CST* is implementation-defined.

21.2 <value type>

- 8) For each character string type *CST*, there is a character string type *CSNF(CST)*, known as the *normal form* of *CST* (which may be *CST* itself), such that *CSNF(CST)* and *CST* have the same maximum length and the declared name of *CSNF(CST)* is the preferred name of character string types.
- 9) Every character string type with maximum length *CSMAXL* only includes character strings with a length that is less than or equal to *CSMAXL*.
- 10) The maximum length of a character string is implementation-defined but shall be greater than or equal to $2^{14}-1 = 16383$.
- 11) Characters in a character string are numbered beginning with 1 (one).
- 12) The value of every <min length> shall be greater than or equal to 0 (zero).
- 13) If <min length> is omitted, then a <min length> of 0 (zero) is implicit.
- 14) The value of every <fixed length> shall be greater than or equal to 1 (one).
- 15) If <fixed length> is omitted, then a <fixed length> of 1 (one) is implicit.
- 16) The base type of byte string types is named BINARY DATA. The preferred name of fixed-length byte string types is implementation-defined as either BINARY or BYTES. The preferred name of variable-length byte string types is implementation-defined as either VARBINARY or BYTES.
- 17) Every <byte string type> specifies a *byte string type*.
- 18) If both the <min length> *MINL* and the <max length> *MAXL* are specified in a <byte string type>, then *MINL* shall be less than or equal to *MAXL*.
- 19) If the <min length> *BSMINL* is specified in a <byte string type> *BST*, then the minimum length of the byte string type specified by *BST* is the value of *BSMINL*.
- 20) If the <max length> *BSMAXL* is specified in a <byte string type> *BST*, then the maximum length of the byte string type specified by *BST* is the value of *BSMAXL*.
- 21) If the <fixed length> *BSFIXL* is specified in a <byte string type> *BST*, then the minimum length of the byte string type specified by *BST* is the value of *BSFIXL* and the maximum length of the byte string type specified by *BST* is the value of *BSFIXL*.
- 22) If neither the <max length> nor the <fixed length> are specified in a <byte string type> *BST*, then the maximum length of the byte string type specified by *BST* is implementation-defined.
- 23) For each byte string type *BST*, there is a byte string type *BSNF(BST)*, known as the *normal form* of *BST* (which may be *BST* itself), such that *BSNF(BST)* and *BST* have the same minimum length and the same maximum length and

Case:

- a) If *BST* is a fixed-length byte string type, then the declared name of *BSNF(BST)* is the preferred name of fixed-length byte string types.
- b) If *BST* is a variable-length byte string type, then the declared name of *BSNF(BST)* is the preferred name of variable-length byte string types.
- 24) Every byte string type with minimum length *BSMINL* only includes byte strings with a length that is greater than or equal to *BSMINL*.
- 25) Every byte string type with maximum length *BSMAXL* only includes byte strings with a length that is less than or equal to *BSMAXL*.
- 26) The minimum length of every byte string type *BST* shall be less than or equal to the maximum length of *BST*.

- 27) The maximum length of a byte string is implementation-defined but shall be greater than or equal to $2^{16}-2=65534$.
- 28) Bytes in a byte string are numbered beginning with 1 (one).
- 29) The value of every <precision> shall be greater than or equal to 1 (one).
- 30) The base type of exact numeric types with binary precision is named INTEGER DATA. The base type of exact numeric types with decimal precision is named DECIMAL DATA.
- 31) The base type of approximate numeric types is named FLOAT DATA.
- 32) Every <numeric type> specifies the *numeric type* specified by the immediately contained <exact numeric type> or the immediately contained <approximate numeric type>.
- 33) For each exact numeric type *ENT*, there is an implementation-defined exact numeric type, *ENNF(ENT)*, known as the *normal form* of *ENT* (which may be *ENT* itself), such that:
- a) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either SIGNED INTEGER, INTEGER, or INT, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
 - b) For each *p* from the set {16, 32, 64, 128, 256}: If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either SIGNED INTEGER*p*, INTEGER*p*, or INT*p*, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
 - c) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either UNSIGNED INTEGER or UINT, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
 - d) For each *p* from the set {16, 32, 64, 128, 256}: If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either UNSIGNED INTEGER*p* or UINT*p*, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
 - e) If *ENT1* and *ENT2* are exact numeric types whose declared names individually are either DEC or DECIMAL, then *ENNF(ENT1)* is the same as *ENNF(ENT2)*.
 - f) The precision, scale, and radix of *ENNF(ENT)* are the same as the precision, scale, and radix, respectively, of *ENT*.

NOTE 147 — The precision, scale, and radix are determined when an exact numeric type is specified prior to the construction of its descriptor.

- g) *ENNF(ENNF(ENT))* is the same as *ENNF(ENT)*.

- 34) For the <exact numeric type>s:

- a) The maximum value of a <precision> is implementation-defined. <precision> shall not be greater than this value.
- b) The maximum value of a <scale> is implementation-defined. <scale> shall not be greater than this maximum value.

- 35) Every <exact numeric type> specifies the *exact numeric type* specified by the immediately contained <binary exact numeric type> or the immediately contained <decimal exact numeric type>.
- 36) Every <binary exact numeric type> specifies the *binary exact numeric type* specified by the immediately contained <binary exact signed numeric type> or the immediately contained <binary exact unsigned numeric type>.
- 37) Every <binary exact signed numeric type> *BESNT* specifies an *exact signed numeric type with binary precision*.

Case:

- a) If *BESNT* is SIGNED INTEGER8, INTEGER8, or INT8 then *BESNT* specifies the *signed 8-bit integer type* with binary precision of 8 bits and with scale 0 (zero).
 - b) If *BESNT* is SIGNED INTEGER16, INTEGER16, or INT16 then *BESNT* specifies the *signed 16-bit integer type* with binary precision of 16 bits and with scale 0 (zero).
 - c) If *BESNT* is SIGNED INTEGER32, INTEGER32, or INT32 then *BESNT* specifies the *signed 32-bit integer type* with binary precision of 32 bits and with scale 0 (zero).
 - d) If *BESNT* is SIGNED INTEGER64, INTEGER64, or INT64 then *BESNT* specifies the *signed 64-bit integer type* with binary precision of 64 bits and with scale 0 (zero).
 - e) If *BESNT* is SIGNED INTEGER128, INTEGER128, or INT128 then *BESNT* specifies the *signed 128-bit integer type* with binary precision of 128 bits and with scale 0 (zero).
 - f) If *BESNT* is SIGNED INTEGER256, INTEGER256, or INT256 then *BESNT* specifies the *signed 256-bit integer type* with binary precision of 256 bits and with scale 0 (zero).
 - g) If *BESNT* is SIGNED INTEGER, INTEGER, or INT, then *BESNT* specifies the *signed regular integer type* with implementation-defined binary precision greater than or equal to 32 bit and with scale 0 (zero).
 - h) If *BESNT* is SMALLINT, then *BESNT* specifies the *signed small integer type* with implementation-defined binary precision less than or equal to the precision of the signed regular integer type and with scale 0 (zero).
 - i) If *BESNT* is BIGINT, then *BESNT* specifies the *signed big integer type* with implementation-defined precision greater than or equal to the precision of the signed regular integer type and with scale 0 (zero).
 - j) Otherwise, *BESNT* specifies the *signed user-specified integer type* with implementation-defined precision greater than or equal to the value of the <precision> that is immediately contained in *BESNT* as its binary precision in bits and with scale 0 (zero).
- 38) Every <binary exact unsigned numeric type> *BEUNT* specifies an exact unsigned numeric type with binary precision.

Case:

- a) If *BEUNT* is UNSIGNED INTEGER8 or UINT8, then *BEUNT* specifies the *unsigned 8-bit integer type* with binary precision of 8 bits and with scale 0 (zero).
- b) If *BEUNT* is UNSIGNED INTEGER16 or UINT16, then *BEUNT* specifies the *unsigned 16-bit integer type* with binary precision of 16 bits and with scale 0 (zero).
- c) If *BEUNT* is UNSIGNED INTEGER32 or UINT32, then *BEUNT* specifies the *unsigned 32-bit integer type* with binary precision of 32 bits and with scale 0 (zero).
- d) If *BEUNT* is UNSIGNED INTEGER64 or UINT64, then *BEUNT* specifies the *unsigned 64-bit integer type* with binary precision of 64 bits and with scale 0 (zero).
- e) If *BEUNT* is UNSIGNED INTEGER128 or UINT128, then *BEUNT* specifies the *unsigned 128-bit integer type* with binary precision of 128 bits and with scale 0 (zero).
- f) If *BEUNT* is UNSIGNED INTEGER256 or UINT256, then *BEUNT* specifies the *unsigned 256-bit integer type* with binary precision of 256 bits and with scale 0 (zero).
- g) If *BEUNT* is UNSIGNED INTEGER or UINT, then *BEUNT* specifies the *unsigned regular integer type* with implementation-defined binary precision greater than or equal to 32 bits and with scale 0 (zero).

- h) Otherwise, *BEUNT* specifies the *unsigned user-specified integer type* with implementation-defined precision greater than or equal to the value of the <precision> that is immediately contained in *BEUNT* as its binary precision in bits and with scale 0 (zero).
- 39) The value of every <scale> shall be greater than or equal to 0 (zero).
- 40) If an <exact numeric type> *ENT* contains the <scale> *SCALE*, then the value of *SCALE* shall not be greater than the value of the <precision> contained in *ENT*.
- 41) Every <decimal exact numeric type> *DENT* specifies an exact numeric type with decimal precision.
- Case:
- a) If *DENT* immediately contains DECIMAL or DEC and the <precision> *PREC* and the <scale> *SCALE*, then *DENT* specifies the *user-specified decimal exact numeric type* with implementation-defined decimal precision in digits greater than or equal to the decimal precision in digits specified by *PREC* and with implementation-defined decimal scale in digits specified by *SCALE*.
 - b) If *DENT* immediately contains DECIMAL or DEC and the <precision> *PREC* but no <scale>, then *DENT* specifies the *user-specified decimal exact numeric type* with implementation-defined decimal precision in digits greater than or equal to the decimal precision in digits specified by *PREC* and with scale 0 (zero).
 - c) Otherwise, *DENT* specifies the *regular decimal exact numeric type* with implementation-defined decimal precision and with scale 0 (zero).
- 42) The value of every <precision> contained in an <approximate numeric type> shall be equal to or greater than 2.
- NOTE 148 — This accounts for the possible inclusion of a leading bit in the precision describing the size of the mantissa of an approximate numeric value that is implied by but not included in the underlying physical representation.
- 43) For each approximate numeric type *ANT*, there is an implementation-defined approximate numeric type *ANNF(ANT)*, known as the *normal form* of *ANT* (which may be *ANT* itself), such that:
- a) The precision and scale of *ANNF(ANT)* are the same as the precision and scale, respectively, of *ANT*.
- NOTE 149 — The precision and scale are determined when an exact numeric type is specified prior to the construction of its descriptor.
- b) If *ANT1* and *ANT2* are exact numeric types whose declared names individually are either DOUBLE or DOUBLE PRECISION, then *ANNF(ANT1)* is the same as *ANNF(ANT2)*.
 - c) *ANNF(ANNF(ANT))* is the same as *ANNF(ANT)*.
- 44) For the <approximate numeric type>s:
- a) The maximum value of a <precision> is implementation-defined. <precision> shall not be greater than this value.
 - b) The maximum value of a <scale> is implementation-defined. <scale> shall not be greater than this maximum value.
- 45) Every <approximate numeric type> *ANT* specifies an *approximate numeric type*:
- Case:
- a) If *ANT* is FLOAT16, then *ANT* specifies the *16-bit approximate numeric type* with binary precision of 11 bits and with binary scale of 5 bits.
 - b) If *ANT* is FLOAT32, then *ANT* specifies the *32-bit approximate numeric type* with binary precision of 24 bits and with binary scale of 8 bits.

21.2 <value type>

- c) If ANT is FLOAT64, then ANT specifies the *64-bit approximate numeric type* with binary precision of 53 bits and with binary scale of 11 bits.
- d) If ANT is FLOAT128, then ANT specifies the *128-bit approximate numeric type* with binary precision of 113 bits and with binary scale of 15 bits.
- e) If ANT is FLOAT256, then ANT specifies the *256-bit approximate numeric type* with binary precision of 237 bits and with binary scale of 19 bits.
- f) If ANT is FLOAT, then ANT specifies the *regular approximate numeric type* with implementation-defined binary precision greater than or equal to 32 bits and with implementation-defined binary scale.
- g) If ANT is REAL, then ANT specifies the *real approximate numeric type* with an implementation-defined binary precision less than or equal to the precision of the regular approximate numeric type and with implementation-defined binary scale.
- h) If ANT is DOUBLE or DOUBLE PRECISION, then ANT specifies the *double approximate numeric type* with implementation-defined binary precision greater than or equal to the precision of the regular approximate numeric type and with implementation-defined binary scale.
- i) Otherwise, ANT specifies a user-specified approximate numeric type:

Case:

- i) If ANT immediately contains the <precision> $PREC$ but no scale, then ANT specifies the *user-specified approximate numeric type* with implementation-defined binary precision greater than or equal to $PREC$ bits and with implementation-defined binary scale.
- ii) Otherwise, ANT immediately contains the <precision> $PREC$ and the <scale> $SCALE$, then ANT specifies the user-specified approximate numeric type with implementation-defined binary precision greater than or equal to $PREC$ bits and with implementation-defined binary scale greater than or equal to $SCALE$ bits.

- 46) The preferred name of every numeric type is the declared name of its normal form.
- 47) If a <numeric type> NT contains the <precision> $PREC$, then $PREC$ is the explicitly specified precision of the numeric type specified by NT .
- 48) If a <numeric type> NT contains the <scale> $SCALE$, then $SCALE$ is the explicitly specified scale of the numeric type specified by NT .
- 49) DATETIME specifies the datetime type.
- 50) LOCALDATETIME specifies the localdatetime type.
- 51) DATE specifies the date type.
- 52) TIME specifies the time type.
- 53) LOCALTIME specifies the localtime type.
- 54) DURATION specifies the duration type.
- 55) NODE or VERTEX specifies the value type node.
- 56) EDGE or RELATIONSHIP specifies the value type edge.
- 57) PATH specifies the value type path.
- 58) LIST or ARRAY specifies a list value type with the element type specified by the <value type>.
- 59) MULTISSET specifies multiset value type with the element type specified by the <value type>.

- 60) SET specifies set value type with the element type specified by the <value type>.
- 61) ORDERED SET specifies ordered set value type with the element type specified by the <value type>.
- 62) MAP specifies a map value type with the map key type specified by the <map key type> and the map value type specified by the <value type>.
- 63) RECORD specifies a record value type with the field type list specified by <field type list>.
- 64) A <field type list> *FTL* specifies a field type list whose field types are specified by the contained <field type>s of *FTL*.

General Rules

- 1) If any specification or operation attempts to cause an item of a character string type whose character set has a character repertoire of UCS to contain a code point that is a noncharacter, then an exception condition is raised: *data exception — non-character in character string* (22029).
- 2) If <value type> specifies a Boolean type *BT*, then a Boolean data type descriptor is created for *BT* that describes the specified Boolean type and comprises:
 - a) The name of the base type of all Boolean types (BOOLEAN DATA).
 - b) The preferred name of *BT*.
 - c) An indication that *BT* includes the null value.
- 3) If <value type> specifies a character string type *CST*, then a character string data type descriptor is created for *CST* that describes the specified character string type and comprises:
 - a) The name of the base type of all character string types (STRING DATA).
 - b) The preferred name of *CST*.
 - c) An indication that *CST* includes the null value.
 - d) The maximum length in characters of *CST*.
- 4) If <value type> specifies a byte string type *BST*, then a byte string data type descriptor is created for *BST* that describes the specified byte string type and comprises:
 - a) The name of the base type of all byte string types (BINARY DATA).
 - b) The preferred name of *BST*.
 - c) An indication that *BST* includes the null value.
 - d) The minimum length in bytes of *BST*.
 - e) The maximum length in bytes of *BST*.
- 5) If <value type> specifies an exact numeric type *ENT*, then a numeric data type descriptor is created for *ENT* that describes the specified exact numeric type and comprises:
 - a) The name of the base type of *ENT* (INTEGER DATA or DECIMAL DATA).
 - b) The preferred name of *ENT*, which is the declared name of the normal form of *ENT*.
 - c) An indication that *ENT* includes the null value.
 - d) The (implemented) precision of *ENT*.

- e) If ENT specifies an exact numeric type with decimal precision, then the (implemented) scale of ENT .
 - f) An indication of whether the precision and the scale of ENT are expressed in binary or decimal terms.
 - g) The explicit declared precision of ENT , if specified.
 - h) The explicit declared scale of ENT , if specified.
- 6) If <value type> specifies an approximate numeric type ANT , then a numeric data type descriptor is created for ANT that describes the specified approximate numeric type and comprises:
- a) The name of the base type of ANT (FLOAT DATA).
 - b) The preferred name of ANT , which is the declared name of the normal form of ANT .
 - c) An indication that ANT includes the null value.
 - d) The (implemented) precision of ANT .
 - e) The (implemented) scale of ANT .
 - f) An indication that the precision and the scale are expressed in binary terms.
 - g) The explicit declared precision of ANT , if specified.
 - h) The explicit declared scale of ANT , if specified.

Conformance Rules

None.

21.3 Names and identifiers

Function

Specify names.

Format

```
<object name> ::=  
  <identifier>  
  
<schema name> ::=  
  <identifier>  
  
<graph name> ::=  
  <identifier>  
  
<element type name> ::=  
  <type name>  
  
<graph type name> ::=  
  <identifier>  
  
<type name> ::=  
  <identifier>  
  
<binding table name> ::=  
  <identifier>  
  
<value name> ::=  
  <identifier>  
  
<procedure name> ::=  
  <identifier>  
  
<query name> ::=  
  <identifier>  
  
<function name> ::=  
  <identifier>  
  
<label name> ::=  
  <identifier>  
  
<property name> ::=  
  <identifier>  
  
<field name> ::=  
  <identifier>  
  
<path pattern name> ::=  
  <identifier>  
  
<parameter name> ::=  
  <dollar sign> <separated identifier>  
  
<element variable> ::=  
  <variable name>  
  
<path variable> ::=  
  <variable name>
```

21.3 Names and identifiers

```

<subpath variable> ::= 
  <variable name>

<static variable name> ::= 
  <variable name>

<binding variable name> ::= 
  <variable name>

<variable name> ::= 
  <regular identifier>

<identifier> ::= 
  <regular identifier>
 | <delimited identifier>

<separated identifier> ::= 
  <extended identifier>
 | <delimited identifier>

```

Syntax Rules

None.

General Rules

- 1) An <object name> identifies an object.
- 2) A <graph name> identifies a graph.
- 3) A <type name> identifies a type.
- 4) A <binding table name> identifies a binding table.
- 5) A <value name> identifies a value.
- 6) A <procedure name> identifies a procedure.
- 7) A <query name> identifies a query.
- 8) A <function name> identifies a function.
- 9) A <label name> identifies a label.
- 10) A <property name> identifies a property of a graph, a node, or an edge.
- 11) A <field name> identifies a field of a record.
- 12) A <path pattern name> identifies a path that is matched by a path pattern.
- 13) A <parameter name> identifies a parameter.
- 14) A <static variable name> identifies a static variable.
- 15) A <binding variable name> identifies a binding variable.
- 16) An <element variable> identifies an element variable that may be bound to a graph element.
- 17) A <path variable> identifies a path variable that is bound to a path that is matched by a path pattern.
- 18) A <subpath variable> identifies a subpath variable that may be bound to a subpath of a path that is matched by a path pattern.

Conformance Rules

None.

21.4 <token> and <separator>

Function

Specify lexical units (tokens and separators) that participate in the GQL language.

Format

```

<token> ::=

  <non-delimiter token>
  | <delimiter token>

<non-delimiter token> ::=
  <regular identifier>
  | <parameter name>
  | <key word>
  | <unsigned numeric literal>
  | <byte string literal>
  | <multiset alternation operator>

<non-delimited identifier> ::=
  <regular identifier>
  | <extended identifier>

<regular identifier> ::=
  <identifier start> [ <identifier extend>... ]

<extended identifier> ::=
  <identifier extend>...

<identifier start> ::=
  !! See the Syntax Rules.

<identifier extend> ::=
  !! See the Syntax Rules.

<key word> ::=
  <reserved word>
  | <non-reserved word>

<reserved word> ::=
  <case-insensitive reserved word>
  | endNode | inDegree | lTrim | outDegree | percentileCont | percentileDist | rTrim
  | startNode | stDev | stDevP | tail | toLower | toUpper

<case-insensitive reserved word> ::=
  ABS | ACOS | ADD | AGGREGATE | ALIAS | ALL | ALL_DIFFERENT | AND | ANY | ARRAY | AS |
  ASC
  | ASCENDING | ASIN | AT | ATAN | AVG
  | BINARY | BIGINT | BOOL | BOOLEAN | BOTH | BY | BYTE_LENGTH | BYTES
  | CALL | CASE | CAST | CATALOG | CEIL | CEILING | CHARACTER | CHARACTER_LENGTH | CLEAR
  | CLONE | CLOSE | COALESCE | COLLECT | COMMIT | CONSTRAINT | CONSTANT | CONSTRUCT | COPY
  | COS | COSH | COST | COT | COUNT | CURRENT_DATE | CURRENT_GRAPH | CURRENT_PROPERTY_GRAPH
  | CURRENT_ROLE | CURRENT_SCHEMA | CURRENT_TIME | CURRENT_TIMESTAMP | CURRENT_USER | CREATE
  | DATA | DATE | DATETIME | DEC | DECIMAL | DEFAULT | DEGREES | DELETE | DETACH | DESC
  | DESCENDING | DIRECTORIES | DIRECTORY | DISTINCT | DO | DOUBLE | DROP | DURATION
  | ELEMENT_ID | ELSE | END | ENDS | EMPTY_BINDING_TABLE | EMPTY_GRAPH

```

```

| EMPTY_PROPERTY_GRAPH | EMPTY_TABLE | EXCEPT | EXISTS | EXISTING | EXP | EXPLAIN
| FALSE | FILTER | FLOAT | FLOAT16 | FLOAT32 | FLOAT64 | FLOAT128 | FLOAT256
| FLOOR | FOR | FROM | FUNCTION | FUNCTIONS

| GQLSTATUS | GRANT | GROUP

| HAVING | HOME_GRAPH | HOME_PROPERTY_GRAPH | HOME_SCHEMA

| IN | INSERT | INT | INTEGER | INT8 | INTEGER8 | INT16 | INTEGER16 | INT32 | INTEGER32
| INT64 | INTEGER64 | INT128 | INTEGER128 | INT256 | INTEGER256
| INTERSECT | IF | IS

| KEEP

| LEADING | LEFT | LENGTH | LET | LIKE | LIKE_REGEX | LIMIT | LIST | LN
| LOCALDATETIME | LOCALTIME | LOCALTIMESTAMP | LOG | LOG10 | LOWER

| MANDATORY | MAP | MATCH | MERGE | MAX | MIN | MOD | MULTI | MULTIPLE | MULTISET

| NEW | NOT | NORMALIZE | NOTHING | NULL | NULLS | NULLIF | NUMERIC

| OCCURRENCES_REGEX | OCTET_LENGTH | OF | OFFSET | ON | OPTIONAL | OR | ORDER | ORDERED
| OTHERWISE

| PARAMETER | PATH | PATHS | PARTITION | POSITION_REGEX | POWER | PRECISION | PROCEDURE
| PROCEDURES

| PRODUCT | PROFILE | PROJECT

| QUERIES | QUERY

| RADIANS | REAL | RECORD | RECORDS | REFERENCE | REMOVE | RENAME | REPLACE | REQUIRE
| RESET | RESULT | RETURN | REVOKE | RIGHT | ROLLBACK

| SAME | SCALAR | SCHEMA | SCHEMAS | SCHEMATA | SELECT | SESSION | SET | SKIP | SIGNED
| SIN
| SINGLE | SINH | SMALLINT | SQRT | START | STARTS | STRING | SUBSTRING | SUBSTRING_REGEX
| SUM

| TAN | TANH | THEN | TIME | TIMESTAMP | TRAILING | TRANSLATE_REGEX | TRIM | TRUE
| TRUNCATE

| UINT | UINT8 | UINT16 | UINT32 | UINT64 | UINT128 | UINT256 | UNION | UNIT
| UNIT_BINDING_TABLE | UNIT_TABLE | UNIQUE | UNNEST | UNKNOWN | UNSIGNED | UNWIND
| UPPER | USE

| VALUE | VALUES | VARBINARY | VARCHAR

| WHEN | WHERE | WITH | WITHOUT

| XOR

| YIELD

| ZERO

<non-reserved word> ::=
  <case-insensitive non-reserved word>

<case-insensitive non-reserved word> ::=
  ACYCLIC

| BINDING

| CLASS_ORIGIN | COMMAND_FUNCTION | COMMAND_FUNCTION_CODE | CONDITION_NUMBER | CONNECTING

```

21.4 <token> and <separator>

```

| DESTINATION | DIRECTED
| EDGE | EDGES
| FINAL | FIRST
| GRAPH | GRAPHS | GROUPS
| INDEX
| LAST | LABEL | LABELED | LABELS
| MESSAGE_TEXT | MORE | MUTABLE
| NFC | NFD | NFKC | NFKD | NODE | NODES | NORMALIZED | NUMBER
| ONLY | ORDINALITY
| PATTERN | PATTERNS | PROPERTY | PROPERTIES
| READ | RELATIONSHIP | RELATIONSHIPS | RETURNED_GQLSTATUS
| SHORTEST | SIMPLE | SOURCE | SUBCLASS_ORIGIN
| TABLE | TABLES | TIES | TO | TRAIL | TRANSACTION | TYPE | TYPES
| UNDIRECTED
| VERTEX | VERTICES
| WALK | WRITE
| ZONE

<multiset alternation operator> ::==
| + |

<delimiter token> ::==
<GQL special character>
| <bracket right arrow>
| <bracket tilde right arrow>
| <character string literal>
| <concatenation operator>
| <date string>
| <datetime string>
| <delimited identifier>
| <double colon>
| <double minus sign>
| <double period>
| <duration string>
| <greater than operator>
| <greater than or equals operator>
| <left arrow>
| <left arrow bracket>
| <left arrow tilde>
| <left arrow tilde bracket>
| <left minus right>
| <left minus slash>
| <left tilde slash>
| <less than operator>
| <less than or equals operator>
| <minus left bracket>
| <minus slash>
| <not equals operator>
| <right arrow>

```

```

| <right bracket minus>
| <right bracket tilde>
| <slash minus>
| <slash minus right>
| <slash tilde>
| <slash tilde right>
| <tilde left bracket>
| <tilde right arrow>
| <tilde slash>
| <time string>

<bracket right arrow> ::= ]
  ]->

<bracket tilde right arrow> ::= ]
  ]~>

<concatenation operator> ::= ||
  ||

<double colon> ::= ::

<double minus sign> ::= --
  --

<double period> ::= ..
  ..

<greater than operator> ::= <right angle bracket>

<greater than or equals operator> ::= >=
  >=>

<left arrow> ::= <-
  <-

<left arrow tilde> ::= <~
  <~

<left arrow bracket> ::= <-[

<left arrow tilde bracket> ::= <~[

<left minus right> ::= <-->
  <-->

<left minus slash> ::= <--/
  <--/

<left tilde slash> ::= <~/

<less than operator> ::= <left angle bracket>

<less than or equals operator> ::= <=>
  <=>

<minus left bracket> ::= -[
  -[
```

21.4 <token> and <separator>

```

<minus slash> ::= -
  / 

<not equals operator> ::= 
  <>

<right arrow> ::= 
  ->

<right bracket minus> ::= 
  ]-
  ]~ 

<right bracket tilde> ::= 
  ]~ 

<slash minus> ::= 
  /-
  /->

<slash minus right> ::= 
  /->

<slash tilde> ::= 
  /~ 

<slash tilde right> ::= 
  /~>

<tilde left bracket> ::= 
  ~[ 

<tilde right arrow> ::= 
  ~>

<tilde slash> ::= 
  ~/ 

<delimited identifier> ::= 
  <double quoted character sequence>
  | <unbroken accent quoted character sequence>

<double solidus> ::= 
  // 

<separator> ::= 
  { <comment> | <whitespace> }...

<whitespace> ::= 
  !! See the Syntax Rules.

<comment> ::= 
  <simple comment>
  | <bracketed comment>

<simple comment> ::= 
  <simple comment introducer> [ <simple comment character>... ] <newline>

<simple comment introducer> ::= 
  <double solidus>
  | <double minus sign>

<simple comment character> ::= 
  !! See the Syntax Rules.

<bracketed comment> ::= 
  <bracketed comment introducer>

```

```

<bracketed comment contents>
<bracketed comment terminator>

<bracketed comment introducer> ::==
/*
 */

<bracketed comment terminator> ::==
*/
```

<bracketed comment contents> ::==
 !! See the Syntax Rules.
```

<escaped grave accent> ::==
 <reverse solidus> <grave accent>
 | <doubled grave accent>

<doubled grave accent> ::==
``

<newline> ::==
 !! See the *Syntax Rules*.

<edge synonym> ::==
 EDGE | RELATIONSHIP

<node synonym> ::==
 NODE | VERTEX

## Syntax Rules

- 1) An <identifier start> is any character with the Unicode property ID\_Start as defined by UAX31-D1 Default Identifier Syntax in [Unicode Standard Annex #31](#).
 

NOTE 150 — The characters in ID\_Start are those in the Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, and “Ni” together with those with the Unicode property Other\_ID\_Start but none of which have the Unicode properties Pattern\_Syntax or Pattern\_White\_Space.
- 2) An <identifier extend> is any character with the Unicode property ID\_Continue as defined by UAX31-D1 Default Identifier Syntax in [Unicode Standard Annex #31](#).
 

NOTE 151 — The characters in ID\_Continue are those in ID\_Start together with those in the Unicode General Category classes “Mn”, “Mc”, “Nd”, and “Pc” together with those with the Unicode property Other\_ID\_Continue but none of which have the Unicode properties Pattern\_Syntax or Pattern\_White\_Space.
- 3) The *representative form RF* of a <non-delimited identifier> *NDI* is the character string comprising the sequence of characters contained in *NDI*. *RF* shall not be the zero-length character string.
- 4) The maximum length in characters of the representative form of a <non-delimited identifier> shall be  $2^{14} - 1 = 16383$ .
 

NOTE 152 — This maximum length is modified by [Conformance Rule 1](#).
- 5) The *representative form RF* of a <delimited identifier> *DI* is the character string specified by the <single quoted character sequence>, the <double quoted character sequence>, or the <unbroken accent quoted character sequence> contained in *DI*. *RF* shall not be the zero-length character string.
- 6) The maximum length in characters of the representative form of a <delimited identifier> shall be  $2^{14} - 1 = 16383$ .
 

NOTE 153 — This maximum length is modified by [Conformance Rule 2](#).
- 7) <whitespace> is any consecutive sequence of Unicode characters with the property White\_Space.

## 21.4 &lt;token&gt; and &lt;separator&gt;

NOTE 154 — These are the characters the Unicode General Category classes “Zs”, “Zl” and “Zp” together with the characters: \u0009 (Horizontal Tabulation), \u000A (Line Feed), \u000B (Vertical Tabulation), \u000C (Form Feed), \u000D (Carriage Return), and \u0085 (Next Line).

- 8) <simple comment character> is any Unicode character that is not a <newline>.
- 9) <bracketed comment contents> is any character string of Unicode characters that does not contain <bracketed comment terminator>.
- 10) <newline> is the implementation-defined end-of-line indicator.

NOTE 155 — <newline> is typically represented by \u000A (“Line Feed”) and/or \u000D (“Carriage Return”); however, this representation is not required by this document.

- 11) A <token>, other than a <byte string literal>, <character string literal>, or a <delimited identifier>, shall not contain a <separator>.
- 12) Any <token> may be followed by a <separator>. A <non-delimiter token> shall be followed by a <delimiter token> or a <separator>.

NOTE 156 — If the Format does not allow a <non-delimiter token> to be followed by a <delimiter token>, then that <non-delimiter token> shall be followed by a <separator>.

- 13) GQL text containing one or more instances of <simple comment> is equivalent to the same GQL text with each <simple comment> replaced with <newline>.
- 14) GQL text containing one or more instances of <bracketed comment> is equivalent to the same GQL text with each <bracketed comment> *BC* replaced with,

Case:

- a) If *BC* contains a <newline>, then <newline>.
- b) Otherwise, <space>.

- 15) For every <non-delimited identifier> *NDI* there is exactly one corresponding *case-normal form CNF*. *CNF* is a character string derived from the representative form *RF* of *NDI* as follows:

Let *n* be the number of characters in *RF*. For *i* ranging from 1 (one) to *n*, the *i*-th character *M<sub>i</sub>* of *RF* is transliterated into the corresponding character or characters of *CNF* as follows:

Case:

- a) If *M<sub>i</sub>* is a lower-case character or a title case character for which an equivalent upper-case sequence *U* is defined by Unicode, then let *j* be the number of characters in *U*; the next *j* characters of *CNF* are *U*.
- b) Otherwise, the next character of *CNF* is *M<sub>i</sub>*.

NOTE 157 — Any lower-case letters for which there are no upper-case equivalents are left in their lower-case form.

NOTE 158 — The case-normal form of a <non-delimited identifier> is used in excluding <reserved word>s.

- 16) The case-normal form of a <non-delimited identifier> shall not be equal, according to the comparison rules in Subclause 19.3, “<comparison predicate>”, to any <case-insensitive reserved word> (with every letter that is a lower-case letter replaced by the corresponding upper-case letter or letters), treated as a <character string literal>.

NOTE 159 — It is the intention that no <key word> specified in this document or revisions thereto shall end with an <underscore>.

- 17) Two <non-delimited identifier>s are equivalent if their representative forms compare equally according to the comparison rules in Subclause 19.3, “<comparison predicate>”.

- 18) A <non-delimited identifier> and a <delimited identifier> are equivalent if their representative forms compare equally according to the comparison rules in Subclause 19.3, “<comparison predicate>”.
- 19) Two <delimited identifier>s are equivalent if their representative forms compare equally according to the comparison rules in Subclause 19.3, “<comparison predicate>”.
- 20) For the purposes of identifying <case-insensitive reserved word>s, any <simple Latin lower-case letter> contained in a candidate <key word> shall be effectively treated as the corresponding <simple Latin upper-case letter>.
- 21) For the purposes of identifying <case-insensitive non-reserved word>s, any <simple Latin lower-case letter> contained in a candidate <key word> shall be effectively treated as the corresponding <simple Latin upper-case letter>.

## General Rules

*None.*

## Conformance Rules

- 1) Without Feature GB00, “Long identifiers”, in conforming GQL language, the maximum length in characters of the representative form of a <non-delimited identifier> shall be  $2^7 - 1 = 127$ .
- 2) Without Feature GB00, “Long identifiers”, in conforming GQL language, the maximum length in characters of the representative form of a <delimited identifier> shall be  $2^7 - 1 = 127$ .
- 3) Without Feature GB01, “Double minus sign comments”, conforming GQL language shall not contain a <simple comment> introduced with a <double minus sign>.
- 4) Without Feature GB02, “Double solidus comments”, conforming GQL language shall not contain a <simple comment> introduced with a <double solidus>.

## 21.5 <GQL terminal character>

### Function

Define the terminal symbols of the GQL language.

### Format

```

<GQL terminal character> ::==
 <GQL language character>
 | <other language character>

<GQL language character> ::==
 <simple Latin letter>
 | <digit>
 | <GQL special character>

<simple Latin letter> ::==
 <simple Latin lower-case letter>
 | <simple Latin upper-case letter>

<simple Latin lower-case letter> ::=
 a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
 | p | q | r | s | t | u | v | w | x | y | z

<simple Latin upper-case letter> ::=
 A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
 | P | Q | R | S | T | U | V | W | X | Y | Z

<hex digit> ::=
 <standard digit> | A | B | C | D | E | F | a | b | c | d | e | f

<digit> ::=
 <standard digit>
 | <other digit>

<standard digit> ::=
 <octal digit> | 8 | 9

<octal digit> ::=
 <binary digit> | 2 | 3 | 4 | 5 | 6 | 7

<binary digit> ::=
 0 | 1

<other digit> ::=
 !! See the Syntax Rules.

<GQL special character> ::=
 <space>
 | <campersand>
 | <asterisk>
 | <colon>
 | <equals operator>
 | <comma>
 | <dollar sign>
 | <double quote>
 | <exclamation mark>
 | <grave accent>
 | <right angle bracket>
 | <left brace>

```

```
| <left bracket>
| <left paren>
| <left angle bracket>
| <minus sign>
| <period>
| <plus sign>
| <question mark>
| <quote>
| <reverse solidus>
| <right brace>
| <right bracket>
| <right paren>
| <semicolon>
| <solidus>
| <underscore>
| <vertical bar>
| <percent>
| <circumflex>
| <tilde>

<space> ::=

 !! See the Syntax Rules.

<ampersand> ::=

 &

<asterisk> ::=

 *

<circumflex> ::=

 ^

<colon> ::=

 :

<comma> ::=

 ,

<dollar sign> ::=

 $

<double quote> ::=

 "

<equals operator> ::=

 =

<exclamation mark> ::=

 !

<right angle bracket> ::=

 >

<grave accent> ::=

 `

<left brace> ::=

 {

<left bracket> ::=

 [

<left paren> ::=

 (
```

```

<left angle bracket> ::==
 <

<minus sign> ::==
 -

<percent> ::==
 %

<period> ::==
 .

<plus sign> ::==
 +

<question mark> ::==
 ?

<quote> ::==
 '

<reverse solidus> ::==
 \

<right brace> ::==
 }

<right bracket> ::==
]

<right paren> ::==
)

<semicolon> ::==
 ;

<solidus> ::==
 /

<tilde> ::==
 ~

<underscore> ::==
 _

<vertical bar> ::==
 |

<other language character> ::=
 !! See the Syntax Rules.

```

## Syntax Rules

- 1) <other language character> is any Unicode character not contained in <GQL language character>.
- 2) <other digit> is any Unicode character in the Unicode General Category class “Nd” not contained in <standard digit>.
- 3) <space> is the Unicode character \u0020 (Space).
- 4) The <hex digit>s a, b, c, d, e, and f have respectively the same values as the <hex digit>s A, B, C, D, E, and F.

## General Rules

- 1) There is a one-to-one correspondence between the symbols contained in <simple Latin upper-case letter> and the symbols contained in <simple Latin lower-case letter> such that, for every  $i$ , the symbol defined as the  $i$ -th alternative for <simple Latin upper-case letter> corresponds to the symbol defined as the  $i$ -th alternative for <simple Latin lower-case letter>.

## Conformance Rules

*None.*

## 22 Additional common rules

### 22.1 Satisfaction of a <label expression> by a label set

#### Subclause Signature

"Satisfaction of a <label expression> by a label set" [Syntax Rules] (

- Parameter: "LABEL EXPRESSION",
- Parameter: "LABEL SET"
- ) Returns: "TRUTH VALUE"

#### Function

Determine if a label set satisfies a <label expression>.

#### Syntax Rules

- 1) Let *LEXP* be the *LABEL EXPRESSION* and let *LS* be the *LABEL SET* in an application of the Syntax Rules of this Subclause. The result of the application of this Subclause is *TV*, which is returned as *TRUTH VALUE*.
- 2) A label set *LS* satisfies a <label expression> *LE* according to the following recursive definition:
  - a) If *LE* is a <label> *L<sub>2</sub>*, then *L<sub>2</sub>* is a member of *LS*.
  - b) If *LE* is a <wildcard label>, then *LS* is non-empty.

NOTE 160 — This condition is always true; every label set is non-empty. The rule is written this way in case empty label sets are permitted in the future, or for guidance to an implementation that supports graph elements with no labels.

  - c) If *LE* is a <parenthesized label expression> *PLE*, then let *LE<sub>2</sub>* be the <label expression> simply contained in *PLE*. Then *LS* satisfies *LE<sub>2</sub>*.
  - d) If *LE* is a <label negation>, then let *LP* be the <label primary> simply contained in *LE*. Then *LS* does not satisfy *LP*.
  - e) If *LE* is a <label conjunction>, then let *L<sub>1</sub>* be the <label term> and let *L<sub>2</sub>* be the <label factor> simply contained in *LE*. Then *LS* satisfies *L<sub>1</sub>* and *LS* satisfies *L<sub>2</sub>*.
  - f) If *LE* is a <label disjunction>, then let *L<sub>1</sub>* be the <label expression> and let *L<sub>2</sub>* be the <label term> simply contained in *LE*. Then *LS* satisfies *L<sub>1</sub>* or *LS* satisfies *L<sub>2</sub>*.
  - g) *TV* is

Case:

  - i) If *LS* satisfies *LEXP*, then *True*.
  - ii) Otherwise, *False*.

## Access Rules

*None.*

## General Rules

*None.*

## Conformance Rules

*None.*

## 22.2 Machinery for graph pattern matching

### Subclause Signature

```
"Machinery for graph pattern matching" [General Rules] (
 Parameter: "PROPERTY GRAPH",
 Parameter: "PATH PATTERN LIST"
) Returns: "MACHINERY"
```

### Function

Define the infrastructure (alphabet, mappings and related definitions) used in graph pattern matching.

### Syntax Rules

*None.*

### General Rules

- 1) Let  $PG$  be the *PROPERTY GRAPH* and let  $PPL$  be the *PATH PATTERN LIST* in an application of the General Rules of this Subclause. The result of the application of this Subclause is  $MACH$ , which is returned as *MACHINERY*.
- 2) Let  $SNV$  be the set of node variables declared in  $PPL$  at the same depth of graph pattern matching, and let  $SEV$  be the set of edge variables declared in  $PPL$  at the same depth of graph pattern matching.
- 3) For each <subpath variable>  $SPV$  contained in  $PPL$  at the same depth of graph pattern matching, let  $SPVBEGIN$  and  $SPVEND$  be two distinct <identifier>s that are distinct from every <identifier> in  $SNV \cup SEV$  and from every <identifier> created by this rule.  $SPVBEGIN$  is the *begin subpath symbol* and  $SPVEND$  is the *end subpath symbol* of  $SPV$ . Let  $SPS$  be the set of every subpath symbol.
- 4) Let `(` and `)` be mutually distinct <identifier>s that are distinct from every <identifier> in  $SNV \cup SEV \cup SPS$ . These are, respectively, the *anonymous node symbol* and the *anonymous edge symbol*. Let  $SAS$  be the set of anonymous symbols.
- 5) Let  $NPP$  be the number of <parenthesized path pattern expression>s contained in  $PPL$  at the same depth of graph pattern matching. Let  $PPPE_1, \dots, PPPE_{NPP}$  be an enumeration of the <parenthesized path pattern expression>s contained in  $PPL$  at the same depth of graph pattern matching, with every selective <parenthesized path pattern expression> before every non-selective <parenthesized path pattern expression>. For each  $i$ ,  $1 \leq i \leq NPP$ ,  $i$  is the *bracket index* of  $PPPE_i$ .
- 6) Let  $[_1, \dots, [_{NPP}], _1, \dots, ]_{NPP}]$  be  $2 * NPP$  <identifier>s that are mutually distinct, and distinct from every member of  $SNV \cup SEV \cup SPS \cup SAS \cup SEPS$  and from every graph element of  $PG$ . These are called *bracket symbols*. For every bracket symbols  $[_j$  or  $]_j$ ,  $j$  is the *bracket index* of the bracket symbol. There are two bracket symbols for each bracket index  $j$  between 1 (one) and  $NPP$ , corresponding to the <parenthesized path pattern expression>s  $PPPE_j$ . Let  $SBS$  be the set of bracket symbols.
- 7) Let  $GX$  be the set whose members are the graph elements of  $PG$ , the subpath symbols, and the bracket symbols.
- 8) Let  $ABC$  be  $SPS \cup SBS \cup SSV \cup SEV \cup SAS$ .  $ABC$  is the *alphabet*. The members of  $ABC$  are *symbols*.
- 9) A *word* is a string of elements of  $ABC$ .

10) An *elementary binding* is a pair  $(LET, GE)$  where  $LET$  is a member of  $ABC$  and  $GE$  is a member of  $GX$ , such that:

Case:

- a) If  $LET$  is a bracket symbol, then  $LET = GE$ . In this case, the elementary binding is a *bracket symbol binding*.
- b) If  $LET$  is a subpath symbol, then  $LET = GE$ . In this case, the elementary binding is a *subpath variable binding*.
- c) If  $LET$  is a node variable or the anonymous node symbol, then  $GE$  is a node. In this case the elementary binding is a *node binding*.
- d) If  $LET$  is an edge variable or the anonymous edge symbol, then  $GE$  is an edge. In this case, the elementary binding is an *edge binding*.

11) If  $(LET, GE)$  is an elementary binding, then it is said that  $LET$  is *bound* to  $GE$ .

12) When an elementary binding  $(LET, GE)$  binds  $LET$  to a graph element  $GE$  during the evaluation of an `<element pattern where clause>`, temporarily add a new field whose field name is  $LET$  and whose field value is a graph element reference value to  $GE$  to the current working record until the evaluation has finished.

13) A *path binding* is a sequence of zero or more elementary bindings,  $B = (LET_1, E_1), \dots (LET_N, E_N)$ . Given a path binding  $B$ :

NOTE 161 — It is convenient to allow an empty path binding even though paths are generally understood to be non-empty. For example, a quantifier may iterate 0 (zero) times, resulting in an empty path binding. The result of a `<path pattern>`, on the other hand, is required to be non-empty because of a Syntax Rule that its minimum node count shall be at least 1 (one).

- a) The *word* of  $B$  is the sequence  $LET_1, \dots LET_N$ .
- b) The *annotated path* of  $B$  is the sequence  $E_1, \dots E_N$ .

NOTE 162 — If the path binding is consistent (defined subsequently), the annotated path of  $B$  contains within it a path that matches the word of  $B$ , plus mark-up with bracket symbols and subpath symbols indicating how to interpret the path as a match to the word.

- c) The *extracted path*  $XP$  of  $B$  is obtained from the annotated path of  $B$  as follows:
  - i) Let  $XP$  be a copy of the annotated path of  $B$ .
  - ii) All subpath symbols and bracket symbols are deleted from  $XP$ .
  - iii) If two consecutive graph elements in  $XP$  are nodes, then the second is discarded. This step is repeated until there are no adjacent nodes in  $XP$ .

NOTE 163 — For consistent path bindings, two consecutive nodes will be the same.

14) A *reduced path binding* is obtained from a path binding as follows:

- a) All bracket bindings are removed.
- b) The following steps are performed repeatedly until no more anonymous node bindings can be removed:
  - i) If there are two adjacent anonymous node bindings, then the second is removed.
  - ii) If there is a binding of a node variable adjacent to an anonymous node binding, then the anonymous node binding is removed.

## 22.2 Machinery for graph pattern matching

- 15) When a path binding  $PB = (LET_1, E_1), \dots (LET_N, E_N)$  for a path or subpath variable  $POSPV$  is applied in the scope of some element  $E$  during the application of a General Rule  $RULE$ , then:
- Let  $RPB$  be the reduced path binding obtained from  $PB$ .
  - Let  $F$  be a new field whose field name is  $POSPV$  and whose field value is a path value constructed from the graph element reference values corresponding to the graph elements of the extracted path of  $RPB$  in the same order.
  - Add  $F$  temporarily to the current working record until the application of  $RULE$  has completed.
  - For every singleton graph pattern variable  $SGPV$  that is visible in the scope of  $E$  and that is bound by  $PB$  to a single graph element  $SGE$ , temporarily add a new field whose field name is  $SGPV$  and whose field value is a graph element reference value to  $SGE$  to the current working record until the application of  $RULE$  has completed.
  - For every group graph pattern variable  $GGPV$  that is visible in the scope of  $E$  and that is bound by  $PB$  to a set of multiple graph elements  $MGE$ , temporarily add a new field whose field name is  $GGPV$  and whose field value is a list of graph element reference values to the graph elements of  $MGE$  to the current working record until the application of  $RULE$  has completed.
  - For every subpath variable binding  $(SPVBEGIN, SPVBEGIN) (LET_j, E_j), \dots (LET_k, E_k) (SPVEND, SPVEND)$ ,  $1 \leq j \leq k \leq N$  for some subpath variable  $SV$  from  $S$  whose subpath begin symbol is  $SPVBEGIN$  and whose subpath end symbol is  $SPVEND$  and that is contained in  $PB$ :
    - Let  $RSPB$  be the reduced path binding obtained from the subpath binding  $SPB = (LET_j, E_j), \dots (LET_k, E_k)$  for  $SV$ .
    - Let  $SF$  be a new field whose field name is  $SV$  and whose field value is a path value constructed from the graph element reference values corresponding to the graph elements of the extracted path of  $RSPB$  in the same order.
    - Add  $SF$  temporarily to the current working record until the application of  $RULE$  has completed.

16) In a path binding, two node bindings are *separable* if there is an edge binding between them.

17) A path binding  $B$  is *consistent* if the following conditions are true:

- The extracted path of  $B$  is either an empty string or it is a path of  $PG$ ; that is, it begins with a node, it alternates between nodes and edges, each edge connects the node before and after it in the extracted path, and it ends with a node.
  - For every two node bindings  $(LET_1, E_1)$  and  $(LET_2, E_2)$  that are not separable,  $E_1 = E_2$ .
- NOTE 164 — If there are only bracket symbol or subpath variable bindings between two node bindings, the node bindings are required to bind to the same node.
- For every edge binding  $EB = (LET, GE)$ , let  $(LET_{left}, GE_{left})$  be the last node binding to the left of  $EB$  and let  $(LET_{right}, GE_{right})$  be the first node binding to the right of  $EB$ . Then:

Case:

- If  $GE$  is an undirected edge, then  $GE$  is an edge connecting  $GE_{left}$  and  $GE_{right}$ .
- If  $GE$  is a directed edge, then either  $GE_{left}$  is the source node and  $GE_{right}$  is the destination node of  $GE$ , or  $GE_{right}$  is the source node and  $GE_{left}$  is the destination node of  $GE$ .

NOTE 165 — The directionality constraint of the edge binding is fully checked during the generation of the regular language for <path concatenation>.

- d) For every start bracket symbol  $SBS$  contained in  $B$ , let  $j$  be the bracket index of  $SBS$ . Then the following conditions are true:
- i) There is an end bracket symbol contained in  $B$  and following  $SBS$  whose bracket index is  $j$ . Let  $EBS$  be the first such end bracket symbol. Let  $PPS$  be the explicit or implicit <path mode prefix> simply contained in the <parenthesized path pattern expression> whose bracket index is  $j$ . Let  $PM$  be the <path mode> contained in  $PPS$ .
  - ii) Case:
    - 1) If  $PM$  is TRAIL, then there is no edge that is repeated between  $SBS$  and  $EBS$ .
    - 2) If  $PM$  is SIMPLE, then no two separable node bindings between  $SBS$  and  $EBS$  bind the same node, except that the first and last node binding between  $SBS$  and  $EBS$  may bind the same node.
    - 3) If  $PM$  is ACYCLIC, then no two separable node bindings between  $SBS$  and  $EBS$  bind the same node.

NOTE 166 — The <path mode> WALK imposes no constraints on the extracted path.

- 18) A *multi-path binding* is an  $n$ -tuple  $(PB_1, \dots, PB_n)$  for some positive integer  $n$  such that each  $PB_i$  is a path binding.
- 19) If  $S$  and  $T$  are sets of strings, let  $S \cdot T$  be the set of strings formed by concatenating an element of  $S$  followed by an element of  $T$ , that is,  $S \cdot T = \{s t \mid s \text{ is an element of } S, t \text{ is an element of } T\}$ .

NOTE 167 — The  $\cdot$  operator will be used to concatenate words (strings of symbols) and path bindings (strings of elementary bindings).

- 20) If  $S$  is a set of strings, then let  $S^0$  be the set whose only element is the string of length 0 (zero), and for each non-negative integer  $n$ , let  $S^{n+1}$  be  $S^n \cdot S$ . Let  $S^*$  be the union of  $S^n$  for every non-negative integer  $n$ .

NOTE 168 — If  $S$  is not empty, then  $S^*$  is an infinite set; however, a finite result for every <graph pattern> is assured by the syntactic requirement that every <quantified path primary> is bounded, contained in a restrictive <parenthesized path pattern expression> or contained in a selective <parenthesized path pattern expression>.

- 21) Define *REDUCE* as a function that maps path bindings to path bindings, as follows.
- a) Let  $PBIN$  be a path binding.
  - b) Let  $PBOUT$  be a copy of  $PBIN$ .
  - c) All bracket bindings are removed from  $PBOUT$ .
  - d) The following steps are performed on  $PBOUT$  repeatedly until no more anonymous node bindings can be removed:
    - i) If there are two adjacent anonymous node bindings, then the second is removed.
    - ii) If there is a binding of a node variable adjacent to an anonymous node binding, then the anonymous node binding is removed.
  - e)  $REDUCE(PBIN)$  is  $PBOUT$ .
- 22) *REDUCE* is extended to multi-path bindings as follows: if  $MPB = (PB_1, \dots, PB_n)$  is a multi-path binding, then  $REDUCE(MPB) = (REDUCE(PB_1), \dots, REDUCE(PB_n))$ .
- 23) Let  $MACH$  be a data structure comprising the following:
- a)  $ABC$ , the alphabet, formed as the disjoint union of the following:

**22.2 Machinery for graph pattern matching**

- i)  $SNV$ , the set of node variables.
  - ii)  $SEV$ , the set of edge variables.
  - iii)  $SPS$ , the set of subpath symbols.
  - iv)  $SAS$ , the set of anonymous symbols.
  - v)  $SBS$ , the set of bracket symbols.
- b)  $REDUCE$ , the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.

24)  $MACH$  is returned as the  $MACHINERY$ .

## **Conformance Rules**

*None.*

## 22.3 Evaluation of a <path pattern expression>

### Subclause Signature

"Evaluation of a <path pattern expression>" [General Rules] (

- Parameter: "PROPERTY GRAPH",
- Parameter: "PATH PATTERN LIST",
- Parameter: "MACHINERY",
- Parameter: "SPECIFIC BNF INSTANCE",
- Parameter: "PARTIAL MATCH"

) Returns: "SET OF MATCHES"

### Function

Evaluate a <path pattern expression>.

### Syntax Rules

*None.*

### General Rules

- 1) Let  $PG$  be the *PROPERTY GRAPH*, let  $PPL$  be the *PATH PATTERN LIST*, let  $MACH$  be the *MACHINERY*, let  $SBI$  be the *SPECIFIC BNF INSTANCE*, and let  $PM$  be the *PARTIAL MATCH* in an application of the General Rules of this Subclause. The result of the application of this Subclause is  $SM$ , which is returned as *SET OF MATCHES*.

NOTE 169 — The parameters have the following meanings:

*SPECIFIC BNF INSTANCE*: the specific instance of a BNF non-terminal to be evaluated.

*PARTIAL MATCH*: this parameter is reserved for future use to support a sequence of MATCH clauses.

*SET OF MATCHES*: returns the set of partial matches to the *SPECIFIC BNF INSTANCE*.

- 2) The following components of  $MACH$  are identified:
  - a)  $ABC$ , the alphabet, formed as the disjoint union of the following:
    - i)  $SNV$ , the set of node variables.
    - ii)  $SEV$ , the set of edge variables.
    - iii)  $SPS$ , the set of subpath symbols.
    - iv)  $SAS$ , the set of anonymous symbols.
    - v)  $SBS$ , the set of bracket symbols.
  - b)  $REDUCE$ , the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 3) For every instance  $BNT$  of a BNF non-terminal that is a <path pattern expression>, <path term>, <path pattern union>, <path factor>, <path concatenation>, <path primary>, <quantified path primary>, <questioned path primary>, <element pattern>, or <parenthesized path pattern expression> equal to or contained in  $SBI$  at the same depth of graph pattern matching, the following are defined by simultaneous recursion:

## 22.3 Evaluation of a <path pattern expression>

- the *regular language of BNT*, denoted  $RL(BNT)$ , defined as a set of words over  $ABC$ ; and
- the *set of local matches to BNT*, denoted  $SLM(BNT)$ , defined as a set of path bindings.

NOTE 170 —  $SLM(BNT)$  is consistent except when concatenating an edge pattern prior to concatenating the following node pattern. Restrictive path modes are enforced when generating  $SLM(BNT)$  for the <parenthesized path pattern expression> that declares the path mode.

NOTE 171 —  $RL(BNT)$  and  $SLM(BNT)$  may be infinite sets if  $BNT$  contains an effectively unbounded quantifier. Every effectively unbounded quantifier is required to be contained in a selective <path pattern>; the potentially infinite set of local matches is subsequently reduced to a finite set by the General Rules of Subclause 22.4, “Evaluation of a selective <path pattern>”.

NOTE 172 — The BNF non-terminals are listed above in the “top down” order of appearance in the Format of Subclause 16.7, “<path pattern expression>”; the definitions in the following subrules treat the same BNF non-terminals in “bottom up” order.

Case:

- a) If  $BNT$  is a <parenthesized path pattern expression>, then let  $PPE$  be the <path pattern expression> immediately contained in  $BNT$  and let  $j$  be the bracket index of  $BNT$ . Then:

i) Case:

- 1) If  $BNT$  simply contains a <subpath variable declaration>  $SVD$ , then let  $SV$  be the subpath variable declared by  $SVD$ . Let  $BSV$  be the begin subpath symbol associated with  $SV$  and let  $ESV$  be the end subpath symbol associated with  $SV$ . Let  $BSVBINDING$  be  $(BSV, BSV)$  and let  $ESVBINDING$  be  $(ESV, ESV)$ .
- 2) Otherwise, let  $BSV$ ,  $ESV$ ,  $BSVBINDING$  and  $ESVBINDING$  be the empty string.

ii)  $RL(BNT)$  is  $\{ [j] \cdot \{ BSV \} \cdot RL(PPE) \cdot \{ ESV \} \cdot ]_j \}$

iii) Let  $STPB$  be  $SLM(PPE)$ .

Let  $SLMMAYBE$  be  $\{ ([j], [j]) \cdot \{ BSVBINDING \} \cdot STPB \cdot \{ ESVBINDING \} \cdot (]_j, ]_j) \}$

$SLM(BNT)$  is the set of every path binding in  $SLMMAYBE$  that is consistent.

NOTE 173 — That is, the words in  $RL(BNT)$  are formed by surrounding the words of  $RL(PPE)$  by the bracket symbols  $[j]$  and  $]_j$ . If  $BNT$  contains a subpath variable declaration, then the words of  $RL(BNT)$  are also surrounded by the begin and end subpath symbol of that subpath variable. Similarly, the path bindings in  $SLMMAYBE$  are formed by surrounding the path bindings with bracket bindings and, if there is a subpath declaration, with the corresponding begin and end subpath bindings. Eliminating inconsistent bindings from  $SLMMAYBE$  to get  $SLM(BNT)$  has the effect of enforcing restrictive path modes.

- b) If  $BNT$  is an <element pattern>  $EP$ , then

i) Case:

- 1) If  $EP$  declares an element variable  $EV$ , then let  $EPI$  be  $EV$ .
- 2) If  $EP$  is a <node pattern>, then let  $EPI$  be  $\{ \}$ , the anonymous node symbol.
- 3) If  $EP$  is an <edge pattern>, then let  $EPI$  be  $\{ \}$ , the anonymous edge symbol.

ii)  $RL(BNT)$  is  $\{ EPI \}$ , the set whose sole member is  $EPI$ .

iii)  $SLM(BNT)$  is the set of every elementary binding  $(EPI, GE)$  such that:

- 1) If  $EP$  is a <node pattern>, then  $GE$  is a node.
- 2) If  $EP$  is an <edge pattern>, then  $GE$  is an edge.
- 3) If  $EP$  simply contains a <label expression>  $LE$ , then the value of  $TV$  is *True* when the Syntax Rules of Subclause 22.1, “Satisfaction of a <label expression> by a label set”,

## 22.3 Evaluation of a &lt;path pattern expression&gt;

are applied with *LE* as *LABEL EXPRESSION* and the label set of *GE* as *LABEL SET*; let *TV* be the *TRUTH VALUE* returned from the application of those Syntax Rules.

- c) If *BNT* is a <quantified path primary>, then:
  - i) Let *PP* be the <path primary> immediately contained in *BNT*. As a result of the transformations in the Syntax Rules, *PP* is a <parenthesized path pattern expression>. Let *R* be *RL(PP)* and let *S* be *SLM(PP)*.
    - ii) Let *GQ* be the <general quantifier> immediately contained in *BNT*.
    - iii) Let *LB* be the value of the <lower bound> contained in *GQ*.
    - iv) Case:
      - 1) If *GQ* contains an <upper bound>, then let *UB* be the value of the <upper bound>. Then:
        - A) *RL(BNT)* is  $R^{LB} \cup R^{LB+1} \cup \dots \cup R^{UB-1} \cup R^{UB}$ .
        - B) Let *TOOMUCH* be  $S^{LB} \cup S^{LB+1} \cup \dots \cup S^{UB-1} \cup S^{UB}$ .
        - C) *SLM(BNT)* is the set of those path bindings in *TOOMUCH* that are consistent.
      - 2) Otherwise,
        - A) *RL(BNT)* is  $R^{LB} \cdot R^*$ .
        - B) Let *WAYTOOMUCH* be  $S^{LB} \cdot S^*$ .
        - C) *SLM(BNT)* is the set of those path bindings in *WAYTOOMUCH* that are consistent.
  - d) If *BNT* is a <questioned path primary>, then:
    - i) Let *PP* be the <path primary> immediately contained in *BNT*. As a result of the transformations in the Syntax Rules, *PP* is a <parenthesized path pattern expression>. Let *R* be *RL(PP)* and let *S* be *SLM(PP)*.
      - ii) *RL(BNT)* is  $R^\theta \cup R$ .
      - iii) *SLM(BNT)* is  $S^\theta \cup S$ .
  - e) If *BNT* is a <path primary>, then let *BNT2* be the <element pattern> or <parenthesized path pattern expression> that is immediately contained in *BNT*. Then:
    - i) *RL(BNT)* is *RL(BNT2)*.
    - ii) *SLM(BNT)* is *SLM(BNT2)*.
  - f) If *BNT* is a <path concatenation>, then:
    - i) Let *PST* be the <path term> and let *PC* be the <path factor> that are immediately contained in *BNT*.
      - ii) *RL(BNT)* is *RL(PST) · RL(PC)*.
      - iii) Case:
        - 1) If *PC* is an <edge pattern>, then *SLM(BNT)* is *SLM(PST) · SLM(PC)*

NOTE 174 — If *PC* is an <edge pattern>, then there is a <node pattern> to its right, which will be concatenated in a subsequent iteration of this recursion; consistency, including the directionality constraint implied by the <edge pattern>, will be checked at that point.

## 22.3 Evaluation of a &lt;path pattern expression&gt;

- 2) If  $PC$  is a <node pattern>, and the last <element pattern>  $EP$  of  $PST$  is an <edge pattern> then:

NOTE 175 — By transformations in the Syntax Rules of Subclause 16.7, “<path pattern expression>”, an edge binding is always immediately preceded and followed by a node binding. The current rule handles the situation in which the edge has been bound as the last elementary binding of  $PST$ , and therefore  $PC$  is the node binding that immediately follows the edge binding. Also note that the Syntax Rules have transformed every <abbreviated edge pattern> to a <full edge pattern>.

- A) Let  $SLMCONCAT$  be  $SLM(PST) \cdot SLM(PC)$ .
- B) For each path binding  $PB$  contained in  $SLMCONCAT$ ,
  - I) Let  $GE_{right}$  be the node that is bound in the last elementary binding of  $PB$ .
  - II) Let  $GE$  be the edge that is bound in the penultimate elementary binding of  $PB$ .
  - III) Let  $GE_{left}$  be the node that is bound in the antepenultimate elementary binding of  $PB$ .
- C) Let the propositions  $L$ ,  $U$ , and  $R$  be defined as follows:
  - I) Proposition  $L$  is true if  $GE$  is a directed edge,  $GE_{left}$  is the destination node of  $GE$  and  $GE_{right}$  is the source node of  $GE$ .
  - II) Proposition  $U$  is true if  $GE$  is an undirected edge, and  $GE_{left}$  and  $GE_{right}$  are the nodes connected by  $GE$ .
  - III) Proposition  $R$  is true if  $GE$  is a directed edge,  $GE_{left}$  is the source node of  $GE$  and  $GE_{right}$  is the destination node of  $GE$ .
- D) Let the *directionality constraint* of  $EP$  be
 

Case:

  - I) If  $EP$  is a <full edge pointing left>, then proposition  $L$  is true.
  - II) If  $EP$  is a <full edge undirected>, then proposition  $U$  is true.
  - III) If  $EP$  is a <full edge pointing right>, then proposition  $R$  is true.
  - IV) If  $EP$  is a <full edge left or undirected>, then proposition  $L$ , or proposition  $U$ , is true.
  - V) If  $EP$  is a <full edge undirected or right>, then proposition  $U$ , or proposition  $R$ , is true.
  - VI) If  $EP$  is a <full edge left or right>, then proposition  $L$ , or proposition  $R$ , is true.
  - VII) If  $EP$  is a <full edge any direction>, then at least one of proposition  $L$ , proposition  $U$ , or proposition  $R$ , is true.
- E)  $SLM(BNT)$  is the set of those path bindings of  $SLMCONCAT$  that are consistent and satisfy the directionality constraint of  $EP$ .
- 3) Otherwise,  $SLM(BNT)$  is the set of those path bindings in  $SLM(PST) \cdot SLM(PC)$  that are consistent.
- g) If  $BNT$  is a <path factor>, then let  $BNT2$  be the <path primary>, <quantified path primary> or <questioned path primary> immediately contained in  $BNT$ .

Then:

- i)  $RL(BNT)$  is  $RL(BNT2)$ .
- ii)  $SLM(BNT)$  is  $SLM(BNT2)$ .
- h) If  $BNT$  is a <path pattern union>, then let  $NMA$  be the number of <path term>s immediately contained in  $BNT$ . Let  $PMO_1, \dots PMO_{NMA}$  be these <path term>s.

Then

- i)  $RL(BNT)$  is  $RL(PMO_1) \cup RL(PMO_2) \cup \dots \cup RL(PMO_{NMA})$ .
- ii)  $SLM(BNT)$  is  $SLM(PMO_1) \cup SLM(PMO_2) \cup \dots \cup SLM(PMO_{NMA})$ .
- i) If  $BNT$  is <path term>, then let  $BNT2$  be the <path factor> or <path concatenation> immediately contained in  $BNT$ . Then
  - i)  $RL(BNT)$  is  $RL(BNT2)$ .
  - ii)  $SLM(BNT)$  is  $SLM(BNT2)$ .
- j) If  $BNT$  is a <path pattern expression>, then let  $BNT2$  be the <path term> or <path pattern union> immediately contained in  $BNT$ .

Then

NOTE 176 — <path multiset alternation> is transformed into <path pattern union> in the Syntax Rules and does not need to be considered separately here.

- i)  $RL(BNT)$  is  $RL(BNT2)$ .
- ii)  $SLM(BNT)$  is  $SLM(BNT2)$ .

- 4) Let  $SM$  be  $SLM(SBI)$ .
- 5)  $SM$  is returned as the *SET OF MATCHES*.

## Conformance Rules

*None.*

## 22.4 Evaluation of a selective <path pattern>

### Subclause Signature

"Evaluation of a selective <path pattern>" [General Rules] (

- Parameter: "PROPERTY GRAPH",
- Parameter: "PATH PATTERN LIST",
- Parameter: "MACHINERY",
- Parameter: "PARTIAL MATCH",
- Parameter: "SELECTOR",
- Parameter: "INPUT SET OF LOCAL MATCHES"

) Returns: "OUTPUT SET OF LOCAL MATCHES"

### Function

Evaluate a <path pattern> with a selective <path search prefix>.

### Syntax Rules

*None.*

### General Rules

- 1) Let *PG* be the *PROPERTY GRAPH*, let *PPL* be the *PATH PATTERN LIST*, let *MACH* be the *MACHINERY*, let *PM* be the *PARTIAL MATCH*, let *SEL* be the *SELECTOR*, and let *INSLM* be the *INPUT SET OF LOCAL MATCHES* in an application of the General Rules of this Subclause. The result of the application of this Subclause is *OUTSLM*, which is returned as *OUTPUT SET OF LOCAL MATCHES*.

NOTE 177 — The parameters have the following meaning:

*PARTIAL MATCH*: this parameter is reserved for future use to support a sequence of MATCH clauses.

*SEL*: a selective <path pattern>.

*INPUT SET OF LOCAL MATCHES*: the (potentially infinite) set of local matches to *SEL* generated by [Subclause 22.3, "Evaluation of a <path pattern expression>"](#).

*OUTPUT SET OF LOCAL MATCHES*: returns a finite subset of *INPUT SET OF LOCAL MATCHES* selected according to the <path search prefix> of *SEL*.

- 2) It is implementation-defined whether the General Rules of this Subclause are terminated if an exception condition is raised. If an implementation defines that it terminates execution because of an exception condition, it is implementation-dependent which of the members of *CANDIDATES* (defined subsequently) are actually probed to establish whether they might raise an exception.

NOTE 178 — *CANDIDATES* is potentially an infinite set, but there are algorithms to enumerate this set so as to satisfy the selection criterion of the selective <path pattern> without testing all candidate solutions. Even if the implementation defines that it terminates when an exception condition is encountered on a particular candidate solution, the order of enumerating the candidates may be implementation-dependent, and a candidate solution that might raise an exception may never be tested.

- 3) The following components of *MACH* are identified:

- a) *ABC*, the alphabet, formed as the disjoint union of the following:
  - i) *SNV*, the set of node variables.
  - ii) *SEV*, the set of edge variables.

## 22.4 Evaluation of a selective &lt;path pattern&gt;

- iii) *SPS*, the set of subpath symbols.
  - iv) *SAS*, the set of anonymous symbols.
  - v) *SBS*, the set of bracket symbols.
- b) *REDUCE*, the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 4) Let *NP* be the number of <path pattern>s in *PPL*.
- 5) *SEL* is a selective <path pattern>. Let *j* be the bracket index of the <parenthesized path pattern expression> simply contained in *SEL*.
- NOTE 179 — By a syntactic transformation in Subclause 16.6, “<graph pattern>”, this <parenthesized path pattern expression> is the entire content of *SEL* except possibly the declaration of a path variable.
- 6) Let *PSP* be the <path search prefix> simply contained in *SEL*.
- 7) Let *N* be the value of the <number of paths> or the <number of groups> specified in *PSP*. If *N* is not a positive integer, then an exception condition is raised: *data exception — invalid number of paths or groups (22GOF)*.
- 8) Let *p* be such that *SEL* is the *p*-th <path pattern> of *PPL*.
- 9) Let *CANDIDATES* be the set of every path binding *PBX* in *INSLM* such that the following conditions are true:
- a) For every unconditional singleton variable *EV* exposed by *SEL*, *EV* is bound to a unique graph element in *PBX*.
- NOTE 180 — Anonymous symbols are not element variables; there is no requirement that two anonymous symbols bind to the same graph element.
- b) For every <parenthesized path pattern expression> *PPPE* equal to or contained in *SEL*, let *i* be the bracket index of *PPPE*, and let ‘[<sub>*i*</sub>’ and ‘]<sub>*i*</sub>’ be the bracket symbols associated with *PPPE*. A *binding* of *PPPE* is a substring of *PBX* that begins with the bracket binding (‘[<sub>*i*</sub>’, ‘[<sub>*i*</sub>]’) and ends with the next bracket binding (‘]<sub>*i*</sub>’, ‘]<sub>*i*</sub>’).
- For every binding *BPPPE* of *PPPE* contained in *PBX*, the following are true:
- i) For every element variable *EV* that is exposed as an unconditional singleton by *PPPE*, *EV* is bound to a unique graph element by the element variable bindings contained in *BPPPE*.
- NOTE 181 — Anonymous symbols are not element variables; there is no requirement that two anonymous symbols bind to the same graph element.
- ii) If *PPPE* contains a <parenthesized path pattern where clause> *PPPWC*, then the value of *V* is *True* when the General Rules of Subclause 22.5, “*Applying bindings to evaluate an expression*”, are applied with *PPL* as *GRAPH PATTERN*, the <search condition> simply contained in *PPPWC* as *EXPRESSION*, *MACH* as *MACHINERY*, *PBX* as *MULTI-PATH BINDING*, and a reference to *BPPPE* as a subset of *PBX* as *REFERENCE TO LOCAL CONTEXT*; let *V* be the *VALUE* returned from the application of those General Rules.
- NOTE 182 — This is the juncture at which an exception condition might be raised. It is implementation-defined whether to terminate if an exception condition is raised. The order of enumerating the members of *CANDIDATES* is implementation-dependent, and there is no requirement that an implementation test all candidate solutions, which may be an infinite set in any case.
- 10) Each path binding *PBX* of *CANDIDATES* is replaced by *REDUCE(PBX)*.
- 11) Redundant duplicate path bindings are removed from *CANDIDATES*.

## 22.4 Evaluation of a selective &lt;path pattern&gt;

12) *CANDIDATES* is partitioned as follows. For every path binding *PBX* in *CANDIDATES*, the partition of *PBX* is the set of every path binding *PBY* in *CANDIDATES* such that the first and last node bindings of *PBX* bind the same nodes as the first and last node bindings, respectively, of *PBY*.

13) Each partition *PART* of *CANDIDATES* is modified as follows.

Case:

a) If *PSP* is an <any path search>, then

Case:

i) If the number of path bindings in *PART* is *N* or less, then the entire partition *PART* is retained.

ii) Otherwise, it is implementation-dependent which *N* path bindings of *PART* are retained.

b) If *PSP* is a <shortest path search>, then

Case:

i) If *PSP* is a <counted shortest path search>, then

Case:

1) If the number of path bindings in *PART* is *N* or less, then the entire partition *PART* is retained.

2) Otherwise, the path bindings of *PART* are sorted in increasing order of number of edges; the order of path bindings that have the same number of edges is implementation-dependent. The first *N* path bindings in *PART* are retained.

ii) If *PSP* is <counted shortest group search>, then the path bindings in *PART* are grouped, with each group comprising those path bindings having the same number of edges. The groups are ordered in increasing order by the number of edges.

Case:

1) If the number of groups in *PART* is *N* or less, then the entire partition *PART* is retained.

2) Otherwise, the path bindings comprising the first *N* groups of *PART* are retained.

14) Let *OUTSLM* be the set of path bindings retained in *CANDIDATES* after the preceding modifications to its partitions.

15) *OUTSLM* is returned as *OUTPUT SET OF LOCAL MATCHES*.

## Conformance Rules

*None.*

## 22.5 Applying bindings to evaluate an expression

### Subclause Signature

"Applying bindings to evaluate an expression" [General Rules] (

- Parameter: "GRAPH PATTERN",
- Parameter: "EXPRESSION",
- Parameter: "MACHINERY",
- Parameter: "MULTI-PATH BINDING",
- Parameter: "REFERENCE TO LOCAL CONTEXT"

) Returns: "VALUE"

### Function

Evaluate a <value expression> or <search condition> using the bindings of element variables determined by a local context within a multi-path binding.

### Syntax Rules

*None.*

### Access Rules

*None.*

### General Rules

- 1) Let *GP* be the *GRAPH PATTERN*, let *EXP* be the *EXPRESSION*, let *MACH* be the *MACHINERY*, let *MPB* be the *MULTI-PATH BINDING*, and let *RTLC* be the *REFERENCE TO LOCAL CONTEXT* in an application of the General Rules of this Subclause. The result of the application of this Subclause is *V*, which is returned as *VALUE*.

NOTE 183 — The parameters have the following meaning:

*GRAPH PATTERN*: a <graph pattern>.

*EXPRESSION*: a <value expression> or <search condition>.

*MULTI-PATH BINDING*: multi-path binding to the <graph pattern> in which to evaluate the expression.

*REFERENCE TO LOCAL CONTEXT*: an indication of a subset of the multipath binding, the local context. Group bindings are confined to the local context; singleton bindings may look outside the local context. The local context is passed "by reference" in order to correctly evaluate non-local singletons. For example, given the pattern:

[ (A) -> [ (B) -> (C) WHERE A.X = B.X+C.X ] -> (D) ]{2}

then A.X makes a non-local reference to element variable A. A word of this pattern will repeat the outer <parenthesized path pattern expression> twice, requiring two evaluations of the WHERE clause. Each evaluation of the WHERE clause must locate the appropriate non-local reference to A. The bindings to the inner <parenthesized path pattern expression>, if passed "by value", might not be enough information to determine the appropriate binding to the outer <parenthesized path pattern expression>.

- 2) The following components of *MACH* are identified:

- a) *ABC*, the alphabet, formed as the disjoint union of the following:
  - i) *SNV*, the set of node variables.

## 22.5 Applying bindings to evaluate an expression

- ii)  $SEV$ , the set of edge variables.
  - iii)  $SPS$ , the set of subpath symbols.
  - iv)  $SAS$ , the set of anonymous symbols.
  - v)  $SBS$ , the set of bracket symbols.
  - b)  $REDUCE$ , the function mapping path bindings to path bindings, and multi-path bindings to multi-path bindings.
- 3) Let  $LC$  be the subset of  $MPB$  that is indicated by  $RTLC$ .
- 4) For each  $\langle\text{element reference}\rangle ER$  contained in  $EXP$  at the same depth of graph pattern matching,
- a) Let  $DEG$  be the degree of reference of  $ER$ .
  - b) Let  $EV$  be the  $\langle\text{element variable}\rangle$  of  $ER$ .
  - c) Case:
    - i) If  $LC$  is equal to  $MPB$ , then let  $SPACE$  be  $MPB$ .
 

NOTE 184 — That is, the search space is the entire multi-path binding. This case arises in two circumstances: 1) the evaluation of a  $\langle\text{parenthesized path pattern where clause}\rangle$  in the outermost  $\langle\text{parenthesized path pattern expression}\rangle$  of a selective  $\langle\text{path pattern}\rangle$ ; 2) the evaluation of a  $\langle\text{graph pattern where clause}\rangle$ .
    - ii) Otherwise, let  $LCBI$  be the bracket index of the first bracket symbol in  $LC$ . Let  $LCPPPE$  be the  $\langle\text{parenthesized path pattern expression}\rangle$  contained in  $GP$  whose bracket index is  $LCBI$ . Let  $PP$  be the  $\langle\text{path pattern}\rangle$  containing  $LCPPPE$ .
 

Case:

      - 1) If  $EV$  is not declared by  $PP$  then let  $SPACE$  be  $MPB$ .
 

NOTE 185 — In this case,  $EV$  is declared in some other  $\langle\text{path pattern}\rangle$  than the one that contains  $ER$ .
      - 2) If  $EV$  is declared by  $LCPPPE$  then let  $SPACE$  be  $LC$ .
 

NOTE 186 — In this case,  $EV$  is declared locally to  $LCPPPE$  and the binding(s) to  $ER$  can be found by searching the local context  $LC$ .
    - 3) Otherwise, let  $DEFPPPPE$  be the innermost  $\langle\text{parenthesized path pattern expression}\rangle$  that declares  $EV$  and that contains  $LCPPPE$ . Let  $BI$  be the bracket index of  $DEFPPPPE$ . Let  $'[BI'$  and  $']BI'$  be the bracket symbols whose bracket index is  $BI$ . Let  $SPACE$  be the smallest substring of  $MBP$  containing  $LC$  and beginning with  $'[BI'$  and ending with  $']BI'$ .
 

NOTE 187 — In this case,  $EV$  is declared in some outer scope containing  $LCPPPE$ , and the binding of  $ER$ , if any, is found by searching the innermost scope that declares  $EV$ .
- d) Case:
- i) If  $DEG$  is singleton, then
 

Case:

    - 1) If there is an elementary binding of  $EV$  in  $SPACE$ , then let  $LOE$  be a list with a single graph element, the graph element that is bound to  $EV$  in  $SPACE$ .
 

NOTE 188 — Even if  $EV$  is bound multiple times in  $SPACE$  (expressing an equijoin on  $EV$ ), the list has only one graph element.
    - 2) Otherwise, let  $LOE$  be the empty list.

**22.5 Applying bindings to evaluate an expression**

NOTE 189 — This case can only arise if  $DEG$  is conditional singleton.

- ii) If  $DEG$  is group, then

Case:

- 1) If  $SPACE$  does not contain an elementary binding of  $EV$ , then let  $LOE$  be an empty list.
- 2) Otherwise, let  $LOE$  be the list of the graph elements that are bound to  $EV$  in the order that they occur in  $SPACE$ , scanning  $SPACE$  from left to right, and retaining duplicates.

- e)  $LOE$  is the *list of graph elements bound to ER*.

NOTE 190 — This list will be used by the General Rules of those Subclauses that need to evaluate  $ER$ , for example, [Subclause 20.26, “<property reference>](#).

- 5) Let  $V$  be the value of  $EXP$ .
- 6) For each  $<\text{element reference}> ER$  contained in  $EXP$  at the same depth of graph pattern matching, the list of graph elements bound to  $ER$  is destroyed.

## **Conformance Rules**

*None.*

## 22.6 Equality operations

### Function

Specify the prohibitions and restrictions by value type on operations that involve testing for equality.

### Syntax Rules

- 1) An *equality operation* is any of the following:
  - a) A <comparison predicate> that specifies <equals operator> or <not equals operator>.

### General Rules

*None.*

### Conformance Rules

*None.*

## 22.7 Ordering operations

### Function

Specify the prohibitions and restrictions by value type on operations that involve ordering of data.

### Syntax Rules

- 1) An *ordering operation* is any of the following:
  - a) A <comparison predicate> that does not specify <equals operator> or <not equals operator>.
  - b) A <sort specification list>.
  - c) An <aggregate function> that specifies MAX or MIN.

### General Rules

*None.*

### Conformance Rules

*None.*

## 22.8 Result of value type combinations

### Subclause Signature

```
"Result of value type combinations" [General Rules] (
 Parameter: "DTSET"
) Returns: "RESTYPE"
```

### Function

Specify the declared type of the result of the result of certain combinations of values of compatible value types, such as <case expression>s, or <collection value expression>s.

### Syntax Rules

*None.*

### General Rules

- 1) Let *IDTS* be the *DTSET* in an application of the General Rules of this Subclause. The result of the application of this Subclause is the declared type of the result, which is returned as *RESTYPE*.
- 2) Let *DTS* be the set of value types in *IDTS*.
- 3) Case:
  - a) If any of the value types in *DTS* is a character string type, then if any value type in *DTS* is not a character string type, then an exception condition is raised: *data exception — invalid value type (22G03)*.
  - b) If any of the value types in *DTS* is a byte string type, then if any value types in *DTS* are not byte string types an exception condition is raised: *data exception — invalid value type (22G03)*.
  - c) If any value type in *DTS* is numeric, then:
    - i) If any value type in *DTS* is not numeric, then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) The declared type of the result is
 

Case:

      - 1) If any value type in *DTS* is approximate numeric, then approximate numeric with implementation-defined precision.
      - 2) Otherwise, exact numeric with implementation-defined precision and with scale equal to the maximum of the scales of the value types in *DTS*.
  - d) If any value type in *DTS* is a DATETIME or LOCALDATETIME value type, then:
 

Case:

    - i) If any value type in *DTS* is not a DATETIME or LOCALDATETIME value type, then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) If any value type in *DTS* is DATETIME, then the declared type of the result is DATETIME.

- iii) Otherwise, the declared type of the result is LOCALDATETIME.
- e) If any value type in *DTS* is a TIME or LOCALTIME value type, then:
  - Case:
    - i) If any value type in *DTS* is not a TIME or LOCALTIME value type then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) If any value type in *DTS* is TIME, then the declared type of the result is TIME.
    - iii) Otherwise, the declared type of the result is LOCALTIME.
- f) If any value type in *DTS* is DURATION, then:
  - Case:
    - i) If any value type in *DTS* is not DURATION then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) Otherwise, the declared type of the result is DURATION.
- g) If any value type in *DTS* is a Boolean value, then:
  - Case:
    - i) if any value type in *DTS* is not a Boolean value then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) Otherwise, the declared type of the result is the Boolean type.
- h) If any value type in *DTS* is a list type, then:
  - Case:
    - i) If any value type in *DTS* is not a list type then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) Otherwise, the declared type of the result is the list type with element value type *ETR*, where *ETR* is the value type resulting from the application of this Subclause to the set of element types of the list types of *DTS*.
- i) If any value type in *DTS* is a multiset type, then:
  - Case:
    - i) If any value type in *DTS* is not a multiset type then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) Otherwise, the declared type of the result is the multiset type with element value type *ETR*, where *ETR* is the value type resulting from the application of this Subclause to the set of element types of the multiset types of *DTS*.
- j) If any value type in *DTS* is a set type, then:
  - Case:
    - i) If any value type in *DTS* is not a set type then an exception condition is raised: *data exception — invalid value type (22G03)*.
    - ii) Otherwise, the declared type of the result is the set type with element value type *ETR*, where *ETR* is the value type resulting from the application of this Subclause to the set of element types of the set types of *DTS*.

- k) If any value type in  $DTS$  is a map type, then:

Case:

- i) If any value type in  $DTS$  is not a map type then an exception condition is raised: *data exception — invalid value type (22G03)*.
- ii) Otherwise, declared type of the result is the map type with key element value type  $KETR$ , where  $KETR$  is the value type resulting from the application of this Subclause to the set of key element types of the map types of  $DTS$  and value element value type  $VETR$ , where  $VETR$  is the value type resulting from the application of this Subclause to the set of value element types of the map types of  $DTS$ .

## **Conformance Rules**

*None.*

## 22.9 Bag element grouping operations

### Function

Specify the prohibitions and restrictions by value type on the declared element type of a multiset for operations that involve grouping the elements of a multiset.

### Syntax Rules

- 1) A *multiset element grouping operation* is any of the following:
  - a) An equality operation such that the declared type of an operand of the equality operation is a multiset type.
  - b) A <multiset set function>.
  - c) A <multiset value expression> that specifies MULTISET UNION DISTINCT.
  - d) A <multiset value expression> that specifies MULTISET EXCEPT.
  - e) A <multiset term> that specifies MULTISET INTERSECT.
- 2) A *multiset operand* of a multiset element grouping operation is any of the following:
  - a) A <multiset value expression> immediately contained in a <multiset set function>.
  - b) A <multiset value expression> or a <multiset term> that is immediately contained in a <multiset value expression> that immediately contains MULTISET UNION DISTINCT.
  - c) A <multiset value expression> or a <multiset term> that is immediately contained in a <multiset value expression> that immediately contains MULTISET EXCEPT.
  - d) A <multiset term> or a <multiset primary> that is immediately contained in a <multiset term> that immediately contains MULTISET INTERSECT.
  - e) An operand of an equality operation such that the declared type of the operand is a multiset type.

### General Rules

*None.*

### Conformance Rules

*None.*

## 23 Diagnostics

### General Rules

- 1) When a condition is raised the following map, DIAGNOSTICS, is created with diagnostic information.

```
MAP { COMMAND_FUNCTION: CF,
 COMMAND_FUNCTION_CODE: CFC,
 NUMBER: N,
 CURRENT_SCHEMA: CS,
 HOME_GRAPH: HG,
 CURRENT_GRAPH: CG
}
```

where:

- a) *CF* is a string identifying the command executed. [Table 6, “Command codes”](#) specifies the identifier of the commands.
- b) *CFC* is an integer identifying the code of the GQL-statement executed. [Table 6, “Command codes”](#) identifies the code of the commands. Positive values are reserved for commands defined by this document. Negative values are reserved for implementation-defined commands.
- c) *N* is the number chained GQL-status objects.
- d) *CS* is the value of CURRENT\_SCHEMA.
- e) *HG* is the value of HOME\_GRAPH.
- f) *CG* is the value of CURRENT\_GRAPH.
- g) If the value of GQLSTATUS corresponds to *syntax error or access rule violation — invalid reference (42002)*, then the following map element is added to the map DIAGNOSTICS:

```
INVALID_REFERENCE: R
```

where *R* is the identifier of the reference that caused the exception condition to be raised.

**Table 6 — Command codes**

| Command                        | Identifier            | Code    |
|--------------------------------|-----------------------|---------|
| <session set schema clause>    | SESSION SET SCHEMA    | 1 (one) |
| <session set graph clause>     | SESSION SET GRAPH     | 2       |
| <session set time zone clause> | SESSION SET TIME ZONE | 3       |
| <session set parameter clause> | SESSION SET PARAMETER | 4       |
| <session clear command>        | SESSION CLEAR         | 5       |
| <session close command>        | SESSION CLOSE         | 6       |
| <session remove command>       | SESSION REMOVE        | 7       |

| Command                       | Identifier                  | Code |
|-------------------------------|-----------------------------|------|
| <start transaction command>   | START TRANSACTION           | 8    |
| <rollback command>            | ROLLBACK                    | 9    |
| <commit command>              | COMMIT                      | 10   |
| <create graph statement>      | CREATE GRAPH STATEMENT      | 11   |
| <create graph type statement> | CREATE GRAPH TYPE STATEMENT | 12   |
| <create procedure statement>  | CREATE PROCEDURE STATEMENT  | 14   |
| <create query statement>      | CREATE QUERY STATEMENT      | 15   |
| <create function statement>   | CREATE FUNCTION STATEMENT   | 16   |
| <drop graph statement>        | DROP GRAPH STATEMENT        | 18   |
| <drop graph type statement>   | DROP GRAPH TYPE STATEMENT   | 19   |
| <drop procedure statement>    | DROP PROCEDURE STATEMENT    | 21   |
| <drop query statement>        | DROP QUERY STATEMENT        | 22   |
| <drop function statement>     | DROP FUNCTION STATEMENT     | 23   |
| <match statement>             | MATCH STATEMENT             | 26   |
| <call query statement>        | CALL QUERY STATEMENT        | 27   |
| <call procedure statement>    | CALL PROCEDURE STATEMENT    | 28   |
| <do statement>                | DO STATEMENT                | 29   |
| <insert statement>            | INSERT STATEMENT            | 30   |
| <merge statement>             | MERGE STATEMENT             | 31   |
| <set statement>               | SET STATEMENT               | 32   |
| <remove statement>            | REMOVE STATEMENT            | 33   |
| <delete statement>            | DELETE STATEMENT            | 34   |
| <optional statement>          | OPTIONAL STATEMENT          | 35   |
| <mandatory statement>         | MANDATORY STATEMENT         | 36   |
| <let statement>               | LET STATEMENT               | 37   |
| <for statement>               | FOR STATEMENT               | 38   |
| <aggregate statement>         | AGGREGATE STATEMENT         | 39   |
| <filter statement>            | FILTER STATEMENT            | 40   |

| Command                           | Identifier                                                                                                  | Code     |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------|----------|
| <order by and page statement>     | ORDER BY AND PAGE STATEMENT                                                                                 | 41       |
| Implementation-defined statements | An implementation-defined character string value different from the value associated with any other command | $x^1$    |
| Unrecognized statements           | A zero-length character string                                                                              | 0 (zero) |

<sup>1</sup> An implementation-defined negative number different from the value associated with any other statement in the GQL language.

## 24 Status codes

### 24.1 GQLSTATUS

The character string value returned in a GQLSTATUS parameter comprises a 2-character class code followed by a 3-character subclass code, each is restricted to <standard digit>s and <simple Latin upper-case letter>s. [Table 7, “GQLSTATUS class and subclass codes”](#), specifies the class code for each condition and the subclass code or codes for each class code.

Class codes that begin with one of the <standard digit>s '0', '1', '2', '3', or '4' or one of the <simple Latin upper-case letter>s 'A', 'B', 'C', 'D', 'E', 'F', 'G', or 'H' are returned only for conditions defined in this document or some other International Standard. The range of such class codes is called *standard-defined classes*. Some such class codes are reserved for use by specific International Standards, as specified elsewhere in this Clause. Subclass codes associated with such classes that also begin with one of those 13 characters are returned only for conditions defined in this document or some other International Standard. The range of such subclass codes is called *standard-defined subclasses*. Subclass codes associated with such classes that begin with one of the <standard digit>s '5', '6', '7', '8', or '9' or one of the <simple Latin upper-case letter>s 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', or 'Z' are reserved for implementation-defined conditions and are called *implementation-defined subclasses*.

Class codes that begin with one of the <standard digit>s '5', '6', '7', '8', or '9' or one of the <simple Latin upper-case letter>s 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', or 'Z' are reserved for implementation-defined exception conditions and are called *implementation-defined classes*. All subclass codes except '000', which means *no subclass*, associated with such classes are reserved for implementation-defined conditions and are called *implementation-defined subclasses*. An implementation-defined completion condition shall be indicated by returning an implementation-defined subclass in conjunction with one of the classes *successful completion (00000)*, *warning (01000)*, or *no data (02000)*.

If a subclass code is not specified for a condition, then either subclass '000' or an implementation-defined subclass is returned.

The “Category” column has the following meanings: 'S' means that the class code given corresponds to successful completion and is a completion condition; 'W' means that the class code given corresponds to a successful completion but with a warning and is a completion condition; 'N' means that the class code given corresponds to a no-data situation and is a completion condition; 'X' means that the class code given corresponds to an exception condition.

**Table 7 — GQLSTATUS class and subclass codes**

| Category | Condition                    | Class | Subcondition                         | Subclass |
|----------|------------------------------|-------|--------------------------------------|----------|
| S        | <i>successful completion</i> | 00    | ( <i>no subclass</i> )               | 000      |
| W        | <i>warning</i>               | 01    | ( <i>no subclass</i> )               | 000      |
|          |                              |       | <i>string data, right truncation</i> | 004      |
|          |                              |       | <i>graph already exists</i>          | G01      |
|          |                              |       | <i>graph type already exists</i>     | G02      |
|          |                              |       | <i>graph does not exist</i>          | G03      |

| Category | Condition                   | Class | Subcondition                                  | Subclass |
|----------|-----------------------------|-------|-----------------------------------------------|----------|
|          |                             |       | <i>graph type does not exist</i>              | G04      |
|          |                             |       | <i>procedure does not exist</i>               | G06      |
|          |                             |       | <i>query does not exist</i>                   | G07      |
|          |                             |       | <i>function does not exist</i>                | G08      |
| N        | <i>no data</i>              | 02    | (no subclass)                                 | 000      |
| X        | <i>connection exception</i> | 08    | (no subclass)                                 | 000      |
|          |                             |       | <i>transaction resolution unknown</i>         | 007      |
| X        | <i>data exception</i>       | 22    | (no subclass)                                 | 000      |
|          |                             |       | <i>string data, right truncation</i>          | 001      |
|          |                             |       | <i>numeric value out of range</i>             | 003      |
|          |                             |       | <i>null value not allowed</i>                 | 004      |
|          |                             |       | <i>invalid datetime format</i>                | 007      |
|          |                             |       | <i>invalid time zone displacement value</i>   | 009      |
|          |                             |       | <i>substring error</i>                        | 011      |
|          |                             |       | <i>division by zero</i>                       | 012      |
|          |                             |       | <i>invalid character value for cast</i>       | 018      |
|          |                             |       | <i>invalid argument for natural logarithm</i> | 01E      |
|          |                             |       | <i>invalid argument for power function</i>    | 01F      |
|          |                             |       | <i>trim error</i>                             | 027      |
|          |                             |       | <i>non-character in character string</i>      | 029      |
|          |                             |       | <i>negative limit value</i>                   | G02      |
|          |                             |       | <i>invalid value type</i>                     | G03      |
|          |                             |       | <i>values not comparable</i>                  | G04      |
|          |                             |       | <i>invalid datetime function map key</i>      | G05      |
|          |                             |       | <i>invalid datetime function map value</i>    | G06      |

| Category | Condition                                    | Class | Subcondition                               | Subclass |
|----------|----------------------------------------------|-------|--------------------------------------------|----------|
|          |                                              |       | <i>invalid duration function map key</i>   | G07      |
|          |                                              |       | <i>maximum list cardinality exceeded</i>   | G08      |
|          |                                              |       | <i>invalid graph element reference</i>     | G09      |
|          |                                              |       | <i>invalid session parameter name</i>      | G0A      |
|          |                                              |       | <i>list data, right truncation</i>         | G0B      |
|          |                                              |       | <i>list element error</i>                  | G0C      |
|          |                                              |       | <i>empty binding table returned</i>        | G0D      |
|          |                                              |       | <i>attempt to modify a FINAL parameter</i> | G0E      |
|          |                                              |       | <i>invalid number of paths or groups</i>   | G0F      |
|          |                                              |       | <i>invalid duration format</i>             | G0H      |
| X        | <i>invalid transaction state</i>             | 25    | (no subclass)                              | 000      |
|          |                                              |       | <i>active GQL-transaction</i>              | G01      |
| X        | <i>invalid transaction termination</i>       | 2D    | (no subclass)                              | 000      |
| X        | <i>transaction rollback</i>                  | 40    | (no subclass)                              | 000      |
|          |                                              |       | <i>integrity constraint violation</i>      | 002      |
|          |                                              |       | <i>statement completion unknown</i>        | 003      |
| X        | <i>syntax error or access rule violation</i> | 42    | (no subclass)                              | 000      |
|          |                                              |       | <i>nvalid syntax</i>                       | 001      |
|          |                                              |       | <i>invalid reference</i>                   | 002      |
| X        | <i>procedure call error</i>                  | G0    | (no subclass)                              | 000      |
|          |                                              |       | <i>incorrect number of arguments</i>       | 001      |
|          |                                              |       | <i>invalid parameter type</i>              | 002      |
|          |                                              |       | <i>invalid result type</i>                 | 003      |
| X        | <i>dependent object error</i>                | G1    | (no subclass)                              | 000      |
|          |                                              |       | <i>edges still exist</i>                   | 001      |

| Category | Condition                           | Class | Subcondition                            | Subclass |
|----------|-------------------------------------|-------|-----------------------------------------|----------|
| X        | <i>graph type conformance error</i> | G2    | (no subclass)                           | 000      |
| X        | <i>graph type definition error</i>  | G3    | (no subclass)                           | 000      |
|          |                                     |       | <i>endpoint node type not defined</i>   | 001      |
| X        | <i>invalid action</i>               | G4    | (no subclass)                           | 000      |
|          |                                     |       | <i>attempt to change constant value</i> | 001      |

## 25 Conformance

### 25.1 Introduction to conformance

Conformance may be claimed by a graph, a GQL-program, or a GQL-implementation.

### 25.2 Minimum conformance

Minimum conformance is defined as meeting the requirements of the data model and all syntax and semantics not explicitly identified as belonging to an optional feature.

A claim of minimum conformance shall also include:

- 1) A claim of conformance to at least one of:
  - Feature GA02, “Empty node label set support”
  - Feature GA03, “Singleton node label set support”
- 2) A claim of conformance to at least one of:
  - Feature GA05, “Empty edge label set support”
  - Feature GA06, “Singleton edge label set support”
- 3) A claim of conformance to a specific version of [The Unicode® Standard](#) and the synchronous versions of [Unicode Technical Standard #10](#), [Unicode Standard Annex #15](#), and [Unicode Standard Annex #31](#). The claimed version of [The Unicode® Standard](#) shall not be less than “13.0.0”.
- 4) A claim of conformance to at least one of:
  - Feature GA05, “Empty edge label set support”
  - Feature GA06, “Singleton edge label set support”

### 25.3 Conformance to features

In addition to the data model, syntax and semantics that are mandatory for minimum conformance to this document this document defines optional features. These features are identified by Feature ID and are controlled by explicit or implicit Conformance Rules (see [Subclause 5.3.7, “Feature ID and Feature Name”](#)).

An optional feature *FEAT* is defined by relaxing selected Conformance Rules, as noted at the beginning of each Conformance Rule by the phrase “without Feature *FEAT*, “name of feature”, ...”. An application designates a set of GQL features that the application requires; the GQL language of the application shall observe the restrictions of all Conformance Rules except those explicitly relaxed for the required features. Conversely, conforming GQL-implementations shall identify which GQL features the GQL-implementation supports. A GQL-implementation shall process any application whose required features are a subset of the GQL-implementations supported features.

A feature *FEAT1* may imply another feature *FEAT2*. A GQL-implementation that claims to support *FEAT1* shall also support each feature *FEAT2* implied by *FEAT1*. Conversely, an application need only designate that it requires *FEAT1*, and can assume that this includes each feature *FEAT2* implied by *FEAT1*. The list of features that are implied by other features is shown in [Table 8, “Implied feature relationships”](#). Note that some features imply multiple other features.

### 25.3 Conformance to features

The Syntax Rules and General Rules may define one GQL syntax in terms of another. Such transformations are presented to define the semantics of the transformed syntax and are effectively performed after checking the applicable Conformance Rules, unless otherwise noted in a Syntax Rule that defines a transformation. Transformations may use GQL syntax of one GQL feature to define another GQL feature. These transformations serve to define the behavior of the syntax, and do not have any implications for the feature syntax that is permitted or forbidden by the features so defined, except as otherwise noted in a Syntax Rule that defines a transformation. A conforming GQL-implementation need only process the untransformed syntax defined by the Conformance Rules that are applicable for the set of features that the GQL-implementation claims to support, though with the semantics implied by the transformation.

## 25.4 Requirements for GQL-programs

### 25.4.1 Introduction to requirements for GQL-programs

A conforming GQL-program shall be processed without syntax error by a conforming GQL-implementation if all of the following are satisfied:

- Every command or procedure invoked by the GQL-program is syntactically correct in accordance with this document.
- The GQL-implementation claims conformance to all the optional features to which the GQL-program claims conformance.
- The graph or graphs being processed are conforming and the conformance claims of these graphs do not include any features not included in the GQL-program's claim of conformance.

A conforming GQL application shall not use any additional features beyond the level of conformance claimed.

### 25.4.2 Claims of conformance for GQL-programs

A claim of conformance by a GQL-program to this document shall include all of the following:

- A claim of minimum conformance.
- Zero or more additional claims of conformance to optional features.
- A list of the implementation-defined elements and actions that are relied on for correct performance

## 25.5 Requirements for GQL-implementations

### 25.5.1 Introduction to requirements for GQL-implementations

A conforming GQL-implementation shall correctly translate and execute all GQL-programs conforming to both the GQL language and the implementation-defined features of the GQL-implementation.

A conforming GQL-implementation shall reject all GQL-programs that contain errors whose detection is required by this document.

A conforming GQL-implementation that provides optional features or that provides facilities beyond those specified in this document shall provide a GQL Flagger (See Subclause 25.7, "GQL Flagger").

### 25.5.2 Claims of conformance for GQL-implementations

A claim of conformance by a GQL-implementation to this document shall include all of the following:

- 1) A claim of minimum conformance.
- 2) Zero or more additional claims of conformance to optional features.

- 3) The definition for every element and action, within the scope of the claim, that this document specifies to be implementation-defined.

### 25.5.3 Extensions and options

A GQL-implementation may provide implementation-defined features that are additional to those specified by this document, and may add to the list of reserved words.

NOTE 191 — If additional words are reserved, it is possible that a conforming statement cannot be processed correctly.

A GQL-implementation may provide user options to process non-conforming statements. A GQL-implementation may provide user options to process statements so as to produce a result different from that specified in this document.

It shall produce such results only when explicitly required by the user option.

It is implementation-defined whether a GQL Flagger flags implementation-defined features.

NOTE 192 — A GQL Flagger may flag implementation-defined features using any Feature ID not defined by this document. However, there is no guarantee that some future edition of this document will not use such a Feature ID for a standard-defined feature.

NOTE 193 — The implementation-defined features flagged by a GQL Flagger may identify implementation-defined features from more than one GQL-implementation.

NOTE 194 — The allocation of a Feature ID to an implementation-defined feature may differ between GQL Flaggers.

## 25.6 Requirements for graphs

### 25.6.1 Introduction to requirements for graphs

A graph shall conform to the requirements of the data model in Subclause 4.4.3, “Graphs” as modified by any optional features to which conformance is claimed.

### 25.6.2 Claims of conformance for graphs

A claim of conformance by a graph to this document shall include all of the following:

- 1) A claim of minimum conformance.
- 2) Zero or more additional claims of conformance to optional features.
- 3) A list of the implementation-defined values that are relied on for correct processing.

## 25.7 GQL Flagger

A GQL Flagger is a facility provided by a GQL-implementation that is able to identify GQL language extensions, or other GQL processing alternatives, that may be provided by a conforming GQL-implementation (see Subclause 25.5.3, “Extensions and options”).

A GQL Flagger is intended to assist in the production of GQL language that is both portable and interoperable among different conforming GQL-implementations operating under different levels of this document.

A GQL Flagger is intended to effect a static check of GQL language. There is no requirement to detect extensions that cannot be determined until the General Rules are evaluated.

A GQL-implementation need only flag GQL language that is not otherwise in error as far as that implementation is concerned.

NOTE 195 — If a system is processing GQL language that contains errors, then it can be very difficult within a single statement to determine what is an error and what is an extension. As one possibility, a GQL-implementation may choose to check GQL language in two steps; first through its normal syntax analyzer and secondly through the GQL Flagger. The first step produces error messages for non-standard GQL language that the GQL-implementation cannot process or recognize. The second step processes GQL language that contains no errors as far as that GQL-implementation is concerned; it detects and flags at one

time all non-standard GQL language that could be processed by that GQL-implementation. Any such two-step process should be transparent to the end user.

The GQL Flagger assists identification of conforming GQL language that can perform differently in alternative processing environments provided by a conforming GQL-implementation. It also provides a tool in identifying GQL elements that may have to be modified if GQL language is to be moved from a non-conforming to a conforming GQL processing environment.

## 25.8 Implied feature relationships

The following table lists those features that imply one or more other features.

**Table 8 — Implied feature relationships**

| Feature ID | Feature Name                         | Implied Feature ID | Implied Feature Name             |
|------------|--------------------------------------|--------------------|----------------------------------|
| G030       | Quantified Paths                     | G031               | Quantified Edges                 |
| G060       | Non-local element pattern predicates | G082               | Non-local predicates             |
| GA04       | Unbounded node label set support     | GA03               | Singleton node label set support |
| GA07       | Unbounded edge label set support     | GA06               | Singleton edge label set support |

## Annex A

### (informative)

### **GQL Conformance Summary**

The contents of this Annex summarizes all the Conformance Rules.

Most optional Features of this document are specified by Conformance Rules in Subclauses, however some are specified by implicit Conformance Rules in other text. These are summarized first.

- 1) Specifications for Feature GA00, “Graph label set support”.
  - a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA00, “Graph label set support”, the graph label set of a conforming graph shall be empty.
- 2) Specifications for Feature GA01, “Graph property set support”.
  - a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA01, “Graph property set support”, the graph property set of a conforming graph shall be empty.
- 3) Specifications for Feature G001, “Undirected edge patterns”.
  - a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature G001, “Undirected edge patterns”, a conforming graph shall not contain undirected edges.
- 4) Specifications for Feature GA02, “Empty node label set support”.
  - a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA02, “Empty node label set support”, a node in a conforming graph shall contain a non-empty node label set.
- 5) Specifications for Feature GA03, “Singleton node label set support”.
  - a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA03, “Singleton node label set support”, a node in a conforming graph shall not contain a singleton node label set.
- 6) Specifications for Feature GA04, “Unbounded node label set support”.
  - a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA04, “Unbounded node label set support”, a node in a conforming graph shall not contain a node label set that comprises one or more labels.
- 7) Specifications for Feature GA05, “Empty edge label set support”.
  - a) **Subclause 4.4.3, “Graphs”**

- i) Without Feature GA05, “Empty edge label set support”, an edge in a conforming graph shall contain a non-empty edge label set.
- 8) Specifications for Feature GA06, “Singleton edge label set support”.
- a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA06, “Singleton edge label set support”, an edge in a conforming graph shall not contain a singleton edge label set.
- 9) Specifications for Feature GA07, “Unbounded edge label set support”.
- a) **Subclause 4.4.3, “Graphs”**
    - i) Without Feature GA07, “Unbounded edge label set support”, an edge in a conforming graph shall not contain an edge label set that comprises one or more labels.
- 10) Specifications for Feature GA08, “Named node types in graph types”.
- a) **Subclause 4.15, “Graph types”**
    - i) Without Feature GA08, “Named node types in graph types”, a conforming graph type shall contain an empty node type name dictionary.
- 11) Specifications for Feature GA09, “Named edge types in graph types”.
- a) **Subclause 4.15, “Graph types”**
    - i) Without Feature GA09, “Named edge types in graph types”, a conforming graph type shall contain an empty edge type name dictionary.

The remainder of this Annex recapitulates the Conformance Rules specified in Subclauses throughout this document, organized by feature name and Subclause.

- 1) Specifications for Feature G001, “Undirected edge patterns”:
- a) **Subclause 13.10, “<edge type definition>”:**
    - i) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain an <edge type definition> that simply contains an <edge kind> that is DIRECTED, an <endpoint pair definition> that is an <endpoint pair definition any direction>, or a <full edge type pattern> that is a <full edge type pattern any direction>.
  - b) **Subclause 16.7, “<path pattern expression>”:**
    - i) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain <full edge undirected>, <full edge left or undirected>, <full edge undirected or right>, or <full edge left or right>.
    - ii) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain an <abbreviated edge pattern> that is <tilde>, <left arrow tilde>, <tilde right arrow> or <left minus right>.
  - c) **Subclause 16.11, “<simplified path pattern expression>”:**
    - i) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain <simplified defaulting undirected>, <simplified defaulting left or undirected>, <simplified defaulting undirected or right>, or <simplified defaulting left or right>.
    - ii) Without Feature G001, “Undirected edge patterns”, conforming GQL language shall not contain <simplified override undirected>, <simplified override left or undirected>, <simplified override undirected or right>, or <simplified override left or right>.

- 2) Specifications for Feature G010, “Graph Pattern Keep”:
  - a) Subclause 16.6, “<graph pattern>”:
    - i) Without Feature G010, “Graph Pattern Keep”, conforming GQL language shall not contain a <keep clause>.
- 3) Specifications for Feature G011, “Graph Pattern Where”:
  - a) Subclause 16.6, “<graph pattern>”:
    - i) Without Feature G011, “Graph Pattern Where”, conforming GQL language shall not contain a <graph pattern where clause>.
- 4) Specifications for Feature G020, “Path Multiset Alternation”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G020, “Path Multiset Alternation”, conforming GQL language shall not contain a <path multiset alternation>.
- 5) Specifications for Feature G021, “Path Pattern Union”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G021, “Path Pattern Union”, conforming GQL language shall not contain a <path pattern union>.
- 6) Specifications for Feature G030, “Quantified Paths”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G030, “Quantified Paths”, conforming GQL language shall not contain a <quantified path primary>.
- 7) Specifications for Feature G031, “Quantified Edges”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G031, “Quantified Edges”, conforming GQL language shall not contain a <quantified path primary> that immediately contains a <path primary> that is an <edge pattern>.
- 8) Specifications for Feature G040, “Questioned Paths”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G040, “Questioned Paths”, conforming GQL language shall not contain a <questioned path primary>.
- 9) Specifications for Feature G050, “Simplified Path Pattern Expression”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G050, “Simplified Path Pattern Expression”, conforming GQL language shall not contain a <simplified path pattern expression>.
- 10) Specifications for Feature G060, “Non-local element pattern predicates”:
  - a) Subclause 16.7, “<path pattern expression>”:
    - i) Without Feature G060, “Non-local element pattern predicates”, in conforming GQL language, the <element pattern where clause> of an <element pattern> EP shall only reference the <element variable> declared in EP.

11) Specifications for Feature G070, “Abbreviated Edge Patterns”:

a) Subclause 16.7, “<path pattern expression>”:

- i) Without Feature G070, “Abbreviated Edge Patterns”, conforming GQL language shall not contain an <abbreviated edge pattern>.

12) Specifications for Feature G081, “Parenthesized Path Pattern Where”:

a) Subclause 16.7, “<path pattern expression>”:

- i) Without Feature G081, “Parenthesized Path Pattern Where”, conforming GQL language shall not contain a <parenthesized path pattern where clause>.

13) Specifications for Feature G082, “Non-local predicates”:

a) Subclause 16.7, “<path pattern expression>”:

- i) Without Feature G082, “Non-local predicates”, in conforming GQL language, a <parenthesized path pattern where clause> simply contained in a <parenthesized path pattern expression> *PPPE* shall not reference an <element variable> that is not declared in *PPPE*.

14) Specifications for Feature G083, “Parenthesized Path Pattern Cost”:

a) Subclause 16.7, “<path pattern expression>”:

- i) Without Feature G083, “Parenthesized Path Pattern Cost”, conforming GQL language shall not contain a <parenthesized path pattern cost clause>.

15) Specifications for Feature G091, “Shortest Path Search”:

a) Subclause 16.8, “<path pattern prefix>”:

- i) Without Feature G091, “Shortest Path Search”, conforming GQL language shall not contain a <shortest path search>.

16) Specifications for Feature G092, “Advanced Path Modes: TRAIL”:

a) Subclause 16.8, “<path pattern prefix>”:

- i) Without Feature G092, “Advanced Path Modes: TRAIL”, conforming GQL language shall not contain <path mode> that specifies TRAIL.

17) Specifications for Feature G093, “Advanced Path Modes: SIMPLE”:

a) Subclause 16.8, “<path pattern prefix>”:

- i) Without Feature G093, “Advanced Path Modes: SIMPLE”, conforming GQL language shall not contain <path mode> that specifies SIMPLE.

18) Specifications for Feature G094, “Advanced Path Modes: ACYCLIC”:

a) Subclause 16.8, “<path pattern prefix>”:

- i) Without Feature G094, “Advanced Path Modes: ACYCLIC”, conforming GQL language shall not contain <path mode> that specifies ACYCLIC.

19) Specifications for Feature G100, “ELEMENT\_ID function”:

a) Subclause 20.30, “<element\_id function>”:

- i) Without Feature G100, “ELEMENT\_ID function”, conforming GQL language shall not contain an <element\_id function>.

20) Specifications for Feature G101, “IS DIRECTED predicate”:

a) Subclause 19.7, “<directed predicate>”:

- i) Without Feature G101, “IS DIRECTED predicate”, conforming SQL language shall not contain an <directed predicate>.

21) Specifications for Feature G102, “IS LABELED predicate”:

a) Subclause 19.8, “<labeled predicate>”:

- i) Without Feature G102, “IS LABELED predicate”, conforming GQL language shall not contain an <labeled predicate>.

22) Specifications for Feature G103, “IS SOURCE and IS DESTINATION predicate”:

a) Subclause 19.9, “<source/destination predicate>”:

- i) Without Feature G103, “IS SOURCE and IS DESTINATION predicate”, conforming GQL language shall not contain a <source/destination predicate>.

23) Specifications for Feature G104, “ALL\_DIFFERENT predicate”:

a) Subclause 19.10, “<all\_different predicate>”:

- i) Without Feature G104, “ALL\_DIFFERENT predicate”, conforming GQL language shall not contain an <all\_different predicate>.

24) Specifications for Feature G105, “SAME predicate”:

a) Subclause 19.11, “<same predicate>”:

- i) Without Feature G105, “SAME predicate”, conforming GQL language shall not contain a <same predicate>.

25) Specifications for Feature GB00, “Long identifiers”:

a) Subclause 21.4, “<token> and <separator>”:

- i) Without Feature GB00, “Long identifiers”, in conforming GQL language, the maximum length in characters of the representative form of a <non-delimited identifier> shall be  $2^7 - 1 = 127$ .
- ii) Without Feature GB00, “Long identifiers”, in conforming GQL language, the maximum length in characters of the representative form of a <delimited identifier> shall be  $2^7 - 1 = 127$ .

26) Specifications for Feature GB01, “Double minus sign comments”:

a) Subclause 21.4, “<token> and <separator>”:

- i) Without Feature GB01, “Double minus sign comments”, conforming GQL language shall not contain a <simple comment> introduced with a <double minus sign>.

27) Specifications for Feature GB02, “Double solidus comments”:

a) Subclause 21.4, “<token> and <separator>”:

- i) Without Feature GB02, “Double solidus comments”, conforming GQL language shall not contain a <simple comment> introduced with a <double solidus>.

28) Specifications for Feature GB10, “Explicit request parameters”:

a) Subclause 6.1, “<GQL-request>”:

- i) Without Feature GB10, “Explicit request parameters”, conforming GQL language shall not contain <request parameter set>.

NOTE 196 — This feature only controls the provision of a textual representation of a <request parameter set>. A conforming implementation is required to provide the possibility of specifying a set of request parameters by some means.

## Annex B

(informative)

### **Implementation-defined elements**

This Annex references those features that are identified in the body of this document as implementation-defined.

The term *implementation-defined* is used to identify characteristics that may differ between GQL-implementations, but that shall be defined for each particular implementation.

- 1) References to: "The manner, if it so chooses, in which a GQL-implementation automatically creates a GQL-directory. (A001)":  
 a) Subclause 4.3.5.2, "GQL-directories":  
 i) 5th paragraph
- 2) References to: "The manner, if it so chooses, in which a GQL-implementation automatically populates a GQL-schema upon its creation. (A002)":  
 a) Subclause 4.3.5.3, "GQL-schemas":  
 i) 5th paragraph
- 3) References to: "The result of any operation other than a normalize function or a normalized predicate on an unnormalized character string. (A003)":  
 a) Subclause 4.17.3.2, "Character string types":  
 i) 4th paragraph
- 4) References to: "The rules for determining the actual value of an approximate numeric type from its apparent value. (A004)":  
 a) Subclause 4.17.3.4.2, "Characteristics of numbers":  
 i) 5th paragraph
- 5) References to: "Whether rounding or truncating occurs when least significant digits are lost on assignment. (A005)":  
 a) Subclause 4.17.3.4.2, "Characteristics of numbers":  
 i) 6th paragraph  
 ii) 7th paragraph
- b) Subclause 20.29, "<cast specification>":  
 i) General Rule 3)a)i)  
 ii) General Rule 4)a)i)  
 iii) General Rule 5)a)i)
- c) Subclause 21.1, "<literal>":

- i) General Rule 3)
- 6) References to: "The choice of value selected when there is more than one approximation for an exact numeric type that conforms to the criteria. (A006)":
- a) Subclause 4.17.3.4.2, "Characteristics of numbers":
    - i) 9th paragraph
- 7) References to: "Which which numeric values other than exat numeric types also have approximations. (A007)":
- a) Subclause 4.17.3.4.2, "Characteristics of numbers":
    - i) 10th paragraph
- 8) References to: "The choice of value selected when there is more than one approximation for an approximate numeric type that conforms to the criteria. (A008)":
- a) Subclause 4.17.3.4.2, "Characteristics of numbers":
    - i) 11th paragraph
- 9) References to: "The effect of any additional preamble options provided. (A009)":
- a) Subclause 6.4, "<preamble>":
    - i) General Rule 6)
- 10) References to: "The result of resolving an external object url path. (A010)":
- a) Subclause 17.11, "<external object reference>":
    - i) General Rule 2)
- 11) References to: "Whether rounding or truncating is used on dicision with an approximate mathematical result. (A011)":
- a) Subclause 20.4, "<numeric value expression>":
    - i) General Rule 8)b)
- 12) References to: "The manner in which the result of the concatenation of non-normalized character strings is determined. (A012)":
- a) Subclause 20.7, "<string value expression>":
    - i) General Rule 11)b)ii)2)B)
- 13) References to: "Whether the General Rules of Evaluation of a selective path pattern are terminated if an exception condition is raised. (A013)":
- a) Subclause 22.4, "Evaluation of a selective <path pattern>":
    - i) General Rule 2)
    - ii) General Rule 9)b)ii)
- 14) References to: "The object (principal) that represents a user within a GQL-implementation. (D001)":
- a) Subclause 4.3.4.1, "Principals":
    - i) 1st paragraph
- 15) References to: "The associaation between a principal and its home schema and home graph. (D002)":

- a) Subclause 4.3.4.1, "Principals":
  - i) 3rd paragraph
- 16) References to: "The set of privileges identified by an authorization identifier. (D003)":
  - a) Subclause 4.3.4.2, "Authorization identifiers":
    - i) 2nd paragraph
- 17) References to: "The nesting structure of GQL-directories and GQL-schemas in the GQL-catalog. (D004)":
  - a) Subclause 4.3.5.1, "General description of the GQL-catalog":
    - i) 4th paragraph
- 18) References to: "The kinds of objects an implementation may alias. (D005)":
  - a) Subclause 4.4.7, "Aliases":
    - i) 1st paragraph
- 19) References to: "The period (GQL-Session) in which consecutive GQL-requests are executed by a GQL-client on behalf of a GQL-Agent. (D006)":
  - a) Subclause 4.6.1, "General description of GQL-sessions":
    - i) 1st paragraph
- 20) References to: "Any relaxation of the assumption of the serializable transactional behavior. (D007)":
  - a) Subclause 4.7.1, "General description of GQL-transactions":
    - i) 5th paragraph
- 21) References to: "Whether or not the execution of a data-modifying statement is permitted to occur within the same GQL-transaction as the execution of a catalog-modifying statement. (D008)":
  - a) Subclause 4.7.1, "General description of GQL-transactions":
    - i) 7th paragraph
- 22) References to: "Any additional restrictions, requirements, and conditions imposed on mixed-mode transactions. (D009)":
  - a) Subclause 4.7.1, "General description of GQL-transactions":
    - i) 7th paragraph
- 23) References to: "The conditions raised when the requirements on mixed-mode transactions are violated. (D010)":
  - a) Subclause 4.7.1, "General description of GQL-transactions":
    - i) 7th paragraph
- 24) References to: "The levels of transaction isolation, their interactions, their granularity of application and the means of selecting them. (D011)":
  - a) Subclause 4.7.3, "Transaction isolation":
    - i) 1st paragraph
- 25) References to: "The default preamble. (D012)":

- a) Subclause 4.8.2.1, "Introduction to GQL-request contexts":
    - i) 3rd paragraph
  - b) Subclause 6.3, "<GQL-program>":
    - i) Syntax Rule 1)
- 26) References to: "Any additional preamble options. (D013)":
- a) Subclause 4.8.2.1, "Introduction to GQL-request contexts":
    - i) 4th paragraph
- 27) References to: "The default characteristics for a new execution context. (D014)":
- a) Subclause 4.9.2, "Execution context creation":
    - i) 1) list item
- 28) References to: "Additional conditions for which a completion condition warning (01000) is raised. (D015)":
- a) Subclause 4.10.2, "Conditions":
    - i) 3rd paragraph
- 29) References to: "The translation of condition texts. (D016)":
- a) Subclause 4.10.3, "GQL-status object":
    - i) 2nd list item, of the 2nd list item, of the 3rd paragraph
- 30) References to: "The map of diagnostic information, if provided. (D017)":
- a) Subclause 4.10.3, "GQL-status object":
    - i) 5th list item, of the 3rd paragraph
- 31) References to: "The extent to which further GQL-status objects are chained. (D018)":
- a) Subclause 4.10.3, "GQL-status object":
    - i) 4th paragraph
- 32) References to: "The additional types of local variable definitions provided, if any. (D019)":
- a) Subclause 4.11.4.2, "Variables and parameters":
    - i) 3rd paragraph
- 33) References to: "The additional statements provided, if any. (D020)":
- a) Subclause 4.11.4.3, "Statements":
    - i) 2nd paragraph
- 34) References to: "The preferred name of the Boolean type. (D021)":
- a) Subclause 4.17.3.1, "Boolean types":
    - i) 2nd list item, of the 3rd paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 1)

- 35) References to: "The other cases in which a character string is not assignable only to sites of at least a character string type. (D022)":
- a) Subclause 4.17.3.2, "Character string types":
    - i) 3rd paragraph
- 36) References to: "The preferred name of the character string type. (D023)":
- a) Subclause 4.17.3.2, "Character string types":
    - i) 2nd list item, of the 6th paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 5)
- 37) References to: "The preferred name of the fixed-length byte string type. (D024)":
- a) Subclause 4.17.3.3, "Byte string types":
    - i) 2nd list item, of the 3rd paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 16)
- 38) References to: "Which numeric values have approximations obtained by rounding or truncation for a given approximate numeric type or a given decimal floating-point type. (D025)":
- a) Subclause 4.17.3.4.2, "Characteristics of numbers":
    - i) 12th paragraph
- 39) References to: "Limits on operations on numbers performed according to the normal rules of arithmetic. (D026)":
- a) Subclause 4.17.3.4.2, "Characteristics of numbers":
    - i) 14th paragraph
- 40) References to: "The preferred name of the variable-length byte string type. (D027)":
- a) Subclause 4.17.3.3, "Byte string types":
    - i) 2nd list item, of the 3rd paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 16)
- 41) References to: "The binary precision of a signed regular integer type. (D028)":
- a) Subclause 4.17.3.4.3, "Binary exact numeric types":
    - i) 7th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 37)g)
- 42) References to: "The binary precision of a signed small integer type. (D029)":
- a) Subclause 4.17.3.4.3, "Binary exact numeric types":
    - i) 8th list item, of the 1st paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 37)h)

43) References to: "The binary precision of a signed big integer type. (D030)":

a) Subclause 4.17.3.4.3, "Binary exact numeric types":

i) 9th list item, of the 1st paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 37)i)

44) References to: "The binary precision of a signed user-specified integer type. (D031)":

a) Subclause 4.17.3.4.3, "Binary exact numeric types":

i) 10th list item, of the 1st paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 37)j)

45) References to: "The binary precision of an unsigned regular integer type. (D032)":

a) Subclause 4.17.3.4.3, "Binary exact numeric types":

i) 7th list item, of the 2nd paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 38)g)

46) References to: "The binary precision of an unsigned user-specified integer type. (D033)":

a) Subclause 4.17.3.4.3, "Binary exact numeric types":

i) 8th list item, of the 2nd paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 38)h)

47) References to: "The decimal precision of a regular decimal exact numeric type. (D034)":

a) Subclause 4.17.3.4.4, "Decimal exact numeric types":

i) 1st list item, of the 1st paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 41)c)

48) References to: "The decimal precision of a user-specified decimal exact numeric type without a scale specification. (D035)":

a) Subclause 4.17.3.4.4, "Decimal exact numeric types":

i) 2nd list item, of the 1st paragraph

b) Subclause 21.2, "<value type>":

i) Syntax Rule 41)b)

- 49) References to: "The decimal precision of a user-specified decimal exact numeric type with a scale specification. (D036)":
- a) Subclause 4.17.3.4.4, "Decimal exact numeric types":
    - i) 3rd list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 41)a)
- 50) References to: "The binary precision of a regular approximate numeric type. (D037)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 6th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)f)
  - c) Subclause 22.8, "Result of value type combinations":
    - i) General Rule 3)c)ii)1)
- 51) References to: "The binary scale of a regular approximate numeric type. (D038)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 6th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)f)
- 52) References to: "The binary precision of a real approximate numeric type. (D039)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 7th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)g)
- 53) References to: "The binary scale of a real approximate numeric type. (D040)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 7th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)g)
- 54) References to: "The binary precision of a double approximate numeric type. (D041)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 8th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)h)
- 55) References to: "The binary scale of a double approximate numeric type. (D042)":

**B Implementation-defined elements**

- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 8th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)h)
- 56) References to: "The binary precision of a user-specified approximate numeric type without a scale specification. (D043)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 9th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)j)i)
- 57) References to: "The binary scale of a user-specified approximate numeric type without a scale specification. (D044)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 9th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)j)i)
- 58) References to: "The binary precision of a user-specified approximate numeric type with a scale specification. (D045)":
- a) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)j)ii)
- 59) References to: "The binary scale of a user-specified approximate numeric type with a scale specification. (D046)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 10th list item, of the 1st paragraph
  - b) Subclause 21.2, "<value type>":
    - i) Syntax Rule 45)j)ii)
- 60) References to: "The treatment of language that does not conform to the Formats and Syntax Rules. (D047)":
- a) Subclause 4.17.3.4.5, "Approximate numeric types":
    - i) 10th list item, of the 1st paragraph
  - b) Subclause 5.3.3.2, "Terms denoting rule requirements":
    - i) 1st paragraph
- 61) References to: "The default time zone identifier for a GQL-session context. (D048)":
- a) Subclause 6.1, "<GQL-request>":
    - i) General Rule 2)a)iii)

62) References to: "The session default parameters for a GQL-session context. (D049)":

- a) Subclause 6.1, "<GQL-request>":
  - i) General Rule 2)a)vi)

63) References to: "The session default parameter flags for a GQL-session context. (D050)":

- a) Subclause 6.1, "<GQL-request>":
  - i) General Rule 2)a)vii)

64) References to: "The additional preamble options provided. (D051)":

- a) Subclause 6.4, "<preamble>":
  - i) Syntax Rule 1)

65) References to: "The format, contents and means of reporting the profiling information. (D052)":

- a) Subclause 6.4, "<preamble>":
  - i) General Rule 3)

66) References to: "The format, contents and means of reporting the query plan. (D053)":

- a) Subclause 6.4, "<preamble>":
  - i) General Rule 5)

67) References to: "The Format and Syntax Rules for an implementation-defined access mode. (D054)":

- a) Subclause 8.3, "<transaction characteristics>":
  - i) Syntax Rule 4)

68) References to: "Additional exception condition subclass codes for transaction rollback. (D055)":

- a) Subclause 8.5, "<commit command>":
  - i) General Rule 2)b)

69) References to: "Additional types of binding variable declarations and definitions provided. (D056)":

- a) Subclause 10.5, "Binding variable and parameter declarations and definitions":
  - i) General Rule 1st

70) References to: "The exact numeric type of array element ordinals. (D057)":

- a) Subclause 15.7.6, "<for statement>":
  - i) General Rule 3)e)iii)1)

71) References to: "The exact numeric type of array element indexes. (D058)":

- a) Subclause 15.7.6, "<for statement>":
  - i) General Rule 3)e)iii)2)

72) References to: "The declared type of the result of COUNT function. (D059)":

- a) Subclause 16.19, "<aggregate function>":
  - i) Syntax Rule 7)

73) References to: "The implicit null ordering. (D060)":

- a) Subclause 16.20, "<sort specification list>":  
 i) Syntax Rule 6)

74) References to: "The exception condition(s) to be raised on a failure to resolve an external object url path. (D061)":

- a) Subclause 17.11, "<external object reference>":  
 i) General Rule 2)

75) References to: "The declared type of an unsigned integer specification. (D062)":

- a) Subclause 20.1, "<value specification>":  
 i) Syntax Rule 5)

76) References to: "The declared type of the result of a dyadic arithmetic operator when either operand is approximate numeric. (D063)":

- a) Subclause 20.4, "<numeric value expression>":  
 i) General Rule 3)a)

77) References to: "The declared type of the result of a dyadic arithmetic operator when both operands are exact numeric. (D064)":

- a) Subclause 20.4, "<numeric value expression>":  
 i) General Rule 3)b)

78) References to: "The precision of the result of addition and subtraction. (D065)":

- a) Subclause 20.4, "<numeric value expression>":  
 i) General Rule 3)b)ii)

79) References to: "The precision of the result of multiplication. (D066)":

- a) Subclause 20.4, "<numeric value expression>":  
 i) General Rule 3)b)iii)

80) References to: "The precision and scale of the result of division. (D067)":

- a) Subclause 20.4, "<numeric value expression>":  
 i) General Rule 3)b)iv)

81) References to: "The declared type of the result of a length expression. (D068)":

- a) Subclause 20.6, "<numeric value function>":  
 i) General Rule 1)

82) References to: "The declared type of the result of a trigonometric function. (D069)":

- a) Subclause 20.6, "<numeric value function>":  
 i) General Rule 4)

83) References to: "The declared type of the result of a general logarithm function. (D070)":

- a) Subclause 20.6, "<numeric value function>":

i) General Rule 5)

84) References to: "The declared type of the result of a natural logarithm. (D071)":

- a) Subclause 20.6, "<numeric value function>":

i) General Rule 6)

85) References to: "The declared type of the result of an exponential function. (D072)":

- a) Subclause 20.6, "<numeric value function>":

i) General Rule 7)

86) References to: "The declared type of the result of a power function. (D073)":

- a) Subclause 20.6, "<numeric value function>":

i) General Rule 8)

87) References to: "The precision of an exact numeric result of a numeric value expression. (D074)":

- a) Subclause 20.6, "<numeric value function>":

i) General Rule 9)a)

88) References to: "The precision of an approximate numeric result of a numeric value expression. (D075)":

- a) Subclause 20.6, "<numeric value function>":

i) General Rule 9)b)

89) References to: "The declared type of an element\_id function. (D076)":

- a) Subclause 20.30, "<element\_id function>":

i) Syntax Rule 2)

90) References to: "The Access Rules for element\_id function. (D077)":

- a) Subclause 20.30, "<element\_id function>":

i) Access Rule 1)

91) References to: "The maximum number of digits permitted in an unsigned integer literal. (D078)":

- a) Subclause 21.1, "<literal>":

i) Syntax Rule 9)

92) References to: "The declared type of an approximate numeric literal. (D079)":

- a) Subclause 21.1, "<literal>":

i) Syntax Rule 12)

93) References to: "The decimal precision of a user-specified decimal exact numeric type with a scale specification. (D080)":

- a) Subclause 4.17.3.4.4, "Decimal exact numeric types":

i) 3rd list item, of the 1st paragraph

- b) Subclause 21.2, "<value type>":

## i) Syntax Rule 41)a)

94) References to: "Whether a GQL Flagger flags implementation-defined features. (D081)":

## a) Subclause 25.5.3, "Extensions and options":

## i) 4th paragraph

95) References to: "The maximum precision of an exact numeric type. (D082)":

## a) Subclause 21.2, "&lt;value type&gt;":

## i) Syntax Rule 44)a)

96) References to: "The maximum scale of an exact numeric type. (D083)":

## a) Subclause 21.2, "&lt;value type&gt;":

## i) Syntax Rule 44)b)

97) References to: "The character (code) interpreted as newline. (D084)":

## a) Subclause 21.4, "&lt;token&gt; and &lt;separator&gt;":

## i) Syntax Rule 10)

98) References to: "The mechanism to instruct a GQL-client to create and destroy GQL-sessions to GQL-servers, and to submit GQL-requests to them. (M001)":

## a) Subclause 4.3.2, "GQL-agents":

## i) 1st paragraph

99) References to: "The means of creating and destroying authorization identifiers, and their mapping to principals. (M002)":

## a) Subclause 4.3.4.1, "Principals":

## i) 2nd paragraph

100) References to: "The mechanism and rules by which a GQL-implementation determines when the last request has been received. (M003)":

## a) Subclause 4.6.1, "General description of GQL-sessions":

## i) 2nd paragraph

101) References to: "Alternative means of starting and terminating of transactions. (M004)":

## a) Subclause 4.7.2, "Transaction demarcation":

## i) 1st paragraph

102) References to: "The way in which termination success or failure statuses are made available to the GQL-agent or administrator. (M005)":

## a) Subclause 4.7.2, "Transaction demarcation":

## i) 10th paragraph

103) References to: "The mechanism that is functionally equivalent to the request parameter set specification, if not directly supported. (M006)":

## a) Subclause 4.8.1, "General description of GQL-requests":

- i) 2nd list item, of the 1st paragraph
- b) Subclause 6.1, “<GQL-request>”:
  - i) Syntax Rule 1)

104) References to: “How a GQL-status object is presented to a GQL-client. (M007)”:

- a) Subclause 4.10.1, “Introduction to diagnostics”:
  - i) 3rd paragraph

105) References to: “The way of determining the graph type of a graph without an explicitly specified graph type. (M008)”:

- a) Subclause 20.1, “<value specification>”:
  - i) Syntax Rule 3)

106) References to: “The way in which a binding table type is inferred. (M009)”:

- a) Subclause 20.1, “<value specification>”:
  - i) Syntax Rule 4)

107) References to: “The mechanism by which an external procedure is provided. (M010)”:

- a) Subclause 4.11.2.8, “Procedures classified by type of provisioning”:
  - i) 2nd list item, of the 1st paragraph

108) References to: “The maximum cardinality of a graph label set. (S001)”:

- a) Subclause 4.4.3.1, “Introduction to graphs”:
  - i) 1st list item, of the 3rd paragraph

109) References to: “The maximum cardinality of a graph property set. (S002)”:

- a) Subclause 4.4.3.1, “Introduction to graphs”:
  - i) 1st list item, of the 3rd paragraph

110) References to: “The maximum cardinality of a node label set. (S003)”:

- a) Subclause 4.4.3.1, “Introduction to graphs”:
  - i) 1st list item, of the 3rd list item, of the 3rd paragraph

111) References to: “The maximum cardinality of a node property set. (S004)”:

- a) Subclause 4.4.3.1, “Introduction to graphs”:
  - i) 1st list item, of the 3rd list item, of the 3rd paragraph

112) References to: “The maximum cardinality of an edge label set. (S005)”:

- a) Subclause 4.4.3.1, “Introduction to graphs”:
  - i) 1st list item, of the 4th list item, of the 3rd paragraph

113) References to: “The maximum cardinality of an edge property set. (S006)”:

- a) Subclause 4.4.3.1, “Introduction to graphs”:
  - i) 1st list item, of the 4th list item, of the 3rd paragraph

114) References to: "The maximum cardinality of a graph type label set. (S007)":

- a) Subclause 4.15.1, "Introduction to graph types":  
 i) 1st list item, of the 1st paragraph

115) References to: "The maximum cardinality of a graph type property set. (S008)":

- a) Subclause 4.15.1, "Introduction to graph types":  
 i) 1st list item, of the 1st paragraph

116) References to: "The maximum cardinality of a node type label set. (S009)":

- a) Subclause 4.15.3.3, "Node types":  
 i) 1st list item, of the 2nd paragraph

117) References to: "The maximum cardinality of a node type property set. (S010)":

- a) Subclause 4.15.3.3, "Node types":  
 i) 1st list item, of the 2nd paragraph

118) References to: "The maximum cardinality of an edge type label set. (S011)":

- a) Subclause 4.15.3.6, "Edge types":  
 i) 1st list item, of the 2nd paragraph

119) References to: "The maximum cardinality of an edge type property set. (S012)":

- a) Subclause 4.15.3.6, "Edge types":  
 i) 1st list item, of the 2nd paragraph

120) References to: "The maximum length of a character string. (S013)":

- a) Subclause 4.17.3.2, "Character string types":  
 i) 2nd paragraph
- b) Subclause 20.7, "<string value expression>":  
 i) General Rule 11)b)ii)3)A)
- c) Subclause 20.8, "<string value function>":  
 i) General Rule 14)d)  
 ii) General Rule 15)e)
- d) Subclause 21.2, "<value type>":  
 i) Syntax Rule 7)  
 ii) Syntax Rule 10)

121) References to: "The maximum length of a byte string. (S014)":

- a) Subclause 4.17.3.3, "Byte string types":  
 i) 2nd paragraph
- b) Subclause 20.7, "<string value expression>":

- i) General Rule 12)b)i)
- c) Subclause 21.2, “<value type>”:
  - i) Syntax Rule 22)
  - ii) Syntax Rule 27)

122) References to: “The maximum cardinality of a list type. (S015)”:

- a) Subclause 20.16, “<list value expression>”:
  - i) General Rule 4)b)
- b) Subclause 20.18, “<list value constructor>”:
  - i) General Rule 2)

123) References to: “The maximum precision of an exact numeric type. (S016)”:

- a) Subclause 21.2, “<value type>”:
  - i) Syntax Rule 34)a)
- b) Subclause 22.8, “Result of value type combinations”:
  - i) General Rule 3)c)ii)2)

124) References to: “The maximum scale of an exact numeric type. (S017)”:

- a) Subclause 21.2, “<value type>”:
  - i) Syntax Rule 34)b)

## Annex C (informative)

### Implementation-dependent elements

This Annex references those places where this document states explicitly that the actions of a conforming GQL-implementation are implementation-dependent.

The term *implementation-dependent* is used to identify characteristics that may differ between GQL-implementations, but that are not necessarily specified for any particular implementation.

- 1) Subclause 4.3.1, "General description of GQL-environments":
  - a) 2nd paragraph(paraphrased): The interaction between multiple GQL-environments is implementation-dependent.
- 2) Subclause 4.4.1, "General information about GQL-objects":
  - a) 2nd paragraph: Every GQL-object is identified by one or more effectively immutable *internal object identifiers* that are unique within the GQL-Environment containing the GQL-object. These internal object identifiers of a GQL-object are determined in an implementation-dependent way when the GQL-object is created. Any two internal object identifiers are considered equivalent if they identify the same GQL-object. Any two separately created GQL-objects shall effectively never be identified by the same internal object identifier during a GQL-session. An internal object identifier of a GQL-object is never exposed directly to the user in this document.
- 3) Subclause 4.4.3.1, "Introduction to graphs":
  - a) 1st list item, of the 3rd list item, of the 3rd paragraph: An identifier that is unique within the graph.
 

NOTE 197 — The value of a node identifier is implementation-dependent and is possibly not accessible to the user. This unique identifier is used for definitional purposes to establish the identity of the node.
  - b) 1st list item, of the 4th list item, of the 3rd paragraph: An identifier that is unique within the graph.
 

NOTE 198 — The value of an edge identifier is implementation-dependent and is possibly not accessible to the user. This unique identifier is used for definitional purposes to establish the identity of the edge.
- 4) Subclause 4.4.5, "Binding tables":
  - a) 9th paragraph: If a General Rule *RULE* refers to the *i*-th record *RECORD* of an unordered binding table *TABLE*, then *RECORD* is the *i*-th record in some sequence of all records of *TABLE* in an implementation-dependent order determined for each application of *RULE* and any of its sub rules.
  - b) 10th paragraph: If the application of a General Rule *RULE* processes all records of a binding table using order-independent terms, then the records are to be processed effectively in a sequential, implementation-dependent order determined for that application of *RULE* and any of its sub rules.
- 5) Subclause 4.10.2, "Conditions":
  - a) 9th paragraph: If more than one condition could have occurred as a result of a statement, it is implementation-dependent whether diagnostic information pertaining to more than one condition

is made available. Those addition conditions, if any, are placed in a separate GQL-status object and chained to the GQL-status object containing the GQLSTATUS with the greatest precedence.

6) Subclause 4.13.1, “General information about data types”:

- a) **1st paragraph:** A *data type* is a set of elements with shared characteristics representing data. An *object type* is a *data type* comprising *objects*, a *value type* is a *data type* comprising *values*, a *reference value type* is a *value type* comprising *reference values*, and a *property value type* is a *value type* comprising *property values*. A *data type* characterizes the sites that may be occupied by instances of its elements. Every instance belongs to multiple *data types*. A *data type* characterizes the sites that may be occupied by instances of its elements. The physical representation of an instance of a *data type* is implementation-dependent.

7) Subclause 5.3.2, “Specification of the Information Graph Schema”:

- a) **2nd paragraph:** The only purpose of the Information Graph Schema is to provide access to the definitions contained in the schema. The actual objects on which the Information Graph Schema is based are implementation-dependent.

8) Subclause 5.3.3.3, “Rule evaluation order”:

- a) **4th paragraph:** Where the precedence is not determined by the Formats or by parentheses, effective evaluation of expressions is *generally* performed from left to right. However, it is implementation-dependent whether expressions are *actually* evaluated left to right, particularly when the evaluation of operands or operators causes conditions to be raised or if the results of the expressions may be determined without completely evaluating all parts of the expression.
- b) **10th paragraph:** If evaluation of an inessential part would cause an exception condition to be raised, then it is implementation-dependent whether or not that exception condition is raised.
- c) **11th paragraph:** During the computation of the result of an expression, the GQL-implementation may produce one or more *intermediate results* that are used in determining that result. The declared type of a site that contains an intermediate result is implementation-dependent.

9) Subclause 14.4, “<insert statement>”:

- a) **Syntax Rule 4)c)ii):** Otherwise, let *EVAR* be an implementation-dependent <identifier> distinct from every element variable, subpath variable and path variable contained in *GP*.

10) Subclause 15.6.1, “<match statement>”:

- a) **Syntax Rule 4)c)ii):** Otherwise, let *EVAR* be an implementation-dependent <identifier> distinct from every element variable, subpath variable and path variable contained in *GP*.

11) Subclause 15.7.6, “<for statement>”:

- a) **General Rule 3)c)ii):** If  $FCVER_i$  is a set or a multiset, then  $LV_i$  is an implementation-dependent permutation of all elements of  $FIVER_i$ .

12) Subclause 15.8.2, “<return statement>”:

- a) **General Rule 4)b)iv):** Let *KEYS* be a permutation of *GROUP\_BY* in an implementation-dependent order and let *NKEYS* be the number of elements of *KEYS*.

13) Subclause 16.6, “<graph pattern>”:

- a) **General Rule 12):** Let *MATCHES* be a new binding table comprising the match records obtained for each reduced match of *GP* in an implementation-dependent order.

14) Subclause 16.7, “<path pattern expression>”:

## C Implementation-dependent elements

- a) **Syntax Rule 10)b)**: Let  $NOPMA$  be the number of multiset alternation operands of  $PMA$ . Let  $OPMA_1, \dots OPMA_{NOPMA}$  be an enumeration of the operands of  $PMA$ . Let  $SOPMA_1, \dots SOPMA_{NOPMA}$  be implementation-dependent <identifier>s that are mutually distinct and distinct from every element variable, subpath variable and path variable contained in  $GP$ .

## 15) Subclause 16.20, "&lt;sort specification list&gt;":

- a) **General Rule 1)g)**: If  $PV_i$  and  $QV_i$  are not the null value and the result of " $PV_i <\text{comp op}> QV_i$ " is Unknown, then the relative ordering of  $PV_i$  and  $QV_i$  is implementation-dependent.
- b) **General Rule 1)i)**: Two rows that are not distinct with respect to the <sort specification>s are said to be *peers* of each other. The relative ordering of peers is implementation-dependent.

## 16) Subclause 16.21, "&lt;limit clause&gt;":

- a) **General Rule 2)**: If  $TABLE$  is not ordered, then let  $ORDERED\_TABLE$  be a new ordered binding table created from the result of sorting the collection of all records of  $TABLE$  according to an implementation-dependent order. Otherwise, let  $ORDERED\_TABLE$  be  $TABLE$ .

## 17) Subclause 16.22, "&lt;offset clause&gt;":

- a) **General Rule 2)**: If  $TABLE$  is not ordered, then let  $ORDERED\_TABLE$  be a new ordered binding table created from the result of sorting the collection of all records of  $TABLE$  according to an implementation-dependent order. Otherwise, let  $ORDERED\_TABLE$  be  $TABLE$ .

## 18) Subclause 20.30, "&lt;element\_id function&gt;":

- a) **General Rule 2)b)**: Otherwise, let  $GE$  be the sole graph element in  $LOE$ . The value of <element\_id function> is an implementation-dependent value that encapsulates the identity of  $GE$  in the graph that contains  $GE$  for the duration of the innermost executing statement.

## 19) Subclause 22.4, "Evaluation of a selective &lt;path pattern&gt;":

- a) **General Rule 2)**: It is implementation-defined whether the General Rules of this Subclause are terminated if an exception condition is raised. If an implementation defines that it terminates execution because of an exception condition, it is implementation-dependent which of the members of  $CANDIDATES$  (defined subsequently) are actually probed to establish whether they might raise an exception.

NOTE 199 —  $CANDIDATES$  is potentially an infinite set, but there are algorithms to enumerate this set so as to satisfy the selection criterion of the selective <path pattern> without testing all candidate solutions. Even if the implementation defines that it terminates when an exception condition is encountered on a particular candidate solution, the order of enumerating the candidates may be implementation-dependent, and a candidate solution that might raise an exception may never be tested.

- b) **General Rule 2)**: It is implementation-defined whether the General Rules of this Subclause are terminated if an exception condition is raised. If an implementation defines that it terminates execution because of an exception condition, it is implementation-dependent which of the members of  $CANDIDATES$  (defined subsequently) are actually probed to establish whether they might raise an exception.

NOTE 200 —  $CANDIDATES$  is potentially an infinite set, but there are algorithms to enumerate this set so as to satisfy the selection criterion of the selective <path pattern> without testing all candidate solutions. Even if the implementation defines that it terminates when an exception condition is encountered on a particular candidate solution, the order of enumerating the candidates may be implementation-dependent, and a candidate solution that might raise an exception may never be tested.

- c) **General Rule 9)b)ii)**: If  $PPPE$  contains a <parenthesized path pattern where clause>  $PPPWC$ , then the value of  $V$  is True when the General Rules of Subclause 22.5, "Applying bindings to evaluate an expression", are applied with  $PPL$  as **GRAPH PATTERN**, the <search condition> simply contained in  $PPPWC$  as **EXPRESSION**,  $MACH$  as **MACHINERY**,  $PBX$  as **MULTI-PATH BINDING**, and a reference

to *BPPPE* as a subset of *PBX* as *REFERENCE TO LOCAL CONTEXT*; let *V* be the *VALUE* returned from the application of those General Rules.

NOTE 201 — This is the juncture at which an exception condition might be raised. It is implementation-defined whether to terminate if an exception condition is raised. The order of enumerating the members of *CANDIDATES* is implementation-dependent, and there is no requirement that an implementation test all candidate solutions, which may be an infinite set in any case.

- d) **General Rule 13)a)ii):** Otherwise, it is implementation-dependent which *N* path bindings of *PART* are retained.
- e) **General Rule 13)b)i)2):** Otherwise, the path bindings of *PART* are sorted in increasing order of number of edges; the order of path bindings that have the same number of edges is implementation-dependent. The first *N* path bindings in *PART* are retained.

## Annex D (informative)

### **GQL feature taxonomy**

This Annex describes a taxonomy of features defined in this document.

**Table 9, “Feature taxonomy for optional features”**, contains a taxonomy of the optional features of the GQL language.

In this table, the first column contains a counter that can be used to quickly locate rows of the table; these values otherwise have no use and are not stable — that is, they are subject to change in future editions of or even Technical Corrigenda to this document without notice.

The “Feature ID” column of this table specifies the formal identification of each feature and each subfeature contained in the table.

The “Feature Name” column of this table contains a brief description of the feature or subfeature associated with the Feature ID value.

**Table 9, “Feature taxonomy for optional features”**, does not provide definitions of the features; the definition of those features is found in the Conformance Rules that are further summarized in [Annex A, “GQL Conformance Summary”](#).

**Table 9 — Feature taxonomy for optional features**

|           | <b>Feature ID</b> | <b>Feature Name</b>                         |
|-----------|-------------------|---------------------------------------------|
| <b>1</b>  | <b>G001</b>       | <b>Undirected edge patterns</b>             |
| <b>2</b>  | <b>G010</b>       | <b>Graph Pattern Keep</b>                   |
| <b>3</b>  | <b>G011</b>       | <b>Graph Pattern Where</b>                  |
| <b>4</b>  | <b>G020</b>       | <b>Path Multiset Alternation</b>            |
| <b>5</b>  | <b>G021</b>       | <b>Path Pattern Union</b>                   |
| <b>6</b>  | <b>G030</b>       | <b>Quantified Paths</b>                     |
| <b>7</b>  | <b>G031</b>       | <b>Quantified Edges</b>                     |
| <b>8</b>  | <b>G040</b>       | <b>Questioned Paths</b>                     |
| <b>9</b>  | <b>G050</b>       | <b>Simplified Path Pattern Expression</b>   |
| <b>10</b> | <b>G060</b>       | <b>Non-local element pattern predicates</b> |
| <b>11</b> | <b>G070</b>       | <b>Abbreviated Edge Patterns</b>            |
| <b>12</b> | <b>G081</b>       | <b>Parenthesized Path Pattern Where</b>     |

|    | Feature ID | Feature Name                           |
|----|------------|----------------------------------------|
| 13 | G082       | Non-local predicates                   |
| 14 | G083       | Parenthesized Path Pattern Cost        |
| 15 | G091       | Shortest Path Search                   |
| 16 | G092       | Advanced Path Modes: TRAIL             |
| 17 | G093       | Advanced Path Modes: SIMPLE            |
| 18 | G094       | Advanced Path Modes: ACYCLIC           |
| 19 | G100       | ELEMENT_ID function                    |
| 20 | G101       | IS DIRECTED predicate                  |
| 21 | G102       | IS LABELED predicate                   |
| 22 | G103       | IS SOURCE and IS DESTINATION predicate |
| 23 | G104       | ALL_DIFFERENT predicate                |
| 24 | G105       | SAME predicate                         |
| 25 | GA00       | Graph label set support                |
| 26 | GA01       | Graph property set support             |
| 27 | GA02       | Empty node label set support           |
| 28 | GA03       | Singleton node label set support       |
| 29 | GA04       | Unbounded node label set support       |
| 30 | GA05       | Empty edge label set support           |
| 31 | GA06       | Singleton edge label set support       |
| 32 | GA07       | Unbounded edge label set support       |
| 33 | GA08       | Named node types in graph types        |
| 34 | GA09       | Named edge types in graph types        |
| 35 | GB00       | Long identifiers                       |
| 36 | GB01       | Double minus sign comments             |
| 37 | GB02       | Double solidus comments                |
| 38 | GB10       | Explicit request parameters            |

## **Annex E** (informative)

### **Maintenance and interpretation of GQL**

ISO/IEC JTC 1 provides formal procedures for revision, maintenance, and interpretation of JTC 1 Standards, including creation and processing of "defect reports". Defect reports may result in technical corrigenda, amendments, interpretations, or other commentary on an existing International Standard.

A defect report may be submitted by a national standards body that is a P-member or O-member, a Liaison Organization, a member of the defect editing group for the subject document, or a working group of the committee responsible for the document. A defect identified by the user of the standard, or someone external to the committee, shall be processed via one of the official channels listed above. The submitter shall complete part 2 of the defect report form (see the Defect Report form in the Templates folder at the JTC 1 web site, as well as its attachment 1) and shall send the form to the Convenor or WG Secretariat with which the relevant defect editing group is associated.

Potential new questions or new defect reports addressing the specifications of this document should be communicated to:

Secretariat, ISO/IEC JTC1/SC32

American National Standards Institute

11 West 42nd Street

New York, NY 10036

USA

## Annex F (informative)

### Differences with SQL

This Annex identifies the aspects of the GQL language that differ from the equivalent aspects of the SQL language.

#### 1) Identifiers

SQL has 4 different types of identifiers:

- Regular identifiers
- Delimited identifiers
- Unicode delimited identifiers
- SQL language identifiers

GQL only has two types:

- Non-delimited identifiers
- Delimited identifiers

In SQL, “SQL language identifier” is only used to name character sets and since GQL only recognizes the Unicode character set this form of identifier is unnecessary in GQL.

In both SQL and GQL, a *<regular identifier>* is an *<identifier start>* followed by zero or more *identifier parts* (where an identifier part is either an *<identifier start>* or an *<identifier extend>*). GQL defines an additional element for specifying a non-delimited identifier that is called an *<extended identifier>* and that starts with *<identifier extend>* instead of *<identifier start>*.

However, the definitions of *<identifier start>* and *<identifier extend>* differ slightly between the two languages.

In SQL, *<identifier start>* is any character in the Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, and “Nl” (i.e., upper-case letters, lower-case letters, title-case letters, modifier letters, other letters, and letter numbers).

In GQL, *<identifier start>* is any character in the Unicode property ID\_Start (this includes the Unicode General Category classes “Lu”, “Ll”, “Lt”, “Lm”, “Lo”, and “Nl” but also those with the Unicode property Other\_ID\_Start but none of which have the Unicode properties Pattern\_Syntax or Pattern\_White\_Space).

Thus, GQL excludes \u2E2F (VERTICAL TILDE) from “Lm”, which SQL allows, but adds \u1885, \u1886, \u2118, \u212E, \u309B, and \309C (MONGOLIAN LETTER ALI GALI BALUDA, MONGOLIAN LETTER ALI GAL, SCRIPT CAPITAL P, ESTIMATED SYMBOL, KATAKANA-HIRAGANA VOICED SOUND MARK, and KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK, respectively), which SQL does not.

In SQL, *<identifier extend>* is \u00B7 (Middle Dot), or any character in the Unicode General Category classes “Mn”, “Mc”, “Nd”, “Pc”, and “Cf” (i.e., non-spacing marks, spacing combining marks, decimal numbers, connector punctuations, and formatting codes).

In GQL, <identifier extend> is any character in the Unicode property ID\_Continue (this includes the Unicode General Category classes “Mn”, “Mc”, “Nd”, and “Pc” but also those with the Unicode property Other\_ID\_Continue but none of which have the Unicode properties Pattern\_Syntax or Pattern\_White\_Space).

Thus, GQL excludes the characters in “Cf”, which SQL allows, but adds \u0387, \u1369...\u1371, and \19DA (GREEK ANO TELEIA, ETHIOPIC DIGIT ONE...ETHIOPIC DIGIT NINE, and NEW TAI LUE THAM DIGIT ONE, respectively), which SQL does not.

In GQL, the non-delimited identifiers are case-sensitive, in SQL they are not.

In SQL non-delimited identifiers are limited to either 18 characters or 128 characters depending on the Feature supported and delimited identifiers are limited to either 18 or 128 characters depending on the Feature supported.

In GQL, the length of identifiers is limited to either 127 or 16383 characters depending on the Feature supported.

SQL distinguishes between “delimited identifiers” and “Unicode delimited identifiers” but GQL, because it is Unicode based, does not. GQL allows <escaped character>s including <unicode escape value>s in a delimited identifier whereas SQL does not. SQL, in “Unicode delimited identifiers”, allows the specification of an escape character (which in Unicode and GQL is “\”) and its <unicode escape value>s do not completely conform to the Unicode specification.

- 2) Binding tables in GQL are defined as collections of records whose fields are values of some data type. Records may have zero fields but are not permitted to have multiple fields with the same name. Consequently, a binding table may have zero columns but is not permitted to have multiple columns with the same column name. Furthermore, two binding tables with the same collection of records that only differ in their canonical column sequence are considered equal by most operations.
- 3) Data types

In GQL, the term data type is used more broadly: It encompasses both value types (what SQL calls a “data type”) and object types. Furthermore, references to objects may be represented by reference values in GQL.

a) Character String

SQL has 3 different types of character strings

- CHARACTER
- CHARACTER VARYING
- CHARACTER LARGE OBJECT

All of these have a variety of ways of being specified:

- CHARACTER may also be expressed as CHAR, NATIONAL CHARACTER, NATIONAL CHAR, or NCHAR.
- CHARACTER VARYING may also be expressed as: CHAR VARYING, VARCHAR, NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, or NCHAR VARYING.
- CHARACTER LARGER OBJECT may also be expressed as: CHAR LARGE OBJECT or CLOB, NATIONAL CHARACTER LARGE OBJECT, NCHAR LARGE OBJECT, or NCLOB, but surprisingly not NATIONAL CHAR LARGE OBJECT.

All of the types may specify a character set and a collation, except when the keyword includes NATIONAL, in which case the character set is implicit.

SQL allows support of zero-length character strings to be optional, i.e., implementation-defined.

GQL only has one character string type, specified as either STRING or as VARCHAR. This is roughly equivalent to SQL's CHARACTER VARYING.

GQL requires support for zero-length character strings and considers them to be different from the null value.

GQL supports only one character set: UNICODE.

GQL does not currently support the specification of a collation.

GQL does not currently support character large objects.

b) Byte String

SQL has 3 different types of byte strings

- BINARY
- BINARY VARYING
- BINARY LARGE OBJECT

All of these have a variety of ways of being specified:

- BINARY may also be expressed as: BINARY FIXED.
- BINARY VARYING may also be expressed as: VARBINARY.
- BINARY LARGE OBJECT may also be expressed as: BLOB.

GQL only has one byte string type but distinguishes between fixed-length byte string types specified as either BYTES or as BINARY and variable-length byte string types specified as either BYTES or as VARBINARY. This is roughly equivalent to SQL's BINARY FIXED and BINARY VARYING.

c) JSON

SQL supports a JSON type. GQL does not but provides its own method of support for nested data whose information model is a superset of JSON extending it with a richer set of containers as well as allowing all supported value types as the attributes of records (its equivalent of "JSON objects").

d) Row types

SQL support for row types shows up in multiple places including explicit ROW constructors, row subqueries and in multiple predicates. For example, the SQL <null predicate> has a special case to loop through all of the elements in a row and determine that an instance of a Row type is NULL if all of the elements in that instance are NULL. GQL does not include row types. GQL records are considered to be NOT NULL if they exist, even if all fields in a record are NULL.

4) The type system of GQL is based on subtyping: Structural subtyping is used for record types, graph element content types, and the graph types they constitute and extends to reference value types for references to graphs and graph elements. Furthermore, every data type of GQL is organized in a type hierarchy that includes values, reference values, and the collections containing them, as well as objects.

5) Parameter specification in functions

SQL uses embedded keywords in the parameter list of *some* functions. GQL uses *only* comma separation of procedure parameters.

6) Terminal characters

SQL allows only <standard digit>s as <digit>s. GQL allows any Unicode character in the Unicode General Category class “Nd”.

SQL uses <right bracket trigraph> and <left bracket trigraph> but, since GQL requires Unicode support, these trigraphs not needed and therefore not supported.

7) <string value function>

SQL has various styles of regular expression substring functions but GQL does not have any. See Language Opportunity **GQL-032**.

8) <trim function>

In GQL, the TRIM function parameters are ( <trim source> [ <comma> <trim specification> ] <trim character string> ) while in SQL, the TRIM function parameters are ( [ [ <trim specification> ] [ <trim character> ] FROM ] <trim source> ).

9) <comparison predicate>

SQL has collations that allow the specification of NO PAD or PAD SPACE. Since GQL does not (currently) support collations, every comparison uses PAD SPACE.

10) <as clause>

In SQL, in a <derived column>, the keyword AS is optional. In GQL, in a <select item alias> in a <select item>, the keyword AS is not optional.

11) <set quantifier>

In SQL, the <set quantifier> INTERSECT takes precedence over the <set quantifier>s UNION and EXCEPT, cf. <query expression body>. GQL does not specify the precedence between the INTERSECT, UNION, and EXCEPT operators but prohibits mixed sequences of these operators without explicit nesting, cf. <composite query expression>. In GQL, the effective precedence between these operators is always explicit in the query.

12) SQL includes two forms of comments, “Double minus sign comments” (required) and “Bracketed comments” (via advanced Feature T351). GQL supports three forms of comments, “Bracketed comments” (required), “Double minus sign comments” (via advanced Feature GB01), and “Double solidus comments” (via advanced Feature GB02).

## Annex G (informative)

### **Graph serialization format**

This Annex identifies how parts of the GQL language can be used to serialize graphs.

A serialized graph specifies:

- 1) Optional graph labels that specify every graph label of the serialized graph.
- 2) Optional graph properties that specify every graph property of the serialized graph.
- 3) One optional graph type that specifies the graph type of the serialized graph.
- 4) Optional graph patterns that specify every graph element of the serialized graph.

A graph may be serialized as a character string that is expected to conform to the Format and Syntax specified in this document for a <serialized graph> of the GQL language.

```

<serialized graph> ::==
 [PROPERTY] GRAPH [[{ <identifier> <period> }...] <identifier>]
 [<of graph type>]
 <left brace>
 [<graph attribute specification>] [<simple graph pattern>]
 <right brace>

<graph attribute specification> ::=
 [<label set specification>]
 <left brace> [<literal property specification list>] <right brace>

<label set specification> ::=
 <is or colon> <label set expression>

<literal property specification list> ::=
 <literal property specification> [{ <comma> <literal property specification> }...]

<literal property specification> ::=
 <property name> <colon> <literal> [<of value type>]

```

A <serialized graph> may be deserialized by creating a new (empty) graph of the specified graph type, setting every specified label, setting every specified property, and then inserting every specified graph pattern as if using an <insert statement>.

## Bibliography

- [1] ISO/IEC 646, *Information technology — ISO 7-bit coded character set for information interchange*
- [2] ISO/IEC 6429, *Information technology — Control functions for coded character sets*
- [3] Freed, N. & Dürst, M.. *Character sets* [online]. Los Angeles, California, USA: Internet Assigned Numbers Authority, Available at <http://www.iana.org/assignments/character-sets>

## Index

Index entries appearing in **boldface** indicate the page where the word, phrase, or BNF non-terminal was defined; index entries appearing in *italics* indicate a page where the BNF non-terminal was used in a Format; and index entries appearing in roman type indicate a page where the word, phrase, or BNF non-terminal was used in a heading, Function, Syntax Rule, Access Rule, General Rule, Conformance Rule, Table, or other descriptive text.

### — A —

- ABS • 307, 328, 388
- ACOS • 307, 310, 388
- ACYCLIC • 55, 227, 229, 389, 405, 438
- ADD • 388
- AGGREGATE • 192, 388
- ALIAS • 388
- ALL • 179, 200, 202, 203, 214, 227, 249, 337, 338, 388
- ALL\_DIFFERENT • 295, 388
- AND • 193, 302, 303, 314, 388
- ANY • 227, 228, 375, 388
- ARRAY • 377, 382, 388
- AS • 107, 108, 109, 110, 113, 115, 117, 122, 133, 139, 200, 201, 204, 215, 244, 338, 340, 353, 354, 355, 359, 360, 361, 362, 388, 466
- ASC • 251, 388
- ASCENDING • 388
- ASIN • 307, 310, 388
- AT • 208, 388
- ATAN • 307, 310, 388
- AVG • 249, 388
- Alias resolution • 35
- <abbreviated edge pattern> • 54, **218**, 222, 225, 226, 230, 410, 436, 438
- <abbreviated edge type pattern> • **145**, 146, 147
- <abbreviated edge type pattern any direction> • **145**, **146**, 147
- <abbreviated edge type pattern pointing left> • **145**, **146**
- <abbreviated edge type pattern pointing right> • **145**, **146**
- <absolute url path> • 35, 131, 132, 256, **272**, 273, 277
- <absolute value expression> • **307**, 309
- <accent quoted character representation> • **366**, 369
- active GQL-transaction* • 39, 95, 134
- acyclic • 54
- <aggregate function> • 192, 201, 203, **249**, 306, 419, 449
- <aggregate statement> • 128, **192**, 425
- aggregating • 201
- alias • 35
- alias name • 201
- alias target • 35
- <all path search> • **227**, 228
- <all shortest path search> • 56, **227**, 228
- <all\_different predicate> • 283, **295**, 439
- allowing duplicates • 33
- alphabet • 402
- <ambient linear data-modifying statement> • 100, 136, **160**, 161, 162
- <ambient linear query statement> • 100, 103, 104, 176, 178, **182**
- <ampersand> • 55, 168, 232, 235, 396, **397**
- annotated path • 403
- anonymous edge symbol • 402
- anonymous node symbol • 402
- any object type • 60
- <any path search> • 56, **227**, 228, 414
- any property value type • 60
- <any shortest path search> • **227**, 228
- any value type • 60
- apparent value • 73
- applied • 404
- approximate numbers • 58
- <approximate numeric literal> • 73, 356, 358, **367**, 369, 372
- <approximate numeric type> • 72, 375, **376**, 379, 381
- approximate numeric types • 58
- <arc type any direction> • **145**
- <arc type filler> • **145**
- <arc type pointing left> • **145**
- <arc type pointing right> • **145**
- array • 67
- <as graph type> • **122**, 139
- <as or equals> • **107**, 108, 109, 110, 113, 115, 117, 122
- assignment • 77
- associated • 223, 224
- <asterisk> • 200, 219, 249, 304, 305, 326, 327, 396, **397**
- <at schema clause> • 104, 125, **208**
- at the same depth of graph pattern matching • 213
- atomic • 57
- attempt to modify a FINAL parameter* • 92
- authorization identifier • 26

**— B —**

BIGINT • 58, 74, 376, 380, 388  
 BINARY • 57, 71, 375, 378, 383, 388, 465  
 BINDING • 115, 124, 389  
 BNF non-terminal symbol • 78  
 BOOL • 57, 70, 375, 377, 388  
 BOOLEAN • 57, 70, 375, 377, 383, 388  
 BOTH • 316, 320, 321, 388  
 BY • 201, 246, 248, 388  
 BYTES • 57, 71, 375, 378, 388, 465  
 BYTE\_LENGTH • 307, 388  
 Boolean type • 70, 377  
 Boolean types • 57  
 Booleans • 57  
 be included in • 83  
 begin subpath symbol • 402  
 <binary digit> • 367, 369, 396  
 <binary exact numeric type> • 375, 379  
 <binary exact signed numeric type> • 376, 379  
 <binary exact unsigned numeric type> • 376, 379, 380  
 <binary set function> • 249, 250  
 <binary set function type> • 249  
 binding • 216, 413  
 <binding table initializer> • 115, 116  
 <binding table name> • 263, 264, 385, 386  
 <binding table parameter definition> • 111, 115, 116  
 <binding table parent specification> • 263, 264  
 <binding table reference> • 115, 116, 120, 124, 263  
 <binding table resolution expression> • 263  
 <binding table type> • 124  
 <binding table type expression> • 124, 375  
 <binding table variable> • 115, 116  
 <binding table variable declaration> • 111, 115  
 <binding table variable definition> • 111, 115  
 <binding variable> • 51, 168, 169, 170, 201, 202, 205, 209, 230, 244, 246, 306, 307, 308, 309, 330  
 <binding variable declaration> • 111  
 <binding variable definition> • 50, 105, 111, 112  
 <binding variable definition block> • 49, 50, 51, 105, 190  
 <binding variable definition list> • 111, 112  
 <binding variable name> • 113, 114, 115, 116, 117, 118, 209, 386  
 <boolean factor> • 185, 302  
 <boolean literal> • 365, 369, 373  
 <boolean predicand> • 302  
 <boolean primary> • 302, 303  
 <boolean term> • 302  
 <boolean test> • 302  
 <boolean type> • 70, 375, 377  
 <boolean value expression> • 70, 185, 282, 300, 301, 302, 303  
 bound • 403

bounded quantifier • 220  
 boxed value • 35  
 bracket index • 402  
 <bracket right arrow> • 145, 218, 390, 391  
 bracket symbol binding • 403  
 bracket symbols • 402  
 <bracket tilde right arrow> • 218, 390, 391  
 <bracketed comment> • 392, 394  
 <bracketed comment contents> • 393, 394  
 <bracketed comment introducer> • 392, 393  
 <bracketed comment terminator> • 393, 394  
 <byte length expression> • 307  
 <byte string concatenation> • 313, 314, 315  
 <byte string factor> • 313, 314  
 <byte string function> • 316, 318, 321  
 <byte string literal> • 365, 367, 370, 373, 388, 394  
 <byte string primary> • 313, 314  
 <byte string trim function> • 316, 317, 318, 321  
 <byte string trim source> • 317, 318, 321  
 <byte string type> • 71, 375, 378  
 byte string types • 57  
 <byte string value expression> • 313, 314, 317, 318, 319, 321  
 byte strings • 57  
 <byte substring function> • 316, 317, 318, 319, 321

**— C —**

CALL • 127, 190, 388  
 CASE • 338, 340, 350, 351, 388  
 CAST • 353, 354, 355, 359, 360, 361, 362, 388  
 CATALOG • 47, 108, 388  
 CEIL • 308, 388  
 CEILING • 308, 388  
 CHARACTER • 388  
 CHARACTER\_LENGTH • 307, 360, 388  
 CLASS\_ORIGIN • 389  
 CLEAR • 93, 388  
 CLONE • 388  
 CLOSE • 94, 388  
 COALESCE • 350, 351, 388  
 COLLECT • 249, 388  
 COMMAND\_FUNCTION • 389  
 COMMAND\_FUNCTION\_CODE • 389  
 COMMIT • 39, 99, 388  
 CONDITION\_NUMBER • 389  
 CONNECTING • 146, 389  
 CONSTANT • 388  
 CONSTRAINT • 388  
 CONSTRUCT • 388  
 COPY • 121, 122, 133, 388  
 COS • 307, 310, 388  
 COSH • 307, 310, 388

COST • 218, 388  
 COT • 307, 310, 388  
 COUNT • 249, 250, 388  
 CREATE • 131, 133, 134, 139, 153, 155, 157, 388  
 CSMAXL • 378  
 CURRENT\_DATE • 323, 388  
 CURRENT\_GRAPH • 297, 298, 388  
 CURRENT\_PROPERTY\_GRAPH • 297, 298, 388  
 CURRENT\_ROLE • 388  
 CURRENT\_SCHEMA • 272, 297, 298, 388  
 CURRENT\_TIME • 323, 388  
 CURRENT\_TIMESTAMP • 323, 324, 388  
 CURRENT\_USER • 297, 298, 388  
 Cartesian product • 34  
 Coercion • 59  
 calendar date • 23  
 <call catalog-modifying procedure statement> • 128, 159  
 <call data-modifying procedure statement> • 128, 174  
 <call function statement> • 128, 198  
 <call procedure statement> • 127, 159, 174, 186, 198, 425  
 <call query statement> • 128, 186, 425  
 canonical column sequence • 33  
 cardinality of a data type • 59  
 <case abbreviation> • 350, 351  
 <case expression> • 306, 350, 351, 352, 420  
 <case operand> • 350, 351  
 <case specification> • 350, 351, 352  
 <case-insensitive non-reserved word> • 389, 395  
 <case-insensitive reserved word> • 388, 394, 395  
 case-normal form • 394  
 <cast operand> • 353, 354  
 <cast specification> • 70, 306, 353, 354, 355, 441  
 <cast target> • 353  
 <catalog binding table parent and name> • 263  
 <catalog binding table reference> • 115, 116, 263  
 <catalog function parent and name> • 157, 158, 270  
 <catalog function reference> • 110, 270  
 <catalog graph parent and name> • 133, 138, 258  
 <catalog graph reference> • 46, 113, 114, 258  
 <catalog graph type parent and name> • 139, 152, 261  
 <catalog graph type reference> • 139, 261  
 catalog namespace • 51  
 catalog object • 28  
 catalog object descriptor • 28  
 <catalog object reference> • 51, 117, 118, 272, 274  
 <catalog procedure parent and name> • 153, 154, 266  
 <catalog procedure reference> • 108, 266  
 <catalog query parent and name> • 155, 156, 268  
 <catalog query reference> • 109, 268  
 <catalog schema parent and name> • 131, 132, 256  
 <catalog url path> • 272, 279  
 <catalog-modifying procedure specification> • 100, 101, 159  
 <catalog-modifying statement> • 100, 125  
 <ceiling function> • 307, 308, 309, 312  
 <char length expression> • 307  
 <character representation> • 366  
 <character string concatenation> • 313, 314  
 <character string factor> • 313, 314  
 <character string function> • 316, 318  
 <character string literal> • 365, 369, 370, 373, 390, 394  
 <character string primary> • 313, 314  
 <character string type> • 57, 70, 71, 375, 377  
 character string types • 57  
 <character string value expression> • 307, 313, 314, 316, 317, 318, 319  
 character strings • 57  
 child execution context • 42  
 <circumflex> • 397  
 coercion rule • 59  
 collation • 23  
 <collection type> • 375, 377  
 collection value • 34  
 <collection value constructor> • 306, 331  
 <collection value expression> • 193, 194, 300, 301, 420  
 <colon> • 108, 109, 110, 113, 115, 117, 139, 170, 218, 344, 396, 397, 467  
 <comma> • 87, 89, 97, 111, 141, 151, 168, 170, 172, 200, 204, 211, 213, 218, 219, 230, 242, 244, 246, 249, 251, 295, 296, 307, 308, 316, 317, 318, 334, 336, 341, 342, 343, 344, 346, 350, 375, 376, 377, 396, 397, 466, 467  
 command • 48  
 <comment> • 392  
 <commit command> • 25, 96, 99, 425, 449  
 <common logarithm> • 307, 308  
 <common value expression> • 300, 301  
 <comp op> • 252, 284, 285, 286, 458  
 <compact value variable definition> • 111, 112, 192  
 <compact value variable definition list> • 111, 112, 192  
 <compact variable declaration> • 111, 112, 211  
 <compact variable declaration list> • 111, 112  
 <compact variable definition> • 111, 112, 190, 211  
 <compact variable definition list> • 111, 112, 190  
 <comparison predicate> • 184, 252, 283, 284, 395, 418, 419, 466  
 <comparison predicate part 2> • 284, 350  
 completion condition • 44  
 <composite query expression> • 175, 178, 180, 181, 466  
 <composite query statement> • 125, 161, 175  
 <concatenation operator> • 313, 332, 390, 391  
 <conditional data-modifying statement> • 125, 162, 163  
 <conditional query statement> • 125, 161, 176, 177  
 conditions • 45  
 conforms • 58  
 connection exception • 25  
 <connector any direction> • 146

<connector pointing right> • 146

consistent • 404

constant object • 35

constitute • 81

constructed • 57

contain • 81

contained in • 81

containing • 81

<copy graph expression> • 121, 133

<copy graph type expression> • 122, 123

<cost clause> • 218, 219

<counted shortest group search> • 56, 227, 228, 414

<counted shortest path search> • 56, 227, 228, 414

<create function statement> • 128, 157, 425

<create graph statement> • 128, 133, 135, 425

<create graph type statement> • 128, 134, 139, 140, 142, 425

<create procedure statement> • 128, 153, 425

<create query statement> • 128, 155, 425

<create schema statement> • 131

current execution context • 41

current request context • 40

current session context • 37

current x • 37, 40, 41

## — D —

DATA • 56, 57, 58, 70, 71, 72, 377, 378, 379, 383, 384, 388

DATE • 76, 323, 324, 359, 361, 362, 368, 371, 376, 382, 388

DATETIME • 76, 323, 324, 362, 368, 371, 376, 382, 388, 420

DEC • 58, 75, 376, 379, 381, 388

DECIMAL • 57, 58, 72, 75, 376, 379, 381, 383, 388

DEFAULT • 218, 388

DEGREES • 307, 310, 388

DELETE • 172, 388

DESC • 251, 252, 388

DESCENDING • 388

DESTINATION • 293, 294, 390

DETACH • 172, 173, 388

DIRECTED • 146, 147, 149, 291, 390, 436

DIRECTORIES • 388

DIRECTORY • 388

DISTINCT • 178, 179, 203, 249, 250, 337, 338, 340, 388, 423

DO • 164, 168, 170, 172, 388

DOUBLE • 58, 75, 376, 381, 382, 388

DROP • 132, 134, 138, 140, 152, 154, 156, 158, 388

DURATION • 76, 328, 359, 360, 361, 368, 371, 372, 376, 382, 388, 421

*data exception* • 91, 92, 169, 171, 187, 194, 277, 285, 298, 299, 301, 303, 304, 305, 309, 310, 311, 312, 314, 315, 318, 319, 320, 321, 322, 324, 325, 326, 328, 330, 332, 333, 334, 335, 336, 337, 344, 354, 355, 356, 357, 358, 359, 360, 361, 362, 383, 413, 420, 421, 422

data object • 29

data type • 56, 457

<data-modifying procedure specification> • 100, 101, 102, 174

<data-modifying statement> • 100, 125

<date function> • 323, 324

<date function parameters> • 323, 324, 325

<date literal> • 367, 368, 371, 373

<date string> • 323, 325, 368, 371, 373, 390

<datetime factor> • 322

<datetime function> • 323, 324

<datetime function parameters> • 323, 324, 325

<datetime literal> • 368, 371, 373

<datetime primary> • 322

<datetime string> • 323, 325, 368, 371, 373, 390

<datetime term> • 322, 326, 327

<datetime value expression> • 300, 301, 322, 326, 327

<datetime value function> • 322, 323, 325

<decimal exact numeric type> • 375, 376, 379, 381

decimal numbers • 58

decimal types • 58

decimals • 58

declared • 223

declared names • 57

declares • 223

degree of exposure • 224

degree of reference • 280

<delete item> • 172

<delete item list> • 172

<delete statement> • 128, 172, 173, 425

<delimited identifier> • 277, 386, 390, 392, 393, 394, 395, 439

<delimiter token> • 221, 388, 390, 394

*dependent object error* • 173

<dependent value expression> • 205, 249, 250

descriptor • 83

<destination node type name> • 146, 147, 148

<destination node type reference> • 145, 146, 147, 148

<destination predicate part 2> • 293, 350, 351

<digit> • 356, 358, 367, 369, 372, 396, 466

<directed predicate> • 283, 291, 439

<directed predicate part 2> • 291, 350, 351

directionality constraint • 410

distinct array • 68

distinct list • 68

*division by zero* • 305, 309

<do statement> • 128, 164, 168, 170, 172, 425

<dollar sign> • 385, 396, 397

double approximate numeric type • 75, 382

<double colon> • 375, 390, 391

<double minus sign> • 390, 391, 392, 395, 439

<double period> • 272, 273, 277, 390, 391

<double quote> • 366, 369, 370, 396, 397

<double quoted character representation> • 366, 369

<double quoted character sequence> • 365, 369, 392, 393  
 <double solidus> • 392, 395, 439  
 <doubled grave accent> • 393  
 <drop function statement> • 128, 158, 425  
 <drop graph statement> • 128, 134, 138, 425  
 <drop graph type statement> • 128, 138, 140, 152, 425  
 <drop procedure statement> • 128, 154, 425  
 <drop query statement> • 128, 156, 425  
 <drop schema statement> • 132  
 duplicate-free • 33  
 duration • 23  
 <duration absolute value function> • 328  
 <duration factor> • 326, 327  
 <duration function> • 328  
 <duration function parameters> • 328  
 <duration literal> • 365, 368, 371, 372, 373  
 <duration primary> • 326  
 <duration string> • 328, 368, 371, 390  
 <duration term> • 322, 326, 327  
 <duration term 1> • 326  
 <duration term 2> • 326, 327  
 <duration value expression> • 300, 301, 322, 326, 327, 328  
 <duration value expression 1> • 326, 327  
 <duration value function> • 326, 328  
 dynamic object • 30  
 dynamic object type • 30  
 dynamic site • 30

## — E —

EDGE • 6, 377, 382, 390, 393  
 EDGES • 390  
 ELEMENT\_ID • 364, 388  
 ELSE • 162, 176, 338, 340, 350, 351, 352, 388  
 EMPTY\_BINDING\_TABLE • 297, 298, 388  
 EMPTY\_GRAPH • 133, 297, 298, 388  
 EMPTY\_PROPERTY\_GRAPH • 297, 298, 389  
 EMPTY\_TABLE • 297, 298, 389  
 END • 199, 338, 340, 350, 351, 388  
 ENDS • 388  
 EXCEPT • 178, 179, 337, 338, 389, 423, 466  
 EXISTING • 389  
 EXISTS • 90, 91, 92, 113, 114, 115, 116, 117, 118, 131, 132, 133, 134, 138, 139, 140, 152, 153, 154, 155, 156, 157, 158, 287, 389  
 EXP • 308, 311, 389  
 EXPLAIN • 89, 389  
 edge • 232  
 edge binding • 403  
 <edge kind> • 145, 146, 149, 436  
 <edge pattern> • 54, 55, 217, 218, 220, 221, 222, 223, 225, 230, 236, 237, 238, 408, 409, 410, 437  
 <edge reference> • 293  
 <edge synonym> • 145, 393

<edge type definition> • 141, 142, 145, 146, 148, 149, 436  
 <edge type filler> • 145, 148  
 <edge type label set definition> • 145, 147, 148, 149  
 <edge type name> • 141, 145, 147, 148  
 <edge type property type set definition> • 145, 147, 149  
 edge variable • 223  
 effectively • 82  
 <element pattern> • 54, 55, 217, 220, 221, 222, 223, 224, 226, 230, 407, 408, 409, 410, 437  
 <element pattern cost clause> • 217, 218, 221, 230  
 <element pattern filler> • 54, 165, 184, 217, 218, 221  
 <element pattern predicate> • 165, 184, 217, 218, 221, 230  
 <element pattern where clause> • 55, 218, 221, 223, 226, 230, 280, 403, 437  
 <element property specification> • 165, 184, 218  
 <element reference> • 280, 291, 292, 293, 295, 296, 350, 351, 364, 416, 417  
 <element type definition> • 141  
 <element type definition list> • 141  
 <element type name> • 143, 145, 146, 385  
 <element variable> • 54, 217, 221, 223, 226, 230, 280, 385, 386, 416, 437, 438  
 <element variable declaration> • 54, 165, 184, 214, 217, 221, 223, 230  
 <element\_id function> • 306, 364, 438, 458  
 elementary binding • 403  
 <else clause> • 350, 351, 352  
 <else linear data-modifying statement branch> • 162  
 <else linear query branch> • 176  
 empty • 221  
*empty binding table returned* • 187  
 empty data type • 59  
 <empty grouping set> • 246  
 <end node function> • 330  
 end subpath symbol • 402  
 <end transaction command> • 88, 96  
 endNode • 330, 388  
 <endpoint definition> • 145, 146  
 <endpoint pair definition> • 146, 147, 149, 436  
 <endpoint pair definition any direction> • 146, 147, 149, 436  
 <endpoint pair definition pointing left> • 146  
 <endpoint pair definition pointing right> • 146  
 equality operation • 418  
 <equals operator> • 54, 89, 107, 111, 168, 213, 219, 284, 286, 302, 396, 397, 418, 419  
 <escaped backspace> • 366, 370  
 <escaped carriage return> • 366, 370  
 <escaped character> • 366, 369, 370, 464  
 <escaped double quote> • 366, 370  
 <escaped form feed> • 366, 370  
 <escaped grave accent> • 393  
 <escaped newline> • 366, 370

<escaped quote> • 366, 370  
 <escaped reverse solidus> • 366, 370  
 <escaped tab> • 366, 370  
 exact numbers • 57  
 <exact numeric literal> • 73, 355, 357, 367, 369, 372  
 <exact numeric type> • 72, 375, 379, 381  
 exact numeric types • 57  
 exact signed numeric type with binary precision • 379  
 exception condition • 44  
 <exclamation mark> • 55, 232, 235, 396, 397  
 executable branch • 162, 176  
 executing GQL-request • 36  
 execution context • 41  
 execution outcome • 43  
 execution stack • 41  
 <exists predicate> • 283, 287, 288  
 <exponent> • 73, 367, 372  
 <exponential function> • 307, 308, 309, 311  
 expression • 201  
 <extended identifier> • 386, 388, 463  
 extensionally equal • 59  
 exterior variable • 229  
 <external object reference> • 256, 258, 259, 261, 263, 264,  
     266, 268, 270, 279, 450  
 <external object url> • 279  
 extracted path • 403

## — F —

FALSE • 302, 357, 359, 365, 373, 389  
 FILTER • 165, 168, 170, 172, 189, 193, 389  
 FINAL • 90, 91, 92, 93, 114, 116, 118, 390  
 FIRST • 251, 252, 390  
 FLOAT • 57, 58, 72, 75, 76, 376, 379, 382, 384, 389  
 FLOAT128 • 58, 75, 376, 382, 389  
 FLOAT16 • 58, 75, 376, 381, 389  
 FLOAT256 • 58, 75, 382, 389  
 FLOAT32 • 58, 75, 376, 381, 389  
 FLOAT64 • 58, 75, 376, 382, 389  
 FLOOR • 308, 389  
 FOR • 193, 194, 389  
 FROM • 204, 206, 338, 340, 389  
 FUNCTION • 47, 110, 157, 158, 270, 389  
 FUNCTIONS • 389  
 <factor> • 304, 326, 327  
 failed outcome • 44  
 <field> • 346, 372  
 <field list> • 346, 372  
 <field name> • 346, 377, 385, 386  
 <field type> • 377, 383  
 <field type list> • 377, 383  
 <field value> • 346  
 <filter statement> • 128, 165, 189, 425

<fixed length> • 375, 378  
 fixed length path pattern • 220  
 <fixed quantifiers> • 219, 220, 224  
 floating point numbers • 58  
 <floor function> • 307, 308, 309, 311  
 <focused linear data-modifying statement> • 100, 101, 160,  
     162, 207  
 <focused linear data-modifying statement body> • 160  
 <focused linear query statement> • 100, 103, 104, 176, 178,  
     182, 206  
 <focused linear query statement body> • 182  
 <fold> • 316, 318, 319, 320  
 <for item> • 193, 194  
 <for item alias> • 193, 194  
 <for item list> • 193  
 <for ordinality or index> • 193, 194  
 <for statement> • 128, 193, 194, 425, 457  
 <formal parameter declaration> • 211  
 <formal parameter declaration list> • 211  
 <formal parameter definition> • 211  
 <formal parameter definition list> • 211  
 <formal parameter list> • 211  
 <from graph clause> • 104, 182, 204, 206, 232  
 <full edge any direction> • 218, 410  
 <full edge left or right> • 218, 225, 410, 436  
 <full edge left or undirected> • 218, 225, 410, 436  
 <full edge pattern> • 54, 218, 222, 230, 410  
 <full edge pointing left> • 218, 230, 410  
 <full edge pointing right> • 218, 230, 410  
 <full edge type pattern> • 145, 147, 149, 436  
 <full edge type pattern any direction> • 145, 149, 436  
 <full edge type pattern pointing left> • 145, 147, 148  
 <full edge type pattern pointing right> • 145, 148  
 <full edge undirected> • 218, 225, 230, 410, 436  
 <full edge undirected or right> • 218, 225, 410, 436  
 function does not exist • 158, 336  
 <function initializer> • 110, 157  
 <function name> • 158, 270, 271, 385, 386  
 <function parent specification> • 158, 270, 271  
 <function reference> • 110, 270  
 <function resolution expression> • 270  
 <function specification> • 100, 101, 104, 198  
 <function variable> • 110  
 <function variable definition> • 107, 110

## — G —

Feature G001, “Undirected edge patterns” • 32, 64, 149, 225,  
     238, 435, 436  
 Feature G010, “Graph Pattern Keep” • 216, 437  
 Feature G011, “Graph Pattern Where” • 216, 437  
 Feature G020, “Path Multiset Alternation” • 225, 437  
 Feature G021, “Path Pattern Union” • 225, 437  
 Feature G030, “Quantified Paths” • 225, 434, 437

- Feature G031, "Quantified Edges" • 225, 434, 437  
 Feature G040, "Questioned Paths" • 226, 437  
 Feature G050, "Simplified Path Pattern Expression" • 226, 437  
 Feature G060, "Non-local element pattern predicates" • 226, 434, 437  
 Feature G070, "Abbreviated Edge Patterns" • 226, 438  
 Feature G081, "Parenthesized Path Pattern Where" • 226, 438  
 Feature G082, "Non-local predicates" • 226, 434, 438  
 Feature G083, "Parenthesized Path Pattern Cost" • 226, 438  
 Feature G091, "Shortest Path Search" • 229, 438  
 Feature G092, "Advanced Path Modes: TRAIL" • 229, 438  
 Feature G093, "Advanced Path Modes: SIMPLE" • 229, 438  
 Feature G094, "Advanced Path Modes: ACYCLIC" • 229, 438  
 Feature G100, "ELEMENT\_ID function" • 364, 438  
 Feature G101, "IS DIRECTED predicate" • 291, 439  
 Feature G102, "IS LABELED predicate" • 292, 439  
 Feature G103, "IS SOURCE and IS DESTINATION predicate" • 294, 439  
 Feature G104, "ALL\_DIFFERENT predicate" • 295, 439  
 Feature G105, "SAME predicate" • 296, 439  
 Feature GA00, "Graph label set support" • 31, 61, 435  
 Feature GA01, "Graph property set support" • 31, 61, 435  
 Feature GA02, "Empty node label set support" • 31, 63, 431, 435  
 Feature GA03, "Singleton node label set support" • 31, 63, 431, 434, 435  
 Feature GA04, "Unbounded node label set support" • 31, 63, 434, 435  
 Feature GA05, "Empty edge label set support" • 32, 64, 431, 435, 436  
 Feature GA06, "Singleton edge label set support" • 32, 64, 431, 434, 436  
 Feature GA07, "Unbounded edge label set support" • 32, 64, 434, 436  
 Feature GA08, "Named node types in graph types" • 61, 436  
 Feature GA09, "Named edge types in graph types" • 61, 436  
 Feature GB00, "Long identifiers" • 395, 439  
 Feature GB01, "Double minus sign comments" • 395, 439, 466  
 Feature GB02, "Double solidus comments" • 395, 439, 466  
 Feature GB10, "Explicit request parameters" • 86, 439, 440  
 <GQL language character> • 396, 398  
 <GQL special character> • 390, 396  
 <GQL terminal character> • 396, 399  
 GQL-agent • 25  
 GQL-catalog • 26  
 GQL-catalog root • 26  
 GQL-client • 25  
 GQL-data • 28  
 GQL-directory • 27  
 GQL-directory descriptor • 27  
 GQL-environment • 24  
 GQL-implementation • 25  
 GQL-object • 29  
 <GQL-program> • 85, 88, 444  
 <GQL-request> • 44, 85, 87, 453  
 GQL-request context • 40  
 GQL-schema • 28  
 GQL-server • 26  
 GQL-session • 36  
 GQL-status object • 45  
 GQL-transaction • 38  
 GQLSTATUS • 44, 45, 389, 427  
 GRANT • 389  
 GRAPH • 113, 122, 133, 134, 136, 138, 139, 140, 141, 152, 258, 261, 390, 467  
 GRAPHS • 390  
 GROUP • 201, 214, 228, 246, 389  
 GROUPS • 228, 390  
 Graph pattern matching • 53  
 Graphs • 30  
 Gregorian calendar • 23  
 <general literal> • 365  
 <general logarithm argument> • 308, 310  
 <general logarithm base> • 308, 310  
 <general logarithm function> • 307, 309, 310  
 <general quantifier> • 219, 220, 224, 409  
 <general set function> • 205, 249, 250  
 <general set function type> • 249  
 generally depends on • 83  
 generally includes • 83  
 graph descriptor • 32  
*graph does not exist* • 138, 194, 298, 301, 303, 304, 309, 314, 318, 322, 326, 330, 332, 334, 337, 344, 420, 421, 422  
 <graph element function> • 329, 330  
 <graph element primary> • 329, 348  
 <graph element type> • 375, 376  
 <graph element value expression> • 300, 301, 329  
 <graph expression> • 113, 114, 120, 121, 122, 123, 133, 204, 206, 207  
 <graph initializer> • 113, 114  
 <graph name> • 133, 138, 258, 259, 385, 386  
 <graph parameter definition> • 111, 113, 114  
 <graph parent specification> • 133, 138, 258, 259  
 <graph pattern> • 50, 51, 52, 53, 184, 213, 214, 215, 232, 280, 287, 405, 415, 457  
 <graph pattern quantifier> • 55, 214, 217, 219, 220, 223, 224, 235, 237  
 <graph pattern where clause> • 213, 216, 223, 280, 416, 437  
 <graph reference> • 121, 139, 232, 258  
 <graph resolution expression> • 90, 91, 258  
 <graph source> • 133  
 <graph specification> • 121, 136, 137  
 graph type • 61  
*graph type already exists* • 140, 299

*graph type conformance error* • 134  
*graph type definition error* • 148  
*graph type does not exist* • 152, 285, 318, 326  
*<graph type expression>* • 122, 123, 134, 139, 375  
*<graph type initializer>* • 139  
*<graph type name>* • 139, 152, 261, 262, 385  
*<graph type parent specification>* • 139, 152, 261, 262  
*<graph type reference>* • 122, 123, 261  
*<graph type resolution expression>* • 261  
*<graph type specification>* • 122, 123, 141, 142  
*<graph type specification body>* • 141, 146  
*<graph variable>* • 113, 114  
*<graph variable declaration>* • 111, 113  
*<graph variable definition>* • 111, 113, 114  
*<grave accent>* • 366, 369, 393, 396, 397  
*<greater than operator>* • 252, 284, 390, 391  
*<greater than or equals operator>* • 284, 390, 391  
*<group by clause>* • 200, 201, 204, 246, 247  
*grouping* • 201  
*<grouping element>* • 201, 246  
*<grouping element list>* • 246  
*grouping record* • 202

**— H —**

*HAVING* • 204, 389  
*HOME\_GRAPH* • 297, 298, 389  
*HOME\_PROPERTY\_GRAPH* • 297, 298, 389  
*HOME\_SCHEMA* • 297, 298, 389  
*hard links* • 35  
*<having clause>* • 204  
*<hex digit>* • 367, 369, 370, 373, 396, 398

**— I —**

*IF* • 90, 91, 92, 113, 114, 115, 116, 117, 118, 131, 132, 133, 134, 138, 139, 140, 152, 153, 154, 155, 156, 157, 158, 389  
*IN* • 193, 389  
*INDEX* • 193, 194, 390  
*INSERT* • 165, 389  
*INT* • 58, 74, 376, 379, 380, 389  
*INT128* • 58, 74, 376, 380, 389  
*INT16* • 58, 74, 376, 380, 389  
*INT256* • 58, 74, 376, 380, 389  
*INT32* • 58, 74, 376, 380, 389  
*INT64* • 58, 74, 376, 380, 389  
*INT8* • 74, 376, 380, 389  
*INTEGER* • 57, 58, 72, 74, 376, 379, 380, 383, 389  
*INTEGER128* • 58, 74, 376, 380, 389  
*INTEGER16* • 58, 74, 376, 380, 389  
*INTEGER256* • 58, 74, 376, 380, 389  
*INTEGER32* • 58, 74, 376, 380, 389  
*INTEGER64* • 58, 74, 376, 380, 389  
*INTEGER8* • 74, 376, 380, 389

*INTERSECT* • 178, 179, 337, 338, 389, 423, 466  
*IS* • 218, 237, 238, 289, 290, 291, 292, 293, 302, 303, 314, 320, 338, 340, 351, 389  
*<identifier>* • 50, 89, 165, 184, 193, 194, 200, 201, 204, 221, 230, 244, 272, 385, 386, 402, 457, 458, 467  
*<identifier extend>* • 388, 393, 463, 464  
*<identifier start>* • 388, 393, 463  
*immediately contain* • 81  
*<implementation-defined access mode>* • 97  
*inDegree* • 308, 388  
*<inDegree function>* • 307, 308  
*include* • 83  
*<independent value expression>* • 249, 250  
*inessential* • 82  
*<inline procedure call>* • 159, 174, 186, 198, 240, 241  
*innermost* • 81  
*innermost executing command* • 36  
*innermost executing operation* • 37  
*innermost executing procedure* • 36  
*innermost executing procedure or command* • 36  
*innermost executing statement* • 36  
*<insert statement>* • 128, 165, 425, 457, 467  
*instant* • 23  
*integer numbers* • 58  
*integer types* • 58  
*integers* • 58  
*interior variable* • 228  
*intermediate results* • 82, 457  
*internal object identifiers* • 29, 456  
*invalid action* • 91  
*invalid argument for natural logarithm* • 311  
*invalid argument for power function* • 311  
*invalid character value for cast* • 354, 355, 357, 359, 362  
*invalid datetime function map key* • 324, 325  
*invalid duration format* • 362  
*invalid graph element reference* • 169, 171  
*invalid number of paths or groups* • 413  
*invalid reference* • 44, 99, 178, 202, 209, 223, 230, 242, 273, 424  
*invalid session parameter name* • 92  
*invalid time zone displacement value* • 91  
*invalid transaction state* • 39, 95  
*invalid transaction termination* • 98, 99  
*is generally dependent on* • 83  
*<is label expression>* • 150, 217, 218, 221, 224, 230  
*<is or colon>* • 218, 467

**— J —**

*join* • 59

**— K —**

*KEEP* • 213, 389  
*<keep clause>* • 213, 214, 216, 437

<key word> • 57, 89, 293, 302, **388**, 394, 395  
 known not nullable • 77

## — L —

LABEL • 150, 390

LABLED • 292, 390

LABELS • 150, 390

LAST • 251, 252, 390

LEADING • 316, 317, 318, 320, 321, 389

LEFT • 316, 317, 318, 389

LENGTH • 307, 389

LET • 190, 389

LIKE • 122, 124, 133, 134, 389

LIKE\_REGEX • 389

LIMIT • 254, 389

LIST • 377, 382, 389

LN • 308, 311, 389

LOCALDATETIME • 76, 323, 324, 361, 371, 376, 382, 389, 420, 421

LOCALTIME • 76, 323, 324, 360, 361, 371, 376, 382, 389, 421

LOCALTIMESTAMP • 323, 324, 389

LOG • 308, 389

LOG10 • 308, 389

LOWER • 316, 319, 389

lTrim • 316, 317, 318, 388

<label> • 54, 150, 168, 169, 171, **209**, 232, 235, 238, 400

<label conjunction> • 232, 400

<label disjunction> • 150, 230, **232**, 400

<label expression> • 54, 150, 218, 224, **232**, 233, 237, 238, 292, 400, 401, 408

<label factor> • 232, 400

<label name> • 209, **385**, 386

label namespace • 51

<label negation> • 150, 230, **232**, 400

<label primary> • 232, 400

<label set definition> • 143, 145, 149, **150**

<label set expression> • **168**, 170, 467

<label term> • 232, 400

<labeled predicate> • 283, **292**, 439

<labeled predicate part 2> • **292**, 350, 351

<left angle bracket> • 235, 377, 391, 397, **398**

<left arrow> • 146, 218, 222, 390, **391**

<left arrow bracket> • 145, 218, 390, **391**

<left arrow tilde> • 219, 222, 225, 235, 390, **391**, 436

<left arrow tilde bracket> • 218, 390, **391**

left boundary variable • 228

<left brace> • 100, 103, 104, 141, 151, 218, 219, 341, 342, 343, 344, 346, 377, 396, **397**, 467

<left bracket> • 91, 219, 232, 235, 325, 336, 343, 371, **397**

<left minus right> • 219, 222, 225, 390, **391**, 436

<left minus slash> • 234, 236, 390, **391**

<left paren> • 143, 146, 211, 217, 219, 232, 235, 242, 246, 249, 287, 295, 296, 302, 306, 307, 308, 316, 317, 323, 326, 328, 330, 334, 340, 350, 353, 364, 375, 376, **397**

<left tilde slash> • 234, 236, 390, **391**

left to right • 54

<length expression> • **307**, 308

<less than operator> • 252, 284, 390, **391**

<less than or equals operator> • 284, 390, **391**

<let statement> • 128, **190**, 425

library • 34

library default procedure name • 34

library procedure object • 34

<like binding table shorthand> • **124**

<like binding table type> • **124**

<like graph expression> • **122**, 123

<like graph expression shorthand> • **122**, 123

<limit clause> • 196, 204, **254**, 458

<linear catalog-modifying statement> • **125**, **130**

<linear data-modifying statement> • 51, **125**, **160**, 161, 162, 163

<linear query expression> • 176, 178, 179, **181**

<linear query statement> • 51, 181, **182**, 183

list • 67

<list concatenation> • **332**, 333

*list data, right truncation* • 333

<list element> • **336**, 372

*list element error* • 334, 335

<list element list> • **336**, 372

<list literal> • 365, **368**, 372, 373

list of graph elements bound to ER • 417

<list primary> • **332**

<list value constructor> • 331, **336**, 455

<list value constructor by enumeration> • **336**, 368, 372, 373

<list value expression> • 300, 301, **332**, 334, 335, 455

<list value expression 1> • **332**

<list value function> • 332, **334**, 335

<list value type> • **377**

<list value type name> • 336, **377**

<literal> • 89, 228, 297, 298, 354, 355, 356, 358, 359, 360, 361, 362, **365**, 372, 451, 467

<local binding table reference> • **263**, 264

<local datetime function> • **323**, 324, 325

<local function reference> • **270**, 271

<local graph reference> • **258**, 259

<local graph type reference> • **261**, 262

local namespace • 51

<local procedure reference> • **266**, 267

<local query reference> • **268**, 269

<local time function> • **323**, 324

<lower bound> • **219**, 220, 223, 409

## — M —

MANDATORY • 127, 184, 187, 190, 193, 389  
 MAP • 344, 377, 383, 389, 424  
 MATCH • 184, 287, 389  
 MAX • 249, 389, 419  
 MERGE • 167, 389  
 MESSAGE\_TEXT • 390  
 MIN • 249, 389, 419  
 MOD • 307, 389  
 MORE • 390  
 MULTI • 211, 389  
 MULTIPLE • 211, 389  
 MULTISET • 337, 338, 340, 341, 377, 382, 389, 423  
 MUTABLE • 90, 114, 116, 118, 390  
 <main activity> • 88  
 <mandatory formal parameter list> • 211, 212  
 <mandatory statement> • 127, 128, 184, 187, 193, 425  
 <mantissa> • 73, 356, 358, 367, 369, 372  
 <map element> • 344, 345, 372  
 <map element list> • 344, 372  
 <map key> • 324, 325, 328, 344, 345  
 <map key type> • 377, 383  
 <map literal> • 365, 368, 372, 373  
 <map value> • 325, 344, 345  
 <map value constructor> • 323, 328, 331, 344, 345  
 <map value constructor by enumeration> • 344, 368, 372, 373  
 <map value expression> • 300, 301  
 <map value type> • 375, 377  
 match • 216  
 <match statement> • 50, 128, 184, 204, 280, 425, 457  
 material • 57  
 <max length> • 375, 377, 378  
 meet • 59  
 <merge statement> • 128, 167, 425  
 meta type • 56  
 <min length> • 375, 378  
 minimum node count • 222  
 minimum path length • 220  
 <minus left bracket> • 145, 218, 390, 391  
 <minus sign> • 219, 222, 235, 304, 305, 322, 326, 367, 372, 397, 398  
 <minus slash> • 234, 236, 390, 392  
 <modulus expression> • 307, 309  
 most specific • 59  
 most specific type • 56  
 multi-path binding • 405  
 multiset • 66  
 multiset alternation operand • 221  
 <multiset alternation operator> • 55, 217, 234, 388, 390  
 <multiset element> • 341, 372  
 multiset element grouping operation • 423  
 <multiset element list> • 341, 372  
 <multiset literal> • 365, 368, 372, 373

multiset operand • 423  
 <multiset primary> • 337, 338, 423  
 <multiset set function> • 340, 423  
 <multiset term> • 337, 338, 339, 423  
 <multiset value constructor> • 331, 341  
 <multiset value constructor by enumeration> • 341, 368, 372, 373  
 <multiset value expression> • 300, 301, 337, 338, 339, 340, 423  
 <multiset value function> • 337, 338, 340  
 <multiset value type> • 377

## — N —

NEW • 389  
 NFC • 290, 316, 317, 390  
 NFD • 316, 390  
 NFKC • 316, 390  
 NFKD • 316, 390  
 NODE • 6, 376, 382, 390, 393  
 NODES • 390  
 NORMALIZE • 314, 316, 320, 389  
 NORMALIZED • 290, 314, 320, 390  
 NOT • 90, 91, 113, 114, 115, 116, 117, 118, 131, 133, 134, 139, 140, 153, 155, 157, 284, 289, 290, 291, 292, 293, 302, 303, 351, 389, 465  
 NOTHING • 375, 389  
 NULL • 289, 302, 338, 340, 350, 351, 352, 354, 368, 371, 389, 465  
 NULLIF • 350, 351, 389  
 NULLS • 251, 252, 389  
 NUMBER • 390  
 NUMERIC • 389  
 named graph descriptor • 32  
 named graph type descriptor • 62  
 <named procedure call> • 50, 51, 159, 174, 186, 198, 240, 242, 243  
 named procedure descriptor • 46  
 <natural logarithm> • 307, 308, 309, 311  
 negative duration • 23  
 <nested ambient data-modifying procedure specification> • 136  
 <nested catalog-modifying procedure specification> • 100, 101  
 <nested data-modifying procedure specification> • 100, 101, 136, 160, 161, 162, 164  
 <nested function specification> • 104, 110  
 <nested graph query specification> • 113, 114, 122, 136  
 <nested graph type specification> • 122, 141  
 <nested procedure specification> • 100, 101, 108, 159, 174, 186, 198, 241  
 <nested query specification> • 103, 109, 115, 116, 117, 118, 136, 176, 182, 204, 287, 349  
 <newline> • 190, 370, 392, 393, 394  
 no data • 44, 45, 427

node • 232  
 node binding • 403  
 <node pattern> • 54, 55, 217, 220, 222, 223, 229, 230, 232, 408, 409, 410  
 <node reference> • 293  
 <node synonym> • 143, 393  
 <node type definition> • 141, 142, 143, 144, 148  
 <node type filler> • 143, 146, 148  
 <node type label set definition> • 143  
 <node type name> • 141, 142, 143, 148  
 <node type property type set definition> • 143  
 node variable • 223  
*non-character in character string* • 355, 356, 357, 358, 359, 383  
 <non-delimited identifier> • 388, 393, 394, 395, 439  
 <non-delimiter token> • 18, 388, 394  
 <non-parenthesized value expression primary> • 284, 285, 302, 303, 306, 350, 351  
 <non-reserved word> • 388, 389  
 <normal form> • 70, 290, 316, 317, 377, 378, 379, 381  
 <normalize function> • 70, 316, 317, 318, 319  
 <normalized predicate> • 70, 283, 290  
 <normalized predicate part 2> • 290  
 <not equals operator> • 284, 286, 302, 390, 392, 418, 419  
 nothing type • 60  
 <null literal> • 353, 365, 368, 371, 373  
 <null ordering> • 251  
 <null predicate> • 283, 289, 465  
 <null predicate part 2> • 289, 350  
*null value not allowed* • 298, 299, 356, 358, 363  
 nullability characteristic • 77  
 nullable • 57  
 <number of groups> • 227, 228, 413  
 <number of paths> • 227, 228, 413  
 numbers • 57  
 <numeric literal> • 367  
 <numeric primary> • 304, 305  
 <numeric type> • 72, 375, 379, 382  
 numeric types • 57  
 <numeric value expression> • 249, 300, 301, 304, 305, 307, 308, 309, 310, 311, 312, 317, 334, 335, 450  
 <numeric value expression base> • 308, 311  
 <numeric value expression dividend> • 307, 309  
 <numeric value expression divisor> • 307, 309  
 <numeric value expression exponent> • 308, 311  
 <numeric value function> • 304, 307, 451  
*nvalid syntax* • 44, 91, 148, 173, 242, 314, 315, 319, 320, 356, 357, 358, 359

## — O —

OCCURRENCES\_REGEX • 389  
 OCTET\_LENGTH • 307, 389  
 OF • 121, 122, 124, 133, 293, 375, 389

OFFSET • 255, 389  
 ON • 389  
 ONLY • 97, 390  
 OPTIONAL • 127, 165, 184, 188, 190, 193, 211, 389  
 OR • 133, 134, 139, 140, 153, 155, 157, 284, 285, 302, 303, 338, 351, 352, 389  
 ORDER • 248, 389  
 ORDERED • 196, 343, 377, 383, 389  
 ORDINALITY • 193, 194, 390  
 OTHERWISE • 178, 179, 389  
 <object name> • 275, 385, 386  
 object type • 56, 457  
 <octal digit> • 367, 369, 396  
 <of binding table type> • 115, 116, 124  
 <of graph type> • 113, 114, 122, 133, 134, 467  
 <of type prefix> • 122, 124, 211, 375, 377  
 <of type signature> • 108, 109, 110, 153, 155, 157, 211  
 <of value type> • 117, 118, 300, 375, 467  
 <offset clause> • 196, 204, 255, 458  
 <offset synonym> • 255  
 omitted result • 43  
 <optional binding table variable definition> • 111, 115  
 <optional binding variable definition> • 111, 112  
 <optional binding variable definition list> • 111, 112  
 <optional formal parameter list> • 211, 212  
 <optional graph variable definition> • 111, 113  
 <optional parameter cardinality> • 211  
 <optional statement> • 127, 128, 165, 184, 188, 193, 425  
 <optional value variable definition> • 111, 117  
 <order by and page statement> • 128, 196, 199, 426  
 <order by clause> • 196, 204, 248  
 ordered • 33  
 <ordered set element> • 343, 372  
 <ordered set element list> • 343, 372  
 <ordered set literal> • 365, 368, 372, 373  
 <ordered set value constructor> • 331, 343  
 <ordered set value constructor by enumeration> • 343, 368, 372, 373  
 <ordered set value expression> • 300, 301  
 <ordered set value type> • 377  
 ordering operation • 419  
 <ordering specification> • 251  
 <other digit> • 396, 398  
 <other language character> • 396, 398  
 outDegree • 308, 388  
 <outDegree function> • 307, 308  
 outermost • 81  
 owning schema • 28

## — P —

PARAMETER • 90, 389  
 PARTITION • 389  
 PATH • 227, 228, 382, 389

PATHS • 227, 228, 389  
 PATTERN • 390  
 PATTERNS • 390  
 POSITION\_REGEX • 389  
 POWER • 308, 389  
 PRECISION • 58, 75, 376, 381, 382, 389  
 PROCEDURE • 47, 108, 153, 154, 266, 389  
 PROCEDURES • 389  
 PRODUCT • 249, 389  
 PROFILE • 89, 389  
 PROJECT • 205, 389  
 PROPERTIES • 390  
 PROPERTY • 113, 122, 133, 134, 136, 139, 140, 141, 152, 390, 467  
 <parameter> • 51, 92, 114, 115, 116, 117, 118, 209, 277, 297  
 <parameter cardinality> • 211  
 <parameter definition> • 87, 90, 91, 111  
 <parameter name> • 91, 92, 113, 209, 298, 385, 386, 388  
 parameter namespace • 51  
 <parameter value specification> • 297, 306  
 <parameterized url path> • 272, 273  
 parameterized view • 48  
 <parent catalog object reference> • 258, 259, 261, 262, 263, 264, 266, 267, 268, 269, 270, 271, 272  
 parent directory • 27  
 <parent object relative url path> • 272, 273  
 <parenthesized Boolean value expression> • 302  
 <parenthesized formal parameter list> • 211  
 <parenthesized label expression> • 232, 400  
 <parenthesized path pattern cost clause> • 219, 226, 438  
 <parenthesized path pattern expression> • 55, 214, 215, 216, 217, 219, 220, 221, 223, 224, 225, 226, 228, 229, 280, 402, 405, 407, 408, 409, 413, 415, 416, 438  
 <parenthesized path pattern where clause> • 216, 219, 221, 223, 225, 226, 280, 413, 416, 438, 458  
 <parenthesized value expression> • 306  
 path binding • 403  
 <path concatenation> • 217, 220, 222, 223, 224, 404, 407, 409, 411  
 <path factor> • 217, 224, 407, 409, 410, 411  
 <path length expression> • 307, 309  
 <path mode> • 53, 55, 214, 225, 227, 228, 229, 405, 438  
 <path mode prefix> • 214, 219, 224, 227, 228, 405  
 <path multiset alternation> • 217, 220, 221, 223, 224, 225, 229, 411, 437  
 <path or paths> • 227  
 <path pattern> • 53, 54, 55, 213, 214, 215, 223, 225, 228, 229, 230, 403, 408, 412, 413, 416, 458  
 <path pattern expression> • 54, 55, 213, 214, 217, 219, 220, 221, 223, 225, 230, 407, 408, 411, 457  
 <path pattern list> • 213, 215  
 <path pattern name> • 385, 386  
 <path pattern prefix> • 54, 213, 214, 227, 228, 438

<path pattern union> • 217, 220, 221, 223, 224, 225, 229, 407, 411, 437  
 path pattern union operand • 221  
 <path primary> • 217, 220, 221, 223, 224, 225, 407, 409, 410, 437  
 <path search prefix> • 53, 56, 213, 227, 228, 412, 413  
 <path term> • 217, 221, 222, 224, 407, 409, 411  
 <path variable> • 54, 213, 214, 385, 386  
 path variable in the global scope • 214  
 peers • 252, 458  
 <percent> • 232, 397, 398  
 percentileCont • 249, 388  
 percentileDist • 249, 388  
 <period> • 73, 168, 170, 258, 261, 263, 266, 268, 270, 272, 275, 277, 348, 356, 358, 367, 369, 397, 398, 467  
 persistent • 77  
 <plus sign> • 219, 304, 305, 322, 326, 367, 397, 398  
 possibly nullable • 77  
 possibly variable length path pattern • 220  
 <power function> • 307, 308, 309, 311  
 <preamble> • 40, 88, 89, 449  
 <preamble option> • 89  
 <preamble option identifier> • 89  
 precede • 252  
 <precision> • 72, 376, 379, 380, 381, 382  
 predefined • 57  
 <predefined graph parameter> • 258, 297  
 <predefined parameter> • 297  
 <predefined parent object parameter> • 272, 273, 297  
 <predefined schema parameter> • 256, 297  
 <predefined table parameter> • 263, 264, 297  
 <predefined type> • 344, 353, 375, 377  
 <predefined type literal> • 365  
 <predicate> • 70, 283, 302  
 predicative production rule • 79  
 preferred name • 57  
 primary • 29  
 primary object • 29  
 <primary result object expression> • 120, 300, 301  
 <primitive catalog-modifying statement> • 128  
 <primitive data-modifying statement> • 128  
 <primitive data-transforming statement> • 128  
 <primitive result statement> • 51, 125, 160, 161, 182, 183, 199  
 principal • 26  
 <procedure argument> • 242  
 <procedure argument list> • 242  
 <procedure body> • 49, 51, 100, 101, 102, 103, 104, 105, 106, 108, 109, 110, 162, 176, 178, 182, 187, 205, 249  
 <procedure call> • 127, 159, 174, 186, 187, 188, 198, 240  
 procedure call error • 242  
 procedure descriptor • 46  
 procedure does not exist • 154, 325

<procedure initializer> • 108, 153  
 procedure logic • 46  
 <procedure name> • 154, 266, 267, 385, 386  
 <procedure parent specification> • 154, 266, 267  
 <procedure reference> • 108, 159, 174, 186, 198, 242, 266  
 <procedure resolution expression> • 266  
 <procedure result type> • 211, 212  
 procedure signature • 46  
 procedure signature descriptor • 47  
 <procedure specification> • 88, 100, 101, 287  
 <procedure variable> • 108  
 <procedure variable definition> • 107, 108  
 procedures • 30  
 production rule • 78  
 <project statement> • 199, 205  
 property graph • 31  
 <property key value pair> • 165, 184, 218  
 <property key value pair list> • 165, 184, 218  
 <property name> • 151, 166, 168, 169, 170, 171, 184, 218, 348, 385, 386, 467  
 property namespace • 51  
 <property reference> • 104, 223, 306, 348, 417  
 <property type definition> • 151  
 <property type definition list> • 151  
 <property type set definition> • 143, 145, 149, 151  
 property value type • 56, 457

## — Q —

QUERIES • 389  
 QUERY • 47, 109, 155, 156, 268, 389  
 <qualified binding table name> • 263, 264  
 <qualified function name> • 270, 271  
 <qualified graph name> • 258, 259  
 <qualified graph type name> • 261, 262  
 <qualified name prefix> • 275  
 <qualified object name> • 258, 259, 261, 262, 263, 264, 266, 267, 268, 269, 270, 271, 275, 276  
 <qualified procedure name> • 266, 267  
 <qualified query name> • 268, 269  
 <quantified path primary> • 214, 217, 220, 221, 223, 224, 225, 405, 407, 409, 410, 437  
 <query conjunction> • 178  
*query does not exist* • 156, 328  
 <query initializer> • 109, 155  
 <query name> • 156, 268, 269, 385, 386  
 <query parent specification> • 156, 268, 269  
 <query reference> • 109, 268  
 <query resolution expression> • 268  
 <query specification> • 100, 101, 103, 186  
 <query statement> • 100, 103, 104, 125, 161  
 <query variable> • 109  
 <query variable definition> • 107, 109  
 <question mark> • 55, 217, 235, 397, 398

<questioned path primary> • 55, 217, 220, 221, 223, 224, 226, 407, 409, 410, 437  
 <quote> • 365, 366, 367, 369, 370, 397, 398

## — R —

RADIAN • 307, 310, 389  
 READ • 95, 97, 390  
 REAL • 58, 75, 376, 382, 389  
 RECORD • 346, 377, 383, 389  
 RECORDS • 389  
 REFERENCE • 389  
 RELATIONSHIP • 6, 377, 382, 390, 393  
 RELATIONSHIPS • 390  
 REMOVE • 92, 170, 389  
 RENAME • 389  
 REPLACE • 133, 134, 139, 140, 153, 155, 157, 389  
 REQUIRE • 389  
 RESET • 389  
 RESULT • 389  
 RETURN • 127, 165, 184, 193, 200, 287, 389  
 RETURNED\_GQLSTATUS • 390  
 REVOKE • 389  
 RIGHT • 316, 317, 318, 389  
 ROLLBACK • 39, 98, 389  
 rTrim • 316, 317, 318, 388  
 real approximate numeric type • 75, 382  
 <record literal> • 365, 368, 372, 373  
 record ordinal • 33  
 record type • 57  
 <record value constructor> • 331, 346, 347  
 <record value constructor by enumeration> • 346, 368, 372, 373  
 <record value expression> • 300, 301  
 <record value type> • 124, 375, 377  
 reduced match • 216  
 reduced path binding • 403  
 <reference value expression> • 300, 301  
 reference value type • 56, 457  
 regular approximate numeric type • 75, 382  
 regular decimal exact numeric type • 75, 381  
 <regular identifier> • 386, 388, 463  
 regular language of BNT • 408  
 regular result • 43  
 <relative url path> • 35, 272, 273, 277  
 <remove item> • 170, 171  
 <remove item list> • 170  
 <remove label item> • 170, 171  
 <remove property item> • 170, 171  
 <remove statement> • 128, 170, 171, 425  
 representation with reduced precision • 23  
 representative form • 393  
 <request parameter> • 87  
 <request parameter set> • 40, 85, 86, 87, 440

<reserved word> • **388**, 394  
 restrictive • 228  
 <result> • **350**, 351, 352  
 <result expression> • **350**, 352  
 <return item> • **200**, 201, 202, 203  
 <return item alias> • **200**, 201  
 <return item list> • **200**  
 <return statement> • **199**, **200**, 457  
 <return statement body> • **200**  
 <reverse solidus> • 366, 367, 369, 370, 393, 397, **398**  
 <right angle bracket> • 235, 377, 391, 396, **397**  
 <right arrow> • 146, 219, 222, 390, **392**  
 right boundary variable • 229  
 <right brace> • 100, 103, 104, 141, 151, 218, 219, 341, 342,  
     343, 344, 346, 377, 397, **398**, 467  
 <right bracket> • 91, 219, 232, 235, 325, 336, 343, 371, 397,  
     **398**  
 <right bracket minus> • 145, 218, 391, **392**  
 <right bracket tilde> • 145, 218, 391, **392**  
 <right paren> • 143, 146, 211, 217, 219, 232, 235, 242, 246,  
     249, 287, 295, 296, 302, 306, 307, 308, 316, 317, 323, 326,  
     328, 330, 334, 340, 350, 353, 364, 375, 376, 397, **398**  
 right to left • 54  
 <rollback command> • 39, 96, **98**, 425  
 root directory • 26  
 root schema • 26

## — S —

SAME • 296, 389  
 SCALAR • 389  
 SCHEMA • 90, 131, 132, 389  
 SCHEMAS • 389  
 SCHEMATA • 389  
 SELECT • 204, 338, 340, 389  
 SESSION • 90, 92, 93, 94, 389  
 SET • 90, 166, 168, 340, 342, 343, 377, 383, 389  
 SHORTEST • 214, 227, 228, 390  
 SIGNED • 58, 74, 376, 379, 380, 389  
 SIMPLE • 55, 227, 229, 390, 405, 438  
 SIN • 307, 310, 389  
 SINGLE • 211, 389  
 SINH • 307, 310, 389  
 SKIP • 255, 389  
 SMALLINT • 58, 74, 376, 380, 389  
 SOURCE • 293, 294, 390  
 <SQL-interval literal> • **368**, 371, 372  
 SQRT • 308, 389  
 START • 95, 389  
 STARTS • 389  
 STRING • 57, 71, 359, 360, 361, 362, 375, 377, 383, 389, 465  
 SUBCLASS\_ORIGIN • 390  
 SUBSTRING • 57, 316, 317, 318, 359, 360, 361, 389  
 SUBSTRING\_REGEX • 389

SUM • 249, 389  
 <same predicate> • 283, **296**, 439  
 satisfied • 282  
 satisfies • 400  
 <scale> • 72, **376**, 379, 381, 382  
 <schema name> • 131, 132, 256, **385**  
 <schema reference> • 46, 90, 208, **256**  
 <search condition> • 55, 70, 162, 165, 168, 170, 172, 176,  
     189, 190, 193, 204, 213, 216, 218, 219, 239, **282**, 314, 320,  
     350, 352, 413, 415, 458  
 <searched case> • **350**, 351, 352  
 <searched when clause> • **350**, 351, 352  
 secondary • 29  
 secondary object • 29  
 <select graph match> • **204**  
 <select graph match list> • **204**  
 <select item> • **204**, 466  
 <select item alias> • **204**, 466  
 <select item list> • **204**  
 <select query specification> • **204**  
 <select statement> • 161, 182, 183, **204**, 206  
 <select statement body> • **204**  
 selective • 228  
 &ltsemicolon> • 397, **398**  
 separable • 404  
 <separated identifier> • 385, **386**  
 <separator> • 304, 365, 367, 370, **392**, 394, 452  
 <session activity> • **88**  
 <session clear command> • 88, **93**, 424  
 <session close command> • 37, 88, **94**, 424  
 session command • 48  
 session context • 37  
 <session parameter> • 87, **90**  
 <session parameter command> • **88**  
 <session parameter flag> • **90**, 91  
 session parameters • 29, 30  
 <session remove command> • 88, **92**, 424  
 <session set command> • 88, **90**, 91  
 <session set graph clause> • **90**, 91, 424  
 <session set parameter clause> • **90**, 91, 424  
 <session set schema clause> • **90**, 424  
 <session set time zone clause> • **90**, 424  
 set • 67  
 <set all properties item> • **168**, 169  
 <set element> • **342**, 372  
 <set element list> • **342**, 372  
 <set item> • 166, **168**, 169  
 <set item list> • 166, **168**  
 <set label item> • **168**, 169  
 <set literal> • 365, **368**, 372, 373  
 set of local matches to BNT • 408  
 <set operator> • **178**  
 <set property item> • **168**, 169

<set quantifier> • 178, 200, 204, **249**, 250, 466  
 <set statement> • 128, 166, **168**, 169, 425  
 <set time zone value> • 90, 91  
 <set value constructor> • 331, **342**, 372  
 <set value constructor by enumeration> • **342**, 368, 372, 373  
 <set value expression> • **300**, 301  
 <set value type> • 377  
 shall • 81  
 <shortest path search> • 227, 228, 229, 414, 438  
 <sign> • 304, 326, **367**, 373  
 signed 128-bit integer type • 74, 380  
 signed 16-bit integer type • 74, 380  
 signed 256-bit integer type • 74, 380  
 signed 32-bit integer type • 74, 380  
 signed 64-bit integer type • 74, 380  
 signed 8-bit integer type • 74, 380  
 signed big integer type • 74  
 <signed decimal integer> • 73, **367**  
 signed exact numbers • 58  
 signed exact numeric types • 58  
 <signed numeric literal> • 354, 355, 365, **367**, 373  
 signed regular integer type • 74, 380  
 signed small integer type • 74, 380  
 signed user-specified integer type • 380  
 signed user-specified integer types • 74  
 simple • 54  
 <simple Latin letter> • **396**  
 <simple Latin lower-case letter> • 395, **396**, 399  
 <simple Latin upper-case letter> • 395, **396**, 399, 427  
 <simple case> • **350**, 351  
 <simple catalog-modifying statement> • **128**, 130  
 <simple comment> • **392**, 394, 395, 439  
 <simple comment character> • **392**, 394  
 <simple comment introducer> • **392**  
 <simple data-accessing statement> • **128**, 160, 161  
 <simple data-modifying statement> • 125, **128**, 160, 161  
 <simple data-reading statement> • **128**  
 <simple data-transforming statement> • **128**  
 <simple graph pattern> • 50, 51, 52, 165, 167, **230**, 231, 467  
 <simple linear query statement> • 160, 161, **182**  
 <simple path pattern> • **230**, 231  
 <simple path pattern list> • **230**  
 <simple query statement> • **128**, 182  
 <simple relative url path> • **272**, 273  
 <simple url path> • **272**, 273  
 simple view • 48  
 <simple when clause> • **350**, 351  
 <simplified concatenation> • **234**, 236, 237  
 <simplified conjunction> • **234**, **235**, 236, 237  
 <simplified contents> • **234**, 235, 236, 238  
 <simplified defaulting any direction> • **234**  
 <simplified defaulting left> • **234**  
 <simplified defaulting left or right> • **234**, 238, 436  
 <simplified defaulting left or undirected> • **234**, 238, 436  
 <simplified defaulting right> • **234**  
 <simplified defaulting undirected> • **234**, 238, 436  
 <simplified defaulting undirected or right> • **234**, 238, 436  
 <simplified direction override> • **235**, 236, 237  
 <simplified factor high> • **234**, **235**  
 <simplified factor low> • **234**, **235**, 237  
 <simplified multiset alternation> • **234**, 236, 237  
 <simplified negation> • **235**, 236, 238  
 <simplified override any direction> • **235**, 238  
 <simplified override left> • **235**, 237  
 <simplified override left or right> • **235**, 237, 238, 436  
 <simplified override undirected> • **235**, 237, 238, 436  
 <simplified override undirected or right> • **235**, 237, 238, 436  
 <simplified path pattern expression> • 217, 226, **234**, 236, 436, 437  
 <simplified path union> • **234**, 237  
 <simplified primary> • **235**, 238  
 <simplified quantified> • **235**, 236, 237  
 <simplified questioned> • **235**, 236, 237  
 <simplified secondary> • **235**, 237  
 <simplified term> • **234**, 237  
 <simplified tertiary> • **235**, 237  
 simply contained in • 81  
 simply containing • 81  
 simply contains • 81  
 <single quoted character representation> • 365, **366**, 369  
 <single quoted character sequence> • **365**, 369, 393  
 site • 77  
 <slash minus> • 234, 236, 391, **392**  
 <slash minus right> • 234, 236, 391, **392**  
 <slash tilde> • 234, 236, 391, **392**  
 <slash tilde right> • 234, 236, 391, **392**  
 <solidus> • 256, 272, 304, 305, 326, 397, **398**  
 sort direction • 252  
 <sort key> • **251**, 252  
 <sort specification> • 33, **251**, 252, 458  
 <sort specification list> • 248, **251**, 419, 458  
 <source node type name> • **146**, 147, 148  
 <source node type reference> • **145**, **146**, 147, 148  
 <source predicate part 2> • **293**, 350, 351  
 <source/destination predicate> • 283, **293**, 294, 439  
 <space> • 285, 355, 356, 357, 367, 370, 394, 396, **397**, 398  
 specified by • 81  
 specify • 81  
 <square root> • **307**, **308**  
 stDev • 249, 388  
 stDevP • 249, 388  
 <standard digit> • **396**, 398, 427, 466

standard-defined classes • 427  
 standard-defined features • 84  
 standard-defined subclasses • 427  
 <start node function> • 330  
 <start position> • 316, 317, 318, 319, 321  
 <start transaction command> • 88, 95, 425  
 startNode • 330, 388  
 <statement> • 100, 103, 104, 105, 106, 125, 163, 208  
 <statement block> • 49, 50, 51, 105  
*statement completion unknown* • 25, 242, 305, 309, 310, 311, 312, 354, 355  
 <statement mode> • 127, 184, 190, 193  
 static object • 30  
 static object type • 30  
 static site • 30  
 <static variable> • 51, 209  
 <static variable definition> • 50, 105, 107  
 <static variable definition block> • 49, 50, 51, 105  
 <static variable name> • 108, 109, 110, 209, 386  
 strict elements • 56  
 strict interior variable • 229  
 strict subtype • 58  
 strict supertype • 59  
 strictly conforms • 58  
 <string length> • 316, 317, 318, 319, 321  
 <string literal character> • 366, 370  
 <string value expression> • 90, 91, 290, 300, 301, 307, 313, 314, 454  
 <string value function> • 313, 316, 318, 454, 466  
 <subpath variable> • 219, 386, 402  
 subpath variable binding • 403  
 <subpath variable declaration> • 219, 221, 408  
*substring error* • 319, 321  
 <substring function> • 316, 317, 318, 319  
 subtype relation • 58  
*successful completion* • 44, 45, 427  
 successful outcome • 43  
 supertype • 59  
 supertype relation • 59  
 supported result • 43  
 symbolic links • 35  
 symbols • 402  
*syntax error or access rule violation* • 44, 178, 202, 209, 223, 230, 273, 424

## — T —

Feature T351, “Bracketed comments” • 466  
 TABLE • 115, 124, 263, 390  
 TABLES • 390  
 TAN • 307, 310, 389  
 TANH • 307, 310, 389  
 THEN • 105, 162, 176, 338, 340, 350, 351, 352, 389  
 TIES • 390

TIME • 76, 90, 323, 324, 360, 368, 371, 376, 382, 389, 421  
 TIMESTAMP • 368, 389  
 TO • 146, 390  
 TRAIL • 55, 214, 227, 229, 390, 405, 438  
 TRAILING • 316, 317, 318, 320, 321, 389  
 TRANSACTION • 95, 390  
 TRANSLATE\_REGEX • 389  
 TRIM • 316, 317, 318, 334, 389  
 TRUE • 302, 357, 359, 365, 373, 389  
 TRUNCATE • 389  
 TYPE • 122, 124, 134, 138, 139, 140, 141, 143, 145, 152, 261, 390  
 TYPES • 390  
 tail • 334, 388  
 <tail list function> • 334  
 <temporal literal> • 365, 367  
 <temporal type> • 375, 376  
 temporary • 77  
 <term> • 304, 326, 327  
 <then statement> • 105  
 <tilde> • 146, 219, 222, 225, 235, 397, 398, 436  
 <tilde left bracket> • 145, 218, 391, 392  
 <tilde right arrow> • 219, 222, 225, 391, 392, 436  
 <tilde slash> • 234, 236, 391, 392  
 time • 23  
 <time function> • 323, 324  
 <time function parameters> • 323, 324, 325  
 <time literal> • 367, 368, 371, 373  
 time of day • 23  
 time scale • 23  
 time shift • 23  
 <time string> • 323, 325, 368, 371, 373, 391  
 toLower • 316, 388  
 toUpper • 316, 388  
 <token> • 388, 394, 452  
 trail • 54  
 <transaction access mode> • 97  
 <transaction activity> • 88, 101  
 <transaction characteristics> • 95, 97, 449  
 transaction command • 48  
 <transaction mode> • 97  
*transaction resolution unknown* • 25, 359, 360, 361  
*transaction rollback* • 25, 99  
 transient • 77  
 <trigonometric function> • 307, 309, 310  
 <trigonometric function name> • 307, 310  
 <trim byte string> • 317, 321  
 <trim character string> • 316, 318, 320, 466  
*trim error* • 320, 321  
 <trim function> • 316, 317, 318, 319, 320, 466  
 <trim list function> • 334, 335  
 <trim source> • 316, 317, 318, 319, 320, 466

<trim specification> • 316, 317, 320, 321, 466  
 <truth value> • 302  
 truth values • 57  
 type hierarchy • 60  
 <type name> • 151, 385, 386  
 <type signature> • 49, 50, 211, 212  
 type signature descriptor • 47

## — U —

UINT • 58, 74, 376, 379, 380, 389  
 UINT128 • 58, 74, 376, 380, 389  
 UINT16 • 58, 74, 376, 380, 389  
 UINT256 • 58, 74, 376, 380, 389  
 UINT32 • 58, 74, 376, 380, 389  
 UINT64 • 58, 74, 376, 380, 389  
 UINT8 • 74, 376, 380, 389  
 UNDIRECTED • 146, 147, 390  
 UNION • 178, 179, 337, 338, 389, 423, 466  
 UNIQUE • 389  
 UNIT • 346, 389  
 UNIT\_BINDING\_TABLE • 297, 298, 389  
 UNIT\_TABLE • 297, 298, 389  
 UNKNOWN • 302, 365, 373, 389  
 UNNEST • 338, 340, 389  
 UNSIGNED • 58, 74, 376, 379, 380, 389  
 UNWIND • 389  
 UPPER • 316, 319, 389  
 USE • 207, 389  
 unbounded quantifier • 220  
 <unbroken accent quoted character sequence> • 366, 369, 392, 393  
 <unbroken character string literal> • 365, 368, 369, 371, 373  
 <unbroken double quoted character sequence> • 365, 366, 369  
 <unbroken single quoted character sequence> • 365, 369  
 <underscore> • 367, 368, 369, 394, 397, 398  
 undirected • 54  
 <unicode 4 digit escape value> • 366  
 <unicode 6 digit escape value> • 366, 367  
 <unicode escape value> • 366, 370, 464  
 union type • 60  
 unit value • 69  
 unordered • 33

unsigned 128-bit integer type • 74, 380  
 unsigned 16-bit integer type • 74, 380  
 unsigned 256-bit integer type • 74, 380  
 unsigned 32-bit integer type • 74, 380  
 unsigned 64-bit integer type • 74, 380  
 unsigned 8-bit integer type • 74, 380  
 <unsigned binary integer> • 367, 369  
 <unsigned decimal integer> • 73, 367, 368, 369, 375, 376  
 unsigned exact numbers • 58

unsigned exact numeric types • 58  
 <unsigned hexadecimal integer> • 355, 357, 367, 368, 369  
 <unsigned integer> • 73, 219, 220, 228, 297, 356, 358, 367, 369, 372  
 <unsigned integer specification> • 56, 227, 254, 255, 297, 298, 299  
 <unsigned literal> • 297, 365  
 <unsigned numeric literal> • 355, 365, 367, 388  
 <unsigned octal integer> • 367, 368, 369  
 unsigned regular integer type • 74, 380  
 unsigned user-specified integer type • 381  
 unsigned user-specified integer types • 74  
 <unsigned value specification> • 297, 298, 306  
 <untyped value expression> • 300, 301  
 <upper bound> • 219, 220, 224, 409  
 <url path parameter> • 256, 258, 261, 263, 266, 268, 270, 272, 277, 278  
 <url segment> • 272, 273  
 <use graph clause> • 104, 160, 207, 232  
 user-defined • 57  
 user-specified approximate numeric type • 382  
 user-specified approximate numeric types • 75  
 user-specified decimal exact numeric type • 381  
 user-specified decimal exact numeric types • 75

## — V —

VALUE • 111, 112, 117, 349, 389  
 VALUES • 389  
 VARBINARY • 57, 71, 375, 378, 389, 465  
 VARCHAR • 57, 71, 375, 377, 389, 465  
 VERTEX • 6, 377, 382, 390, 393  
 VERTICES • 390  
 <value expression> • 50, 51, 52, 111, 117, 118, 166, 168, 169, 172, 184, 192, 200, 201, 204, 205, 218, 230, 242, 249, 250, 251, 285, 300, 301, 306, 336, 341, 342, 343, 344, 346, 350, 352, 353, 354, 372, 415  
 <value expression primary> • 289, 300, 301, 304, 306, 313, 322, 326, 329, 332, 337, 338  
 <value initializer> • 117, 118  
 <value name> • 385, 386  
 <value parameter definition> • 111, 117, 118  
 <value query expression> • 306, 349  
 <value specification> • 297, 298, 453  
 <value type> • 56, 211, 300, 375, 377, 382, 383, 384, 455, 457  
 <value variable> • 111, 117, 118, 192  
 <value variable declaration> • 111, 117  
 <value variable definition> • 111, 117  
 <variable name> • 244, 245, 385, 386  
 variable quantifier • 220  
 <verbose binary exact numeric type> • 376  
 <vertical bar> • 55, 217, 232, 234, 397, 398  
 view • 48

— W —

WALK • 55, 227, 228, 390, 405  
WHEN • 162, 338, 340, 350, 351, 389  
WHERE • 185, 189, 193, 213, 218, 219, 239, 389, 415  
WITH • 193, 194, 389  
WITHOUT • 389  
WRITE • 95, 97, 390  
*warning* • 44, 45, 134, 138, 140, 152, 154, 156, 158, 356, 358,  
  363, 427  
<when clause> • 162, 165, 168, 170, 172, 176  
<when operand> • 350, 351  
<when operand list> • 350, 351  
<when then linear data-modifying statement branch> • 162  
<when then linear query branch> • 176  
<where clause> • 189, 190, 192, 193, 204, 239  
<whitespace> • 314, 320, 321, 354, 355, 356, 358, 359, 360,  
  361, 362, 392, 393  
<wildcard label> • 54, 232, 400  
with an intervening instance of • 81  
without an intervening instance of • 81  
word • 402, 403

— X —

XOR • 302, 389

— Y —

YIELD • 215, 244, 389  
<yield clause> • 105, 106, 213, 214, 215, 223, 242, 243, 244,  
  245  
<yield item> • 215, 244  
<yield item alias> • 244, 245  
<yield item list> • 215, 244  
<yield item name> • 244

— Z —

ZERO • 389

ZONE • 90, 390