Open Geospatial Consortium Inc.

Date: 2026-08-28

Reference number of this Open Geospatial Consortium © Project Document: **OGC 26-048**

Version: 1.2.0

Category: Open Geospatial Consortium© Interface Standard

Editor: Peter Baumann

Web Coverage Processing Service (WCPS)
Geo Datacube Query Language Standard

Document type:	Open Geospatial Consortium ® Interface Standard
Document subtype:	Interface
Document stage:	Draft
Document language:	English

Warning

OGC official documents use a triple decimal-dot notation (i.e. MM.xx.ss). This document may be identified as MM.xx (Major.minor) and may include increments to the third dot series (schema changes) without any modification to this document, or the version displayed on the document. Corrections to the document are registered via corrigendums.

This document is an OGC Standard. Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation

.

Contents

Page

Tables

Page

i. Abstract

The OGC Web Coverage Processing Service (WCPS) defines a protocol-independent, declarative language for on-demand extraction, processing, and analysis of multi-dimensional gridded coverages ("datacubes") representing – among others – spatio-temporal sensor, image, simulation output, or statistics data. Key principles of this language include implicit iteration and spatio-temporal semantics.

ii. Submitting organizations

The following organizations have submitted this Interface Standard to the Open Geospatial Consortium, Inc.

- Constructor University
- Feng-Chia University

iii. Document Contributor Contact Points

Name	Organization
Peter Baumann	Constructor University
Chen-Yu Hao	Feng-Chia University

iv. Revision history

Date	Release	Author	Paragraph modified	Description
2008-04-29	0.0.1	Peter Baumann		created from 07-151r1 and 08-059r3
2008-09-01	0.0.2	Peter Baumann	Many	Final version following adoption
2008-09-22	0.0.3	Peter Baumann	Result type specs	More accurate phrasing, not just table reference
2009-01-08	1.0.0	Peter Baumann	many	Reworked scalar operations; incorporated e-vote comments; fixed syntax inconsistencies; final editorial brush-up
2019-05-30	1.1.0	Peter Baumann	many	Extended to allow processing of OGC CIS 1.1
2020-08-03	1.1.0	Peter Baumann	Several	Proofreading fixes
2026-xx-xx	1.2.0	Peter Baumann	Several	Fixed typos, covered corner cases, enhanced semantics, adjusted to OGC CIS 1.2, removed deprecated parts

v. Changes to the Open Geospatial Consortium ® Abstract Specification

The Open Geospatial Consortium ® Abstract Specification does not require any changes to accommodate the technical contents of this document.

vi. Future Work

This WCPS framework will be enhanced and extended including the following features:

1) Add support for the *regions-of-validity* concept [10].

2) Lift some of the constraints on coverages in Subclause 6, such as by adding support for further coverage types (point, curve, surface, solid).

3) Add support for inserting, updating, and deleting coverages through expressions.

Foreword

This OGC WCPS Language Standard 1.2 defines a datacube analytics language based on OGC Coverage Implementation Schema (CIS) 1. This document supersedes WCPS version 1.1. It is as powerful and expressive as WCPS 1, but has the following differences:

- revised in the model basis: relying on CIS 1.2, rather than on pure mathematical foundations.
- enhanced semantics by capturing and excluding invalid situations as per CIS.
- Enhanced functionality, in particular: polygon clipping has been added with several variants, including curtains and corridors.
- The xWCPS part of WCPS 1 is now the standard syntax, the deprecated variant is removed.
- Coverage probing functions and requirements have been adjusted to the CIS logical structure.
- The non-normative text has been reworked for more clarity and conciseness.
- The syntax is more lenient towards simple syntax errors.

The language capabilities have been enhanced, affecting the language elements in detail as follows:

- unmodified: *processCoveragesExpr, processingExpr, encodedCoverageExpr, decodeCoverageExpr, coverageConstructorExpr, coverageIdentifier, Metadata, rangeConstructorExpr, subsetExpr, trimExpr, sliceExpr, extendExpr, rangeExpr, unaryInducedExpr, unaryArithmeticExpr, trigonometricExpr, exponentialExpr, booleanExpr, castExpr, fieldExpr, binaryInducedExpr, switchExpr, reduceExpr, scalarExpr, booleanScalarExpr, numericScalarExpr, stringScalarExpr, condenseExpr, generalCondenseExpr*.
- extended: *domainSetExpr, rangeTypeExpr, rangeSetExpr, domainOpExpr*.
- minor syntax fix (no functional change): *scaleExpr, crsTransformExpr*.
- added: *clipPolygonExpr, clipMultipolygonExpr, clipLinestringExpr, clipCurtainExpr, clipCorridorExpr, polygonizeExpr, getExpr*.

This document includes two normative Annexes, A and B.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium Inc. shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Introduction

The OGC Web Coverage Processing Service (WCPS) defines a geo datacube analytics language for server-side retrieval, filtering, processing, and fusion of multi-dimensional geospatial grid coverages representing, such as spatio-temporal sensor, image, simulation, and statistics data. WCPS is model-based, aligned with the OCG/ISO Coverage Implementation Schema (CIS) standard [OGC 26-041][4].

The WCPS language is independent from any particular request and response encoding as no concrete request/response protocol is assumed, such as the WCS Processing Extension [7] of the Web Coverage Service (WCS) Standard [8].

Open Geospatial Consortium Interface: Web Coverage Processing Service (WCPS)

1 Scope

This document defines a protocol-independent language for retrieving and processing geospatial coverage datacubes.

Normatively, WCPS relies on the coverage model as defined in the OGC Coverage Implementation Schema (CIS) Standard [OGC 26-041] (which is identical to ISO 19123-2 [4]) where coverages are defined as "digital geospatial information representing space-varying phenomena", practically speaking: regular and irregular multi-dimensional grids, which are supported by WCPS currently. In terms of processing, WCPS is an implementation target of the OGC Abstract Specification Topic 6.3 "Schema for Coverage Geometry and Functions - Part 3: Processing Fundamentals" [OGC 1-060r2] (which is identical to ISO 19123-3 [5]).

NOTE Following OGC's rules of compatibility among minor release numbers this WCPS standard applies to all CIS 1.x versions.

2 Compliance

Annex A (normative) specifies compliance tests which shall be passed by any service claiming to implement WCPS.

3 Normative references

The following normative documents contain provisions that, through reference in this text, constitute provisions of this standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies, with the highest minor version number.

- OGC 26-041 Coverage Implementation Schema (CIS), version 1.2
- ISO/IEC 13249-3:2016 Information technology — Database languages — SQL multimedia and application packages Part 3: Spatial

4 Terms, definitions, and abbreviations

4.1 Terms and definitions

For the purposes of this document, the terms and definitions given in the above references (in particular: CIS) apply.

4.1.1
WCPS query
coverage processing string conforming to the syntax and semantics of WCPS

4.2 Abbreviations

CIS	Coverage Implementation Schema
CRS	Coordinate Reference System
UML	Unified Modeling Language
UoM	Unit of Measure
WCS	Web Coverage Service
WCPS	Web Coverage Processing Service
XML	Extensible Markup Language

5 Conventions

5.1 UML notation

All class diagrams that appear in this standard use Unified Modeling Language (UML) static class structure diagrams.

5.2 Platform-neutral and platform-specific specifications

WCPS is a model-based, high-level expression language, independent from any client/server transmission protocol and server-side implementation paradigm.

Data treated (coverages / datacubes) can be encoded in many alternative ways, as defined in CIS. One service embedding and encoding is defined in the Web Coverage Service (WCS) Processing Extension [7]. Other encodings may specify an API (Application Programming Interface).

6 WCPS coverage data model

6.1 Coverage model

The coverage model of WCPS relies on the OGC Coverage Implementation Schema (CIS). As per CIS, a coverage is a set of locations, each one bearing some value, together with some constraints. Following the mathematical notion of a function that maps elements of a domain (here: spatio-temporal coordinates) to a range (here: "pixel", "voxel", … values), a coverage can be structured into (Figure 1):

- a *domain set* of coordinate points: "*where in the multi-dimensional space can I find values?*"
- a *range set*: "*what are the values?*"
- a *range type*: "*what do those values mean?*"
- optional *metadata*: "*what else should I know about these data?*"

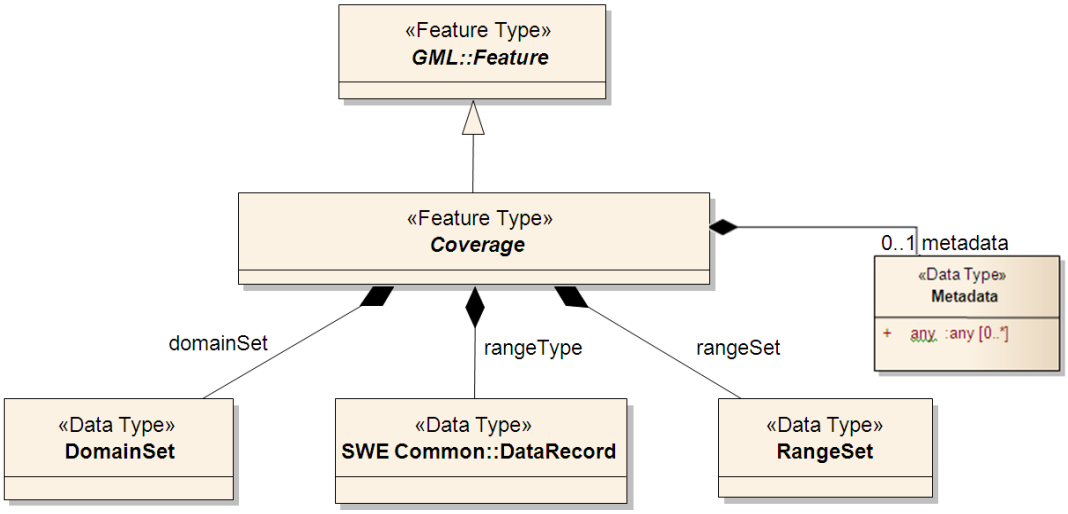

Figure 1 – Schematic coverage UML diagram

WCPS specifically focuses on the spatio-temporal datacubes of CIS ("grid coverages") where all the locations sit on some multi-dimensional raster. The term "grid" or "raster" refers to a topological structure where each grid position has exactly one upper and one lower nearest neighbor, except at the borders of the grid/raster where less neighbors exist. Figure 1 shows sample 2-D and 3-D regular and irregular grid structures.

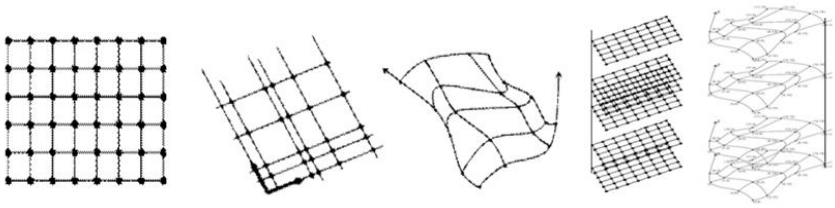

Figure 2 – Sample regular and irregular grid structures

WCPS supports the *General Grid Coverage* of CIS introduced with CIS 1.1; this structure replaces the legacy *Rectified Grid Coverage* and *Referenceable Grid Coverage* (CIS 1.2 Annex B) in a unifying, simplifying, and at the same time more expressive structure. For the purpose of this document, "coverage" always means "general grid coverage". Some restrictions apply over the CIS coverage model [OGC 26-041][4]):

- Optional *envelope* is discarded; it may get added in a future version.
- Optional *gridLimits* is discarded; it may get added in a future version.
- Redundant attribute *dimension* is discarded, as it can be derived from the number of axes.
- Optional *coverageFunction* is discarded as it addresses physical data layout which is not in scope of WCPS.
- Only index, regular, and irregular axes are supported (see Figure 2), not sensor transformation models.
- The range set is restricted to a flat record of atomic values, with at least one record element.
- Beyond the current CIS version, axis-specific interpolation is supported, in preparation of future CIS versions.

Figure 3 shows the *General Grid Coverage* excerpt used from the CIS UML model. This is the data model of WCPS around which operations and syntax are designed.

Requirement 1
A `coverageExpr` in a WCPS query **shall** evaluate to a structure according to Figure 3, with range field types according to Table 1.

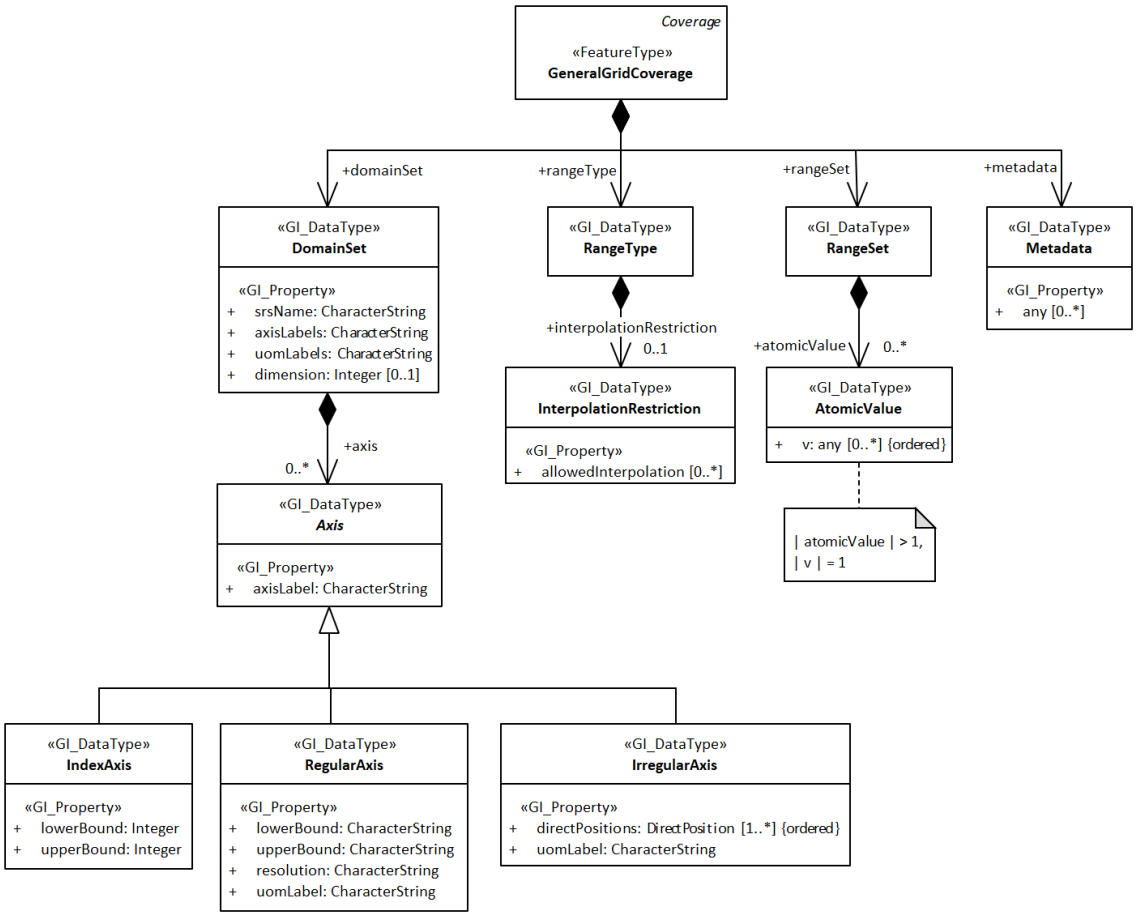

Figure 3 – UML model CIS subset of WCPS

NOTE While *General Grid Coverage* is the model on which WCPS is based for processing, an implementation may additionally offer query results structured as legacy CIS *Rectified* or *Referenceable Grid Coverage*.

6.1.1 Domain set

The domain (set) describes the grid and its properties. A grid is based on a Coordinate Reference System (CRS) defining an ordered list of axes (directions) on which grid point coordinates are expressed. Consequently, a coordinate is a tuple where the i^{th} component is defined by the i^{th} axis using its numbering scheme (such as "42.5", "42°30'", "2026-07-15T18:06"). A grid has at least one axis, which is delimited by a lower and upper bound each. The grid point coordinates within these bounds, referred to as *direct positions*, thus define a *datacube*. All axis names must be pairwise different.

An axis can be of three types, which can be arbitrarily combined to form a grid:

- A Cartesian ("index") axis has integer coordinates, so is discrete. The unit of measure is 1. An index coordinate interval is described by its lower and upper bound.

- A regular axis is based on some totally ordered set of values, such as degrees, meters, or time, with some suitable unit of measure which can be discrete or continuous. A regular axis is described by lower and upper bound and a resolution (aka grid point distance).

- An irregular axis likewise based on some totally ordered value set with some unit of measure, which can be discrete or continuous. An irregular axis is described by a list of coordinate positions, in ascending order. First and last list element determine the lower and upper bound, respectively.

WCPS and CIS do not define CRSs nor their syntax. Recommended notations are either OGC URLs such as https://www.opengis.net/def/crs/EPSG/0/4326, or CURIE notation, such as "EPSG:4326".

NOTE OGC provides a resolver, operated by the Constructor University L-SIS group on behalf of OGC and using an open-source service created by rasdaman.

Beyond CIS 1.2, interpolation can be applied on a per-axis basis for regular and irregular axes. See Subclause 6.1.4 for details.

NOTE Among others, this allows differentiating interpolation along Lat/Lon versus time axes.

6.1.2 Range set

Each direct position in a coverage has a value assigned which is a record with one or more atomic components. All range values in a coverage adhere to one common structure, described in the range type.

6.1.3 Range type

The range value semantics is defined in the range type. The range value structure consists of a record with atomic components named *field*, each described through the following components:

- Record component name, unique within the record, called *field label*;

- The data type, one of the types listed in Table 1 - all common numeric and Boolean data types are supported in WCPS;

- The *nature* of the values, which describes the statistical classification of the range values, one of:

    - *Quantity* (synonym: *numeric*): data have meaning as a measurement, like height or temperature. All operations are possible.

    - *Category* (synonym: *categorial*): characteristics such as land use, mapped to numbers which, however, do not have mathematical meaning. Hence, no arithmetics is possible, just equality comparison.

- *Count*: Data which fall into categories, but they have a certain numerical meaning; for example, rankings can be compared, but also averaged.

Example Temperature is *quantity* - it can be added, subtracted, multiplied. Time is *count* - a date addition like *'2023' + '2025'* does not make sense while a test *$c.date < '2025'* is valid; contrarily, with time durations as defined in ISO 8601, such as "*D1m*" for 1 month, time becomes a quantity and *'2023' + D1m'* is valid. Land classification is categorial - adding *$c.road + $c.water* and comparing *$c.road > $c.water* do not make sense, however equality check *$c.use = $c.road* is valid, assuming *road* and *water* stand for classification numbers.

- A (possibly empty) set of *nil* values which are to be ignored, e.g., in aggregations;
- An optional *unit of measure* (*uom*), applicable only to quantities and counts, a unit of measure can be provided optionally. This is a string whose syntax and semantics is not defined by this standard, although UCUM is recommended;

Example Temperature can be measured in Kelvin ("K") or Celsius ("C"), distance in meters ("m") or feet ("ft"), etc.

- A (possibly empty) set of interpolation methods applicable to this field, in case *nature* is *quantity*.

Requirement 2
In a coverage addressed by a WCPS query the range data type of each field **shall** be one of the types listed in Table 1.

NOTE It is not required that all range fields within a coverage are of the same type.

Table 1 – Coverage range field data types.

Data type name (case insensitive)	Alias (case insensitive)	Comment
`bool`	`boolean`	Boolean
`char`	`int8`	8-bit signed integer
`unsigned char`	`uint8`	8-bit unsigned integer
`short`	`int16`	16-bit signed integer
`unsigned short`	`uint16`	16-bit unsigned integer
`long`	`int32`	32-bit signed integer
`unsigned long`	`uint32`	32-bit unsigned integer
`long long`	`int64`	64-bit signed integer
`unsigned long long`	`uint64`	64-bit unsigned integer
`float`	`float32`	Single precision floating point number

`double`	`float64`	Double precision floating point number
`complex short`	`CInt16`	2*16-bit signed complex number
`complex long`	`CInt32`	Single precision complex number
`complex ( float )?`	`CFloat32`	Single precision complex number
`complex double`	`CFloat64`	Double precision complex number

6.1.4 Interpolation

Interpolation implicitly is applied in several coverage operations, in particular scaling and CRS transformation, where new values sitting between grid positions are derived. A set of interpolation methods which are admissible on principle can be specified in both coverage domain and range type. The method actually applied in an operation can be specified in the operations affected. It must be consistent with the methods the coverage allows.

Function *effint*(*c,a,f*) determines the set of interpolation methods effectively applicable to field *f* along axis *a* in coverage *c*. It is given by the intersection of methods applicable to the axis and to the field; a non-existing interpolation set (in the formalism denoted as *null*) is interpreted as "no constraint imposed".

Requirement 3
Only those interpolation methods **shall** be applied in a **coverageExpr** on field *f* along axis *a* in coverage *c* on a server supporting method set *s* which are contained in *effint*(*c,a,f*) = *e* constructed sequentially as follows:

- if *c.domain.axis*[*a*].*nature*≠*quantity*
  then *e* = ∅
- if *c.domain.axis*[*a*].*int*≠*null* and *c.rangetype.fieldLabels*[*f*].*int*≠*null*
  then *e*= *c.domain.axis*[*a*].*int*∩ *c.rangetype.fields*[*f*].*int*∩*s*
- if *c.domain.axis*[*a*].*int*=*null* and *c.rangetype.fields*[*f*].*int*≠*null*
  then *e*= *c.rangetype*.fields[*f*].*int*∩*s*
- if *c.domain.axis*[*a*].*int*≠*null* and *c.rangetype.fields*[*f*].*int*=*null*
  then *e*= *c.domain.axis*[*a*].*int*∩*s*
- if *c.domain.axis*[*a*].*int*=*null* and *c.rangetype.fields*[*f*].*int*=*null*
  then *e*=*s*

Example Assume a domain axis has interpolation methods { *nearest, linear* } and a range component has { *linear*, *quadratic* }. Then, only *linear* can be applied in scaling. If domain has only { *nearest* } and range interpolation is as before, then the intersection is empty, and no interpolation is applicable (which will constrain some operations).

NOTE 1 Numerical effects have to be expected when applying interpolation, with implementation-defined results.

NOTE 2 Legacy coverages without interpolation stored are representd as having a null value as interpolatino method set, which in the conditions above enables all methods supported by the server.

NOTE 3 A coverage without any interpolation allowed (i.e., *effint*(*c*,*a*,*f*) =∅) is said to be *discrete*: only at the direct positions range values are available, everywhere else access will lead to an error. Otherwise, the coverage is called *continuous*.

6.1.5 Metadata

The metadata component of a coverage can contain any information - its syntax and semantics is not known to CIS nor WCPS. Consequently, the metadata component is treated as a byte string.

6.2 Probing functions and semantics definition

The semantics of WCPS expressions is defined via so-called "probing functions" which extract information from a coverage.

Note Probing functions are not language elements, rather they serve to define what the actual WCPS coverage expressions evaluate to.

A serialized equivalent of the UML model of Figure 3 is defined in Table 2, together with the probing functions available on the coverage components, relying on common dot (".") notation for functional nesting. Further, for notational convenience in this document, on the list and set valued items the usual list and set functions are assumed for extraction and manipulation, such as union, intersection. Further, application of some function to a list or set which is defined on the elements denotes simultaneous application of this function to all list or set elements. Such a functional extension is called an *induced function* with the understanding that the function on the value type "induces" an operation on lists/sets of such values.

Semantics definition makes use of common mathematical notation:

- <*a*,…,*z*> denotes an ordered list (aka tuple) of elements *a*,…,*z*. A list may be empty.
- {*a*,…,*z*} denotes an unordered set of elements *a*,…,*z*. A set may be empty.

In case the elements are assigned a name, denoted as *n*:*a* where *n* is unique within the list or set, such items can be addressed in a set or list *sl* as *sl*[*n*]. In lists, components additionally can be addressed by their position, with numbering starting with 1.

Requirement 4
Coverage probing functions **shall** be defined as given by Figure 3 and Table 2.

Table 2 – WCPS probing functions[1].

Coverage UML component	Probing function, for coverage *c*	Data type	Comment
(implicit,	*c.type*	name	In this standard, fixed

[1] These probing functions are available in the language through the *getExpr* branch (Subclause 7.5.8).

through coverage subtype)			to *GeneralGridCoverage*
id	*c.id*	string	Default: *null*
domainSet	*c.domain*	record	Abstract, only access to components defined
- dimension		integer	Number of dimensions, equal to | axisLabels |, thus redundant and not needed for semantics
- srsName	*c.domain.crs*	string	Ex: "[EPSG:4326]"
- axisLabels	*c.domain.axisLabels*	list of names	Axis names in proper order of CRS Ex: <Lat, Lon, Date>
- uomLabels	*c.domain.uomLabels*	list of strings	List of units per axis, in axis sequence[2] Ex: <deg, deg, d>
-axis	*c.domain.axes*[*a*] for *a*∈*c.domain.axisLabels*	axis definition	See **domainExpr** for the structure Ex: *c.domain*[*red*]
- - axisLabel		name	Captured by *axisLabels* above, so redundant
- - axisType	*c.domain.axes*[*a*].*axisType*	One of: *index*, *regular*, *irregular*	
- - indexAxis			Index axis type
- - - lowerBound	*c.domain.axis*[*a*].*lo*	integer	
- - - upperBound	*c.domain.axis*[*a*].*hi*	integer	
- - regularAxis			Regular axis type
- - - lowerBound	*c.domain.axes*[*a*].*lo*	Number or string, determined by uom	Lowest coordinate in coverage extent along this axis
- - - upperBound	*c.domain.axes*[*a*].*hi*	Number or string, determined by uom	Highest coordinate in coverage extent along this axis
- - - resolution	*c.domain.axes*[*a*].*res*	Number or string, determined by uom	Only for axis type *regular*
- - - uomLabel	*c.domain.axes*[*a*].*uom*	string	Unit of coordinates along this axis, in UCUM Ex: "deg"

[2] Redundant, in WCPS constructed from the *uomLabel* components in *axis.*

- - irregularAxis			Irregular axis type
- - - direct- - - - Positions	*c.domain.axes*[*a*].*pos*	list of coordinate values	Coordinate values on axis *a* of all direct positions
rangeType	*c.rangetype*	list of range field definitions	
-/-	*c.rangetype.fieldLabels*	list of names	Not explicitly present in CIS, can be deduced from DataRecord
- fields	*c.rangetype.fields*[*f*] for *f*∈*c.rangetype.fieldLabels*	list of names	Ex: <red, green, blue>
- definition	*c.rangetype.fields*[*f*].*def* for *f*∈*c.rangetype.fieldLabels*	string	Data type, as per Table 1
- nilValues	*c.rangetype.fields*[*f*].*nil* for *f*∈*c.rangetype.fieldLabels*	List of values	
- nature	*c.rangetype.fields*[*f*].*nature* for *f*∈*c.rangetype.fieldLabels*	name	One of: *quantity*, *count*, *category*[3]
- uom	*c.rangetype.fields*[*f*].*uom* for *f*∈*c.rangetype.fieldLabels*	string	UCUM notation
- interpolationRestriction	*c.rangetype.fields*[*f*].*int* for *f*∈*c.rangetype.fieldLabels*	set of identifiers	Admissible interpolation methods
rangeSet	*c.range*	Tensor of values, all of same type	Technically, an *n*-D array
- atomicValue.v	*c.range*[*i,j,k,…*].*f*	atomic type	Range field at some index coordinate
metadata	*c.metadata*	string	Content not interpreted by CIS nor WCPS
Components beyond CIS[4]:			
./.	*c.domain.axis*[*a*].*int*	set of identifiers	interpolation methods admissible for this (regular or iregular)

[3] For *quantity*, *numericl* is a synonym; for *category*, *categorial* is a synonym.

[4] Components beyond CIS are elements not defined in CIS currently, but under discussion for standardization and thus possibly available in future versions of CIS. Until then, coverages stored and delivered in WCPS are set to null; processing expressions may define such values, but when delivered as a query result to clients they will be discarded.

			axis
./.	*c.domain.index*[*a*]	pair <*lo*,*hi*> of integer	For some axis *a*, if of type *index* then <*a.lo*,*a.hi*> else if regular then <0,*ceil*(*a.hi*–*a.lo*/*a.res*)> else (irregular) <0,|*a.pos*|–1>
- lo	*c.domain.index*[*a*].*lo*	integer	Lower index bound, typically 0
- hi	*c.domain.index*[*a*].*hi*	integer	Upper index bound
WCPS auxiliary functions, not part of CIS:			
./.	*c.domain.pos*	set of coordinate tuples	All direct position coordinate tuples
./.	*c.domain.index*[*a*].*pos*[*i*]	number or string, determined by uomLabel	Coordinate at position *i*, for *i* < |*c.domain.axis*|
./.	*effint*(*c*,*a*,*f*)	set of identifiers	interpolation methods effectively applicable (cf. Requirement 3)

NOTE SWE Common, from which the DataRecord structure is adopted for the range type, foresees several more optional components which are disregarded by CIS and WCPS. Further, the DataRecord structure in CIS (and hence WCPS) are non-recursive, limited to a nesting level 1.

NOTE 2 Some structures and naming conventions in CIS are legacy, in an attempt to achieve a smooth transition from the early GML-centric coverage model to a more flexible, enhanced model. This GML model also did not clearly differentiate between logical and physical level, whereas WCPS is defined on logical level and strictly independent from the concrete data structures. Consequently, WCPS considers only coverage items on logical level and leaves all physical aspects implementation-defined.

7 WCPS Coverage Processing Language

7.1 Query Syntax and Semantics

7.1.1 Overview

The WCPS coverage processing language allows clients to request processing of one or more coverages available on a server supporting WCPS. Such a server evaluates an expression and returns an appropriate response to the client. The result returned upon a successful request consists of an ordered sequence of either zero or more coverages or zero or more scalar values (no mix).

NOTE Data items within a WCPS response list can be heterogeneous in size and structure. In particular, the coverages within a response list can have different dimensions, domains, range types, etc. However, a response always consists of either coverages or scalar values, no mix of both.

The WCPS primitives plus the nesting capabilities form an expression language independent from any particular coverage encoding, collectively referred to as the *WCPS language*. In the subsequent subclauses, the syntax and semantics of all language elements are detailed. The complete syntax is listed in (normative) Annex B.

Having a model-based language enables additional safety, trust, and efficiency (in particular, in unsupervised processing inside automated processing chains):

- Implicit iterations - this simplifies expressions and avoids non-terminating loops. At the same time, this opens up opportunities in the server for choosing optimal evaluation plans.

  NOTE WCPS has been designed to be "safe in evaluation" – i.e., implementations are possible where any valid WCPS request can be evaluated in a finite number of steps, based only on the operation primitives. Hence, WCPS implementations can be constructed in a way that no single request can render the service permanently unavailable. Notwithstanding, it still is possible to send requests that will impose a high workload on a server.

- Spatio-temporal semantics: processing is informed about the semantics of coverage domain and range, allowing for additional semantic checking in the server prior to query execution.

  NOTE Among the errors the language does not allow are: mathematically invalid operations; combining incompatible data; statistically wrong aggregation over categorial and count data; and many more.

Requirement 5
A server claiming to support WCPS **shall** implement `processCoveragesExpr` as defined in Subclause 7.1.2.

NOTE This standard does not specify the communication protocol nor the mode of returning responses, such as synchronous or asynchronous delivery. That level of behavior is defined in protocol bindings such as the WCS Processing extension [7].

7.1.2 processCoveragesExpr

The semantics of a WCPS query - i.e.: the result delivered - is given by function *eval*(*p*) for some *processCoveragesExpr p*.

The *processCoveragesExpr* element processes a list of coverage tuples in turn. To this end, variables are defined which iterate over a list of coverages associated with each. In each evaluation step, the variables have a specific coverage assignment, jointly referred to as the *coverage assignment*.

Each coverage assignment is optionally checked first for fulfilling some predicate specified in the *where* clause, and gets selected – i.e., contributes to an element of the result list – only if the predicate evaluates to *true*. Each coverage selected gets processed, and the single result per coverage assignment evaluation is appended to the result list. This result list, finally, is returned as the overall response unless an error was generated.

In the **let** clause, named constants can be defined and assigned a single value each. Constants must be defined before use in an expression.

Example The following statement is not valid: **let** `$x:=$x+1.`

Optionally, a server may allow queries in different, implementation-defined languages in WCPS queries for retrieving the names of coverages to be processed, thus bridging data and metadata.

NOTE Among the candidates for such an external retrieval are SQL, XPath, STAC queries, etc.

Requirement 6
A WCPS server **may** support query expressions in some non-WCPS language, denoted as a string, in *processCoveragesExprs* in place of a *coverageName* in the *for* clause. Such external queries **shall** evaluate to zero or more comma-separated *coverageNames*.

Example The following query shows a sample interleaving of WCPS and XPath performing a STAC access for obtaining the coverage names bound to `$C`:

```
for $C in
  ( "//stac:Item[stac:resolution<10]/stac:id/text()" )
return $C.metadata
```

Requirement 7
A *processCoveragesExpr* **shall** be defined as below.

Let

$p_1, \ldots p_x$, be *xpathExpr*s for $x \geq 1$,
$v_1, \ldots v_n$ be *iteratorVar*s for $n \geq 1$ with *eval*(v_1), … *eval*(v_n) pairwise different,
$L_1, \ldots L_n$ be *coverageList*s for $n \geq 1$,
$c_1, \ldots c_m$, be pairwise different *variableName*s with *eval*(c_1), … *eval*(c_m) pairwise different,

$e_1, \ldots e_m$, be *letAssignment*s for $m \geq 1$ where e_i must not contain any c_j for $j \geq i$,
b be a *booleanScalarExpr* possibly containing occurrences of one or more v_i,
P be a *processingExpr* possibly containing occurrences of v_i ($1 \leq i \leq n$).

Then

for any *processCoveragesExpr* E,
where

```
E = 'for' v1 in '(' L1 ')',
              …
              vn 'in' '(' Ln ')'
    ( 'let' c1 ':=' e1 ',' … ',' cm ':=' em )?
    ( 'where' b )?
    'return' P
```

the result R of evaluating E is constructed as follows:

Let R be the empty sequence;
textually replace all occurrences of $p_1,\ldots,p_x$ in $L_1,\ldots,L_n$, by its evaluation result, adjusting commas in the XPath result name lists as necessary for *coverageList* syntax;
while L_1 is not empty:
{ assign the first element in L_1 to iteration variable v_1;
  while L_2 is not empty:
  { assign the first element in L_2 to iteration variable v_2;
    …
    { assign the first element in L_n to iteration variable v_n;
      while L_n is not empty:
      { assign the first element in L_n to iteration variable v_n;
        textually replace any occurrence of $p_1,\ldots,p_x$ in b and P;
        evaluate all e_i and b and P;
        textually replace any occurrence of c_i and P by '(' e_i ')';
        substitute any occurrence of v_i by its current coverage,
        yielding P';
        if *eval*(b)
        then
          append evaluation result *eval*(P') to R;
      }
      remove the first element from L_n;
    }
    remove the first element from L_2;
    …
  }
  remove the first element from L_1;
}

The elements contained in the *coverageList* clause, constituting coverage identifiers, are taken from the coverage identifiers advertised by the server.

Requirement 8
In a *processCoveragesExpr*, every coverage referenced in some *coverageList* **shall** exist in the server processing it.

NOTE Coverage identifiers may occur more than once in a **coverageList**. The coverage referenced will be evaluated each time it appears, in inspection sequence.

Coverages assigned through the **for** clause are assumed to be General Grid Coverages as per CIS 1.2, so the probing functions are well-defined. Below the semantics of extra WCPS-specific probing functions are defined for coverages referenced.

Requirement 9
A coverage referenced in some *coverageList* of a *processCoveragesExpr* **shall** be defined as below.

Let

C_1, … C_n be coverage references in *coverageList*s for $n \geq 1$.

Then

eval(C_i) is defined as follows:

Coverage constituent
eval(C_i).*type* = *GeneralGridCoverage*
eval(C_i).*id* = *id* of the coverage referenced
eval(C_i).*domain* = *domain* of the coverage referenced, with all items not present set to default
eval(C_i).*rangetype* = *rangetype* of the coverage referenced, with all items not present set to default
eval(C_i).*range* = *range* of the coverage referenced
eval(C_i).*metadata* = *metadata* of the coverage referenced
for $a \in$ *eval*(C_i).*axisLabels*: *eval*(C_i).*domain.axis*[a].*int* = ∅[5]
for $a \in$ *eval*(C_i).*axisLabels*: *eval*(C_i).*domain.index*[a].*lo* = *eval*(C_i).*domain.axis*[a].*lo* if *eval*(C_i).*domain.axis*[a].*axisType* = *index*, 0 otherwise
for $a \in$ *eval*(C_i).*axisLabels*: *eval*(C_i).*domain.index*[a].*hi* =

[5] As of CIS 1.2, no interpolation is assigned with axes; future CIS versions could support this, in which case the interpolation set is available accordingly.

| *eval*(C_i).*domain.axis*[*a*] | - *eval*(C_i).*domain.index*[*a*].*lo* -1

Example Assume a WCPS server offers same-size satellite images A, B, and C and a coverage M acting as a mask (i.e., with range values between 0 and 1 and the same extent as A, B, and C). Then, masking each satellite image can be performed with a request like the following:

```
for $s in ( A, B, C ),
    $m in ( M )
return encode( $s * $m, "image/tiff" )
```

7.1.3 processingExpr

Requirement 10
A *processingExpr* element **shall** be either a *encodedCoverageExpr* (see Subclause 7.1.4), or a *scalarExpr* (Subclause **Error! Reference source not found.**).

7.1.4 encodedCoverageExpr

The *encodedCoverageExpr* element specifies encoding of a coverage-valued request result by means of a data format and possible extra encoding parameters. The output coverage may be unable to contain some components of a coverage.

Example For some georeferenced coverage, a GeoTIFF result file will contain the coverage's geo coordinate and resolution information as per the GeoTIFF standard but cannot contain uom information, thus not allowing to reconstruct the complete coverage.. CIS XML and JSON encodings are complete in the sense that they can express all coverage information

Requirement 11
An *encodedCoverageExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
f be a string,
where
f is the name of a data format allowed for *C*,
the data format specified by *f* supports encoding of a coverage of *C*'s domain and range.

Then

for any *byteString S*
where *S* is one of
S_e = 'encode' '(' *C* ',' *f* ')'
S_{ee} = 'encode' '(' *C* ',' *f* ',' *extraParams* ')'
with *f* and *extraParams* being strings enclosed in quotes.

eval(*S*) is defined as that byte string which encodes *C* into the data format specified by *formatName* and the optional *extraParams*. Both *f* and *extraParams* are not defined in this standard.

NOTE In a WCS framework, the data encoding formats supported can be obtained from the *supportedFormats* list contained in the response to a *GetCapabilities* request.

Example The following expression specifies retrieval of coverage C encoded in NetCDF:

```
encode( $C, "application/x-netcdf" )
```

Example A WCPS implementation may allow applying a color ramp to a JPEG image as:

```
encode( $c,
        "image/jpeg",
        '{ "colorMap":
           { "type": "ramp",
             "colorTable":
             { "0":   [0,0,255,255],
               "200": [255,0,0,255]
             }
           }
        }'
)
```

7.1.5 Evaluation result

The response to a WCPS query is a coverage or scalar list, or an error.

Requirement 12
The result of evaluating a *processCoveragesExpr* **shall** consist of one of the following alternatives:

- A (possibly empty) list of encoded coverages.
- A (possibly empty) list of scalars (where scalar summarizes all non-coverage type data, such as numbers, strings, URLs) or records of such scalars.
- An error information.

Requirement 13
A coverage-valued *processCoveragesExpr* response **shall** validate against the OGC Coverage Implementation Schema (CIS) standard [OGC 26-041].

NOTE In practice, a WCPS server may refuse to evaluate some expressions, in particular if the volume of intermediate or result data would exceed some internal limits. A typical example is retrieval of the complete range set.

7.2 coverageConstructorExpr

A *coverageConstructorExpr* creates a *d*-dimensional general grid coverage, for some $d \geq 1$, by defining the coverage's domain set, range type, range set, and metadata

through expressions. This allows creating coverages with entirely new shapes, dimensions, and values.

The coverage domain set is built from a Coordinate Reference System (CRS) defining the multi-dimensional axes and the meaning of coordinates, including units of measure; indicating the coordinates of the direct positions, i.e., the points where values sit.

Axis names can be chosen according to the rules of CIS 1.1; it is recommended to keep Native CRS and grid CRS axis names disjoint.

A range type expression optionally creates the coverage range type. In the scope of the embedding WCPS condensers this expression defines the range component names as known (immutable) variables. Values derived for some such range component will automatically be attempted to cast to the target type of that range component; if not possible, a cast error will be raised.

A range set expression creates the coverage range set. A *rangeSetExpr* is evaluated at every direct position of the coverage's domain set.

An optional metadata expression creates the coverage metadata component. As such metadata are not interpreted by the coverage they are represented as a string which may contain any character, depending on the character set supported (which is out of scope of this standard).

A *coverageExpr* always evaluates to a single coverage.

Requirement 14
A *coverageExpr* **shall** be either a *coverageIdentifier* (Subclause 7.2.1), or a *coverageConstructorExpr* (Subclause 7.2.2), or a *domainOpExpr* (Subclause 7.3.1), or a *rangeExpr* (Subclause 7.3.9).

7.2.1 coverageIdentifier

The *coverageIdentifier* element represents the name of a single coverage offered by the server addressed. It is represented by a coverage variable indicated in the *processCoveragesExpr* clause (Subclause 7.1.2).

Requirement 15
A *coverageIdentifier* **shall** be defined as below.

Let

id be a *variableName* bound to a coverage C_1 offered by the server.

Then

for any *coverageExpr* C_2,
where
C_2 = *id*

eval(C_2) is defined as follows:

Coverage constituent
eval(C_2).*type* = *eval*(C_1).*type*
eval(C_2).*id* = *eval*(C_1).*id*
eval(C_2).*domain* = *eval*(C_1).*domain*
eval(C_2).*rangetype* = *eval*(C_1).*rangetype*
eval(C_2).*range* = *eval*(C_1).*range*
eval(C_2).*metadata* = *eval*(C_1).*metadata*

Example The following coverage expression evaluates to the complete, unchanged coverage to which coverage iteration variable `$c` is bound at the time of evaluation:

```
$c
```

7.2.2 coverageConstructorExpr

Requirement 16
A *coverageConstructorExpr* **shall** be defined as follows.

Let

t be an *identifier* which is the constant string *GeneralGridCoverage*,
id be an *identifier*,
D be a *domainSetExpr*,
T be a *rangeTypeExpr*,
R be a *rangeSetExpr*,
M be a *metadataExpr***.**

Where

C is a *coverageConstructorExpr*
with
C = ('coverage' (*t*)? *id*)? (*D*)? (*T*)? *R* (*M*)?

7.2.3 domainSetExpr

A coverage domain is described through a list of axes defined through a single, common CRS. An axis is of type index, regular, or irregular. Index axes have integer coordinates and therefore no unit of measure and no interpolation, whereas regular and irregular axes must have a unit associated and can have admissible interpolation methods.

Let further

d be an *integer* with $d>0$,
c be a *crsName* representing a *d* -dimensional CRS,

a_i be pairwise distinct *axisName*s for $1 \le i \le d$,
x_i be pairwise distinct *axisName*s for $1 \le i \le d$,
$ce_{i,1}$, $ce_{i,2}$ be *arithmeticExpr*s for $1 \le i \le d$, which are valid coordinates for axis i as per *eval*(c) with *eval*($ce_{i,1}$) ≤ *eval*($ce_{i,2}$),
res_i be *arithmeticExpr*s with *eval*(res_1)<…< *eval*(res_d) for $1 \le i \le d$ valid for the i^{th} axis as per *eval*(c),
$xe_{i,1}$,… be *arithmeticExpr*s for $1 \le i \le d$, which are valid coordinates for axis a_i as per *eval*(c) with *eval*($xe_{i,1}$) < *eval*($xe_{i,2}$) <…,
e be a *getDomainExpr*,
u_i be *uomExpr*s for $1 \le i$,
$imd_{i,1}$, … be *interpolationMethod*s for $1 \le i$.

Where

D is a *domainSetExpr*
with D one of

D_1 = 'domain' ('set')?
('crs' c 'axes')?
a_1 ':' ad_1 ('uom' u_1)? ('interpolation' $imd_{1,1}$, …)?
listSeparator
…
a_d ':' ad_d ('uom' u_d)? ('interpolation' $imd_{d,1}$, …)?

D_2 = e

and ad_i one of

$ad_{i,index}$ = 'index' '(' $ie_{i,1}$ ':' $ie_{i,2}$ ')'
$ad_{i,regular}$ = 'regular' '(' $ce_{i,1}$ ':' $ce_{i,2}$ ')'
resolution res_i
$ad_{i,irregular}$ = 'irregular' '(' $xe_{i,1}$ ',' … ',' $xe_{i,n}$ ')'

with
no uom clause with u_i is present for ad_i=$ad_{i,index}$,
c defines a CRS with all axes a_i in their sequence listed.

Then

eval(D) is a domain set as follows:

Coverage constituent
eval(D_1).*crs* = *eval*(c).*crs* if present, *Index-d-D* otherwise[6] *eval*(D_2).*crs* = *eval*(e).*crs*
eval(D_1).*axisLabels* = <*eval*(a_1) ,…, *eval*(a_d)> *eval*(D_2).*axisLabels* = *eval*(e).*axisLabels*
eval(D_1).*uomLabels* = <*eval*(u_1) ,…, *eval*(u_d)> with *eval*(u_i)=*null* for u_i omitted *eval*(D_2).*uomLabels* = *eval*(e).*uomLabels*
for $a \in$ *eval*(D).*axisLabels* :

[6] "Index1D", "Index2D", …

eval(D_1).*axis*[a].*axisLabel* = *eval*(a) *eval*(D_2).*axis*[a].*axisLabel* = *eval*(e).*axis*[a].*axisLabel*
for *eval*(D).*axisLabels* = <a_1,…,a_d>: *eval*(D_1). *axis*[a_i]. *hi* = *eval*($ie_{i,2}$) if ad_i is a $ad_{i,index}$, else *eval*($ce_{i,2}$) if ad_i is a $ad_{i,regular}$, else *eval*($xe_{i,n}$) if ad_i is a $ad_{i,irregular}$, *eval*(D_2).*axis*[a_i].*hi* = *eval*(e).*axis*[a_i].*hi*
for *eval*(D).*axisLabels* = <a_1,…,a_d>: *eval*(D_1).*axis*[a_i].*lo* = *eval*($ie_{i,1}$) if ad_i is a $ad_{i,index}$, else *eval*($ce_{i,1}$) if ad_i is a $ad_{i,regular}$, else *eval*($xe_{i,1}$) if ad_i is a $ad_{i,irregular}$, *eval*(D_2).*axis*[a_i].*lo* = *eval*(e).*axis*[a_i].*lo*
for *eval*(D).*axisLabels* = <a_1,…,a_d>: if ad_i is a $ad_{i,regular}$: *eval*(D_1).*axis*[a_i].*res* = *eval*(res_i) *eval*(D_2).*axis*[a_i].*res* = *eval*(e).*axis*[a_i].*res*
for *eval*(D).*axisLabels* = <a_1,…,a_d>: if ad_i is a $ad_{i,regular}$ or $ad_{i,irregular}$: *eval*(D_1).*axis*[a_i].*uom* = *eval*(u_i) *eval*(D_2).*axis*[a_i].*uom* = *eval*(e.*axis*[a_i].*uom*)
for *eval*(D).*axisLabels* = <a_1,…,a_d>: if ad_i is a $ad_{i,irregular}$: *eval*(D_1).*axis*[a_i].*pos*[*j*] = *eval*($xe_{i,j}$) for 1≤ *j*≤*n* *eval*(D_2).*axis*[a_i].*pos*[*j*] = *eval*(e).*axis*[a_i].*pos* [*j*] for 1≤ *j*≤*n*
for *eval*(D).*axisLabels* = <a_1,…,a_d>: *eval*(D_1).*axis*[a_i].*index.lo* = *eval*(D_1).*axis*[a_i].*lo* if ad_i is a $axisdef_{i,index}$ else 0 if ad_i is a $ad_{i,regular}$ else 0 if ad_i is a $ad_{i,irregular}$ *eval*(D_2).*axis*[a_i].*index.lo*) = *eval*(e).*axis*[a_i].*index.lo*
for *eval*(C).*domain.axisLabels* = <a_1,…,a_d>: *eval*(D_1).*axis*[a_i].*index.hi* = *eval*(D).*axis*[a_i].*hi* if ad_i is a $ad_{i,index}$ else *ceil*(*eval*(D).*axis*[a_i].*hi* - *eval*(D).*axis*[a_i].*lo*) / *abs*(*eval*(D).*axis*[a_i].*res*)) +1 if ad_i is a $ad_{i,regular}$ else | *eval*(D).*axis*[a_i].*pos* | -1 if ad_i is a $ad_{i,irregular}$: *eval*(D_2).*axis*[a_i].*index.hi* = *eval*(e).*axis*[a_i].*index.hi*
for *eval*(D).*axisLabels* = <a_1,…,a_d>: *eval*(D_1).*axis*[a_i].*int* = {*eval*($imd_{i,1}$), …} if an interpolation clause is present, ∅ otherwise *eval*(D_2).*axis*[a_i].*int* = *eval*(e).*axis*[a_i].*int*

7.2.4 rangeTypeExpr

Let further

n be an *integer* with n>0,
f_1,…, f_n be *fieldName*s,
t_1,…, t_n be *rangeType*s,
n_1,…, n_n be *valueNature*s,
u_1,…, u_n be *stringConstant*s,
$nil_{1,1}$,… and $nil_{n,1}$,… be *constant*s compatible with type q_i,
$imr_{1,1}$,… and $imr_{n,1}$,… be *interpolationMethod*s.

Then

for any *rangeTypeExpr T*
where

```
T = 'range' 'type'
      f1 : ( n1 )? t1 ( 'uom' u1 )? ( 'nil' nil1,1, … )?
           ( 'interpolation' imr1,1, … )? ','
      …
      fn : ( n1 )? tn ( 'uom' un )? ( 'nil' nil1,1, … )?
           ( 'interpolation' imrn,1, … )?
```

with
uom clause u_i present only if n_i=`quantity` or absent.

eval(*T*) is a range type defined as follows:

Coverage constituent
eval(*T*).*fieldLabels* = < *eval*(f_1),…, *eval*(f_n) >
for *f*∈*eval*(*T*).*fieldLabels*: *eval*(*T*).*fields*[*f*] = *eval*(f_i)
for *f*∈*eval*(*T*.*fieldLabels* *eval*(*T*).*fields*[*f*].*def* = *eval*(t_i)
for *f*∈*eval*(*T*).*fieldLabels*: *eval*(*T*).*fields*[*f*].*nil* = { *eval*($nil_{i,1}$), … } if a *nil* clause is present, ∅ otherwise
for *f*∈*eval*(*T*).*fieldLabels*: *eval*(*T*).*fields*[*f*].*uom* = *eval*(u_i) if a *uom* clause is present, *null* otherwise
for *f*∈*eval*(*T*).*fieldLabels*: *eval*(*T*).*fields*[*f*].*nature* = *eval*(n_i) if a *nature* clause is present, *quantity* otherwise
for *f*∈*eval*(*T*).*fieldLabels*: *eval*(*T*).*fields*[*f*].*int* =

{ *eval*($imr_{1,1}$), … } if a *interpolation* clause is present, ∅ otherwise

NOTE DataRecords in SWE Common, on which the CIS range type relies, define several more components which, however, have turned out to be unused in WCPS practice. Hence, there is a restriction to the practically relevant parts leaving other parts implementation dependent. A future version of this standard may add definitions for further SWE DataRecord components.

7.2.5 rangeSetExpr

Let further

r be a *rangeExpr* possibly containing axis iterators a_i defined in D and range field identifiers f_j defined in T.

Where

R is a *rangeSetExpr*
with
R = 'range' ('set')? r

Then

eval(R) is a coverage C defined as follows:

Coverage constituent
eval(C).*type* = *GeneralGridCoverage*
eval(C).*id* = *eval*(r).*id*
eval(C).*domain* = *eval*(r).*domain*
eval(C).*rangetype* = *eval*(r).*rangetype*
eval(C).*range* = *eval*(r).*range*
eval(C).*metadata* = *eval*(r).*metadata*

7.2.6 Metadata

Let further

m be a *stringExpr*

Where

M is a *metadataExpr*
with
M = **metadata** m

NOTE If the metadata value resembles valid XML or JSON then an implementation may support that XPath can address into these data.

7.2.7 Coverage constructor semantics

For every coverage constituent (except range values), first the value indicated is used, if any; if there is none, inherited values from input coverage(s) are used; if not available, defaults are used; if there is none, an exception is returned.

Then

`C` is defined as a coverage of type `t` as follows:

Coverage constituent
eval(`C`).*type* = *GeneralGridCoverage*

if `id` is present:
  eval(`C`).*id* = `id`
else
  null

if `D` is present:
  eval(`C`).*domain* = *eval*(`D`)
else
  eval(`C`).*domain* = *eval*(`r`).*domain*

if `T` is present:
  eval(`C`).*rangetype* = *eval*(`T`)
else
  eval(`C`).*rangetype* = *eval*(`r`).*rangetype*

eval(`C`).*range* is the range set resulting from evaluating *eval*(`r`)

if `M` is present:
  eval(`C`).*metadata* = *eval*(`m`)
else
  eval(`C`).*metadata* = *eval*(`r`).*metadata*

Requirement 17
In range field definitions, if the value nature is *count* or *category* then the interpolation method list **shall** be empty or contain only *nearest*.

7.2.8 Range value enumeration

A coverage can be defined ad-hoc by enumerating its range values. In this variant of the coverage constructor the domain and range type clauses are mandatory.

Requirement 18
A `coverageEnumExpr` **shall** be defined as follows.

Let

`t` be an `identifier` which is the constant string `GeneralGridCoverage`,
`id` be an `identifier`,

`D` be a `domainSetExpr`,
`T` be a `rangeTypeExpr`,
`E` be a `rangeEnumExpr` possibly containing occurrences of $axis_i$ iterators defined in `D` and range component identifiers $field_j$ defined in `T`,
`M` be a `metadataExpr`,

`n` be an `integer` with `n` = | `D`.*domain.pos* |,
$v_1, \ldots, v_n$ be `scalarExpr`s pairwise coercion-compatible (Requirement 79).

Where

`C` is a `coverageEnumExpr`
with
`C = 'coverage' ( t )? id D T E ( M )?`
and
`E = 'range' ( 'set' )? '<' v1 ',' … ',' vn '>'`

Then

eval(`C`) is defined as follows:

Coverage constituent
eval(`C`).*type* = *GeneralGridCoverage*
eval(`C`).*id* = `id`
eval(`C`).*domain* = *eval*(`D`)
eval(`C`).*rangetype* = *eval*(`T`) if `T` is present, and determined from `E` otherwise
if `T` is not present*:* *eval*(C_2).*rangetype.fieldLabels* = <`f`> where `f` is an implementation-defined `fieldName`
if `T` is not present: *eval*(C_{plus}).*rangetype.fields*[f_i].*def* = the largest data type encompassing all values in <*eval*(v_1),…,*eval*(v_n)>
if `T` is not present: *eval*(C_2).*rangetype.fields*[f_i].*nil* = ∅
if `T` is not present: *eval*(C_2).*rangetype.fields*[f_i].*uom* = *null*
if `T` is not present: *eval*(C_2).*rangetype*.fields[f_i].*nature* = *eval*(C_1).*rangetype*.fields[f_i].*nature*
for *p*∈*eval*(`C`).*domain.pos*: *eval*(`C`).*range*[*p*] = < *eval*(v_1), …, *eval*(v_n) > where the v_i get assigned to *range*[*p*] from low to high index, first to last axis ("row-major").
eval(`C`).*metadata* = *eval*(`M`)

NOTE Only single-component range fields are supported.

Example The following 1-D coverage contains the first 13 Fibonacci numbers.

```
coverage Fibonacci
domain
  number: index ( 1:13 )
range type
  fib: unsigned long
range
  < 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 >
```

7.2.9 decodeCoverageExpr

A *decodeCoverageExpr* evaluates a byte stream passed as parameter to a coverage by decoding the byte stream. Encoding formats supported, and their flavors, are implementation-defined.

NOTE Implementations will be able to recognize the encoding format used from analyzing the input byte stream, hence no format indication parameter is required.

Requirement 19
Syntax and semantics of a *decodeCoverageExpr* **shall** be given as follows.

Let

> *d* be a *variableName*,
> *extraParams* is a *stringConstant* containing (format- and implementation-defined) decoding directives.

Then

> for any *decodeCoverageExpr C*
> where
> *C* = 'decode' '(' *d* (',' *extraParams*)? ')'
>
> *eval*(*C*) is defined as the decoded coverage equivalent to the contents of *d* while applying the directives of the *extraParams*.

This function can be used to provide input coverages accompanying the query.

Example Assume a NetCDF file is passed as positional parameter `$1` in the WCS Processing Extension [7] . This decodes the byte stream and establishes the corresponding coverage:

```
decode( $1 )
```

7.3 domainOpExpr

Domain expressions modify the domain set, but not the range values and range type, except for scaling and CRS transform.

7.3.1 Overview

Requirement 20
A *domainOpExpr* **shall** be either a *subsetExpr* (Subclause 7.3.3) or a *scaleExpr* (Subclause 7.3.7) or a *crsTransformExpr* (Subclause 7.3.8) or a *polygonExpr* (Subclause 7.3.9).

7.3.2 subsetExpr

The *subsetExpr* element specifies spatial and temporal domain subsetting. It encompasses spatial and temporal trimming (i.e., constraining the result coverage domain to a subinterval), slicing (i.e., cutting out a hyperplane from a coverage), extending, and scaling of a coverage expression.

Requirement 21
A *subsetExpr* **shall** be either a *trimExpr* (Subclause 7.3.4), or a *sliceExpr* (Subclause 7.3.4), or an *extendExpr* (Subclause 7.3.5).

All of the *subsetExpr* elements allow to make use of coordinate reference systems other than a coverage's image CRS. A coverage's individual mapping from some supported CRS coordinates to its Index CRS coordinates does not need to be disclosed by the server, hence coordinate transformation **should** be considered a "black box" by the client.

NOTE 1 The special case that subsetting leads to a single point remaining still resembles a coverage by definition; this coverage is viewed as being of dimension 0.

NOTE 2 Range subsetting is accomplished via the unary induced *fieldExpr* (Subclause 7.4.8).

7.3.3 trimExpr

The *trimExpr* element extracts a subset from a given coverage expression along the dimension indicated, specified by a lower and upper bound for each dimension affected. Interval limits can be expressed in the coverage CRS or any other CRS explicitly indicated, as long as a transformation to the coverage CRS exists.

The new, reduced extent limits can be given as an interval or cloned from some other coverage domain. In the latter case, only the extent limits are adopted from the domain, no other components (axis type, etc.).

Requirement 22
In a *trimExpr* for each axis the lower and upper trim limits **shall** lie inside the coverage's domain.

Requirement 23
A *trimExpr* **shall** be defined as below.

Let

C_1 be a *coverageExpr*,
n be an *integer* with $n>0$,
$axis_i$ be pairwise distinct *axisName*s for $1 \le i \le n \le d$,
$(lo_1:hi_1),\ldots,(lo_n:hi_n)$ be *dimensionIntervalExpr*s with $lo_i \le hi_i$ for $1 \le i \le n$,
d is a *getDimensionExpr*,
$d_1,\ldots, d_n$ are *getAxisExpr*s.

Where

$axis_i \in eval(C_1).domain.axisLabels$ for $1 \le i \le n$.

Then

for any *coverageExpr* C_2
where C_2 is one of
$C_{2,1} = C_1$ '[' p_1 ',' … ',' p_n ']'
$C_{2,2} = C_1$ '[' d ']'
with p_i one of
$p_{i,1} = axis_i$ '(' lo_i ':' hi_i ')'
$p_{i,2} = axis_i$ '(' d_i ')'
$p_{i,3} = axis_i$ '.' 'index' '(' lo_i ':' hi_i ')'
$p_{i,4} = axis_i$ '.' 'index' '(' d_i ')'
and
each $[lo_i:hi_i]$ interval is completely within the C_1 bounds,
the extent of d is completely within the C_1 bounds,
every d_i has an axis label matching with some axis label of C_1,
every d_i extent is within the limits of the corresponding axis of C_1.
d and all d_i are compatible with the axis addressed in the coverage in axis type, resolution, uom.

eval(C_2) is defined as follows:

Coverage constituent
eval(C_2).*type* = *eval*(C_1).*type*
eval(C_2).*id* = *eval*(C_1).*id*
eval(C_2).*domain.crs* = *eval*(C_1).*domain.crs*
eval(C_2).*domain.axisLabels* = *eval*(C_1).*domain.axisLabels*
eval(C_2).*domain.uomLabels* = *eval*(C_1).*domain.uomLabels*
for $a \in eval(C_2).domain.axisLabels$: *eval*($C_2$).*domain.axes*[$a$].*axisLabel* = *eval*(a)
for $a \in eval(C_2).domain.axisLabels$: if $a = axis_i$ for some $1 \le i \le n$: *eval*($C_{2,1}$).*domain.axes*[a].*hi* = hi_i for $p_{i,1}$ *eval*($C_{2,1}$).*domain.axes*[a].*hi* = *eval*(d_i).*hi* for $p_{i,2}$ *eval*($C_{2,1}$).*domain.axes*[a].*hi* = hi_i for $p_{i,3}$ *eval*($C_{2,1}$).*domain.axes*[a].*hi* = *eval*(d_i).*hi* for $p_{i,4}$

else *eval*($C_{2,1}$).*domain.axes*[a].*hi* = *eval*(C_1).*domain.axes*[a].*hi*

for $a \in$ *eval*(C_2).*domain.axisLabels*:
if $a = axis_i$ for some $1 \le i \le n$:
eval($C_{2,1}$).*domain.axes*[a].*lo* = lo_i for $p_{i,1}$
eval($C_{2,1}$).*domain.axes*[a].*lo* = *eval*(d_i).*lo* for $p_{i,2}$
eval($C_{2,1}$).*domain.axes*[a].*lo* = lo_i for $p_{i,3}$
eval($C_{2,1}$).*domain.axes*[a].*lo* = *eval*(d_i).*lo* for $p_{i,4}$
else
eval($C_{2,1}$).*domain.axes*[a].*lo* = *eval*(C_1).*domain.axes*[a].*lo*

for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*res* = *eval*(C_1).*domain.axis*[a].*res* if a_i is regular
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*uom* = *eval*(C_1).*domain.axis*[a].*uom*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.hi* = *eval*(C_1).*axis*[a].*index.hi*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.lo* = *eval*(C_1).*domain.axis*[a].*index.lo*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*int* = *eval*(C_1).*domain.axis*[a].*int*
for $a \in$ *eval*(C_2).*axisLabels*: *eval*(C_2).*domain.index*[a].*lo* = *eval*(C_2).*domain.axis*[a].*lo* if *eval*(C_2).*domain.axis*[a].*axisType* = *index*, 0 otherwise
for $a \in$ *eval*(C_2).*axisLabels*: *eval*(C_2).*domain.index*[a].*hi* = | *eval*(C_2).*domain.axis*[a] | - *eval*(C_2).*domain.index*[a].*lo* -1
eval(C_2).*rangetype* = *eval*(C_1).*rangetype*
for $p \in$ *eval*(C_2).*domain.pos*: *eval*(C_2).*range*[p] = *eval*(C_1).*range*[p]
eval(C_2).*metadata* = *eval*(C_1).*metadata*

Example The following is a syntactically valid trim expression:

```
$C[ Long (-120: -80), Lat (-10: +10) ]
```

7.3.4 sliceExpr

The *sliceExpr* element extracts a spatial slice (i.e., a hyperplane) from a given coverage expression along one of its dimensions, specified by one or more slicing dimensions and a slicing position thereon. For each slicing dimension indicated, the resulting coverage has a dimension reduced by 1; its dimensions are the dimensions of the original coverage, in the same sequence, with the section dimension being removed from the list. CRSs / axes not used by any of the remaining dimensions are removed from the coverage's CRS set.

The slice point can sit on a direct position or between two direct positions on a coverage axis. In the former case, range values are used unchanged, in the latter case interpolation is applied. First, the case of matching a direct position is addressed.

Requirement 24
In a *sliceExpr* the slicing coordinates **shall** lie inside the coverage's domain.

For syntactic convenience, both array-style addressing using brackets and function-style syntax are provided; both are equivalent in semantics.

Requirement 25
A *sliceExpr* slicing at direct positions only **shall** be defined as below.

Let

C_1 be a *coverageExpr*,
n be an *integer* with $0 \leq n$,
$axis_1, \ldots, axis_n$ be pairwise distinct *axisName*s,
$s_1, \ldots, s_n$ be *arithmeticExpr*s resolving to some direct position in C_1.

Where

$axis_i \in eval(C_1).domain.axisLabels$ for $1 \leq i \leq n$.

Then

for any *coverageExpr* C_2
where
$C_2 = C_1$ '[' S_1 ',' … ',' S_n ']'
with S_i one of
$S_{i,1} = axis_i$ '(' s_i ')'
$S_{i,2} = axis_i$ '.' 'index' '(' s_i ')'

$eval(C_2)$ is defined as follows:

Coverage constituent
$eval(C_2).type = eval(C_1.type$
$eval(C_2).id = eval(C_1.id$
$eval(C_2).domain.crs = crs$
$eval(C_2).domain.axisLabels = eval(C_1).domain.axisLabels$ / { $axis_1, \ldots, axis_n$ }
$eval(C_2).domain.uomLabels =$ $< eval(C_1).domain.axis[axis_1].uom, \ldots, eval(C_1).domain.axis[axis_n].uom>$
for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].axisLabel = eval(a)$
for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].hi = eval(C_1).domain.axis[a].hi$

for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].lo = eval(C_1).domain.axis[a].lo$
for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].res = eval(C_1).domain.axis[a].res$ if a is regular
for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].uom = eval(C_1).domain.axis[a].uom$
for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].index.lo = eval(C_1).axis[a].index.lo$
for $a \in eval(C).domain.axisLabels$: $eval(C_2).domain.axis[a].index.hi = eval(C_1).axis[a].index.hi$
for $a \in eval(C_2).domain.axisLabels$: $eval(C_2).domain.axis[a].int = eval(C_1).domain.axis[a].int$
for $a \in eval(C_2).axisLabels$: $eval(C_2).domain.index[a].lo =$ $eval(C_2).domain.axis[a].lo$ if $eval(C_2).domain.axis[a].axisType = index$, 0 otherwise
for $a \in eval(C_2).axisLabels$: $eval(C_2).domain.index[a].hi =$ $\lvert eval(C_2).domain.axis[a] \rvert - eval(C_2).domain.index[a].lo - 1$
$eval(C_2).rangetype = eval(C_1.rangetype$
$C' = C_1[A_1(s_1:s_1),\ldots,A_n(s_n:s_n)]$ where A_i is one of $axis_i$ and $axis_i.\texttt{index}$; for all $p = \langle p_1,\ldots,p_{d-n} \rangle \in eval(C_2).domain.pos$ where p is obtained from $q = \langle q_1,\ldots,q_d \rangle \in eval(C_1).domain.pos$ by removing the coordinate components of axes $axis_i$: $eval(C_2).range[p] = eval(C').range[q]$
$eval(C_2).metadata = eval(C_1).metadata$

Example The following is a syntactically valid slice expression:

```
$c[ Lat ( 120 ) ]
```

Next, the case of a slice point coordinate falling between two direct positions is addressed.

Requirement 26
A *sliceExpr* containing at least one slice point between direct positions **shall** be defined as below.

Let

C_1 be a *coverageExpr*,
n be an *integer* with $1 \le n \le \lvert eval(C_1).domain.axisLabels \rvert = d$,
m be an *integer* with $m = \lvert eval(C_1).rangetype.fieldLabels \rvert$,
$axis_1,\ldots,axis_n$ be pairwise distinct *axisName*s,
$s_1,\ldots,s_n$ be *arithmeticExpr*s.

Where

$axis_1,\ldots,axis_n$ are not of type *index*,
$\{axis_1,\ldots,axis_n\} \subseteq eval(C_1).domain.axisLabels$ for $1 \le i \le d$.

Then

for any *coverageExpr* C_2
where
$C_2 = C_1$ '[' S_1 ',' … ',' S_n ']'
with
$S_i = axis_i$ '(' s_i ')'

eval(C_2) is defined as follows:

Coverage constituent
eval(C_2).*type* = *eval*(C_1.*type*
eval(C_2).*id* = *eval*(C_1.*id*
eval(C_2).*domain.crs* = the CRS obtained from *eval*(C_1).*domain.crs* by removing axes $axis_1,\ldots,axis_n$ if such a CRS exists, the *n*-D Index CRS `Index-n-D` otherwise
eval(C_2).*domain.axisLabels* = *eval*(C_1).*domain.axisLabels* / { $axis_1,\ldots,axis_n$ }
eval(C_2).*domain.uomLabels* = < *eval*(C_1).*domain.axis*[a_1].*uom*,…, *eval*(C_1).*domain.axis*[a_n].*uom*>
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*axisLabel* = *eval*(a)
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*hi* = *eval*(C_1).*domain.axis*[a].*hi*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*lo* = *eval*(C_1).*domain.axis*[a].*lo*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*res* = *eval*(C_1).*domain.axis*[a].*res* if a is regular
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*uom* = *eval*(C_1).*domain.axis*[a].*uom*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.lo* = *eval*(C_1).*axis*[a].*index.lo*
for $a \in$ *eval*(C).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.hi* = *eval*(C_1).*axis*[a].*index.hi*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*int* = *eval*(C_1).*domain.axis*[a].*int*
eval(C_2).*rangetype* = *eval*(C_1.*rangetype*
for *eval*(C_1).*domain.axisLabels*=<$x_1,\ldots,x_d$> and *eval*(C_1).*rangetype.fieldLabels*=<$f_1,\ldots,f_m$> with $1 \le i \le d$ and $1 \le j \le m$:

interpolationMethod $e_{i,j}$ is *nearest* if $x_i \notin \{a_1,\ldots,a_n\}$, else some implementation-defined selection from $E_{i,j}=effint(C_1, x_i, f_j)$ if $E_{i,j}\neq\emptyset$, else error; for $<f_1,\ldots,f_m>=eval(C_1).rangetype.fieldLabels$, for $p=<p_1,\ldots,p_{d-n}> \in eval(C_2).domain.pos$ where p is obtained from $q=<q_1,\ldots,q_d> \in eval(C_1).domain.pos$ by removing the coordinate components of axes $axis_i$ from the q coordinate tuples: $eval(C_2).range[p]$ is obtained by applying interpolation method $e_{i,j}$ on field f_i along axis $axis_i$ using some neighbourhood around position q in $eval(C_1)$.
$eval(C_2).metadata = eval(C_1).metadata$

NOTE Applying interpolation method nearest on values associated with points sitting at direct positions is equivalent to taking the original, unmodified value.

7.3.5 extendExpr

The *extendExpr* element extends a coverage to new lower and upper bounds along the axes selected. The new direct positions are filled with one of the coverage's nil values or, if none available, with 0.

In case of index and regular axes, new direct positions are generated up to the new limit with the appropriate resolution; in case of irregular axes, only one new step is generated at the new limit position.

Requirement 27
In an *extendExpr* if the coverage's nil value set is empty but whenever a nil value is needed then the server **shall** use a value of 0.

There is no restriction on the position and size of the new bounding box; in particular, it does not need to lie outside the coverage; it may intersect with the coverage; it may lie completely inside the coverage; it may not intersect the coverage at all (in which case a coverage completely filled with null values will be generated).

NOTE In this sense the *extendExpr* is a generalization of the *trimExpr*; still the *trimExpr* should be used whenever the application needs to be sure that a proper subsetting has to take place.

Requirement 28
An *extendExpr* **shall** be defined as below.

Let

C_1 be a *coverageExpr*,
n be an integer with $1 \leq n$,
$E_1,\ldots,E_n$ be *trimExpr*s evaluating to pairwise distinct *axisName*s $x_1,\ldots,x_n$ and pairs of integers lo_i, hi_i with $lo_i \leq hi_i$ for $1 \leq i \leq n$.

Where

$\{x_1,\ldots,x_n\} \subseteq$ *eval*(C_1).*domain.axisLabels* for $1 \leq i \leq d$.

Then

for any *coverageExpr* C_2
where
C_2 = 'extend' '(' C_1 ',' '[' E_1 ',' … ',' E_n ']' ')'

eval(C_2) is defined as follows:

Coverage constituent
eval(C_2).*type* = *eval*(C_1).*type*
eval(C_2).*id* = *eval*(C_1).*id*
eval(C_2).*domain.crs* = *eval*(C_1).*domain.crs*
eval(C_2).*domain.axisLabels* = *eval*(C_1).*domain.axisLabels*
eval(C_2).*domain.uomLabels* = *eval*(C_1).*domain.uomLabels*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*axisLabel* = a
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*hi* = hi_i if $a=x_i$ for some $i \leq n$, *eval*(C_1).*domain.axis*[a].*hi* otherwise
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*lo* = lo_i if $a=x_i$ for some $i \leq n$, *eval*(C_1).*domain.axis*[a].*lo* otherwise
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*res* = *eval*(C_1).*domain.axis*[a].*res* if a is regular
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*uom* = *eval*(C_1).*domain.axis*[a].*uom*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.lo* = *eval*(C_1).*domain.axis*[a].*index.lo* for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.lo* = *eval*(C_1).*domain.axis*[a].*index.lo* if $a=x_i$ for some $i \leq n$ and regular or irregular axis a, lo_i if $a=x_i$ for some $i \leq n$ and index axis a, *eval*(C_1).*domain.axis*[a].*index.lo* otherwise
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*index.hi* = *eval*(C_1).*domain.axis*[a].*index.hi* +1 for irregular axis a, (*eval*(C_2).*domain.axis*[a].*hi* - *eval*(C_2).*domain.axis*[a].*lo* + 1) / *eval*(C_2).*domain.axis*[a].*res*

+ *eval*(C_2).*domain.axis*[a].*lo* for regular axis a, hi_i if $a=x_i$ for some $i \le n$ and index axis a, *eval*(C_1).*domain.axis*[a].*index.hi* otherwise
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*int* = *eval*(C_1).*domain.axis*[a].*int*
for $a \in$ *eval*(C_2).*domain.axisLabels*: *eval*(C_2).*domain.axis*[a].*int* = *eval*(C_1).*domain.axis*[a].*int*
for $a \in$ *eval*(C_2).*axisLabels*: *eval*(C_2).*domain.index*[a].*lo* = *eval*(C_2).*domain.axis*[a].*lo* if *eval*(C_2).*domain.axis*[a].*axisType* = *index*, 0 otherwise
for $a \in$ *eval*(C_2).*axisLabels*: *eval*(C_2).*domain.index*[a].*hi* = | *eval*(C_2).*domain.axis*[a] | - *eval*(C_2).*domain.index*[a].*lo* -1
eval(C_2).*rangetype* = *eval*(C_1).*rangetype*
for $p \in$ *eval*(C_2).*domain.pos*, for $f \in$ *eval*(C_2).*domain.fieldLabels*: *eval*(C_2).*range*[p].f = *eval*(C_1).*range*[p].f if $p \in$ *eval*(C_1).*domain.pos*, some implementation-defined nil value $n \in$ *eval*(C_1).*rangetype*[f].*nil* if *eval*(C_1).*rangetype*[f].*nil* $\neq \varnothing$, 0 otherwise
eval(C_2).*metadata* = *eval*(C_1).*metadata*

Example The following is a syntactically valid *extend*() expression:

```
extend( $c, [ Lat(-100:+100), i(-10:+10) ] )
```

7.3.6 subsetExpr

Requirement 29
A *subsetExpr* **shall** be either a *scaleExpr* (Subclause Requirement 29) or a *crsTransformExpr* (Subclause 7.3.8).

7.3.7 scaleExpr

A *scaleExpr* reduces resolution while leaving the geographic extent unchanged; hence, it is applicable only to regular axes. There are three different variants:

- Scaling to a target extent (i.e., number of grid points along each axis selected)
- Scaling by factors applied individually to axes selected
- Scaling by a factor applied to all axes.

Interpolation methods to be applied during evaluation of the expression are matched against the allowed interpolation methods stored in the target coverage, per domain axis and range field. The intersection of all methods available is effectively applied; if the intersection is empty, an error is returned.

Scaling supports selection of only interpolation method. If different interpolation methods are to be applied to different range fields within the same coverage, the recommended approach is to split them, scale them individually, and then rejoin.

Example This operation scales using different interpolation in the image and mask fields:

```
struct
{ image: scale( $C.image, $factor, linear ),
  mask:  scale( $C.mask,  $factor, nearest )
}
```

Requirement 30
A *scaleExpr* **shall** be a *scaleToExtentExpr* or a *scaleByFactorsExpr* or a *scaleByCommonFactorExpr*.

A *scaleToExtentExpr* expression indicates a target index extent along selected axes by specifying the number and distribution of the new direct positions, thus impacting the index upper bound while leaving the index lower bound as well as the non-index CRS lower and upper bounds unchanged.

NOTE A scale factor > 1 results in a denser grid (i.e., increasing resolution or "zoom in") whereas a facor < 1 reduces resolution (less grid points, "zoom out"). This behavior has been adopted to remain consistent with WCS-Scaling where OGC has decided to adopt this behavior.

Requirement 31
A *scaleToExtentExpr* **shall** be defined as below.

Let

C be a coverageExpr,
n be an integer with $1 \leq n$,
$E_1, \ldots, E_n$ be *trimExpr*s evaluating to pairwise distinct *axisName*s $x_1, \ldots, x_n$ and pairs of integers lo_i, hi_i with $lo_i \leq hi_i$ for $1 \leq i \leq n$,
im be an *interpolationMethod*.

Where

eval(C).*domain.axes*[a].*axisType=regular* for all $a \in$ *eval*(C).*domain.axisLabels*,
$\{x_1, \ldots, x_n\} \subseteq$ *eval*(C_1).*domain.axisLabels*,
$im \in$ *effint*(C,a,f)
for all $a \in$ *eval*(C).*domain.axisLabels*, $f \in$ *eval*(C).*rangetype.fieldLabels*.

Then

For any *scaleToExtentExpr* S
where S is one of

S_1 = 'scale' '(' C ',' '[' E_1 ',' … ',' E_m ']' ')'
S_2 = 'scale' '(' C ',' '[' E_1 ',' … ',' E_m ']' ',' im ')'

eval(S) is defined as follows:

Coverage constituent
eval(S).*type* = *eval*(C).*type*
eval(S).*id* = *eval*(C).*id*
eval(S).*domain.crs* = *eval*(C).*domain.crs*
eval(S).*domain.axisLabels* = *eval*(C).*domain.axisLabels*
eval(S).*domain.uomLabels* = *eval*(C).*domain.uomLabels*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axes*[a].*axisLabel* = a
for $a \in$ *eval*(S)*domain.axisLabels*: *eval*(S).*domain.axis*[a].*hi* = *eval*(C).*domain.axis*[a].*hi*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*lo* = *eval*(C).*domain.axis*[a].*lo*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*uom* = *eval*(C).*domain.axis*[a].*uom*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*res* = (*eval*(hi_i) - *eval*(hi_i) + 1) if $a=x_i$ and regular for some $i \le n$ *eval*(C).*domain.axis*[a].*res* if a is regular
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*index.hi* = hi_i if $a=x_i$ for some $i \le n$ *eval*(C).*domain.axis*[a].*index.hi* otherwise
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*index.lo* = lo_i if $a=x_i$ for some $i \le n$ *eval*(C).*domain.axis*[a].*index.lo* otherwise
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*int* = *eval*(C).*domain.axis*[a].*int*
eval(S).*rangetype* = *eval*(C).*rangetype*
Interpolation method IM = an implementation-defined choice with $IM \in$ *effint*(C_1,a,f) for $a \in$ *eval*(C).*domain.axisLabels*, $f \in$ *eval*(C).*rangetype.fieldLabels* if $S = S_1$, im if $S = S_2$; for $p \in$ *eval*(S).*domain.pos*: *eval*(S).*range*[p] is obtained by interpolating the range values of *eval*(C) at the direct positions of *eval*(S).*domain*, applying interpolation IM on all axes and

fields.
for *a*∈*C*2.*domain.axisLabels*: *C*2.*domain.axis*[*a*].*uom* = *C*1.*domain.axis*[*a*].*uom*
for *a*∈*C*2.*domain.axisLabels*: *C*2.*domain.axis*[*a*].*int* = *C*1.*domain.axis*[*a*].*int*
*C*2.*rangetype* = *C*1.*rangetype*
Interpolation method *IM* = an implementation-defined choice with *IM*∈*effint*(*C*1,*a*,*f*) for all *a*∈*eval*(*C*).*domain.axisLabels*, *f*∈*eval*(*C*).*rangetype.fieldLabels* if *S* = *S*1, *im* if *S* = *S*2; for *p*∈*eval*(*S*).*domain.pos*: *eval*(*S*).*range*[*p*] is obtained by interpolating the range values of *eval*(*C*) at the direct positions of *eval*(*S*).*domain*, applying interpolation *IM* on all axes x_i and all fields.
*C*2.*metadata* = *C*1.*metadata*

Example The following expression performs x/y scaling of some coverage referenced by variable *$C* using interpolation type `cubic` in both `x` and `y` dimension. Note that *$C* might have other axes, such as time, which would remain unaffected.

```
scale($C,[Lat.index(100:200),Lon.index(300:400)],cubic)
```

A receives a scale vector, syntactically similar to a slicing, indicating an individual scale factor for selected axes, but across all range fields.

Requirement 32
A *scaleByFactorsExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
n be an *integer* with 1≤*n*,
$E_1,\ldots,E_n$ be *sliceExpr*s evaluating to pairwise distinct *axisName*s $x_1,\ldots,x_n$ and *arithmeticExpr*s $s_1,\ldots,s_n$,
im be an *interpolationMethod*.

Where

eval(*C*).*domain.axes*[*a*].*axisType=regular* for all *a*∈*eval*(*C*).*domain.axisLabels*,
$\{x_1,\ldots,x_n\}\subseteq$ *eval*(*C*).*domain.axisLabels*,
im∈*effint*(*C*,*a*,*f*)
for all *a*∈*eval*(*C*).*domain.axisLabels*, *f*∈*eval*(*C*).*rangetype.fieldLabels*.

Then

For any *scaleByFactorsExpr* S,
where S is one of

S_1 = 'scale' '(' C ',' '[' x_1 '(' s_1 ')' ',' … ',' x_n '(' s_n ')' ']' ')'

S_2 = 'scale' '(' C ',' '[' x_1 '(' s_1 ')' ',' … ',' x_n '(' s_n ')' ']' ',' *im* ')'

eval(C_2) is defined as follows:

Coverage constituent
eval(S).*type* = *eval*(C).*type*
eval(S).*id* = *eval*(C).*id*
eval(S).*domain.crs* = *eval*(C).*domain.crs*
eval(S).*domain.axisLabels* = *eval*(C).*domain.axisLabels*
eval(S).*domain.uomLabels* = *eval*(C).*domain.uomLabels*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axes*[a].*axisLabel* = a
for $a \in$ *eval*(S)*domain.axisLabels*: *eval*(S).*domain.axis*[a].*hi* = *eval*(C).*domain.axis*[a].*hi*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*lo* = *eval*(C).*domain.axis*[a].*lo*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*res* = *eval*(C).*domain.axis*[a].*res* / *eval*(s_i) if $a=x_i$ and regular for some $i \le n$ *eval*(C).*domain.axis*[a].*res* if a is regular
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*uom* = *eval*(C).*domain.axis*[a].*uom*
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a].*index.hi* = (*eval*(C).*axis*[a].*index.hi* - *eval*(C).*axis*[a].*index.lo* +1) / *eval*(C).*domain.axis*[a].*res* + *eval*(C).*axis*[a].*index.lo* if $a=x_i$ and regular for some $i \le n$ *eval*(C).*domain.axis*[a].*res* if a is regular
for $a \in$ *eval*(S).*domain.axisLabels*: *eval*(S).*domain.axis*[a_i].*index.lo* = *eval*(C).*domain.axis*[a_i].*index.lo*
for *eval*(S).*domain.axisLabels* = <a_1,…,a_d>: *eval*(S).*domain.axis*[a_i].*int* = *eval*(C).*domain.axis*[a_i].*int*
eval(S).*rangetype* = *eval*(C).*rangetype*
Interpolation method IM = an implementation-defined choice with $IM \in$ *effint*(C_1,a,f) for all $a \in$ *eval*(C).*domain.axisLabels*, $f \in$ *eval*(C).*rangetype.fieldLabels* if $S = S_1$, *im* if $S = S_2$; for $p \in$ *eval*(S).*domain.pos*: *eval*(S).*range*[p] is obtained by interpolating the range values of *eval*(C) at

the direct positions of *eval*(*S*).*domain*, applying interpolation *IM* on all axes x_i and all fields.
eval(*S*).*metadata* = *eval*(*C*).*metadata*

Example The expression below rescales a timeseries datacube by 2 in *Lat* and *Lon*: using linear interpolation along these axes; all other axes (such as time) remain unchanged:

```
scale( $XYT, [ Lat(2), Lon(2) ], linear )
```

In a *scaleByCommonFactorExpr*, a single scale factor gets applied to al coverage axes simultaneously, optionally with a common interpolation method to be used on all axes and fields. If no valid interpolation is available on the input coverage, an error is raised.

Requirement 33
A *scaleByCommonFactorExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
s be an *arithmeticExpr*,
im be an *interpolationMethod*.

Where

eval(*C*).*domain.axes*[*a*].*axisType*=*regular* for all *a*∈*eval*(*C*).*domain.axisLabels*, *im*∈*effint*(*C*,*a*,*f*)
for all *a*∈*eval*(*C*).*domain.axisLabels*, *f*∈*eval*(*C*).*rangetype.fieldLabels*.

Then

For any *scaleByCommonFactorExpr S*
where *S* is one of
S_1 = 'scale' '(' *C* ',' s ')'
S_2 = 'scale' '(' *C* ',' s ',' *im* ')'

eval(*S*) is defined as follows:

Coverage constituent
eval(*S*).*type* = *eval*(*C*).*type*
eval(*S*).*id* = *eval*(*C*).*id*
eval(*S*).*domain.crs* = *eval*(*C*).*domain.crs*
eval(*S*).*domain.axisLabels* = *eval*(*C*).*domain.axisLabels*
eval(*S*).*domain.uomLabels* = *eval*(*C*).*domain.uomLabels*
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axes*[*a*].*axisLabel* = *a*
for *a*∈*eval*(*S*)*domain.axisLabels*:

eval(*S*).*domain.axis*[*a*].*hi* = *eval*(*C*).*domain.axis*[*a*].*hi*
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axis*[*a*].*lo* = *eval*(*C*).*domain.axis*[*a*].*lo*
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axis*[*a*].*res* = *eval*(*C*).*domain.axis*[*a*].*res* * *eval*(*s*) if *a* is regular
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axis*[*a*].*uom* = *eval*(*C*).*domain.axis*[*a*].*uom*
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axis*[*a*].*index.hi* = (*eval*(*C*).*axis*[*a*].*index.hi* - *eval*(*C*).*axis*[*a*].*index.lo* +1) / *eval*(*C*).*domain.axis*[*a*].*res* + *eval*(*C*).*axis*[*a*].*index.lo* if *a* is regular
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axis*[*a*].*index.lo* = *eval*(*C*).*domain.axis*[*a*].*index.lo*
for *a*∈*eval*(*S*).*domain.axisLabels*: *eval*(*S*).*domain.axis*[*a*].*int* = *eval*(*C*).*domain.axis*[*a*].*int*
eval(*S*).*rangetype* = *eval*(*C*).*rangetype*
Interpolation method *IM* = an implementation-defined choice with *IM*∈*effint*(*C*,*a*,*f*) for all *a*∈*eval*(*C*).*domain.axisLabels*, *f*∈*eval*(*C*).*rangetype.fieldLabels* if *S* = S_1, *im* if *S* = S_2; for *p*∈*eval*(*S*).*domain.pos*: *eval*(*S*).*range*[*p*] is obtained by interpolating the range values of *eval*(*C*) at the direct positions of *eval*(*S*).*domain*, applying interpolation *IM* on all axes and fields.
eval(*S*).*metadata* = *eval*(*C*).*metadata*

Example The expression below reduces resolution of a map by a factor of 2:

```
scale( $Map, 2 )
```

7.3.8 crsTransformExpr

A *crsTransformExpr* reprojects a coverage into a different CRS. This CRS may address only some of the axes, in which case all other axes of the input coverage are retained unmodified. Optionally, an interpolation method can be applied if the coverage supports it, otherwise an error is raised.

NOTE A service can refuse to accept some CRS combinations.

Requirement 34
A *crsTransformExpr* **shall** be defined as below.

Let

`C` be a `coverageExpr`,
`crs` be a `crsName`,
`im` be an `interpolationMethod`.

Where

`im`∈*effint*(`C`,`a`,`f`)
for all `a`∈*eval*(`C`).*domain.axisLabels*, `f`∈*eval*(`C`).*rangetype.fieldLabels*.

Then

for any `crsTransformExpr` `T`
where `T` is one of:
T_1 = `'crsTransform' '(' C ',' crs ')'`
T_2 = `'crsTransform' '(' C ',' crs ',' im ')'`

eval(`T`) is defined as follows:

Coverage constituent
eval(`T`).*type* = *eval*(`C`).*type*
eval(`T`).*id* = *eval*(`C`).*id*
eval(`T`).*domain.crs* = `crs`
for `a`∈*eval*(`T`).*domain.axisLabels*: `a` = *eval*(`C`).*domain.axis*[`a`] if `a` is not defined by `crs` the axis label defined by `crs` otherwise
for `a`∈*eval*(`T`).*domain.uomLabels*: `u` = *eval*(`C`).*domain.axis*[`a`].*uom* if `a` is not defined by `crs` the uom defined by `crs` otherwise
for `a`∈*eval*(`T`).*domain.axisLabels*: *eval*(`T`).*domain.axis*[`a`].*axisLabel* = `a`
for `a`∈*eval*(`T`).*domain.axisLabels*: *eval*(`T`).*domain.axis*[`a`].*hi* = *eval*(`C`).*domain.axis*[`a`].*hi* if `a` is not defined by `crs` implementation-defined by the `crs` transformation algorithm otherwise
for `a`∈*eval*(`T`).*domain.axisLabels*: *eval*(`T`).*domain.axis*[`a`].*lo* = *eval*(`C`).*domain.axis*[`a`].*lo* if `a` is not defined by `crs` implementation-defined by the `crs` transformation algorithm otherwise
for `a`∈*eval*(`T`).*domain.axisLabels*: *eval*(`T`).*domain.axis*[`a`].*res* = *eval*(`C`).*domain.axis*[`a`].*res* if `a` is regular and not defined by `crs` implementation-defined by the `crs` transformation algorithm otherwise
for `a`∈*eval*(`T`).*domain.axisLabels*:

eval(*T*).*domain.axis*[*a*].*uom* = *eval*(*C*).*domain.axis*[*a*].*uom*
for *a*∈*eval*(*T*).*domain.axisLabels*: *eval*(*T*).*domain.axis*[*a*].*int* = *eval*(*C*).*domain.axis*[*a*].*int*
for *a*∈*eval*(*T*).*domain.axisLabels*: *eval*(*T*).*domain.axis*[*a*].*index.lo* = *eval*(*C*).*domain.axis*[*a*].*index.lo* if *a* is not defined by *crs* implementation-defined by the *crs* transformation algorithm otherwise
for *a*∈*eval*(*T*).*domain.axisLabels*: *eval*(*T*).*domain.axis*[*a*].*index.hi* = *eval*(*C*).*domain.axis*[*a*].*index.hi* if *a* is not defined by *crs* implementation-defined by the *crs* transformation algorithm otherwise
for *a*∈*eval*(C_2).*axisLabels*: *eval*(C_2).*domain.index*[*a*].*hi* = | *eval*(C_2).*domain.axis*[*a*] | - *eval*(C_2).*domain.index*[*a*].*lo* -1
C_2.*rangetype* = C_1.*rangetype*
Interpolation method *IM* = an implementation-defined choice with *IM*∈*effint*(*C*,*a*,*f*) for all *a*∈*eval*(*C*).*domain.axisLabels*, *f*∈*eval*(*C*).*rangetype.fieldLabels* if *T* = T_1, *im* if *T* = T_2; for *p*∈*eval*(*T*).*domain.pos*: *eval*(*T*).*range*[*p*] is obtained by reprojecting coverage C_1 into CRS *crs* along those axes defined by *crs*, applying interpolation *IM* on all axes and fields.
eval(*T*).*metadata* = *eval*(*C*).*metadata*

Example The following expression transforms coverage `$c` (which is assumed to be 2D) into the CRS identified by "[EPSG:3035]":

```
crsTransform( $c, "[EPSG:3035]" )
```

7.3.9 polygonExpr

Requirement 35
A *polygonExpr* **shall** be either a *clipExpr* (Subclause 7.3.10) or a *clipCurtainExpr* (Subclause 7.3.14) or a *clipCorridorExpr* (Subsection 7.3.15) or a *polygonizeExpr* (Subclause 7.3.16).

7.3.10 clipExpr

The *clipExpr* extracts coverage range values along a polygon. Syntactically, all clipping variants rely on the same *clip*() function accepting a coverage and a clipping polygon. Operations are differentiated by the polygon types.

The coverage CRS must describe a 2-D geodetic horizontal surface (such as Lat/Lon).

NOTE Via induced operations, a clipping can be executed, e.g., on time series by applying the clipping polygon to each time slice in turn.

Polygons can be of type *polygon*, *multipolygon*, and *linestring*. They all have in common a representation as a list of vertices using the ISO 13249-3 *Well-Known Text* (WKT) format where the polygon segments are defined as the straight lines connecting any two neighbor vertices. Polygons of type *polygon* and *multipolygon* are closed rings, assuming an implicit line between last and first vertex. All polygon types allow self-intersection and vertex re-visitation.

Polygon coordinates are expressed in the CRS of the coverage. The polygon bounding box must intersect with the coverage domain.

The output coverage has the input coverage range values for all direct positions inside the polygon (but outside polygon holes, if any) as well as clipped areas inside the polygon, but outside the coverage will be set to null values if available, otherwise 0.

The extent of the output coverage is reduced (trimmed) in its domain extent to the minimal bounding box of the polygon.

Requirement 36
A *clipExpr* **shall** be a *clipPolygonExpr*, or a *clipMultipolygonExpr*, or a *clipLinestringExpr*, or a *clipCurtainExpr*, or a *clipCorridorExpr*.

7.3.11 clipPolygonExpr

A *clipPolygonExpr* clears all direct positions of the input coverage which are exterior to the polygon passed, and reduces the coverage domain to the minimal bounding box of the polygon.

The input consists of a coverage and a polygon. The polygon consists of one or more polygon rings which must be mutually disjoint (non-overlapping). A direct position of a coverage is inside the polygon if it is inside one of the polygon rings, otherwise outside.

The result is a 2-D coverage where range values of direct positions inside the polygon are identical to the input range values, whereas range values outside the polygon are set to nil. A direct position is inside the polygon if it is inside any of its polygon rings. The coverage extent is cropped to the intersection of the coverage domain and multipolygon minimal bounding box.

Requirement 37
A *clipPolygonExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
P be a *polygonExpr* with bounding box *B*.

Where

C has a CRS which defines a 2-D geodetic surface,
P coordinates are expressed in the CRS of *C*,

`C` domain and `P` bounding box have a non-empty intersection,
`P` contains polygon rings $p_1,\ldots,p_n$ which are mutually disjoint,
every polygon ring in `P` contains at least three points which are not colinear.

Then

for any `clipPolygonExpr C'`
where
`C'` = `'clip' '(' C ',' P ')'`

eval(`C'`) is defined as follows:

Coverage constituent
eval(`C'`).*type* = *eval*(`C`).*type*
eval(`C'`).*id* = *eval*(`C`).*id*
eval(`C'`).*domain.crs* = *eval*(`C`).*domain.crs*
eval(`C'`).*domain.axisLabels* = *eval*(`C`).*domain.axisLabels*
eval(`C'`).*domain.uomLabels* = *eval*(`C`).*domain.uomLabels*
eval(`C'`).*domain.axes* = *eval*(`C`).*domain.axes*
for all `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*axisLabel* = `a`
for all `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axes*[`a`].*uom* = *eval*(`C`).*domain.axes*[`a`].*uom*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*hi* = *min*(*eval*(`C`).*domain.axis*[`a`].*hi* , `B.a.hi`)
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*lo* = *max*(*eval*(`C`).*domain.axis*[`a`].*lo*, `B.a.lo`)
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*res* = *eval*(`C`).*domain.axis*[`a`].*res* if `a` is regular
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*uom* = *eval*(`C`).*domain.axis*[`a`].*uom*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*int* = *eval*(`C`).*domain.axis*[`a`].*int*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[a_i].*index.lo* = 0
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*index.hi* = | `B.a` |
eval(`C'`).*rangetype* = *eval*(`C`).*rangetype*
for *p*∈*eval*(`C'`).*domain.pos*: *eval*(`C'`).*range*[`p`] = *eval*(`C`).*range*[`p`] if `p` is inside any polygon ring in `P` nil or 0 otherwise
eval(`C'`).*metadata* = *eval*(`C`).*metadata*

Example: The following expressions clips a coverage by a polygon:

```
clip(
  $C,
  polygon(( 51.645 10.772, 51.641 10.784, 51.641 10.790, … ))
)
```

7.3.12 clipMultipolygonExpr

A multipolygon consists of a non-empty set of polygon rings, each with possible holes. The clip function clears all direct positions outside the first *polygonExpr* and inside the remaining *polygonExpr*s.

The result is a 2-D coverage where range values of direct positions inside the multipolygon are identical to the input range values, whereas range values outside the multipolygon are set to nil. A direct position is inside the multipolygon if it is inside the first polygon and outside any of the remaining polygons. The coverage extent is cropped to the intersection of the coverage domain and multipolygon minimal bounding box.

Requirement 38
A *clipMultipolygonExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
n be an *integer* with $n>0$,
M be a *multiPolygonExpr* with bounding box *B*,
$P_0, P_1, \ldots, P_n$ be *polygonExpr*s.

Where

P_i each contains mutually disjoint polygon rings, for $0 \le i \le n$,
every polygon ring in every P_i contains at least three points which are not colinear, for $0 \le i \le n$,
each P_i is completely inside P_0, for $0 \le i \le n$.

Then

for any *clipMultipolygonExpr C'*
where
C' = 'clip' '(' *C* ',' *M* ')'
with
M = 'multipolygon' '(' P_0 ',' P_1 ',' … ',' P_n ')'

eval(*C'*) is defined as follows:

Coverage constituent
eval(*C'*).*type* = *eval*(*C*).*type*

eval(`C'`).*id* = *eval*(`C`).*id*
eval(`C'`).*domain.crs* = *eval*(`C`).*domain.crs*
eval(`C'`).*domain.axisLabels* = *eval*(`C`).*domain.axisLabels*
eval(`C'`).*domain.uomLabels* = *eval*(`C`).*domain.uomLabels*
eval(`C'`).*domain.axes* = *eval*(`C`).*domain.axes*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*axisLabel* = `a`
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axes*[`a`].*uom* = *eval*(`C`).*domain.axes*[`a`].*uom*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*hi* = *min*(*eval*(`C`).*domain.axis*[`a`].*hi* , `B.a.hi`)
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*lo* = *max*(*eval*(`C`).*domain.axis*[`a`].*lo*, `B.a.lo`)
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*res* = *eval*(`C`).*domain.axis*[`a`].*res* if `a` is regular
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*uom* = *eval*(`C`).*domain.axis*[`a`].*uom*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*int* = *eval*(`C`).*domain.axis*[`a`].*int*
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[a_i].*index.lo* = 0
for `a`∈*eval*(`C'`).*domain.axisLabels*: *eval*(`C'`).*domain.axis*[`a`].*index.hi* = | `B.a` |
eval(`C'`).*rangetype* = *eval*(`C`).*rangetype*
for *p*∈*eval*(`C'`).*domain.pos*: *eval*(`C'`).*range*[`p`] = *eval*(`C`).*range*[`p`] if `p` is inside any polygon ring in P_0 but not inside any polygon ring in any of $P_1, \ldots, P_n$ nil or 0 otherwise
eval(`C'`).*metadata* = *eval*(`C`).*metadata*

Example The following is a sketch of a multipolygon expression, meaning to evaluate `$C` over Germany, but only land (by excluding all water bodies):

```
clip(
  $C,
  multipolygon(
    (( <germany-boundary> )),
    (( <river-or-lake-1> )), (( <river-or-lake-2> )), …
  )
)
```

7.3.13 clipLinestringExpr

A linestring is a (not necessarily closed) polygon, with self-crossings allowed. The clip function extracts the range values along the polygon, accepting some implementation defined distance ("buffer"). The linestring must be completely inside the coverage domain.

The result is a 1-D coverage with an index axis, lining up the values in sequence.

Requirement 39
A `clipLinestringExpr` **shall** be defined as below.

Let

`C` be a `coverageExpr`,
`L` be a `linestringExpr`.

Then

for any `clipLinestringExpr C'`
where
`C' = 'clip' '(' C ',' L ')'`

eval(`C'`) is defined as follows:

Coverage constituent
eval(`C'`).*type* = *eval*(`C`).*type*
eval(`C'`).*id* = *null*
eval(`C'`).*domain.crs* = "[OGC:Index1D]"
eval(`C'`).*domain.axisLabels* = < `i` >
eval(`C'`).*domain.uomLabels* = < `"1"` >
eval(`C'`).*domain.axes*[`i`].*axisType* = *index*
eval(`C'`).*domain.axes*[`i`].*lo* = 0
eval(`C'`).*domain.axes*[`i`].*hi* = `n-1))` where `n` is the number of direct positions visited by the linestring
eval(`C'`).*rangetype* = *eval*(`C`).*rangetype*
eval(`C'`).*domain.pos* is the sequence of range values from `C` visited by `L`
eval(`C'`).*metadata* = *eval*(`C`).*metadata*

Example For a query "*Wind strength along a flight path in space and time, from a 4D x/y/z/t wind datacube*" with a 4-D linestring trajectory, the *clip*() call might look like this:

```
clip(
  sqrt( pow( $C.u, 2.0 ) + pow( $C.v, 2.0 ) ),
  linestring(
    53.3899  -6.2183  5000   "2024-08-01T02:00",
    51.5156  -0.1099  35990  "2024-08-01T03:00",
    48.8647  2.3428   35990  "2024-08-01T04:00",
    48.2137  6.3696   5000   "2024-08-01T05:00"
```

```
    )
)
```

7.3.14 clipCurtainExpr

A curtain expression takes a coverage and a polygon or linestring input and extracts hyperplanes with full extent along those coverage axes not matched by the polygon axes, so-called curtains; see [3] for the semantics foundation.

The input coverage must be at least 3-D and contain the axes listed in the polygon axis list, which is a string containing a comma-separated list of the polygon's/linestring's axis names. The Polygon coordinates are expressed in the axes provided as separate function parameter.

The output coverage inherits the CRS from the input coverage, however with all axes in the polygon axis list removed and an index axis added along which the values harvested along the linestring/polygon are collected.

Requirement 40
A *clipCurtainExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
L be a *linestringExpr*,
P be a *polygonExpr*,
A be a *axisListString*,
x is an axisName.

Where

C domain contains at least one additional axis not occurring in *A*,
L and *P* have a CRS with the axes defined in *A*, in proper sequence,
P contains at least three points which are not collinear,
A contains a comma-separated list of axis names which all occur in the axis list of *C*,
x does not appear in the axis label list of *C*.

Then

for any *clipCurtainExpr C'*
where
C' = 'curtain' '(' *C* ',' *X* ',' *A* ')'
with *X* one of
X_1 = *P*
X_2 = *L*

eval(*C'*) is defined as follows:

Coverage constituent
eval(C′).*type* = *eval*(C).*type*
eval(C′).*id* = *eval*(C).*id*
eval(C′).*domain.crs* = the CRS modified from *eval*(C).*domain.crs* so as to describe all axes in *eval*(C′).*domain.axisLabels*
eval(C′).*domain.axisLabels* = x + *eval*(C).*domain.axisLabels* - *eval*(A)
eval(C′).*domain.uomLabels* = "1" + *eval*(C).*domain.uomLabels* with all uom items corresponding to *eval*(A) axes removed
for all a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a].*axisLabel* = a
for all a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axes*[a].*uom* = *eval*(C).*domain.axes*[a].*uom*
for a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a].*hi* is given by *corridor*() is as defined in [3]
for a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a].*lo* = 0 if a=x, *eval*(C).*domain.axis*[a].*lo* if a≠x
for a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a].*axisType* = *index* if a=x, *eval*(C).*domain.axis*[a].*axisType* if a≠x
for a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a].*uom* = *eval*(C).*domain.axis*[a].*uom*
for a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a_i].*index.lo* = 0
for a∈*eval*(C′).*domain.axisLabels*: *eval*(C′).*domain.axis*[a].*index.hi* if a≠x
eval(C′).*rangetype* = *eval*(C).*rangetype*
for p∈*eval*(C′).*domain.pos*: *eval*(C′).*range*[p] = *curtain*(*eval*(C), *eval*(X), *eval*(A)) where *curtain*() is as defined in [3]
eval(C′).*metadata* = *eval*(C).*metadata*

Example The following are examples of a curtain expression with linestring and polygon:

```
curtain(
  $c,
  linestring( 25 40, 30 40, 30 45, 30 42 )
  "Lat,Lon"
)
```

```
curtain(
  $c,
  polygon(( 25 40, 30 40, 30 45, 30 42 ))
  "Lat, Long"
)
```

7.3.15 clipCorridorExpr

A corridor expression "drills tunnels" (so-called *corridors*) through a coverage by moving ("sweeping") a linestring or polygon along a linestring, harvesting the data found along and around the path; see [3] for the semantics foundation. With a 1-D linestring, the corridor forms a shape of topological dimension 2 (1-D carrier line, 1-D sweeping line); a polygon passed must be a 2-D ring, and so the corridor has a topological dimension 3 (1-D carrier line, 2-D sweeping area).

The carrier linestring must have the same dimension, axes, axis sequence, and CRS as the input coverage; its coordinates thus are expressed in the coverage's CRS. The sweeping line/polygon input must have a subset of the coverage axes, enumerated in the last input parameter as a comma-separated list of axis names.

In case of a (1-D) linestring sweeping line, the corridor output is a coverage with index-only axes where the first dimension counts along the carrier linestring and the second dimension extends along the sweeping linestring. In case of a (2-D) sweeping polygon, the output is a 3-D index-only coverage where the first dimension counts along the carrier linestring and the second and third dimension extends along the sweeping polygon.

Requirement 41
A *clipCorridorExpr* **shall** be defined as below.

Let

> *A* is a *axisListString* containing comma-separated axis names which all occur in the axis list of *C*,
> *C* be a *coverageExpr*,
> L_1, L_2 be *linestringExpr*s,
> *P* be a *polygonExpr*.

Where

> L_1 has coordinates expressed in the CRS of *C*,
> L_2 and *P* have a CRS with the axes defined in *A*, in proper sequence,
> *P* contains at least three points which are not collinear,
> *A* is a *axisListString* containing comma-separated axis names which all occur in the axis list of *C*.

Then

> for any *clipCorridorExpr C'*
> where

```
   C' = 'corridor' '(' C ',' L1 ',' X ',' A ')'
```

with X one of

```
   X1  = P
   X2  = L2
```

eval(C') is defined as follows:

Coverage constituent
eval(C').*type* = *eval*(C).*type*
eval(C').*id* = *eval*(C).*id*
eval(C').*domain.crs* = *Index2D* for $X=L_2$, *Index3D* for $X=P$
eval(C').*domain.axisLabels* = <"i","j"> for $X=L_2$, <"i","j","k"> for $X=P$
eval(C').*domain.uomLabels* = <"1","1"> for $X=L_2$, <"1","1","1"> for $X=P$
for $a \in$ *eval*(C').*domain.axisLabels*: *eval*(C').*domain.axis*[a].*axisLabel* = a
for $a \in$ *eval*(C').*domain.axisLabels*: *eval*(C').*domain.axes*[a].*uom* = "1"
for $a \in$ *eval*(C').*domain.axisLabels*: *eval*(C').*domain.axis*[a].*hi* is given by *corridor*() is as defined in [3]
for $a \in$ *eval*(C').*domain.axisLabels*: *eval*(C').*domain.axis*[a].*lo* = 0
for $a \in$ *eval*(C').*domain.axisLabels*: *eval*(C').*domain.axis*[a].*uom* = "1"
for $a \in$ *eval*(C').*domain.axisLabels*: *eval*(C').*domain.axis*[a].*axisType* = *index*
eval(C').*rangetype* = *eval*(C).*rangetype*
for $p \in$ *eval*(C').*domain.pos*: *eval*(C').*range*[p] = *corridor*(*eval*(C), *eval*(L_1), *eval*(X), *eval*(A)) where *corridor*() is as defined in [3]
eval(C').*metadata* = *eval*(C).*metadata*

Example The two clip expressions below show use of polygon and linestring. The first one assumes a UTM meter-based surface CRS combined with a date axis into a DateTime/E/N timeseries datacube, the second assumes a Lat/Lon/DateTime timeseries datacube.

```
corridor(
  $c,
  LineString(
    "2008-01-01T02:01:20.000Z" 75042.7273594 5094865.55794,
    "2008-01-01T02:01:20.000Z" 75042.7273594 5194865.55794
```

```
  ),
  LineString(
    75042.7273594 5094865.55794, 75042.7273594 5094865.55794,
    85042.7273594 5194865.55794, 95042.7273594 5194865.55794
  ),
  "E,N"
)

corridor(
  $c,
  LineString( 26 41 "1950-01-01", 28 41 "1950-01-02"),
  Polygon(( 25 40, 30 40, 30 45, 25 45)),
  "Lat,Lon",
)
```

7.3.16 polygonizeExpr

Function *polygonize*() converts boundaries of regions recognized in a coverage into polygon rings. To achieve a maximum contrast and hence best results, the coverage range type must be Boolean. The coverage must have exactly one range field component.

The connectedness parameter can be set to 4 (default) or 8. Setting it to 4 ensures a *true*-cell can only be considered a neighbor if it shares at least a corner with some other *true*-cell. With a value of 8 a *true*-cell can only be a neighbor if it shares a least an edge with some other *true*-cell.

If the result encoding consists of multiple files, as is the case with ESRI Shapefile, the files get returned as a single zip archive. Formats supported and their identifiers are implementation-defined.

Requirement 42
A `polygonizeExpr` **shall** be defined as below.

Let

> C be a `coverageExpr`,
> f be a `stringConstant`,
> c be an `integerConstant`.

Where

> $|\ eval(C).rangetype.fieldLabels\ | = 1$,
> $eval(C).rangetype.fields[f].def = bool$ for $f \in eval(C_2).rangetype.fields$,
> $eval(c) \in \{4,8\}$.

Then

> for any `polygonizeExpr` P
> where
> P = `'polygonize' '(' C ',' f ( ',' c )? ')'`

eval(*P*)=*E* is defined as follows:

E is a multipolygon with a set of outer polygon rings which enclose areas of *true* values in the input coverage and a (potentially empty) set of inner polygon rings enclosing areas of *false* within *true* regions in the coverage.

The connectedness parameter *c* is assumed to be 4 if omitted. A connectedness of 4 ensures a *true* cell can only be considered a neighbor if it shares at least a corner with some other *true* cell. A connectedness of 8 means a *true* cell can only be a neighbor if it shares a least an edge with some other *true* cell.

Example By providing a suitable coverage predicate, any aspect can be vectorized, such as heatmaps:

```
polygonize( $Temperature > 40, "ESRI Shapefile" )
```

7.4 rangeExpr

7.4.1 Overview

Range expressions modify the range values and possibly the range type, but leave the domain set unchanged.

Requirement 43
A *rangeExpr* **shall** be either a *unaryInducedExpr* (Subclause 7.4.2), or a *binaryInducedExpr* (Subclause 7.4.9), or a *booleanExpr* (Subclause 7.4.6), or a *switchExpr* (Subclause 7.4.10), or a *rangeConstructorExpr* (Subclause 7.4.10), or a *condenseExpr* (Subclause 7.5).

Induced operations allow to simultaneously apply a function originally working on a single value to all grid point values of a coverage.

Requirement 44
In a *rangeExpr*, in case the range type contains more than one range component, the function **shall** be applied to each point simultaneously.

Requirement 45
In a *rangeExpr*, whenever a numeric argument is expected (such as a coverage with numeric range fields), Boolean *false* and *true* **shall** be interpreted as 0 and 1, resp. Conversely, whenever a Boolean argument is expected (such as a coverage with numeric range fields), then 0 and 1 **shall** be interpreted as Boolean *false* and *true*, resp.

Requirement 46
In a *rangeExpr*, whenever one of the point values ("pixels", etc.) participating in an induced operation is equal to one of the nil values of its coverage then the result **shall** be one of the values in the participating coverage's nil value set (for a unary induced operation) or one of the values in the nil value set intersection of both participating coverages (for a binary induced operation) if said intersection is not empty.
If no nil value is available (for a unary induced operation) or the intersection of both in-

put coverages' nil values is empty (for a binary induced operation) then the server **shall** use a value of 0.

Requirement 47
In a *rangeExpr* the result coverage **shall** have the same domain as the input coverage(s).

NOTE The term "induced" is based on the idea that for each operation available on a range field data type, a corresponding operation is provided applying the operation to the whole coverage, thus "lifting" the operation to coverage level.

7.4.2 unaryInducedExpr

The *unaryInducedExpr* element specifies a unary induced operation, i.e., an operation where only one coverage argument occurs.

NOTE The term "unary" refers only to coverage arguments; it is well possible that further non-coverage parameters occur, such as an integer number indicating the shift distance in a bit() operation.

Requirement 48
A *unaryInducedExpr* **shall** be either a *unaryArithmeticExpr* (Subclause 7.4.3), or a *trigonometricExpr* (Subclause 7.4.4), or *exponentialExpr* (Subclause 7.4.5), a *booleanExpr* (Subclause 7.4.6), a *castExpr* (Subclause 7.4.7), or a *fieldExpr* (Subclause 7.4.8).

7.4.3 unaryArithmeticExpr

The *unaryArithmeticExpr* element specifies a unary induced arithmetic operation.

Requirement 49
A *unaryArithmeticExpr* **shall** be defined as below.

Let

> C_1 be a *coverageExpr*,
> n be a non-negative *integerExpr*

Where

> C_1.*rangetype*.fields[f].*nature*=*quantity* for all $f \in C_1$.*rangetype.fieldLabels*

Then

> for any *coverageExpr* C_2
> where C_2 is one of
> C_{plus} = '+' C_1
> C_{minus} = '-' C_1
> C_{sqrt} = 'sqrt' '(' C_1 ')'
> C_{abs} = 'abs' '(' C_1 ')'
> C_{bit} = 'bit' '(' C_1 ',' n ')'

```
C_rd  = 'round' '(' C_1 ')'
C_re  = C_1 '.' 're'
C_im  = C_1 '.' 'im'
```

eval(C_2) is defined as follows:

Coverage constituent
eval(C_2).*type* = *eval*(C_1).*type*
eval(C_2).*id* = *eval*(C_1).*id*
eval(C_2).*domain* = *eval*(C_1).*domain*
eval(C_2).*rangetype.fieldLabels* = *eval*(C_1).*rangetype.fieldLabels*
for $f \in$ *eval*(C_2).*rangetype.fields*: *eval*(C_{plus}).*rangetype.fields*[f].*def* = *eval*(C_1).*rangetype.fields*[f].*def*, *eval*(C_{minus}).*rangetype.fields*[f].*def* = *eval*(C_1).*rangetype.fields*[f].*def* if *eval*(C_1).*rangetype.fields*[f].*def* ∈ { *char*, *short*, *int*, *long*, *float*, *double*, *complex*, *complex2* }, *char* if *eval*(C_1), *rangetype.fields*[f] = *unsigned char*, *short* if *eval*(C_1).*rangetype.fields*[f].*def* = *unsigned short*, *int* if *eval*(C_1).*rangetype.fields*[f].*def* = *unsigned int*, *long* if *eval*(C_1).*rangetype.fields*[f].*def* = *unsigned long*, *eval*(C_{sqrt}).*rangetype.fields*[f].*def* = *double* if *eval*(C_1).*rangetype.fields*[f].*def* ∉ {*complex*,*complex2*} else *complex2*, *eval*(C_{abs}).*rangetype.fields*[f].*def* = *eval*(C_1).*rangetype.fields*[f].*def* if *eval*(C_1).*rangetype.fields*[f].*def* ∈ { *boolean*, *unsigned char*, *unsigned short*, *unsigned int*, *unsigned long*, *float*, *double* }, *unsigned char* if *eval*(C_1).*rangetype.fields*[f].*def* = *char*, *unsigned short* if *eval*(C_1).*rangetype.fields*[f].*def* = *short*, *unsigned int* if *eval*(C_1).*rangetype.fields*[f].*def* = *int*, *unsigned long* if *eval*(C_1).*rangetype.fields*[f].*def* = *long*, *float* if *eval*(C_1).*rangetype.fields*[f].*def* = *complex*, *double* if *eval*(C_1).*rangetype.fields*[f].*def* = *complex2* *eval*(C_{bit}).*rangetype.fields*[f_i].*def*= *unsigned char* *eval*(C_{rd}).*rangetype.fields*[f].*def* = *eval*(C_1.*rangetype.fields*[f].*def* *eval*(C_{re}).*rangetype.fields*[f].*def* = *float* if *eval*(C_1.*rangetype.fields*[f].*def*=*complex* else *double*

eval(C_{im}).*rangetype.fields*[f].*def* = *float* if *eval*(C_1).*rangetype.fields*[f].*def=complex* else *double*
for $f \in$ *eval*(C_2).*rangetype.fieldLabels*: *eval*(C_2).*rangetype.fields*[f].*nil* = ∅
for $f \in$ *eval*(C_2).*rangetype.fieldLabels*: *eval*(C_2).*rangetype.fields*[f].*uom* = *null*
for $f \in$ *eval*(C_2).*rangetype.fieldLabels*: *eval*(C_2).*rangetype*.fields[f].*nature* = *eval*(C_1).*rangetype*.fields[f].*nature*
for $f \in$ *eval*(C_2).*rangetype.fields*: *eval*(C_2).*rangetype*.fields[f].*int* = *eval*($C1$).*rangetype*.fields[f].*int*
for $p \in$ *eval*(C_2).*domain.pos*, $f \in$ *eval*(C_2).*rangetype.fieldLabels*: *eval*(C_{plus}).*range*[*p*].f = *eval*(C_1).*range*[*p*].f, *eval*(C_{minus}).*range*[*p*].f = - *eval*(C_1).*range*[*p*].f, *eval*(C_{sqrt}).*range*[*p*].f = *sqrt*(*eval*(C_1).*range*[*p*].f), *eval*(C_{abs}).*range*[*p*].f = *abs*(*eval*(C_1).*range*[*p*].f) *eval*(C_{bit}).*range*[*p*].f = (*eval*(C_1).*range*[*p*].f) >> n) mod 2 *eval*(C_{rd}).*range*[*p*].f = *round*(*eval*(C_1).*range*[*p*].f) *eval*(C_{re}).*range*[*p*].f = (*eval*(C_1).*range*[*p*].f).*re* *eval*(C_{im}).*range*[*p*].f = (*eval*(C_1).*range*[*p*].f).*im*
eval(C_2).*metadata* = *eval*(C_1).*metadata*

Example The following coverage expression evaluates to a float-type coverage where each range value contains the square root of the sum of the corresponding source coverages' values.

```
sqrt( $c + $d )
```

NOTE The operation `bit(a,b)` extracts bit position b (assuming an implementation-defined binary representation) from integer number a and shifts the resulting bit value to bit position 0. Hence, the resulting value is either 0 or 1.

7.4.4 trigonometricExpr

The *trigonometricExpr* element specifies a unary induced trigonometric operation.

Requirement 50
A *trigonometricExpr* **shall** be defined as below.

Let

C_1 be a *coverageExpr*
where
C_1.*rangetype*.fields[f].*nature=quantity* for all $f \in C_1$.*rangetype.fieldLabels.*

Then

for any **coverageExpr** c_2
where c_2 is one of

c_{sin} = 'sin' '(' c_1 ')'
c_{cos} = 'cos' '(' c_1 ')'
c_{tan} = 'tan' '(' c_1 ')'
c_{sinh} = 'sinh' '(' c_1 ')'
c_{cosh} = 'cosh' '(' c_1 ')'
c_{arcsin} = 'arcsin' '(' c_1 ')'
c_{arccos} = 'arccos' '(' c_1 ')'
c_{arctan} = 'arctan' '(' c_1 ')'

c_2 is defined as follows:

Coverage constituent
eval(c_2).*type* = *eval*(c_1).*type*
eval(c_2).*id* = *eval*(c_1).*id*
eval(c_2).*domain* = *eval*(c_1).*domain*
eval(c_2).*rangetype.fieldLabels* = *eval*(c_1).*rangetype.fieldLabels*
for *f*∈*eval*(c_2).*rangetype.fields*: *eval*(c_2).*rangetype.fields*[*f*].*def* = *complex2* if *eval*(c_1).*rangetype.fields*[*f*].*def* ∈{ *complex*, *complex2* }, *double* otherwise
for *f*∈*eval*(c_2).*rangetype.fieldLabels*: *eval*(c_2).*rangetype.fields*[*f*].*nil* = ∅
for *f*∈*eval*(c_2).*rangetype.fieldLabels*: *eval*(c_2).*rangetype.fields*[*f*].*uom* = *null*
for *f*∈*eval*(c_2).*rangetype.fieldLabels*: *eval*(c_2).*rangetype*.fields[*f*].*nature* = *eval*(c_1).*rangetype*.fields[*f*].*nature*
for *f*∈*eval*(c_2.*rangetype.fields*: *eval*(c_2).*rangetype*.fields[*f*].*int* = *eval*($c1$).*rangetype*.fields[*f*].*int*
for *p*∈*eval*(c_2).*domain.pos*, c_2. *f*∈*eval*(c_2).*rangetype.fieldLabels*: *eval*(c_{sin}).*range*[*p*].*f* = *sin*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{cos}).*range*[*p*] .*f* = *cos*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{tan}).*range*[*p*] .*f* = *tan*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{sinh}).*range*[*p*] .*f* = *sinh*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{cosh}).*range*[*p*] .*f* = *cosh*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{arcsin}).*range*[*p*] .*f* = *arcsin*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{arccos}).*range*[*p*] .*f* = *arccos*(*eval*(c_1).*range*[*p*] .*f*) *eval*(c_{arctan}).*range*[*p*] .*f* = *arctan*(*eval*(c_1),.*range*[*p*] .*f*)
eval(c_2).*metadata* = *eval*(c_1).*metadata*

Example The following expression replaces all values of the coverage addressed by `$c` with their sine:

```
sin( $c )
```

Example To enforce a complex result for real-valued arguments the input coverage can be cast to complex:

```
arcsin( (complex) $c )
```

7.4.5 exponentialExpr

The *exponentialExpr* element specifies a unary induced exponential operation.

Requirement 51
An *exponentialExpr* **shall** be defined as below.

Let

C_1 be a *coverageExpr*,
x be a *floatConstant*
where
C_1.*rangetype*.fields[f].*nature*=*quantity* for all $f \in C_1$.*rangetype*.*fieldLabels*.

Then

for any *coverageExpr* C_2
where C_2 is one of

C_{exp} = 'exp' '(' C_1 ')'
C_{log} = 'log' '(' C_1 ')'
C_{ln} = 'ln' '(' C_1 ')'
C_{pow} = 'pow' '(' C_1 ',' x ')'

C_2 is defined as follows:

Coverage constituent
eval(C_2).*type* = *eval*(C_1).*type*
eval(C_2).*id* = *eval*(C_1).*id*
eval(C_2).*domain* = *eval*(C_1).*domain*
eval(C_2).*rangetype*.*fieldLabels* = *eval*(C_1).*rangetype*.*fieldLabels*
for $f \in$ *eval*(C_2).*rangetype*.*fields*: *eval*(C_2).*rangetype*.*fields*[f].*def* = *complex2* if *eval*(C_1).*rangetype*.*fields*[f].*def* $\in$ { *complex*, *complex2* }, *double* otherwise
for $f \in$ *eval*(C_2).*rangetype*.*fieldLabels*: *eval*(C_2).*rangetype*.*fields*[f].*nil* = $\varnothing$
for $f \in$ *eval*(C_2).*rangetype*.*fieldLabels*: *eval*(C_2).*rangetype*.*fields*[f].*uom* = *null*
for $f \in$ *eval*(C_2).*rangetype*.*fieldLabels*: *eval*(C_2).*rangetype*.fields[f].*nature* = *eval*(C_1).*rangetype*.fields[f].*nature*

for $f \in eval(C_2).rangetype.fields$: $eval(C_2).rangetype.\text{fields}[f].int = eval(C_1).rangetype.\text{fields}[f].int$
for $p \in eval(C_2).domain.pos$, $f \in eval(C_2).rangetype.fields$: $eval(C_{exp}).range[p].f = exp(\ eval(C_1).range[p].f\ )$ $eval(C_{log}).range[p].f = log(\ eval(C_1).range[p].f\ )$ $eval(C_{ln}).range[p].f = ln(\ eval(C_1).range[p].f\ )$ $eval(C_{pow}).range[p].f = (\ eval(C_1).range[p].f\ )^x$
$eval(C_2.metadata = eval(C_1.metadata$

Example The following expression replaces all (nonnegative numeric) values of coverage C with their natural logarithm:

```
ln( $c )
```

7.4.6 booleanExpr

The *booleanExpr* element specifies an induced Boolean negation.

Requirement 52
A *booleanExpr* **shall** be defined as below.

Let

C_1 be a *coverageExpr*.

Then

for any *coverageExpr* C_2
where
C_2 = 'not' C_1

C_2 is defined as follows:

Coverage constituent
$eval(C_2).type = eval(C_1).type$
$eval(C_2).id = eval(C_1).id$
$eval(C_2).domain = eval(C_1).domain$
$eval(C_2).rangetype.fieldLabels = eval(C_1).rangetype.fieldLabels$
for $f \in eval(C_2).rangetype.fields$: $eval(C_2).rangetype.fields[f].def = boolean$
for $f \in eval(C_2).rangetype.fieldLabels$: $eval(C_2).rangetype.fields[f].nil = \varnothing$
for $f \in eval(C_2).rangetype.fieldLabels$: $eval(C_2).rangetype.fields[f].uom = null$
for $f \in eval(C_2).rangetype.fieldLabels$: $eval(C_2).rangetype.\text{fields}[f].nature = eval(C_1).rangetype.\text{fields}[f].nature$

for *f*∈*eval*(*C*2).*rangetype.fields*: *eval*(*C*2).*rangetype*.fields[*f*].*int* = *eval*(*C*1).*rangetype*.fields[*f*].*int*
for *p*∈*eval*(*C*2).*domain.pos*, *C*2. *f*∈*eval*(*C*2).*rangetype.fields*: *eval*(*C*2).*range*[*p*].*f* = *not*(*eval*(*C*1).*range*[*p*] .*f*)
eval(*C*2).*metadata* = *eval*(*C*1).*metadata*

Example The following expression inverts all (assumed: Boolean) range field values of coverage `$b`:

```
not $b
```

7.4.7 castExpr

The **castExpr** element specifies a unary induced cast operation, that is: to change the range type of the coverage while leaving all other properties unchanged. All range components are converted to this same type.

NOTE Depending on the input and output types result possibly may suffer from a loss of accuracy through data type conversion.

Requirement 53
A *castExpr* **shall** be defined as below.

Let

*C*1 be a *coverageExpr*,
t be a range field type name.

Then

for any *coverageExpr* *C*2
where
*C*2 = '(' *t* ')' *C*1

*C*2 is defined as follows:

Coverage constituent
eval(*C*2).*type* = *eval*(*C*1).*type*
eval(*C*2).*id* = *eval*(*C*1).*id*
eval(*C*2).*domain* = *eval*(*C*1).*domain*
eval(*C*2).*rangetype.fieldLabels* = *eval*(*C*1).*rangetype.fieldLabels*
for *f*∈*eval*(*C*)2.*rangetype.fields*: *eval*(*C*2).*rangetype.fields*[*f*].*def* = *t*
for *f*∈*eval*(*C*2).*rangetype.fieldLabels*: *eval*(*C*2).*rangetype.fields*[*f*].*nil* = ∅
for *f*∈*eval*(*C*2).*rangetype.fieldLabels*: *eval*(*C*2).*rangetype.fields*[*f*].*uom* = *null*

for *f*∈*eval*(*C*2).*rangetype.fieldLabels*: *eval*(*C*2).*rangetype.fields*[*f*].*nature* = *eval*(*C*1).*rangetype.fields*[*f*].*nature*
for *f*∈*eval*(*C*2).*rangetype.fields*: *eval*(*C*2).*rangetype.fields*[*f*].*int* = *eval*(*C*1).*rangetype.fields*[*f*].*int*
for all *p*∈*eval*(*C*2).*domain.pos*, *f*∈*eval*(*C*2).*rangetype.fields*: *eval*(*C*2).*range*[*p*].*f* = (*t*) *eval*(*C*1).*range*[*p*] .*f*
eval(*C*2).*metadata* = *eval*(*C*1).*metadata*

Example the result range type of the following expression will be char, i.e., 8 bit:

```
(char) ( $c / 2 )
```

Number literals can be attached with a postfix suffix to indicate an exact type for the number. The syntax for *integerConstant* and *floatConstant* is, with no whitespace in between:

```
<number><suffix>
```

Requirement 54
Numerical literals **may** have a suffix acting as a data type cast. Recognized suffixes are listed in Table 3. By default, if no type suffix is specified, integer values have a type `int`, floating-point values have a type `double`, complex numbers have a type `complex2`.

Table 3 – Numerical literal type specifiers[7].

Suffix (case insensitive)	data type	example
c	*char*	-3c
uc	*unsigned char*	255uc
s	*short*	-2000s
us	*unsigned short*	1000us
l	*int*	13l
ul	*unsigned int*	393ul
f	*float*	0.5f
d	*double*	12.3d

[7] "l" is lower-case variant of "L"

7.4.8 fieldExpr

The `fieldExpr` element specifies a unary induced field selection operation.fields are selected by their name.

Requirement 55
A `fieldExpr` **shall** be defined as below.

Let

`C1` be a `coverageExpr`,
`f` be a `fieldName` appearing in `C1`.*rangetype.fieldLabels*.

Then

for any `coverageExpr` `C2`
where
`C2 = C1 '.' f`

`C2` is defined as follows:

Coverage constituent
eval(`C2`).*type* = *eval*(`C1`).*type*
eval(`C2`).*id* = *eval*(`C1`).*id*
eval(`C2`).*domain* = *eval*(`C1`).*domain*
eval(`C2`).*rangetype.fieldLabels* = <`f`>
eval(`C2`).*rangetype.fields*[`f`].*def* = *eval*(`C1`).*rangetype.fields*[`f`].*def*
eval(`C2`).*rangetype.fields*[`f`].*nil* = *eval*(`C1`).*rangetype.fields*[`f`].*nil*
eval(`C2`).*rangetype.fields*[`f`].*uom* = *eval*(`C1`).*rangetype.fields*[`f`].*uom*
eval(`C2`).*rangetype*.fields[`f`].*nature* = *eval*(`C1`).*rangetype.fields*[`f`].*nature*
eval(`C2`).*rangetype*.fields[`f`].*int* = *eval*(`C1`).*rangetype.fields*[`f`].*int*
eval(`C2`).*range*[*p*] .`f` = *eval*(`C1`).*range*[*p*].`f`
eval(`C2`).*metadata* = *eval*(`C1`).*metadata*

Example Let `$c` refer to a coverage with range type integer. Then the following request snippet describes a single-field, integer-type coverage where each grid point value contains the difference between red and green band:

```
$c.red - $c.green
```

7.4.9 binaryInducedExpr

The `binaryInducedExpr` element specifies a binary operation, i.e., an operation involving two coverage-valued arguments, executed on each pair of corresponding direct positions. For this matching to work, all input coverages need to share the same domain set and have compatible range types.

As necessary, appropriate type conflation is performed on the range values prior to performing the binary value operation (cf. Requirement 80).

Requirement 56
In a `binaryInducedExpr` the participating coverages **shall** have the same domain and the same number of range fields.

Requirement 57
A `binaryExpr` **shall** be defined as below.

Let

C_1, C_2 be `coverageExpr`s,
S_1, S_2 be `scalarExpr`s,
where
C_1.*domain* = C_2.*domain*, except possibly for the *int* component,
C_1.*rangetype*.*fields*[f].*def* and C_2.*rangetype*.*fields*[f].*def* are cast-compatible as per Requirement 80 for all $f \in C_1$.*rangetype*.*fieldLabels*,
(C_i .*rangetype*.*fields*[f]. *nature=quantity*}
or
(C_i .*rangetype*.*fields*[f]. = *count* and operation ∈{=,!=,>,>=,<,<=})
or
(C_i .*rangetype*.*fields*[f]. = *category* and operation ∈{=,!=})
) for all $f \in C_2$.*rangetype*.*fields*, $i \in \{1,2\}$.

Then

for any coverageExpr C_3
where C_3 is one of

C_{plusCC} = C_1 '+' C_2
C_{minCC} = C_1 '-' C_2
C_{multCC} = C_1 '*' C_2
C_{divCC} = C_1 '/' C_2
C_{andCC} = C_1 'and' C_2
C_{orCC} = C_1 'or' C_2
C_{xorCC} = C_1 'xor' C_2
C_{eqCC} = C_1 '=' C_2
C_{ltCC} = C_1 '<' C_2
C_{gtCC} = C_1 '>' C_2
C_{leCC} = C_1 '<=' C_2
C_{geCC} = C_1 '>=' C_2
C_{neCC} = C_1 '!=' C_2
C_{ovlCC} = C_1 'overlay' C_2
$C_{atan2CC}$ = ('arctan2' | 'atan2') '(' C_1 ',' C_2 ')'

C_{plusSC} = S_1 '+' C_2
C_{minSC} = S_1 '-' C_2

```
C_multSC    = S1 '*' C2
C_divSC     = S1 '/' C2
C_andSC     = S1 'and' C2
C_orSC      = S1 'or' C2
C_xorSC     = S1 'xor' C2
C_eqSC      = S1 '=' C2
C_ltSC      = S1 '<' C2
C_gtSC      = S1 '>' C2
C_leSC      = S1 '<=' C2
C_geSC      = S1 '>=' C2
C_neSC      = S1 '!=' C2
C_ovlSC     = S1 'overlay' C2
C_atan2SC   = ( 'arctan2' | 'atan2' ) '(' S1 ',' C2 ')'

C_plusCS    = C1 '+' S2
C_mincS     = C1 '-' S2
C_multCS    = C1 '*' S2
C_divCS     = C1 '/' S2
C_andCS     = C1 'and' S2
C_orCS      = C1 'or' S2
C_xorCS     = C1 'xor' S2
C_eqCS      = C1 '=' S2
C_ltCS      = C1 '<' S2
C_gtCS      = C1 '>' S2
C_leCS      = C1 '<=' S2
C_geCS      = C1 '>=' S2
C_neCS      = C1 '!=' S2
C_ovlCS     = C1 'overlay' S2
C_atan2CS   = ( 'arctan2' | 'atan2' ) '(' C1 ','S2 ')'
```

C_3 is defined as follows:

Coverage constituent
eval(C_3).*type* = *eval*(C_1).*type*
eval(C_3).*id* = *eval*(C_1).*id*
eval(C_3).*domain* = *eval*(C_1).*domain*
eval(C_3).*rangetype*.*fieldLabels* = *eval*(C_1).*rangetype*.*fieldLabels* if *eval*(C_1).*rangetype*.*fieldLabels* = *eval*(C_2).*rangetype*.*fieldLabels* < *"band1", "band2", ...* > otherwise
for $f \in$ *eval*(C_3).*rangetype*.*fields*: *eval*(C_{plusCC}).*rangetype*.*fields*[f].*def* is given by Requirement 80 *eval*(C_{minCC}).*rangetype*.*fields*[f].*def* is given by Requirement 80 *eval*(C_{multCC}).*rangetype*.*fields*[f].*def* is given by Requirement 80 *eval*(C_{divCC}).*rangetype*.*fields*[f].*def* is given by Requirement 80

eval(C_{andCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{orCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{xorCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{eqCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{ltCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{gtCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{leCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{geCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{neCC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{ovlCC}).*rangetype*.*fields*[f].*def*	= *eval*(C_1).*rangetype*.*fields*[f_i].*def*
eval($C_{atan2CC}$).*rangetype*.*fields*[f].*def*	= *double*
eval(C_{plusSC}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{minSC}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{multSC}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{divSC}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{andSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{orSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{xorSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{eqSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{ltSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{gtSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{leSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{geSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{neSC}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{ovlSC}).*rangetype*.*fields*[f].*def*	= *rangeFieldType*(C_2)
eval($C_{atan2SC}$).*rangetype*.*fields*[f].*def*	= *double*
eval(C_{plusCS}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{minCS}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{multCS}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{divCS}).*rangetype*.*fields*[f].*def*	is given by Requirement 80
eval(C_{andCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{orCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{xorCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{eqCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{ltCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{gtCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{leCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{geCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{neCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval(C_{ovlCS}).*rangetype*.*fields*[f].*def*	= *boolean*
eval($C_{atan2CS}$).*rangetype*.*fields*[f].*def*	= *double*
for $f \in$ *eval*(C_3).*rangetype*.*fieldLabels*: *eval*(C_3).*rangetype*.*fields*[f].*nil* = ∅	
for $f \in$ *eval*(C_3).*rangetype*.*fieldLabels*: *eval*(C_3).*rangetype*.*fields*[f].*uom* = *null*	

for $f \in eval(C_3).rangetype.fieldLabels$:
$eval(C_3).rangetype.fields[f].nature = eval(C_1).rangetype.fields[f].nature$

for $f \in eval(C_3).rangetype.fields$:
$eval(C_3).rangetype.fields[f].int = eval(C_1).rangetype.fields[f].int \cap eval(C_2).rangetype.fields[f].int$

for all $p \in eval(C_3).domain.pos$:

$eval(C_{plusCC}).range[p] = eval(C_1) + eval(C_2)$
$eval(C_{minCC}).range[p] = eval(C_1) - eval(C_2)$
$eval(C_{multCC}).range[p] = eval(C_1) * eval(C_2)$
$eval(C_{divCC}).range[p] = eval(C_1) / eval(C_2)$
$eval(C_{andCC}).range[p] = eval(C_1)$ and $eval(C_2)$
$eval(C_{orCC}).range[p] = eval(C_1)$ or $eval(C_2)$
$eval(C_{xorCC}).range[p] = eval(C_1)$ xor $eval(C_2)$
$eval(C_{eqCC}).range[p] = eval(C_1).range[p] = eval(C_2).range[p]$
$eval(C_{ltCC}).range[p] = eval(C_1).range[p] < eval(C_2).range[p]$
$eval(C_{gtCC}).range[p] = eval(C_1).range[p] > eval(C_2).range[p]$
$eval(C_{leCC}).range[p] = eval(C_1).range[p] <= eval(C_2).range[p]$
$eval(C_{geCC}).range[p] = eval(C_1).range[p] >= eval(C_2).range[p]$
$eval(C_{neCC}).range[p] = eval(C_1).range[p] != eval(C_2).range[p]$
$eval(C_{ovlCC}).range[p] = eval(C_2).range[p]$ if $eval(C_1).range[p] = 0$, $eval(C_1).range[p]$ otherwise
$eval(C_{atan2CC}).range[p] = atan2(\ eval(C_1).range[p]\ ,\ eval(C_2).range[p]\ )$

$eval(C_{plusSC}).range[p] = eval(S_1) + eval(C_2).range[p]$
$eval(C_{minSC}).range[p] = eval(S_1) - eval(C_2).range[p]$
$eval(C_{multSC}).range[p] = eval(S_1) * eval(C_2).range[p]$
$eval(C_{divSC}).range[p] = eval(S_1 / eval(C_2).range[p]$
$eval(C_{andSC}).range[p] = eval(S_1)$ and $eval(C_2).range[p]$
$eval(C_{orSC}).range[p] = S_1$ or $eval(C_2).range[p]$
$eval(C_{xorSC}).range[p] = eval(S_1)$ xor $eval(C_2).range[p]$
$eval(C_{eqSC}).range[p] = eval(S_1) = eval(C_2).range[p]$
$eval(C_{ltSC}).range[p] = eval(S_1) < eval(C_2).range[p]$
$eval(C_{gtSC}).range[p] = eval(S_1) > eval(C_2).range[p]$
$eval(C_{leSC}).range[p] = eval(S_1) <= eval(C_2).range[p]$
$eval(C_{geSC}).range[p] = eval(S_1) >= eval(C_2).range[p]$
$eval(C_{neSC}).range[p] = eval(S_1) != eval(C_2).range[p]$
$eval(C_{ovlSC}).range[p] = eval(C_2).range[p]$ if $eval(S_1) = 0$, $eval(S_1)$ otherwise
$eval(C_{atan2SC}).range[p] = atan2(\ eval(S_1).range[p]\ ,\ eval(C_2).range[p]\ )$

$eval(C_{plusCS}).range[p] = eval(C_1).range[p] + eval(S_2)$
$eval(C_{minCS}).range[p] = eval(C_1).range[p] - eval(S_2)$
$eval(C_{multCS}).range[p] = eval(C_1).range[p] * eval(S_2)$
$eval(C_{divCS}).range[p] = eval(C_1).range[p] / eval(S_2)$
$eval(C_{andCS}).range[p] = eval(C_1).range[p]$ and $eval(S_{2v})$
$eval(C_{orCS}).range[p] = eval(C_1).range[p]$ or $eval(S_2)$

$eval(C_{xorCS}).range[p]$	$= eval(C_1).range[p]$ xor $eval(S_2)$
$eval(C_{eqCS}).range[p]$	$= eval(C_1).range[p] = eval(eval(S_2)$
$eval(C_{ltCS}).range[p]$	$= eval(C_1).range[p] < eval(S_2)$
$eval(C_{gtCS}).range[p]$	$= eval(C_1).range[p] > eval(S_2)$
$eval(C_{leCS}).range[p]$	$= eval(C_1).range[p] <= eval(S_2)$
$eval(C_{geCS}).range[p]$	$= eval(C_1).range[p] >= eval(S_2)$
$eval(C_{neCS}).range[p]$	$= eval(C_1).range[p]$!= $eval(S_2)$
$eval(C_{ovlCS}).range[p]$	$= eval(S_2)$ if $eval(C_1).range[p]=0$, $eval(C_1).range[p]$ otherwise
$eval(C_{atan2CS}).range[p]$	$= atan2(\ eval(C_1).range[p]\ ,\ eval(S_2).range[p]\ )$
$eval(C_3).metadata = eval(C_1).metadata$	

Example The following expression describes a coverage composed of the sum of the red, green, and blue fields of the coverage referred to by `$c`:

```
$c.red + $c.green + $c.blue
```

7.4.10 switchExpr

The *switchExpr* provides a case distinction for individually choosing range values into a result coverage. As an *n*-ary induced operation, it evaluates the condition and result expressions for every direct position. Therefore, all participating coverages must share the same domain and range type.

Condition expressions in the *case* clause must evaluate to *boolean* scalars. For every direct position, the *case* clauses are evaluated sequentially, and the first *true* alternative is chosen, if any, for evaluation of the associated *return* clause; otherwise, the *default* alternative is chosen.

All *return* clauses must collectively evaluate to either scalar or coverages. All scalar expressions must have the same or a compatible data type (cf. Requirement 80), all coverages must have the same domain and range type. Mixed cases are possible, though; the scalar expressions in such cases are treated as induced operations yielding coverages matching the coverages in the other branches.

- If the *return* expressions all return scalar values, the overall result is scalar.
- If the *return* expressions return coverages (possibly via induced operations), the overall result is coverage-valued.

As in general no single input coverage determines the result completely, the output bears no id and no metadata.

NOTE 1 The conditions of the statement are evaluated in a manner similar to the *if-then-else* statement in programming languages such as Java or C++.

NOTE 2 In case of non-disjoint predicates, it is good practice to specify conditions in order of decreasing selectivity.

Requirement 58
Syntax and semantics of a *switchExpr* **shall** be given as follows.

Let

n be an *integer* with $n \geq 1$,
$B_1, \ldots, B_n$ be *coverageExpr*s,
$R, R_1, \ldots, R_{n+1}$ be *coverageExpr*s,

Where, for $1 \leq i \leq n$, after a lifting of scalar expressions to coverages in an implicit induced operation:

$B_1.domain = \ldots = B_n.domain = R_1.domain = \ldots = R_{n+1}.domain$,
$B_1, \ldots, B_n$ all have a single range field of type *boolean*,
$R_1.rangetype = \ldots = R_{n+1}.rangetype$.

Then

for any coverageExpr *C′*
where

```
   C′ = 'switch'
        'case'  B1 'return'  R1
        …
        'case'  Bn 'return'  Rn
        'default' 'return'  Rn+1
```

C′ is defined as follows:

Coverage constituent
eval(*C′*).*type* = *GeneralGridCoverage*
eval(*C′*).*id* = *null*
eval(*C′*).*domain* = *eval*(R_1).*domain*
eval(*C′*).*rangetype* = *eval*(R_1).*rangetype*
for all $p \in$ *eval*(*C′*).*domain.pos*: *eval*(*C′*).*range*[*p*] = if *eval*(B_1).*range*[*p*] = *true* then *eval*(R_1).*range*[*p*] else if *eval*(B_2).*range*[*p*] = *true* then *eval*(R_2).*range*[*p*] … else if *eval*(B_n).*range*[*p*] = *true* then *eval*(R_n).*range*[*p*] else *eval*(R_{n+1}).*range*[*p*]
eval(*C′*).*metadata* = *null*

Example 1 The expression below performs a traffic light classification on some single-band coverage `$c`.

```
switch
  case $c < 10 return $c * {red: 0;   green: 0;   blue: 255}
```

```
  case $c < 20 return $c * {red: 0;   green: 255; blue: 0}
  case $c < 30 return $c * {red: 255; green: 0;   blue: 0}
  default      return      {red: 0;   green: 0;   blue: 0}
```

Example 2 The example below computes log of all positive values in `$c`, and assigns 0 to the remaining ones. This way it avoids an exception that would otherwise be thrown should any cell not be above zero.

```
switch
  case $c>0 return log($c)
  default   return 0f
```

7.4.11 rangeConstructorExpr

The *rangeConstructorExpr*, an *n*-ary induced operation, allows building coverages with compound range structures. To this end, coverage range field expressions enumerated are combined into one coverage.

All input coverages must match in their domains, but not in their range type. An input coverage range field may be listed more than once.

Requirement 59
A *rangeConstructorExpr* **shall** be defined as below.

Let

> n be an *integer* with $n \geq 1$,
> $C_1, \ldots, C_n$ be *coverageExpr*s,
> $f_1, \ldots, f_n$ be *fieldName*s

Where, for $1 \leq i,j \leq n$,

> $C_1.domain = \ldots = C_n.domain$,
> $|\ C_1.rangeFieldLabels\ | = \ldots = |\ C_n.rangeFieldLabels\ | = 1$.

Then

> for any *coverageExpr* C'
> where
> C' = ('struct')? '{' f_1 ':' C_1 ',' … ',' f_n ':' C_n '}'
>
> C' is defined as follows:

Coverage constituent
eval(C').*type = GeneralGridCoverage*
eval(C').*id = null*
eval(C').*domain = eval*(C_1).*domain*
for $1 \leq i \leq n$: *eval*(C').*rangetype.fieldLabels*[i] = *eval*(C_i).*rangetype.fieldLabels*[1]

for $f \in eval(C').rangetype.fieldLabels$: $eval(C').rangetype.fields[f] = eval(C_i).rangetype.fields[1]$
for all $p \in eval(C').domain.pos$: $eval(C').range[p] = < eval(C_1).range[p], \ldots, eval(C_n).range[p] >$
$eval(C').metadata = null$

NOTE For convenience, both semicolon ";" and comma "," are allowed as field separators.

Example 1: The expression below does a false color encoding by combining near-infrared, red, and green bands into a 3-band image of 8-bit channels each, which can be visually interpreted as RGB:

```
struct
{ red:   (char) $c.nir,
  green: (char) $c.red,
  blue:  (char) $c.green,
}
```

Example 2: The following expression transforms a greyscale image referred to by variable `$g` containing a range field `panchromatic` into an 8-bit RGB-structured image:

```
{ red:   (unsigned char) $g.panchromatic,
  green: (unsigned char) $g.panchromatic,
  blue:  (unsigned char) $g.panchromatic
}
```

7.5 condenseExpr

7.5.1 Overview

Condensers serve to summarize over coverages or parts thereof. There is a general, highly configurable condenser as well as a set of common shorthands (also known as reduce expressions). The value returned is scalar, i.e.: a single scalar value or a record of values, reflecting the number of the input coverage's range type components.

Condensers are restricted by the coverage value nature: a *quantity* nature allows all operations whereas a *count* nature allows only min, max, or, and, and a *category* nature does not allow any summarization.

Requirement 60
A *condenseExpr* **shall** be a *generalCondenseExpr* (Subclause 7.5.2) or a *reduceExpr* (Subclause0).

Requirement 61
In a *condenseExpr* whenever a coverage range value participating in a condense operation is equal to one of the nil values of its coverage range field then this value **shall** be ignored and not contribute to the result of the condenser.
If all participating range values of a field are nil then the result **shall** be nil or, if the result type has no nil value defined, 0.

7.5.2 generalCondenseExpr

The general *generalCondenseExpr* consolidates the grid point values of a coverage along selected dimensions to a scalar value based on the condensing operation indicated. It iterates over a given domain while combining the result values of the *scalarExpr*s through the *condenseOpType* indicated. Admissible *condenseOpTypes* are the binary operations `+`, `*`, `max`, `min`, `and`, `or`, as well as the special operator `avg`.

The summation expression usingExpr can be a coverage or scalar. Both are defined separately below. Further, the special case of averaging is treated separately.

Requirement 62
A non-avg *generalCondenseExpr* over coverages **shall** be defined as below.

Let

op be a *condenseOpType* ≠ `'avg'`,
n be some *integer* with $n \geq 0$,
d be some *integer* with $d > 0$,
a_i be pairwise distinct *axisName*s for $1 \leq i \leq d$,
I_i be *intervalExpr*s for $1 \leq i \leq d$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
Dom be an *overExpr*,
f_i be pairwise distinct *fieldLabel*s for $1 \leq i \leq n$,
P be a *booleanExpr* possibly containing occurrences of a_i,
X be a *coverageExpr* possibly containing occurrences of a_i
D be a *coverageExpr*.

Where

for $f \in$ *eval*(*X*).*rangetype*.field*Labels*:
¬ (*eval*(*X*).*rangetype*.fields[*f*].*nature* = *count*
∧ *op* ∈ { `'+'`, `'*'`, `'and'`, `'or'`, `'avg'` })
∧
eval(*X*).*rangetype.fields*[*f*].*nature* ≠ *category*.

NOTE In case of value nature *quantity*, no restriction applies on *op*.

Then

For any *generalCondenseExpr S*
where

```
S   =   'condense' op
        'over' Dom
        ( 'where' P )?
        'using' X
Dom =   Dom' | Dom"
```

```
Dom'=    a1 '(' I1 ')' ',' … ',' ad '(' Id ')'
Dom"=    D '.' 'domain'
```

and

eval(D).*domain.axisLabels* ⊆ *eval*(C).*domain.axisLabels*,
for $a_i \in \langle a_1,\ldots,a_d \rangle$ = *eval*(D).*domain.axisLabels*:
eval(D).*domain.axes*[a_i].*lo* = lo_i,
eval(D).*domain.axes*[a_i].*hi* = hi_i

with

for $a_i \in \langle a_1,\ldots,a_d \rangle$ = *eval*(D).*domain.axisLabels*:
$lo_i \geq$ *eval*(X).*domain.axes*[a_i].*lo*,
$hi_i \leq$ *eval*(X).*domain.axes*[a_i].*hi*

eval(S)=$\langle f_1:v_1,\ldots,f_d:v_d \rangle$ is constructed as follows:

```
init := false;
for all a1 ∈ {lo1,… ,hi1}
  for all a2 ∈ {lo2,… ,hi2}
    …
      for all ad ∈ {lod,… ,hid}
        if P is present
        then
          let predicate P' be obtained from evaluating expression P
          by substituting all occurrences of ai by their current
          iteration assignment
        else
          P' := true
        fi;
        if eval(P')
        then
          let X' be the coverageExpr obtained from X by
          substituting all occurrences of ai by their current iteration
          assignment;
          v := eval(X') = <f1:v1,…,fd:vd>;
          for f∈<f1,…,fd>:
            if v.f ∉ eval(X).rangetype.fields[f].nil
            then
              if init
              then
                if data type of v.f is a complex type
                then
                  E.f := (complex double) E.f op v.f
                else
                  E.f := (double) E.f op v.f
                fi
              else
```

```
                        if data type of v.f is a complex type
                        then
                            E.f := (complex double) v.f
                        else
                            E.f := (double) v.f
                        fi;
                        init := true
                    fi
                fi
            endfor
        fi
    endfor
    …
    endfor
  endfor
return E
```

NOTE For simplicity, the algorithm is assuming that *C* evaluates to a single value or value record and does not contain any induced operation; however, it extends naturally into such cases.

Requirement 63

A non-avg *generalCondenseExpr* over scalars **shall** be defined as below.

Let

op be a *condenseOpType* ≠ 'avg',
d be some *integer* with *d*>0,
a_i be pairwise distinct *axisName*s for 1≤*i*≤*d*,
I_i be *intervalExpr*s for 1≤*i*≤*d* which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
Dom be an *overExpr*,
P be a *booleanExpr* possibly containing occurrences of a_i,
X be a *scalarExpr* possibly containing occurrences of a_i
D be a *coverageExpr*.

Then

For any *generalCondenseExpr S*
where

```
S    =   'condense' op
         'over' Dom
         ( 'where' P )?
         'using' X
Dom  =   Dom' | Dom"
Dom' =   a1 '(' I1 ')' ',' … ',' ad '(' Id ')'
Dom" =   D '.' 'domain'
```

and

eval(`D`).*domain.axisLabels* ⊆ *eval*(`X`).*domain.axisLabels*,
for $a_i \in \langle a_1,\ldots,a_d \rangle$ = *eval*(`D`).*domain.axisLabels*:
eval(`D`).*domain.axes*[a_i].*lo* = lo_i,
eval(`D`).*domain.axes*[a_i].*hi* = hi_i

eval(`S`) is constructed as follows:

```
init := false;
for all a1 ∈ {lo1,… ,hi1}
  for all a2 ∈ {lo2,… ,hi2}
    …
      for all ad ∈ {lod,… ,hid}
        if P is present
        then
          let predicate P′ be obtained from evaluating expression P
          by substituting all occurrences of ai by their current
          iteration assignment
        else
          P' := true
        fi;
        if eval(P')
        then
          let S′ be the scalarExpr obtained from S by
          substituting all occurrences of ai by their current iteration
          assignment;
          v := eval(S′);
          if init
          then
            if data type of v is a complex type
            then
              E := (complex double) E op v
            else
              E := (double) E.f op v
            fi
          else
            if data type of v.f is a complex type
            then
              E := (complex double) v
            else
              E := (double) v
            fi;
            init := true
          fi
        fi
      endfor
    …
```

```
    endfor
  endfor
return E
```

NOTE In this version, only condensers over atomic values are supported, not records.

Requirement 64

A *generalCondenseExpr* with *op*='avg' over coverages **shall** be defined as follows:

Let

n be some *integer* with $n \geq 0$,
d be some *integer* with $d > 0$,
a_i be pairwise distinct *axisName*s for $1 \leq i \leq d$,
I_i be *intervalExpr*s for $1 \leq i \leq d$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
Dom is an *overExpr*,
f_i be pairwise distinct *fieldLabel*s for $1 \leq i \leq n$,
P be a *booleanExpr* possibly containing occurrences of a_i,
X be a *coverageExpr* possibly containing occurrences of a_i
D be a *coverageExpr*.

Then

For any *generalCondenseExpr S*
where

```
S    =   'condense' 'avg'
         'over' Dom
         ( 'where' P )?
         'using' X
Dom  =   Dom' | Dom"
Dom' =   a1 '(' I1 ')' ',' … ',' ad '(' Id ')'
Dom" =   D '.' 'domain'
```

and

eval(*D*).*domain.axisLabels* $\subseteq$ *eval*(*X*).*domain.axisLabels*,
for $a_i \in \langle a_1,\ldots,a_d \rangle$ = *eval*(*D*).*domain.axisLabels*:
eval(*D*).*domain.axes*[a_i].*lo* = lo_i,
eval(*D*).*domain.axes*[a_i].*hi* = hi_i

with
for $a_i \in \langle a_1,\ldots,a_d \rangle$ = *eval*(*D*).*domain.axisLabels*:
$lo_i \geq$ *eval*(*X*).*domain.axes*[a_i].*lo*,
$hi_i \leq$ *eval*(*X*).*domain.axes*[a_i].*hi*

eval(*S*) is constructed as follows[8]:

[8] NaN = Not-a-Number as defined in IEEE 754

```
sum := eval( condense + over Dom [ where P ] using X );
cnt := eval( condense + over Dom [ where P ] using 1 );
if cnt = 0
then
        if data type of sum or cnt is a complex type
        then
            E := (complex double) (NaN,NaN)
        else
            E := (double) NaN
        fi
else
        if data type of sum or cnt is a complex type
        then
            E := (complex double) sum / cnt
        else
            E := (double) sum / cnt
        fi
fi;
return E
```

Requirement 65

A *generalCondenseExpr* with *op*='avg' over scalars **shall** be defined as follows:

Let

n be some *integer* with $n \geq 0$,
d be some *integer* with $d > 0$,
a_i be pairwise distinct *axisName*s for $1 \leq i \leq d$,
I_i be *intervalExpr*s for $1 \leq i \leq d$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$,
Dom be an *overExpr*,
P be a *booleanExpr* possibly containing occurrences of a_i,
X be a *scalarExpr* possibly containing occurrences of a_i
D be a *coverageExpr*.

Then

For any *generalCondenseExpr S*
where

```
S    =   'condense' 'avg'
         'over' Dom
         ( 'where' P )?
         'using' X
Dom  =   Dom' | Dom"
Dom' =   a1 '(' I1 ')' ',' … ',' ad '(' Id ')'
Dom" =   D '.' 'domain'
```

and
for $a_i \in \langle a_1,\ldots,a_d \rangle$ = *eval*(*D*).*domain.axisLabels*:
eval(*D*).*domain.axes*[a_i].*lo* = lo_i,
eval(*D*).*domain.axes*[a_i].*hi* = hi_i

eval(*S*) is constructed as follows:

```
sum := eval( condense + over Dom [ where P ] using X );
cnt := eval( condense + over Dom [ where P ] using 1 );
if cnt = 0
then
        if data type of sum or cnt is a complex type
        then
            E := (complex double) (NaN,NaN)
        else
            E := (double) NaN
        fi
else
        if data type of sum or cnt is a complex type
        then
            E := (complex double) sum / cnt
        else
            E := (double) sum / cnt
        fi
fi;
return E
```

7.5.3 reduceExpr

A *reduceExpr* element derives a summary value from the coverage passed; in this sense it "reduces" a coverage to a scalar value.

Requirement 66
A *reduceExpr* **shall** use either an `add`, `avg`, `min`, `max`, `count`, `some`, or `all` operation as per Table 4.

Table 4 – reduceExpr definition via generalCondenseExpr

`$A` and `$B` are expressions evaluating to a coverage with domain < D_1,…,D_d>, axis labels < a_1,…,a_d>, a single range field *f* of nature *quantity*, and a numeric type in `$A` and Boolean in `$B`.

reduceExpr definition	Meaning

add($A) = **condense** + **over** $A.$D_1$, …, $A.$D_d$ **using** $A[$A.D_1, …, $A.$D_d$]	sum over all values in *$a*
avg($A) = add($A) / | $A.domain.pos |	Average of all values in *$a*
min($A) = **condense** min **over** $A.$D_1$, …, $A.$D_d$ **using** $A[$A.D_1, …, $A.$D_d$]	Minimum of all values in *$a*
max($A) = **condense** max **over** $A.$D_1$, …, $A.$D_d$ **using** $A[$A.D_1, …, $A.$D_d$]	Maximum of all values in *$a*
count($B) = **condense** + **over** $B.$D_1$, …, $B.$D_d$ **where** $B[$B.D_1, …, $B.$D_d$] **using** 1	Number of *true* values in *$b*
some($B) = **condense** or **over** $B.$D_1$, …, $B.$D_d$ **using** $B[$B.D_1, …, $B.$D_d$]	is there any point in *$b* with value *true*?
all($B) = **condense** and **over** $B.$D_1$, …, $B.$D_d$ **using** $B[$B.D_1, …, $B.$D_d$]	do all points of *$b* have value *true*?

7.5.4 scalarExpr

Requirement 67
A *scalarExpr* **shall** be either a *booleanScalarExpr* (Subclause 7.5.5), or a *numericScalarExpr* (Subclause 7.5.6) or a *stringScalarExpr* (Subclause 7.5.6), or *getExpr* (Subclause 7.5.8).

NOTE As such, such an expression returns a (simple or composite) result value, that is: not a coverage.

7.5.5 booleanScalarExpr

Requirement 68
A *booleanScalarExpr* **shall** be a *scalarExpr* whose result type is Boolean. Operations **shall** be the well-known Boolean functions `and`, `or`, `xor`, and `not` , arithmetic

comparison (>, <, >=, <=, =, !=) on strings and numbers, and parenthesing, all bearing the well-known standard semantics.

7.5.6 numericScalarExpr

Requirement 69
A *numericScalarExpr* **shall** be a *scalarExpr* whose result type is numeric (i.e., integer, float, or complex).

Example Numeric constants and numerically-valued metadata retrieval functions deliver numbers.

Operations provided are the well-known arithmetic, trigonometric, and exponential / logarithmic operations bearing the standard mathematical semantics. The rounding function, `round()`, rounds a real (not complex) number to the next integer number towards zero. A *condenseExpr* (see Subclause 7.5) derives a summary value from a coverage expression.

7.5.7 stringScalarExpr

Requirement 70
A *stringScalarExpr* **shall** be a *scalarExpr* whose result type is character string of length greater or equal to zero.

Example *IdentifierExpr*s deliver strings.

7.5.8 getExpr

A *getExpr* extracts domain information from a coverage: its domain in total, a specific axis, or lower/upper bound along an axis. The set of functions and their syntax are tuned for convenient use in defining the coordinate iteration of constructor and condenser (Subclauses 7.2.3 and 7.5.2).

Requirement 71
A *getExpr* **shall** be defined as below.

Let

C be a *coverageExpr*,
a be an *axisName*,
f be a *fieldName*.

Then

for any *getExpr G*
where *G* is one of:

G_{id} = *C* '.' id'

```
G_domain     = C '.' 'domain'
G_crs        = C '.' 'domain' '.' 'crs'
G_axes       = C '.' 'domain' '.' 'axes'
G_axis       = C '.' 'domain' '.' 'axes'.' a
G_axistype   = C '.' 'domain' '.' 'axes'.' a '.' 'axistype'
G_res        = C '.' 'domain' '.' 'axes'.' a '.' 'res'
G_axisuom    = C '.' 'domain' '.' 'axes'.' a '.' 'uom'
G_axispos    = C '.' 'domain' '.' 'axes'.' a '.' 'pos'
G_int        = C '.' 'domain' '.' 'axes'.' a '.' 'int'
G_lo         = C '.' 'domain' '.' 'axes' '.' a '.' 'lo'
G_hi         = C '.' 'domain' '.' 'axes' '.' a '.' 'hi'
G_index      = C '.' 'domain' '.' 'axes' '.' a '.' 'index'
G_lo         = C '.' 'domain' '.' 'axes' '.' a '.' 'index'
                  '.' 'lo'
G_hi         = C '.' 'domain' '.' 'axes' '.' a '.' 'index'
                  '.' 'hi'

G_rangetype  = C '.' 'rangetype'
G_fields     = C '.' 'rangetype' '.' 'fields'
G_field      = C '.' 'rangetype' '.' 'fields' '.' f
G_uom        = C '.' 'rangetype' '.' 'fields' '.' f '.' 'uom'
G_nil        = C '.' 'rangetype' '.' 'fields' '.' f '.' 'nil'
G_nature     = C '.' 'rangetype' '.' 'fields' '.' f
                  '.' 'nature'

G_metadata   = C '.' 'metadata'
```

eval(G) is defined as follows:

Coverage constituent
eval(G_{id}) = *eval*(C).*id*
eval(G_{domain}) = *eval*(C).*domain*
eval(G_{crs}) = *eval*(C).*domain.crs*
eval(G_{axes}) = *eval*(C).*domain.axisLabels*
eval(G_{axis}) = *eval*(C).*domain.axes*[*eval*(a)]
eval($G_{axistype}$) = *eval*(C).*domain.axes*[*eval*(a)].*axisType*
eval(G_{lo}) = *eval*(C).*domain.axes*[*eval*(a)].*lo*
eval(G_{hi}) = *eval*(C).*domain.axes*[*eval*(a)].*hi*
eval(G_{res}) = *eval*(C).*domain.axes*[*eval*(a)].*res* if a is regular
eval($G_{axisuom}$) = *eval*(C).*domain.axes*[*eval*(a)].*uom* if a is regular or irregular

eval($G_{axispos}$) = *eval*(C).*domain.axes*[*eval*(a)].*pos* if a is irregular
eval(G_{index}) = *eval*(C).*domain.axes*[*eval*(a)].*index*
eval($G_{indexlo}$) = *eval*(C).*domain.index*[*eval*(a)].*index.lo*
eval($G_{indexhi}$) = *eval*(C).*domain.index*[*eval*(a)].*index.hi*
eval($G_{axisint}$) = *eval*(C).*domain.axes*[*eval*(a)].*int* if a is regular or irregular
eval($G_{rangetype}$) = *eval*(C).*rangetype*
eval(G_{fields}) = *eval*(C).*rangetype.fields*
eval(G_{field}) = *eval*(C).*rangetype.fields*[*eval*(f)]
eval(G_{uom}) = *eval*(C).*rangetype.fields*[*eval*(f)].*uom*
eval(G_{nil}) = *eval*(C).*rangetype.fields*[*eval*(f)].*nil*
eval(G_{nature}) = *eval*(C).*rangetype.fields*[*eval*(f)].*nature*
eval(G_{int}) = *eval*(C).*rangetype.fields*[*eval*(f)].*int*
eval($G_{metadata}$) = *eval*(C).*metadata*

Example The following expression yields the *Lat* extent of coverage $C, ready for use in iterations:

```
$C.domain.Lat
```

NOTE Not all probing functions are "lifted" to syntax level, in particular not those delivering redundant information, such as *c.domain.uomLabels.*

For axis and range field access shorthands are defined: middle elements can be omitted as long as the resulting expression may be associated with the original expression unambiguously.

Requirement 72
In a *getExpr* the equivalent shorthands **shall** be defined as in Table 5, allowing to omit the components *axes* and *fields*.

Table 5 – *getExpr* shorthands.

Get expression	Shorthand for coverage *C*, axis label *a*, range field label *f*
C.domain.axes.*a*	*C*.domain.*a*
C.domain.axes.*a*.axistype	*C*.domain.*a*.axistype
C.domain.axes.*a*.uom	*C*.domain.*a*.uom

C.domain.axes.*a*.res	*C*.domain.*a*.res
C.domain.axes.*a*.pos	*C*.domain.*a*.pos
C.domain.axes.*a*.lo	*C*.domain.*a*.lo
C.domain.axes.*a*.hi	*C*.domain.*a*.hi
C.domain.axes.*a*.index	*C*.domain.*a*.index
C.domain.axes.*a*.index.lo	*C*.domain.*a*.index.lo
C.domain.axes.*a*.index.hi	*C*.domain.*a*.index.hi
C.domain.axes.*a*.int	*C*.domain.*a*.int
C.rangetype.fields.*f*	*C*.rangetype.*f*
C.rangetype.fields.*f*.uom	*C*.rangetype.*f*.uom
C.rangetype.fields.*f*.nil	*C*.rangetype.*f*.nil
C.rangetype.fields.*f*.nature	*C*.rangetype.*f*.nature
C.rangetype.fields.*f*.int	*C*.rangetype.*f*.int

The result of a *getExpr* can be a query output, represented as a printable string.

Requirement 73
The output of a *getExpr* when used directly in a **return** clause **shall** be a string adhering to the *coverageConstructorExpr* syntax.

Example The coverage domain can be retrieved as

```
for $c in ( MapCoverage )
return $c.domain
```

The output might look as follows:

```
crs "[EPSG:4326],[OGC:DateTime]" axes
Lat: regular (10:30) resolution -0.5
  interpolation nearest, linear,
Lon: regular (10:30) resolution 0.5
  interpolation nearest, linear,
Date: irregular
  ( "2017-01-01", "2017-02-01", "2017-07-01", "2017-11-01" )
  interpolation linear
```

7.6 Expression evaluation

This Sublause defines additional rules for expression evaluation.

7.6.1 Character encoding

The character set used depends on the embedding protocol.

Requirement 74
The character set of a *processingExpr* **shall** follow the rules of the embedding protocol binding context; default is US ASCII.

Example An extra parameter in a request transporting the query may specify the character set, such as Unicode.

A quoted string may contain any character, even national special characters and non-printable characters, as long as these are properly encoded following http URL rules.

Requirement 75
Non-printable characters in a *processingExpr* **shall** be represented according to http rules.

7.6.2 Evaluation sequence and operator precedence

Requirement 76
Operators **shall** have the following precedence (listed in descending strength of binding):

- range field selection, trimming, slicing
- unary –
- unary arithmetic, trigonometric, and exponential functions
- `*`, `/`
- `+`, `-`
- `<`, `<=`, `>`, `>=`, `!=`, `=`
- `and`
- `or`, `xor`
- `:` (interval constructor), condense, coverage constructor
- overlay, switch
- In all remaining cases, evaluation **shall** be done left to right.

7.6.3 Nesting

Requirement 77
A *processingExpr* **shall** allow nesting all operators, constructors, and functions to arbitrary depth, provided that each sub-expression's result type matches the required type at the position where the sub-expression occurs.

NOTE This holds without limitation for all arithmetic, Boolean, string, and coverage valued expressions.

7.6.4 Parentheses

A **processingExpr** may contain parentheses to enforce a particular evaluation sequence.

Requirement 78
Parentheses enforcing evaluation sequence in a *processingExpr* **shall** be defined as below.

Let

C_1 be a *coverageExpr*.

Then

For any *coverageExpr* C_2
where
C_2 = '(' C_1 ')'

eval(C_2) = *eval*(C_1).

Example `$c * ( $c > 0 )`

7.6.5 Range type compatibility and coercion

Frequently, coverages processed - for example, when combined with other coverages - are of different data types. Whenever possible, this is remedied automatically by adjusting the data types to match. This procedure is called type coercion.

NOTE See also the explicit type change through a *castExpr* (Subclause 7.4.7).

Type coercion only operates on the *def* component of a range field; behavior of other components, like *uom*, *nil*, and *int* is governed by other requirements.

Requirement 79
If use of a *coverageExpr* C in some expression *expr*(C) with result type $T \in$ Table 1 results in $T \neq C$.*rangetype.fields*[f].*def* for some field $f \in C$.*rangetype.fieldLabels*, then *coercion of* C.*rangetype.fields*[f].*def to* T **shall** take place, defined as follows:

- look up C.*rangetype.fields*[f].*def* as per Requirement 80;
- change it to its right-hand neighbour type or, if it is the last type in line, by the first type of the next line;
- repeat until it is equal to T, or the end of Table 6 is reached and an error is raised.

Example For three single-field coverages `$F`, `$I`, and `$B` with range types `float`, `integer`, and `boolean`, resp., the result type of the following expression is `float`:

```
$F + $I + $B
```

Requirement 80
On both arguments to binary operations type coercion **shall** be applied until both argument types are equal, thereby determining the result coverage's data type.

Table 6 – Type extension sequence.

```
boolean > char
char > boolean
boolean > unsigned char
unsigned char > boolean
char > short
char > unsigned short
unsigned char > short
unsigned char > unsigned short
short > long
short > unsigned long
unsigned short > long
unsigned short > unsigned long
long > long long
long > unsigned long long
unsigned long > long long
unsigned long > unsigned long long
long long > float
float > double
float > complex
double > complex
complex > complex double
complex char > complex short
complex short > complex int
complex int > complex
complex > complex double
```

Requirement 81
Whenever rounding from floating-point to integer numbers is required, rounding **shall** be applied towards zero.

7.6.6 Variable scope

Variables defined are valid and can be used in the scope given by the defining operation. If, in a nested scope, a variable with a name already existing in the nested scope is defined then it *shadows* the outer variable, making it inaccessible within the nested scope.

Requirement 82
A variable **shall** have its scope given as follow:

- If defined in the *for* clause of a *processingExpr*, the variable **can** be used in the *let* an *where* and *return* clauses;
- If defined in the *let* clause of a *processingExpr*, the variable **can** be used in the *where* and *return* clauses;
- If defined as *axisName* in the *domainSetExpr* of a *coverageConstructorExpr*, the variable **can** be used in the *rangeSetExpr*.

- If defined as an axis name in the *overExpr* of a *generalCondenseExpr*, the ariable **can** be used in the *where* and *using* clauses.

Requirement 83
A *getDomainExpr* implicitly defines the axis names of its domain, and these **can** be used in the same way as explicitly defined axis variables.

Requirement 84
If a variable is defined and another variable with same name is defined in the embedding scope then in the scope of the inner variable the inner variable **can** be used and the outer variable **cannot** be used (shadowing).

7.6.7 Metadata

Requirement 85
An implementation **may** choose to not return the complete metadata component of a coverage, or none of it at all.

NOTE A possible reason is to differentiate between global metadata (which are valid for the whole coverage and get delivered always) and local metadata (which are valid for a part of a coverage and may get only delivered in a subsetting when the area of their validity is retrieved).

7.6.8 Evaluation exceptions

Not all syntactically valid *processingExpr* expressions are semantically admissible. Possible errors include:

- A cast is requested or implicitly needed, but not possible.
- A numeric overflow condition, such as in summation or float to int conversion.
- A path expression attempts to retrieve coverage names, but fails for some external reason;
- Access rights or quota are violated.

Requirement 86
Whenever a *processingExpr* cannot be evaluated by a server, an error **shall** be reported.

Example Syntax error: `$C $C`
Semantic error (illegal trigonometric argument): `arcsin( 2 )`
Valid only if coverage bound to `$C` exists and has a quantity field `red`:
`$C.red*2.5`

NOTE No detailed exception handling is defined here as this is the task of a concrete protocol binding of a WCPS service, such as WCS with GET/KVP, XML/POST, SOAP, or OAPI-Coverages.

Annex A
(normative)

Abstract Test Suite

The Abstract Test Suite for WCPS defines a single conformance class, WCPS, which is also the core conformance class.

Test:

- For every language element and every requirement, build a sample query based on coverages existing in the system under test.
- Test queries on 1D through 5D coverages in the system under test, including coverage joins.
- Check for correct result.
- Test succeeds if every query responds in accordance with the standard.

Annex B
(normative)

WCPS Expression Syntax

B.1 Overview

This Annex defines the WCPS expression syntax using common EBNF syntax using "|" for alternative choices, "(…)?" for optional elements, and "(…)*" for optional repetitions.

Tokens enclosed in single quotes represent literals which appear "as is" in a valid WCPS expression ("terminal symbols"), all other tokens represent either sub-expressions to be substituted according to the grammar production rules ("non-terminals") or terminal symbol classes like identifiers, strings, and numbers as listed at the end of this Annex. All terminals, except strings, are case insensitive. Whitespace characters (blank, tabulator, newline) **may** appear between tokens more than once.

NOTE Between language tokens (such as "for") and names there must be at least one whitespace character, whereas between names and non-alphanumeric tokens (such as opening parenthesis, "("), no whitespace is required.

Requirement 87
A WCPS query **shall** adhere to the syntax defined in this Annex B, with the *processCoveragesExpr* non-terminal as the start of the production system.

B.2 Queries

```
processCoveragesExpr ::=
        'for' variableName 'in' '(' coverageList ')'
            ( ',' variableName 'in' '(' coverageList ')' )*
        ( letExpr )?
        ( 'where' booleanScalarExpr )?
        'return' processingExpr

coverageList ::=
        coverageOrXpath ( ',' coverageOrXpath )*

coverageOrXpath ::=
          coverageName
        | xpathExpr

letExpr ::=
        'let' letBinding ( listSeparator letBinding )*

letBinding ::=
        variableName ':=' letAssignment

letAssignment ::=
          coverageExpr
```

```
        | '[' trimOrSliceList ']'
        | rangeExpr
        | getExpr
        | constant

processingExpr ::=
          encodedCoverageExpr
        | scalarExpr
        | polygonizeExpr

encodedCoverageExpr ::=
        'encode' '(' coverageExpr ',' formatName
                     ( ',' extraParams )? ')'

coverageExpr ::=
          coverageConstructorExpr
        | domainOpExpr
        | rangeExpr
        | decodeCoverageExpr

decodeCoverageExpr  ::=
        'decode' '(' variableName ( ',' extraParams )? ')'
```

B.3 Coverage Constructor Expressions

```
coverageConstructorExpr ::=
        ( 'coverage' ( coverageTypeName )? coverageName )?
        ( domainSetExpr )?
        ( rangeTypeExpr )?
        rangeSetExpr
        ( metadataExpr )?

domainSetExpr ::=
        'domain' ( 'set' )? domainExpr ( listSeparator )?

domainExpr ::=
          ( 'crs' stringConstant 'axes' )? axisList
        | getDomainExpr

axisList ::=
        axisExpr ( listSeparator axisExpr )*

axisExpr ::=
          axisName ':' axisdefExpr
          ( uomExpr )? ( intExpr )?
        | getDomainExpr

axisdefExpr ::=
          'index' '(' indexExpr ':' indexExpr ')'
        | 'regular' '(' arithmeticExpr ':' arithmeticExpr ')'
          'resolution' arithmeticExpr
        | 'irregular' '(' arithmeticExpr ( ',' arithmeticExpr )*
')'
```

```
uomExpr ::=
        'uom' stringConstant

intExpr ::=
        'interpolation'
        interpolationMethod ( ',' interpolationMethod )*

interpolationMethod ::=
          'nearest'
        | 'linear'
        | 'quadratic'
        | 'cubic'
        | nameOrString

rangeTypeExpr ::=
        'range' 'type' rangeComponent ( ',' rangeComponent )*
        ( listSeparator )?

rangeSetExpr ::=
        'range' ( 'set' )? rangeExpr

metadataExpr ::=
        'metadata' ( stringConstant )

coverageEnumExpr ::=
        'coverage' ( coverageTypeName )? coverageName
        domainSetExpr
        rangeTypeExpr
        rangeEnumExpr
        ( metadataExpr )?

rangeEnumExpr ::=
        '<' scalarExpr ( listSeparator scalarExpr )* '>'
```

B.4 Range Type Expressions

```
rangeComponent ::=
        variable  ':' ( valueNature )? definition ( uomExpr )?
        ( nilExpr )? ( intExpr )?

definition ::=
          'boolean'                | 'bool'
        | 'char'                   | 'int8'
        | 'unsigned' 'char'        | 'uint8'
        | 'short'                  | 'int16'
        | 'unsigned' 'short'       | 'uint16'
        | 'long'                   | 'int32'
        | 'unsigned' 'long'        | 'uint32'
        | 'long' 'long'            | 'int64'
        | 'unsigned' 'long' 'long' | 'uint64'
        | 'float'                  | 'float32'
        | 'double'                 | 'float64'
        | 'complex' 'short'        | 'CInt16'
        | 'complex' 'int'          | 'CInt32'
```

```
        | 'complex' ( 'float' )?   | 'CFloat32'
        | 'complex' 'double'       | 'CFloat64'

valueNature ::=
          'quantity' | ' numeric' | 'count'
        | 'category' | 'categorial'

nilExpr ::=
        'nil' nilConstant ( ',' nilConstant )*
```

B.5 Domain Expressions

```
domainOpExpr ::=
          trimOrSliceExpr
        | extendExpr
        | scaleExpr
        | crsTransformExpr

trimOrSliceExpr ::=
          coverageExpr '[' trimOrSliceList ']'
        | coverageExpr '[' '[' trimOrSliceList ']' ']'

trimOrSliceList ::=
          trimOrSliceExpr ( ',' trimOrSliceExpr )*

trimOrSliceExpr ::=
          trimExpr
        | sliceExpr

trimExpr ::=
          axisName ( '.' 'index' )?
             '(' coordinateExpr ':' coordinateExpr ')'
        | axisName ( '.' 'index' )? '(' getAxisExpr ')'
        | axisName ( '.' 'index' )? '(' getIndexExpr ')'
        | getDomainExpr

sliceExpr ::=
        axisName ( '.' 'index' )? '(' coordinateExpr ')'

arithmeticExpr ::=
          arithmeticExpr '+' arithmeticTerm
        | arithmeticExpr '-' arithmeticTerm
        | arithmeticTerm

arithmeticTerm ::=
          arithmeticTerm '*' arithmeticFactor
        | arithmeticTerm '/' arithmeticFactor
        | arithmeticFactor

arithmeticFactor ::=
          'round' '(' arithmeticExpr ')'
        | '(' arithmeticExpr ')'
        | scalarExpr
```

```
        | stringConstant
        | numericalConstant
        | getHiExpr
        | getLoExpr
        | getIndexLoExpr
        | getIndexHiExpr

extendExpr ::=
        'extend' '(' coverageExpr ',' '['
        axisName '(' arithmeticExpr ':' arithmeticExpr ')'
        ( axisName '(' arithmeticExpr ':' arithmeticExpr ')' )*
        ']' ')'

scaleExpr ::=
          scaleToExtentExpr
        | scaleByFactorsExpr
        | scaleByCommonFactorExpr

scaleToExtentExpr ::=
        'scale' '(' coverageExpr ','
        '[' trimExpr ',' ( ',' trimExpr)? ')'
        ( ',' interpolationMethod )?
        ')'

scaleByFactorsExpr ::=
        'scale' '(' coverageExpr ','
        '[' sliceExpr ',' ( ',' sliceExpr)?
        ( ',' interpolationMethod )?
        ')'

scaleByCommonFactorExpr ::=
        'scale' '(' coverageExpr ',' scalarExpr
                ( ',' interpolationMethod )?
        ')'

crsTransformExpr ::=
        'crsTransform' '(' coverageExpr ( ',' crsName )? ')'

polygonExpr ::=
          clipExpr
        | clipCurtainExpr
        | clipCorridorExpr

clipExpr ::=
          clipPolygonExpr
        | clipMultipolygonExpr
        | clipLinestringExpr

clipPolygonExpr ::=
        'clip' '(' coverageExpr ',' polygonExpr ')'

polygonExpr ::=
        'polygon' '('
```

```
          '(' vertexList ')' ( ',' '(' vertexList ')' )*
        ')'

clipMultipolygonExpr ::=
        'clip' '(' coverageExpr ',' multipolygonExpr ')'

multipolygonExpr ::=
        'multipolygon' '('
          '(' polygonExpr ')' ( ',' '(' polygonExpr ')' )*
        ')'

clipLinestringExpr ::=
        'clip' '(' coverageExpr ',' linestringExpr ')'

linestringExpr ::=
        'linestring' '(' vertexList ')'

clipCurtainExpr ::=
          'curtain' '(' coverageExpr ',' linestringExpr ','
          axisListString ')'
        | 'curtain' '(' coverageExpr ',' polygonExpr ','
          axisListString ')'

axisListString ::=
        stringLiteral

clipCorridorExpr ::=
          'corridor' '(' projectionExpr ',' linestringExpr ','
          polygonExpr ',' axisListString ')'
        | 'corridor' '(' projectionExpr ',' linestringExpr ','
          linestringExpr',' axisListString ')'

vertexList ::=
        vertexTuple ( ',' vertexTuple )*

vertexTuple ::=
        '(' coordinateExpr ( coordinateExpr )* ')'

polygonizeExpr ::=
        'polygonize' '(' coverageExpr ',' stringConstant
        ( ',' integerConstant )? ')'
```

B.6 Range Expressions

```
rangeExpr ::=
          unaryInducedExpr
        | binaryInducedExpr
        | booleanExpr
        | switchExpr
        | rangeConstructorExpr
        | condenseExpr
        | '(' rangeExpr ')'
```

```
unaryInducedExpr ::=
          unaryArithmeticExpr
        | exponentialExpr
        | trigonometricExpr
        | booleanExpr
        | castExpr
        | fieldExpr
        | rangeConstructorExpr

unaryArithmeticExpr ::=
          '+' coverageExpr
        | '-' coverageExpr
        | 'sqrt' '(' coverageExpr ')'
        | 'abs' '(' coverageExpr ')'
        | 'bit' '(' coverageExpr ',' indexExpr ')'
        | 'round' '(' coverageExpr ')'
        | coverageExpr '.' 're'
        | coverageExpr '.' 'im'
        | numericalConstant

exponentialExpr ::=
          'exp' '(' coverageExpr ')'
        | 'log' '(' coverageExpr ')'
        | 'ln' '(' coverageExpr ')'

trigonometricExpr ::=
         'sin' '(' coverageExpr ')'
        | 'cos' '(' coverageExpr ')'
        | 'tan' '(' coverageExpr ')'
        | 'sinh' '(' coverageExpr ')'
        | 'cosh' '(' coverageExpr ')'
        | 'tanh' '(' coverageExpr ')'
        | 'arcsin' '(' coverageExpr ')'
        | 'arccos' '(' coverageExpr ')'
        | 'arctan' '(' coverageExpr ')'

binaryInducedExpr ::=
          binaryInducedExpr '+' binaryInducedTerm
        | binaryInducedExpr '-' binaryInducedTerm
        | binaryInducedTerm

binaryInducedTerm ::=
          binaryInducedExpr '*' binaryInducedFactor
        | binaryInducedExpr '/' binaryInducedFactor
        | binaryInducedFactor

binaryInducedFactor ::=
          binaryInducedFactor 'overlay' binaryInducedUnary
        | ( 'arctan2' | 'atan2' )
          '(' binaryInducedFactor ',' binaryInducedUnary ')'
        | numericalConstant
        | '(' binaryInducedExpr ')'
```

```
booleanExpr ::=
          booleanExpr 'or' booleanTerm
        | booleanExpr 'xor' booleanTerm
        | booleanTerm

booleanTerm ::=
          booleanTerm 'and' booleanFactor
        | booleanFactor

booleanFactor ::=
          coverageExpr compOp coverageExpr
        | 'not' booleanExpr
        | booleanConstant
        | '(' booleanExpr ')'

compOp ::=
          '='
        | '!='
        | '>'
        | '>='
        | '<'
        | '<='

fieldExpr ::=
        coverageExpr '.' fieldName

rangeConstructorExpr ::=
          ( 'struct' )? '{'
          fieldName ':' coverageExpr
          ( listSeparator fieldName ':' coverageExpr )* '}'
        | '{' coverageExpr ( listSeparator coverageExpr )* '}'

castExpr ::=
        '(' rangeType ')' coverageExpr

switchExpr ::=
        'switch'
        'case' coverageExpr 'return' coverageExpr
        *( 'case' coverageExpr 'return' coverageExpr )
        'default' 'return' coverageExpr
```

B.7 Scalar Expressions

```
scalarExpr ::=
          condenseExpr
        | constant
```

B.8 Condenser Expressions

```
condenseExpr ::=
          generalCondenseExpr
        | reduceExpr
```

```
generalCondenseExpr ::=
        'condense' condenseOpType
        'over' overExpr ( listSeparator )?
        ( 'where' booleanScalarExpr )?
        'using' usingExpr

condenseOpType ::=
          '+'
        | '*'
        | 'max'
        | 'min'
        | 'and'
        | 'or'
        | 'avg'

overExpr ::=
       dimensionIntervalElement ( ',' dimensionIntervalElement )*
        | getDomainExpr

usingExpr ::=
          coverageExpr
        | scalarExpr

reduceExpr ::=
          'all' '(' coverageExpr ')'
        | 'some' '(' coverageExpr ')'
        | 'count' '(' coverageExpr ')'
        | 'add' '(' coverageExpr ')'
        | 'avg' '(' coverageExpr ')'
        | 'min' '(' coverageExpr ')'
        | 'max' '(' coverageExpr ')'
```

B.9 Get Expressions

```
getExpr ::=
          getIdExpr
        | getDomainExpr
        | getCrsExpr
        | getAxesExpr
        | getAxisExpr
        | getAxisTypeExpr
        | getResExpr
        | getAxisUomExpr
        | getAxisPosExpr
        | getAxisIntExpr
        | getLoExpr
        | getHiExpr
        | getIndexExpr
        | getIndexLoExpr
        | getIndexHiExpr
        | getRangetypeExpr
        | getFieldsExpr
        | getFieldExpr
        | getUomExpr
```

```
        | getNilExpr
        | getNatureExpr
        | getIntExpr
        | getMetadataExpr

getIdExpr ::=
       coverageExpr '.' 'id'

getDomainExpr ::=
       coverageExpr '.' 'domain'

getCrsExpr ::=
       coverageExpr '.' 'domain' '.' 'crs'

getAxesExpr ::=
       coverageExpr '.' 'domain' '.' 'axes'

getAxisExpr ::=
         coverageExpr '.' 'domain' '.' 'axes '.' axisName
       | coverageExpr '.' 'domain' '.' axisName

getAxisTypeExpr ::=
         coverageExpr '.' 'domain' '.' 'axes'.' axisName
         '.' 'axistype'
       | coverageExpr '.' 'domain' '.' axisName '.' 'axistype'

getResExpr ::=
         coverageExpr '.' 'domain' '.' 'axes'.' axisName
         '.' 'res'
       | coverageExpr '.' 'domain' '.' axisName '.' 'res'

getAxisUomExpr ::=
         coverageExpr '.' 'domain' '.' 'axes'.' axisName
         '.' 'uom'
       | coverageExpr '.' 'domain' '.' axisName '.' 'uom'

getAxisPosExpr ::=
         coverageExpr '.' 'domain' '.' 'axes'.' axisName
         '.' 'pos'
       | coverageExpr '.' 'domain' '.' axisName '.' 'pos'

getLoExpr ::=
         coverageExpr '.' 'domain' '.' 'axes '.' axisName
         '.' 'lo'
       | coverageExpr '.' 'domain' '.' axisName '.' 'lo'

getHiExpr ::=
         coverageExpr '.' 'domain' '.' 'axes '.' axisName
         '.' 'hi'
       | coverageExpr '.' 'domain' '.' axisName '.' 'hi'

getIndexExpr
         coverageExpr '.' 'domain' '.' 'axes '.' axisName
```

```
          '.' 'index'
        | coverageExpr '.' 'domain' '.' axisName '.' 'index'

getIndexLoExpr
          coverageExpr '.' 'domain' '.' 'axes '.' axisName
          '.' 'index' '.' 'lo'
        | coverageExpr '.' 'domain' '.' axisName
          '.' 'index' '.' 'lo'

getIndexHiExpr
          coverageExpr '.' 'domain' '.' 'axes '.' axisName
          '.' 'index' '.' 'hi'
        | coverageExpr '.' 'domain' '.' axisName
          '.' 'index' '.' 'hi'

getAxisIntExpr ::=
          coverageExpr '.' 'domain' '.' 'axes'.' axisName
          '.' 'int'
        | coverageExpr '.' 'domain' '.' axisName '.' 'int'

getRangetypeExpr ::=
        coverageExpr '.' 'rangetype'

getFieldsExpr ::=
        coverageExpr '.' 'rangetype' '.' 'fields'

getFieldExpr ::=
          coverageExpr '.' 'rangetype' '.' 'fields' '.' fieldName
        | coverageExpr '.' 'rangetype' '.' fieldName

getUomExpr ::=
          coverageExpr '.' 'rangetype' '.' 'fields' '.' fieldName
          '.' 'uom'
        | coverageExpr '.' 'rangetype' '.' fieldName '.' 'uom'

getNilExpr ::=
          coverageExpr '.' 'rangetype' '.' 'fields' '.' fieldName
          '.' 'nil'
        | coverageExpr '.' 'rangetype' '.' fieldName '.' 'nil'

getNatureExpr ::=
          coverageExpr '.' 'rangetype' '.' 'fields' '.' fieldName
          '.' 'nature'
        | coverageExpr '.' 'rangetype' '.' fieldName '.' 'nature'

getIntExpr ::=
          coverageExpr '.' 'rangetype' '.' 'fields' '.' fieldName
          '.' 'int'
        | coverageExpr '.' 'rangetype' '.' fieldName '.' 'int'

getMetadataExpr ::=
        coverageExpr '.' 'metadata'
```

B.10 General

```
coordinateExpr :=
          arithmeticExpr
        | stringExpr

listSeparator ::=
          ','
        | ';'

coverageTypeName ::=
        'GeneralGridCoverage'

coverageName ::=
          name
        | stringConstant

variableName ::=
        name

formatName ::=
        stringConstant

extraParams ::=
        stringConstant

crsName ::=
        stringConstant

axisName ::=
        name

fieldName ::=
        name

nilConstant ::=
        nameOrString

stringExpr ::=
          stringExpr '.' stringTerm
        | stringTerm

stringTerm ::=
          '(' string Expr ')'
        | stringConstant

constant ::=
          stringConstant
        | numericalConstant
        | booleanConstant

stringConstant ::=
          '"' bytes '"'   // arbitrary string
        | ''' bytes '''   // arbitrary string
```

```
numericalConstant ::=
          integerConstant
        | floatConstant
        | complexConstant

complexConstant ::=
          '(' floatConstant ',' floatConstant ')'
        | '(' integerConstant ',' integerConstant ')'

queryExpr ::=
        stringConstant
```

B.11 WCPS terminal symbols

The following are terminal symbols, in addition to the underlined terminal literals: *name; stringConstant; booleanConstant; integerConstant; floatConstant.*

Requirement 88
Keywords **shall** be treated case insensitive.

Requirement 89
A `name` **shall** given by the regular expression:
`([a-zA-Z_][0-9a-zA-Z:\_\.-]*)|("[^"]*")|('[^']*')`
not identical to any reserved (in the grammar: quoted) word.

Requirement 90
A *booleanConstant* **shall** represent a logical truth value expressed as one of the case-insensitive literals `true` and `false`, resp.

Requirement 91
An *integerConstant* **shall** represent an integer number expressed in either decimal, octal (with a "0" prefix), or hexadecimal notation (with a "0x" or "0X" prefix), optionally suffixed with a type designator as defined in Table 3.

Requirement 92
A *floatConstant* **shall** represent a floating point number following the syntax of the Java programming language, optionally suffixed with a type designator as defined in Table 3.

Requirement 93
A *stringConstant* **shall** represent a character sequence expressed by enclosing it into single or double quotes, with no mix of both in a single constant and possible quotes inside the string escaped through a backslash ("\") prefix.

Bibliography

[1] Baumann, P.: *A Database Array Algebra for Spatio-Temporal Data and Beyond*. The Fourth International Workshop on Next Generation Information Technologies and Systems (NGITS '99), July 5-7, 1999, Zikhron Yaakov, Israel, Springer Lecture Notes on Computer Science 1649, pp. 76 – 93

[2] Baumann, P.: *The OGC Web Coverage Processing Service (WCPS) Standard*. Geoinformatica, 14(4)2010, pp 447-479

[3] Baumann, P., Bell, B.: *Polygon Clipping in Datacubes*. (submitted), available on https://coverages.info/pdf/polygon-clipping-in-datacubes.pdf

[4] ISO: Geographic information — Schema for coverage geometry and functions — Part 2: Coverage Implementation Schema. ISO 19123-2:2026, https://committee.iso.org/sites/tc211/home/projects/projects---complete-list/iso-19123-2.html

[5] ISO: *Geographic information — Schema for coverage geometry and functions — Part 3: Coverage Processing Fundamentals*. ISO 19123-3:2023, https://committee.iso.org/sites/tc211/home/projects/projects---complete-list/iso-19123-3.html

[6] N.n.: *UCUM*. https://ucum.org

[7] OGC: *WCS Processing Extension*, version 2.1. OGC 08-059r4, DOI 10.62973/08-059r4

[8] OGC: *Web Coverage Service (WCS) Core*, version 2.1. OGC 17-089r1, DOI 10.62973/17-089r1

[9] P. Baumann: *Modeling Multi-Dimensional, Spatio-Temporal Big Data: The Coverage Data Standards*. Big Earth Data, 2025, pp. 1 - 54, DOI 10.1080/20964471.2025.2585732

[10] P. Baumann: *Enhanced Calendar Support for Temporal Datacube Queries*. Transactions in GIS, 28(7)2025, pp. 2089-2112, DOI 10.1111/TGIS.13215

[11] W3C: *XQuery 3.1*. W3C Recommendation, 2017, https://www.w3.org/TR/2017/REC-xquery-31-20170321/