Open Geospatial Consortium

Submission Date 2023-04-19

Approval Date: <yyyy-mm-dd>

Publication Date: <yyyy-mm-dd>

External identifier of this OGC® document: <http://www.opengis.net/doc/[{doc-type/}]{standard}/{m.n}>

Internal reference number of this OGC® document: 21-060r1

Version: 1.0

Category: OGC® Abstract Specification

Editor: Peter Baumann

OGC Abstract Specification Topic 6: Schema for Coverage Geometry and Functions – Part 3: Processing Fundamentals

Warning

This document is not an OGC Standard. This document is distributed for review and comment. This document is subject to change without notice and may not be referred to as 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.

Document type: OGC® Abstract Specification
Document subtype:
Document stage: Draft
Document language: English

Contents

Foreword

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

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

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

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

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

This document was prepared by Technical Committee ISO/TC 211, *Geographic information/Geomatics*, in collaboration with the European Committee for Standardization (CEN) Technical Committee CEN/TC 287, *Geographic Information,* in accordance with the Agreement on technical cooperation between ISO and CEN (Vienna Agreement), and in collaboration with the Open Geospatial Consortium (OGC).

A list of all parts in the ISO 19123 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

This document defines, at a high, implementation-independent level, operations on coverages, i.e. digital representations of space-time varying geographic phenomena, as defined in ISO 19123-1. Specifically, regular and irregular grid coverages are addressed. The operations can be applied through an expression language allowing composition of unlimited complexity and combining an unlimited number of coverages for data fusion.

The language is functionally defined and free of any side effects. Its conceptual foundation relies on only two constructs: A “coverage constructor” builds a coverage, either from scratch or by deriving it from one or more other coverages. A “coverage condenser” derives summary information from a coverage by performing an aggregation like count, sum, minimum, maximum and average.

The coverage processing language is independent from any particular request and response encoding, as no concrete request/response protocol is assumed. Hence, this document does not define a concrete service, but acts as the foundation for defining service standards functionality. One such standardization target is the OGC Web Coverage Service (WCS).[3]

Throughout this document, the following formatting conventions apply.

— Bold-Face in the text, such as **processCoveragesExpr**, represents syntax elements, normatively defined in Annex B.
— Text in italics, such as *succ*(), represents mathematical functions and variables.
— Courier font, such as **`return`** and `encode`(), is used for code in the sense of the coverage processing language.

Geographic information — Schema for coverage geometry and functions —

Part 3: **Processing fundamentals**

1 Scope

This document defines a coverage processing language for server-side extraction, filtering, processing, analytics, and fusion of multi-dimensional geospatial coverages representing, for example, spatio-temporal sensor, image, simulation, or statistics datacubes. Services implementing this language provide access to original or derived sets of coverage information, in forms that are useful for client-side consumption.

This document relies on the abstract coverage model defined in ISO 19123-1. In this edition, regular and irregular multi-dimensional grids are supported for axes that can carry spatial, temporal or any other semantics. Future editions will additionally support further axis types as well as further coverage types from ISO 19123-1, in particular, point clouds and meshes.

2 Normative references

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

ISO 19111, *Geographic information — Referencing by coordinates*

ISO 19123-1, *Geographic information — Schema for coverage geometry and functions — Part 1: Fundamentals*

3 Terms, definitions, abbreviated terms and notation

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 19123-1 and the following apply.

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

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1.1
probing function
<coverage> function extracting information from the coverage

4 Conformance

4.1 Notation

Table 1 lists the other International Standards and packages in which UML classes used in this document have been defined.

Table 1 — Sources of externally defined UML classes

Prefix	International Standard	Package
	ISO 19123-1	Coverage Core, Grid Coverage

4.2 Interoperability and conformance testing

As this document is an abstract standard, it allows for multiple different implementations and does not define a standardized interoperable implementation. Rather, standardization targets are specifications of coverage operations and services which may use this language to describe the semantics of their operations.

Conformance testing shall be accomplished by validating a candidate concretization against all requirements by exercising the tests set out in Annex A. As a prerequisite, a candidate shall also pass all conformance tests of ISO 19123-1 Coverage Core and Grid Coverage.

4.3 Organization

Table 2 provides details of the conformance classes described in this document. The name and contact information of the maintenance agency for this document can be found at www.iso.org/maintenance_agencies.

Table 2 — Conformance classes

Conformance class	Clause	Identifying URL
Coverage Processing	6	https://standards.isotc211.org/19123/-3/1/conf/coverage-processing

5 Coverage model

5.1 Overview

This document defines a language whose expressions accept any number of input coverages (together with further common inputs like numbers and strings) to generate any number of output coverages or non-coverage results. Coverages are defined in ISO 19123-1.

5.2 Coverage model

Following the mathematical notion of a function that maps elements of a domain (such as spatio-temporal coordinates) to a range (such as values of a “pixel”, “voxel”, etc.), a coverage consists of (Figure 1):

— an *identifier* which uniquely identifies a coverage in some context (here, the context of an expression);

— a *domain* of coordinate points (expressed in a common Coordinate Reference System, CRS): "*where in the multi-dimensional space can I find values?*";

— a probing function which answers for each coverage coordinate in the domain ("*direct position*"): "*what is the value here?*";

— a *range type*: "*what do those values mean?*".

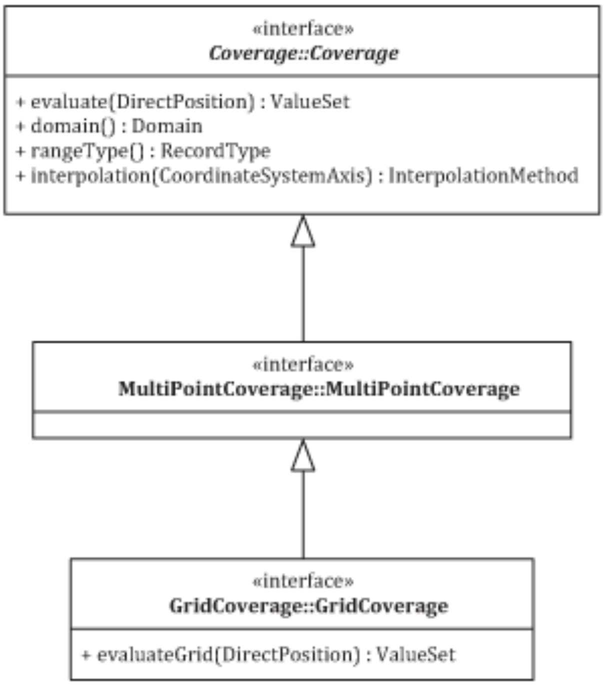

Figure 1 — Coverage and GridCoverage (ISO 19123-1)

NOTE 1 Coverage in ISO 19123-1 defines an interface which describes such an object's behaviour, but does not yet assume any particular data structure. One interoperable concretization of it is the implementation standard ISO 19123-2.

Below "probing functions" are introduced which extract components from a given coverage. For every component of a coverage a corresponding probing function exists so that altogether all properties of a coverage can be retrieved. They serve to define the document's language semantics.

NOTE 2 In the processing definition of this document, further probing functions, beyond the ISO 19123-1 probing function *evaluate*(), are used as a concise means to describe all aspects of coverage-valued function results.

5.3 Coverage identifier

Coverages in this document have an identifier which is used in a query to address a coverage to derive from. Therefore, it is necessary for this identifier to be unique within some context (here: a query). Beyond this, no particular assumption is made on the realization of this identifier. In particular, when the context of the coverage object changes (such as during delivery to a client) uniqueness is not necessarily guaranteed any longer, and therefore querying the object in the new context is potentially no longer possible.

NOTE In a concrete service, coverages available would typically be those which are stored on this server, where access control allows addressing the coverage according to the user sending the request, etc. All these aspects are out of scope of this document.

The corresponding probing function for a coverage C is:

$id(C)$

5.4 Domain

5.4.1 Direct Position

A coverage offers values for particular positions in its domain; these are called “direct positions”; further values can be derived through interpolation, depending on whether and what type of interpolation a coverage allows.

For some direct position $p = (p_1,\ldots,p_d)$ from a domain whose d-dimensional CRS contains axes $(a_1,\ldots,a_d)$, $p[a_i]$ is written for accessing the coordinate tuple component corresponding with axis a_i:

$p[a_i] = p_i$

5.4.2 Grid

The domain contains the coordinate tuples describing the coverage’s direct positions, which for the purpose of this document all sit on a multi-dimensional grid. Informally speaking this means that every direct position inside the grid has exactly one next neighbour in both directions of every axis, except for the rim, where fewer neighbours are available. Figure 2 shows some regular and irregular grid examples.

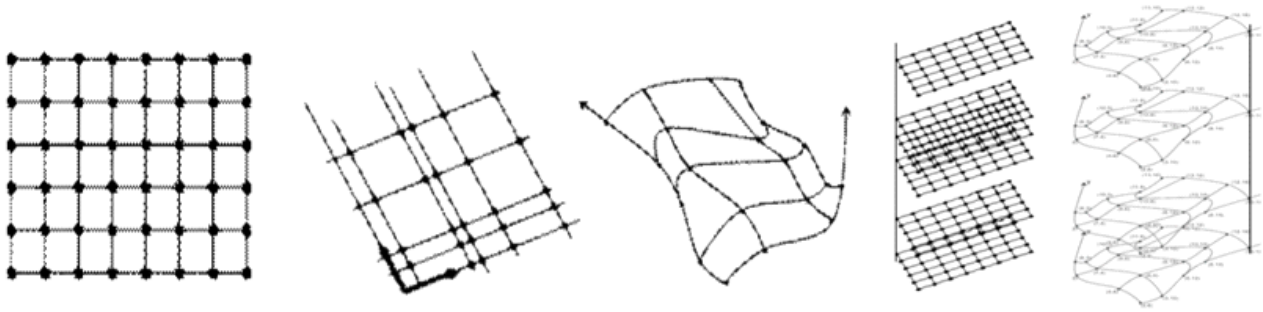

Figure 2 — Sample regular and irregular grid structures (ISO 19123-1)

The grid description depends on the complexity of the grid. As a grid is composed from an ordered sequence of axes, the resulting complexity is determined by the types of axes (such as integer versus Latitude versus time) as well as the rules determining the direct positions along these axes. The following axis types defined in ISO 19123-1 are currently supported by this document:

— a **Cartesian** (“index”) axis, which just requires lower and upper bound (which are of type integer);

— a **regular** axis, which can be described by lower and upper bounds together with a constant distance, the resolution;

— an **irregular** axis, which has individual distances, described by a sequence of coordinates.

As per ISO 19123-1, the coverage domain with its axes has a single CRS which can serve for geo-referencing. The definition and interpretation of CRSs is in accordance with ISO 19111.

The CRS of a domain is obtained through function $crs(C)$.

$crs(C)$

Auxiliary probing function *axisList*()extracts the ordered list of axes (a_1,…,a_d) from a d-dimensional CRS:

axisList(*crs*)

NOTE In accordance with ISO 19123-1, all axis names in such a list are pairwise disjoint so that the names can act as a unique identifier within their CRS.

Each axis contributes coordinates from a nonempty, totally ordered set of values which can be numeric or, in the general case, strings (such as “2020-08-05T”).

For a given coverage C, probing function *domain*() delivers the coverage domain in its CRS:

domain(C)

The domain information describes the coverage’s grid and its extent for each axis:

— the lower and upper bound of the direct positions;

— additionally the following information:

 — for index axes: nothing further;

 — for regular axes: the resolution, expressed in the unit of measure (uom) of the axis;

 — for irregular axes: the sequence of points.

This information is accessible through extended variants of the abovementioned functions. For some coverage domain D with axis a, the following expressions return lower and upper bounds, respectively:

domain(C, a).lo
domain(C, a).hi

For convenience, a function pair identical in effect but based on the domain is defined:

$D[a].lo = domain(C, a).lo$
$D[a].hi = domain(C, a).hi$

The grid of the coverage domain is represented implicitly through functions “walking” the grid from one direct position to one of its neighbours. This is based on the topological structure of a grid where each direct position has exactly one lower and one higher neighbour along each axis, with an exception of the domain rims where no such neighbour is available; therefore, these functions are partial.

Let D be given as the domain of coverage C, so that D = *domain*(C). Let further a be some axis from the CRS of D. Then, functions *pred*() and *succ*() each return a neighbouring direct position for some given position. Function *pred*() returns the immediate preceding direct position along axis a, function *succ*() returns the immediate succeeding direct position along a. Where there is no such direct position (because the input position is sitting at the rim of the domain extent) the value is undefined, written as ⊥.

pred(D, a, p) = x where
 if $p[a] = D[a].lo$ *domain*(C,a).*lo* then $x = \perp$
 else x is given by: $x[a_x] = p[a_x]$ for all $a_x \in domain(C) \setminus \{a\}$, and $x[a] = \max(x' \mid x' \in domain(C, a)$ and $x' < p[a])$

succ(D, a, p) = x where
 if $p[a] = D[a].hi$ *domain*(C,a).*hi* then $x = \perp$

else x is given by: $x[a_x] = p[a_x]$ for all $a_x \in domain(\ C\ ) \setminus \{a\}$, and $x[a] = \min(\ x' \mid x' \in domain(\ C, a\ )$ and $x' > p[a]\ )$

EXAMPLE In Figure 3, neighbours of p in coverage domain D with axes x and y can be reached as follows:
a = *succ*(*D*, *y*, *pred*(*D*, *x*, *p*)) = *pred*(*D*, *x*, *succ*(*D*, *y*, *p*))
b = *succ*(*D*, *y*, *p*)
c = *succ*(*D*, *y*, *succ*(*D*, *x*, *p*)) = *succ*(*D*, *x*, *succ*(*D*, *y*, *p*))
d = *pred*(*D*, *x*, *p*)
e = *succ*(*D*, *x*, *p*)
f = *pred*(*D*, *x*, *pred*(*D*, *y*, *p*)) = *pred*(*D*, *y*, *pred*(*D*, *x*, *p*))
g = *pred*(*D*, *y*, *p*)
h = *succ*(*D*, *x*, *succ*(*D*, *y*, *p*)) = *succ*(*D*, *y*, *succ*(*D*, *x*, *p*))

In this document, for the user's convenience, basic arithmetic functions are assumed on this grid navigation:

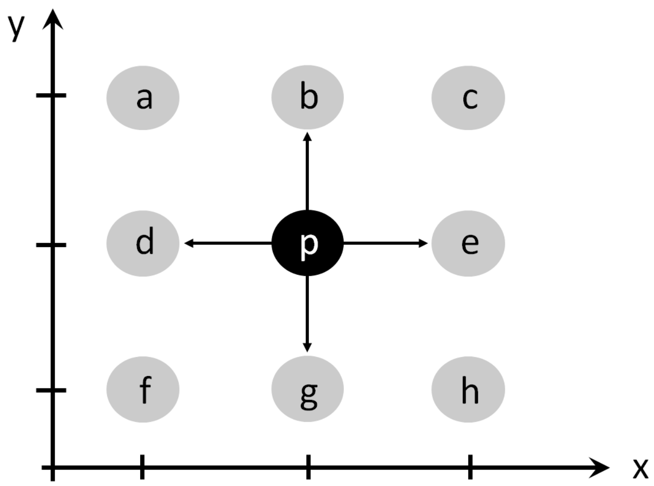

Figure 3 — Sample grid neighbourhood

5.5 Interpolation

In ISO 19123-1 a coverage contains an indication on possible interpolation between direct positions. Such interpolation can be set for all axes in a coverages simultaneously or, following a more fine-grain approach, individually per axis.

NOTE 1 In ISO 19123-1 every coverage has exactly one interpolation method associated (for all axes or per axis). In practice, coverages can allow users to pick one of several interpolation methods, such as with imagery where linear, quadratic and cubic interpolation are applicable on principle, and users can choose any one of those. Conceptually, however, two coverages differing only in the interpolation methods are distinct as they will deliver identical range values on their direct positions, but differing values inbetween those. On the abstract level of ISO 19123-1 and ISO 19123-3, this ambiguity is not desirable.

For the purpose of this document a special interpolation method `none` is assumed as defined, for example, in ISO 19123-1:—[1], Annex B. `None` indicates that no interpolation is possible along the axis under consideration.

NOTE 2 The interpolation method `none` is different from `nearest-neighbor`: An interpolation of `nearest-neighbor` provides values inbetween direct positions which are derived from the closest direct position. Interpolation `none` means that no values are provided between direct positions, in other words: the evaluation function is undefined on any non-direct position and will in practice result in an exception.

[1] Under preparation. Stage at the time of publication: ISO/FDIS 19123-1:2023.

Function *interpolation*(C,a) returns the interpolation method applicable on each axis of coverage C, in order of the CRS axis sequence. For *dimension*(C)=d the probing function delivers interpolation method list (m_1,...,m_d) with interpolation method m_i applying to axis number i:

interpolation(C)

This function is overloaded to extract the interpolation method associated with axis a of C:

interpolation(C, a)

NOTE 3 Interpolation is particularly relevant with functions *scale*() and *project*().

5.6 Range values

The range value at a direct position p can be obtained with function $evaluate_C(p)$ which, for a given coverage C, returns the value associated with $p \in domain(C)$ expressed in the coverage's CRS.

The corresponding probing function is:

value(C, p) = $evaluate_C$(p) for some direct position $p \in domain(C)$

Interpolation guides whether the *value*() function is defined on coordinates outside the set of direct positions, and how this value is determined from the values available at the direct positions.

NOTE The range value set can contain one or more null values, as determined by the range type. This document does not make any assumption on this.

5.7 Range type

A coverage's range type description can be obtained through probing function *rangeType*() which delivers a set of tuples containing at least field names and field type:

rangeType(C)

This function gets overloaded to obtain the coverage range type of a particular range field component f:

rangeType(C, f)

For the purpose of this document, only the common programming language data types are considered, and only on a high, abstract level: Boolean, integer, float, complex, as well as records over those are assumed to be available. However, an implementation specification of this document may add its own data types as long as these are coherent with this document overall.

NOTE The concrete range types available in coverage processing are determined by concretizations of this document. Typically, the standard programming language data types will be available, such as (unsigned) short, int, and long, as well as float and double. For example, the range type (aka pixel) of an 8-bit RGB image normally is given by the triple < red: unsigned char; green: unsigned char; blue: unsigned char>. Further, a concretization can add more information such as null values, accuracy, etc.

5.8 Coverage probing functions synopsis

Requirement 1 https://standards.isotc211.org/19123/-3/1/req/core/probingFunctions
The semantics of the probing functions used for the ISO 19123-1 language semantics definition **shall** be given by Table 3.

Table 3 —Coverage probing functions synopsis

Coverage characteristic	**Probing function for a coverage *C*, based on ISO 19123-1**	**Comment**
Coverage identifier	*id*(*C*)	Identifier of the coverage.
Coverage CRS	*crs*(*C*) = *crs* (*domain*(*C*)) as per ISO 19123-1	CRS of the coverage.
CRS axis list	*axisList*(*c*) = $(a_1,\ldots,a_d)$ for some *d*-dimensional CRS *c* establishing this axis sequence	List of all axis names of the CRS, in proper sequence.
Domain extent of coverage	*domain*(*C*) *domain*(*C*, *a*) = domain extent along axis *a* *domain*(*C*, *a*).lo = lower bound of domain extent along axis *a* *domain*(*C*, *a*).hi = upper bound of domain extent along axis *a*	Extent of the coverage in CRS coordinates.
Grid neighbour	*pred*(*C*, *a*, *p*) *succ*(*C*, *a*, *p*) as defined in 6.4.2	These functions allow to traverse a grid in steps relative to some given position, such as for convolution operations and, generally, Tomlin's non-local operations.
Range type	*rangeType*(*C*) *rangeType*(*C*, *f*) = *t* where $(f{:}t,\ldots) \in$ *rangeType*(*C*)	The range type record is described by a list describing its components in sequence; for the purpose of this document only component name and its data type are considered.
Range field name list	*rangeFieldNames*(*C*) = $(f_1, \ldots, f_n)$ where *rangeType*(*C*) = ($(f_1;t_1,\ldots)$, …, $(f_n{:}t_n,\ldots)$), with field names f_i and types t_i	Ordered list all of the coverage's range fields names and their data types; possible further constituents in a record component are ignored in this document, their values are to be defined elsewhere (e.g. implementation dependent).
Range values	*value*(*C*,*p*) = $evaluate_C(p), p \in domain(C)$ with *evaluate*() as per 19123-1	Range values of the coverage at a direct position (or some position inbetween, interpolation permitting).

Coverage characteristic	Probing function for a coverage *C*, based on ISO 19123-1	Comment
Interpolation	*interpolation*(*C*) as per ISO 19123-1 *interpolation*(*C*, *a*) = interpolation method of axis *a*	List of the interpolation method allowed per axis, in axis order; in case the coverage has only one interpolation defined for all axes, this method is multiplied into all positions of the output list. Interpolation associated with a particular axis.

6 Coverage processing language

This clause establishes the conformance class *Coverage Processing*.

This coverage processing language defines expressions on coverages which evaluate to ordered lists of either coverages or scalars (whereby "scalar" here is used as a summary term of all data structures that are not coverages). In the remainder of this document, the terms *processing expression* and *query* are used interchangeably.

A coverage processing expression consists of a **processCoveragesExpr** (see 6.2). Each International Standard claiming to support this document shall provide the coverage processing operations as specified in the following subclauses. A sample application is provided in Annex D.

NOTE 1 This language has been designed so as to be "safe in evaluation" – i.e. implementations are possible where any valid request can be evaluated in a finite number of steps, based on the operation primitives. Hence, services based on the language constructs can be built in a way that no single request can render the service permanently unavailable. This notwithstanding, it still is possible to send requests that will impose high workload on a server.

NOTE 2 Data items within a query result list can be heterogeneous in size and structure. In particular, the coverages within an evaluation result list can have different dimensions, domains, range types, etc. However, a result list always consists of either coverages or scalar values, not a mix of both.

6.1 Syntax and Semantics Definition Style

6.1.1 Expression Syntax

The language primitives plus the nesting capabilities form an expression language which is independent from any particular encoding and service protocol; collectively it is referred to as the **coverage processing language**. In the following subclauses, the language elements are detailed. The complete syntax is listed in Annex B.

A coverage processing expression is called **admissible** if and only if it adheres to the syntax of the language definition of this document.

Requirement 2 https://standards.isotc211.org/19123/-3/1/req/core/syntax
Coverage processing expressions **shall** adhere to the syntax definition of Annex B.

NOTE A railroad diagram of the syntax in Annex B is provided in Annex C for visualization of the grammar.

EXAMPLE The coverage expression fragment `$c * 2`is admissible as it adheres to language syntax whereas `abc` seen as a coverage expression violates the syntax and, hence, is not admissible.

6.1.2 Expression Semantics

The semantics of a coverage processing expression is defined recursively by indicating, for all admissible expressions, the semantics. An expression is **valid** if and only if it is admissible and complies with all rules imposed by the language semantics.

Requirement 3 https://standards.isotc211.org/19123/-3/1/req/core/semantics
Coverage processing expressions **shall** adhere to all semantics rules of this document.

EXAMPLE The following coverage expression is valid if and only if the coverage bound to variable `$c` has a numeric range component named `red`.

```
$c.red * 2.5
```

NOTE In the remainder of this clause, tables are used to describe the effect of an operation on each coverage constituent.

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

6.2 Coverage Processing Expressions

6.2.1 processCoveragesExpr

A **processCoveragesExpr** element processes a list of coverages in turn. Each coverage is optionally checked first for fulfilling some predicate, and gets selected, i.e. contributes to an element of the result list, only if the predicate evaluates to *true*. Each coverage selected will be processed, and the result will be appended to the result list. This result list, finally, is returned as the *ProcessCoverages* response unless any exception was generated.

Requirement 4 https://standards.isotc211.org/19123/-3/1/req/core/processCoveragesExpr
A **processCoveragesExpr shall** be defined as follows.

Let

$v_1, \dots v_n$ be n pairwise different **iteratorVar**s ($n \geq 1$),
$L_1, \dots L_n$ be n **coverageList**s ($n \geq 1$),
b be a **booleanScalarExpr** possibly containing occurrences of one or more v_i ($1 \leq i \leq n$),
P be a **processingExpr** possibly containing occurrences of v_i ($1 \leq i \leq n$).

Then,

$m, n \geq 1$ be natural numbers,
$v_1, \dots c_n$, be n **iteratorVar**s,
$c_1, \dots c_m$, be n pairwise different **variableName**s,
$e_1, \dots e_m$, be $n+m$ optional **coverageExpr**s or **scalarExpr**s or bracket-enclosed **intervalExpr**s, which may contain occurrences of $v_1, \dots c_n$ and $c_1, \dots c_m$,
c be a **coverageExpr** or **scalarExpr**,
where every c_i is defined before used in an expression.

Then,

for any **processCoveragesExpr** E
where
    E = **for** v_1 **in (** L_1**),**

```
        v2 in ( L2),
         … ,
        vn in ( Ln)
        [ let c1 := e1, …, cm := em ]
        [ where b ]
        return P
```

the result R of evaluating **processCoveragesExpr** E is constructed as:

```
Let R be the empty sequence;
while L1 is not empty:
{  assign the first element in L1 to iteration variable v1;
   while L2 is not empty:
   {   assign the first element in L2 to iteration variable v2;
       …
           while Ln is not empty:
           {   assign the first element in Ln to iteration variable vn;
               substitute every occurrence of ci in E by ei;
               substitute every occurrence of vi in E
                  by the corresponding coverage;
               evaluate b;
               if (b)
               then
                  evaluate P;
                  append evaluation result to R;
                  remove the first element from Ln;
           }
       …
       }
       remove the first element from L2;
   }
   remove the first element from L1;
}
```

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

NOTE 1 Coverage identifiers can occur more than once in a **coverageList**. In this case the coverage will be evaluated each time it appears, respecting the overall inspection sequence.

EXAMPLE 1 Assume availability of coverages *A*, *B* and *C*. Then, the following request:

```
for $c in ( A, B, C )
return encode( $c, "image/tiff" )
```

will produce a result list containing three TIFF-encoded coverages.

Assume availabilityof satellite images *A*, *B*, and *C* and a coverage *M* acting as a mask (i.e. with range values of 0 and 1 and the same extent as *A*, *B*, and *C*). Then, masking each satellite image can be performed with this query:

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

The `let` clause declares a named constant and gives it a value.

EXAMPLE 2 The following statement defines a constant of name `$timeAxis` with value "date".

```
let $timeAxis := "date"
```

NOTE 2 In most cases, named constants are used purely for convenience, to simplify the expressions and make the code more readable.

In a **`let`** clause the named constant only takes one value. This can be a single item or a sequence (there is no real distinction; an item is just a sequence of length one), and the sequence can contain nodes, or atomic values, or a mixture of the two.

Named constants cannot be updated. For example, something like `let $x:=$x+1` is not allowed. More specifically, it will not lead to an evaluation error, but the result will not be as expected (see literature on XPath). This rule can seem very strange if expecting a behaviour as in procedural languages such as JavaScript or python. But the coverage processing language is not that kind of language. It is a declarative language which works at a higher level. This constraint is essential to give optimizers the chance to find execution strategies that can search vast databases in fractions of a second. SQL, XSLT and XQuery users have found that this declarative style of programming enables to code at a higher level by telling the system what results are wanted, rather than telling it how to go about constructing those results.

6.2.2 processingExpr

Requirement 5 https://standards.isotc211.org/19123/-3/1/req/core/processingExpr
A **processingExpr** element **shall** be either an **encodeCoverageExpr** (see 6.8.1) or a **scalarExpr** (see 6.4.1).

6.2.3 coverageExpr

Requirement 6 https://standards.isotc211.org/19123/-3/1/req/core/coverageExpr
A **coverageExpr shall** be either a **coverageIdExpr** (see 6.2.4) or a **coverageConstructorExpr** (see 6.3.1.1) or a **coverageConstantExpr** (see 6.3.1.1) or a **getComponentExpr** (see 6.4.1) or an **inducedExpr** (see 6.5.1) or a **subsetExpr** (see 6.5.6.1) or a **crsTransformExpr** (see 6.6) or a **scaleExpr** (see 6.5.7) or a **decodeCoverageExpr** (see 6.8.2).

NOTE A **coverageExpr** always evaluates to a single coverage.

6.2.4 coverageIdExpr

The **coverageIdExpr** element represents the name of a single coverage available. It is represented by a coverage variable indicated in the **processCoveragesExpr** clause (see 6.2).

Requirement 7 https://standards.isotc211.org/19123/-3/1/req/core/coverageIdentifier
A **coverageIdExpr shall** be defined as follows.

Let

id be a **variableName** bound to a coverage C_1 available.

Then,

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

C_2 is defined as:

Coverage constituent
$id(C_2) = id(C_1)$
$crs(C_2) = crs(C_1)$
$domain(C_2) = domain(C_1)$
$interpolation(C_2) = interpolation(C_1)$
$rangeType(C_2) = rangeType(C_1)$
for all $p \in domain(C_2)$: $value(C_2,p) = value(C_1,p)$

EXAMPLE The following coverage expression evaluates to the complete, unchanged coverage `C`, assuming that coverage iteration variable `$c` is bound to it at the time of evaluation:

```
$c
```

6.3 Coverage-Generating Expressions

6.3.1 coverageConstructorExpr

The **coverageConstructorExpr** element creates a d-dimensional grid coverage for some $d \geq 1$ by defining the coverage's domain, range type and range through expressions. This allows entirely new shapes, dimensions, and values to be derived (see examples below).

The coverage domain is built from a 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 ISO 19123-1.

A range type expression optionally creates the coverage range type. In the scope of the embedding condensers, this expression defines the range component names as known (immutable) variables. Values derived for some such range component will automatically be cast to the target type of that range component.

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

Requirement 8 https://standards.isotc211.org/19123/-3/1/req/core/coverageConstructorExpr
A **coverageConstructorExpr shall** be defined as follows.

Let

`id` be an **identifier**,
D be a **domainExpr**,
T be a **rangeTypeExpr**,
R be a **rangeSetExpr.**

Where

> C is a **coverageConstructorExpr**
> with
> C = **coverage** id [D] [T] R

Let further

> d be an **integer** with d>0,
> c be a **crsName** representing a d-dimensional CRS,
> a_i be pairwise distinct **variableName**s for $1 \le i \le d$,
> $axis_i$ be pairwise distinct **axisName**s for $1 \le i \le d$,
> $ie_{i,1}$, $ie_{i,2}$ be integer-valued **indexExpr**s for $1 \le i \le d$ with $ie_{i,1} \le ie_{i,2}$,
> $ce_{i,1}$, $ce_{i,2}$ be **axisPointExpr**s for $1 \le i \le d$, which are valid coordinates for axis i as per CRS c with $ce_{i,1} \le ce_{i,2}$,
> res_i be **axisPointExpr**s with $res_1 < \ldots < res_d$ for $1 \le i \le d$ valid for the i^{th} axis as per c,
> $xe_{i,1}$,... be **axisPointExpr**s for $1 \le i \le d$, which are valid coordinates for axis $axis_i$ as per CRS c with $xe_{i,1} < xe_{i,2} < \ldots$,
> im_1,..., im_m be (not necessarily distinct) **interpolationMethod**s for $1 \le i \le m$ with m>0 .

Where

> D is a **domainExpr**
> with

```
D =   domain
      crs c with
      axis1 axisdef1 [ interpolation im1 ],
      ...,
      axisd axisdefd [ interpolation imd ]
```

And

> $axisdef_i$ is one of

```
axisdef_i,index     =  index ( ie_i,1 : ie_i,2 )
axisdef_i,regular   =  regular ( ce_i,1 : ce_i,2 ) resolution res_i
axisdef_i,irregular =  irregular( xe_i,1 , ..., xe_i,n )
```

And

> axis names used in the **domainExpr shall** match pairwise against the CRS axes based on their order of occurrence in the D expression.

NOTE The axis names $axis_i$ are made available in the current context for use as iteration variables in the range set computation where coordinate values get bound to each direct position in turn allowing to inspect each direct position of the coverage. Iterator names can use the axis names defined in the CRS, or can define aliases which are matched with the CRS axis names by their position in the expression.

Let further

> n be an **integer** with d>0,
> f_1,..., f_n be **fieldName**s,
> t_1,..., t_n be **rangeType**s .

Where

T is a **rangeTypeExpr**
with

```
T  =   range type
       f1 : t1,
       ...
       fn : tn
```

Let further

r be a **scalarExpr** possibly containing occurrences of direct position coordinates $axis_i$ as defined in D and range component identifiers f_j as defined in T,
c_1, ..., c_m be **constant**s where m=|$domain(C)$|.

Where

R is a **rangeSetExpr**
with R one of

```
R1  =   range r
R2  =   range <c1,..., cm>
```

and

R is part of a **coverageConstructorExpr** containing a **domainExpr**.

Then,

C is defined as the following ISO 19123-1 grid coverage:

Coverage constituent
id(C) = id
crs(C) = c if D is present, otherwise the CRS resulting from evaluating r
domain(C) = domain extent resulting from evaluating D if present, otherwise the domain extent resulting from evaluating r
interpolation(C) = (x_1,…, x_d) where x_i = im_i where im_i is indicated, otherwise x_i = $none$.
rangeType(C) = ((f_1,t_1), …, (f_n,t_n)) if T is present, otherwise the range type resulting from evaluating r ; if no field names are provided (such as with R_2) then the range field names are implementation-dependent.
for all $p \in$ *domain*(C) and **scalarExpr** r: *value*(C,p) = range value resulting from evaluating r, with possible occurrences of a_i substituted by the corresponding $p[i]$ coordinate value. If, for example through computed direct positions, a location outside the domain of coverage addressed gets encountered then the behaviour is implementation dependent (possible options including assuming a null value for such a position or terminating evaluation of the request). for all $p \in$ *domain*(C) and **rangeConstantExpr <**c_1,…, c_m**>**: *value*(C, p) is determined by assigning each value c_i in turn to a grid point location, whereby assignment proceeds in row-major order (per dimension from the lowest to the highest coordinate, and loops over the grid points with the first axis listed as outermost loop, the next axis listed as next-to-outermost loop, etc. and the last axis listed as innermost loop).

NOTE A concretization of this language can extend the capabilities of the coverage constant expression by allowing records at direct positions, rather than only atomic values.

6.3.2 Examples

The following examples illustrate use of the coverage constructor expressions in various practical scenarios relying on common CRSs and data types (both not specified in this document).

The first domain establishes a 2D WGS 84 grid with linear interpolation along both axes.

```
domain
crs "EPSG:4326" with
  Lat  regular (10:30) resolution 0.01 interpolation linear,
  Long regular (10:30) resolution 0.01 interpolation linear
```

In the following example, EPSG:4326 establishes Lat and Long axes. Therefore in the domain expression the first axis will be associated with *Lat* and the second with *Long*, regardless of the axis naming in the domain expression; no interpolation is admissible:

```
domain
  crs "EPSG:4326" with
```

```
  Lat  regular (10:30) resolution 0.5,
  Long regular (10:30) resolution 0.5
```

The next domain establishes a 4D georeferenced timeseries datacube with a spectral dimension, regular in Lat/Long and irregular in time (given the varying number of days a month has and based on the daily resolution specified).

```
domain
crs "EPSG:4326+OGC:unixTtime" with
  Lat  regular (10:30) resolution 0.5,
  Long regular (10:30) resolution 0.5,
  Date irregular ( "2017-01-01", "2017-02-01","2017-03-01", "2017-04-01",
                   "2017-05-01", "2017-06-01","2017-07-01", "2017-08-01",
                   "2017-09-01", "2017-10-01", "2017-11-01", "2017-12-01"
                 )
```

The expression below represents a single-band range type:

```
range type
  panchromatic: integer
```

The following range type defines RGB pixels:

```
range type
  red  :integer,
  green:integer,
  blue :integer
```

The coverage constructor below resembles an induced operation, reducing intensity in all range fields by ½. Coverage type, domain and range type are adopted from the input coverage.

```
coverage Half
range (integer) $c / 2
```

The example below follows a complete coverage constructor representing a 3D georeferenced image timeseries whose range set gets loaded from some input file provided, represented by the positional parameter `$1`. Further, some sketchy INSPIRE XML metadata record is associated:

```
coverage MySatelliteDatacube
domain
  crs "EPSG:4326+OGC:unixTime" with
  Lat  regular (10:30) resolution 0.5,
  Long regular (10:30) resolution 0.5,
  Date regular ("2017-01":"2019-12") resolution "P1M"
range type panchromatic: integer
range decode( $1 )
```

The expression below computes a 256-bucket histogram over band `blue` of some coverage `$c` of unknown domain extent and dimension:

```
coverage histogram
domain
  crs "OGC:Index1D" with bucket index (0:255)
range type
  b :integer
```



```
range
  count( $c.blue = bucket )
```

If constituents can be determined then they do not need to be indicated; in this case input coverage `$C` is copied; assuming it has range type unsigned short then the *log*() operation suggests a float result, so this will be adopted as range type. Along the same line, the domain is adopted from `$C`:

```
coverage LogOfCube
range log( $c )
```

For a Sobel filter, a 3x3 filter kernel can be provided by the expression below. The range value of matrix element (-1/-1) is 1, the value at position (0/-1) is 2, etc.

```
coverage Sobel3x3
domain
  crs "OGC:Index2d" with i index ( -1 : +1 ), j index ( -1 : +1 )
range
  <  1;  2;  1;
     0;  0;  0;
    -1; -2; -1
  >
```

A Sobel filter kernel operation can be expressed as follows:

```
coverage FilteredImage
domain
  crs "OGC:Index2D" with x index ( 0 : 5000 ), y index ( 0 : 5000 )
range
  condense +
  over  i ( -1 : +1 ), j ( -1 : +1 )
  using $c.blue[ x(x+i), y(y+j) ] * Sobel3x3[ i(i), j(j) ]
```

6.4 Coverage Extraction Expressions

6.4.1 scalarExpr

Requirement 9 https://standards.isotc211.org/19123/-3/1/req/core/scalarExpr
A **scalarExpr shall** be either a **getComponentExpr** (see 7.4.2) or a **booleanScalarExpr** (see 7.4.3) or a **numericScalarExpr** (see 7.4.4) or a **stringScalarExpr** (see 7.4.5).

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

6.4.2 getComponentExpr

The **getComponentExpr** element extracts a coverage element from a coverage.

NOTE The grid point value sets ("pixels", "voxels", ...) can be extracted from a coverage using subsetting operations (see Subclause 7.5.5).

Requirement 10 https://standards.isotc211.org/19123/-3/1/req/core/getComponentExpr
A **getComponentExpr shall** be defined as follows.

Let

> *C* be a **coverageExpr**.

Then,

> The following extraction functions are defined;
> the result **shall** be given by the probing functions defined in Table 4;
> strings **shall** be interpreted case-sensitive;
> quotes **shall** be single or double quotes, but no mix per quoted element;
> arbitrary whitespace **may** occur in between any two syntactical elements.

Table 4 — getComponentExpr functions

Coverage processing function for coverage *C*	**Semantics as per Table 3**	**Description**
`id(`*C***`)`**	*id(C)*	Coverage identifier as name (if it does not contain special characters) or a single- or double-quoted string.
`crs``(`*C*`)`	*crs*(*C*)	Identifier of the coverage's CRS.
`domain``(`*C*`)` **`domain`**`(`*C*`,` *a*`)` **`domain`**`(`*C*`,` *a*`)`**`.lo`** **`domain`**`(`*C*`,` *a*`)`**`.hi`**	*domain*(*C*) *domain*(*C*, *a*) *domain*(*C*, *a*).lo *domain*(*C*, *a*).hi	Domain of the coverage's CRS.
`interpolation``(`*C*,*a*`)`	*interpolation*(*C*,a)	Interpolation method assigned to a coverage axis.

EXAMPLE 1 For some coverage named "iamacoverage" bound to variable `$c`, the following expression evaluates to the string "iamacoverage":

```
id( $c )
```

EXAMPLE 2 For some coverage `$c` with native CRS WGS 84 the following expression can evaluate to the string "EPSG:4326", or alternatively "https://www.opengis.net/def/crs/EPSG/0/4326", or some other designation determined by a concretization of this document:

```
nativeCrs( $c )
```

6.4.3 booleanScalarExpr

Requirement 11 https://standards.isotc211.org/19123/-3/1/req/core/booleanScalarExpr
A **booleanScalarExprshall** be a **scalarExpr** (see 7.4.1) 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.

6.4.4 numericScalarExpr

Requirement 12 https://standards.isotc211.org/19123/-3/1/req/core/numericScalarExpr
A **numericScalarExpr shall** be a **scalarExpr** (see 7.4.1) whose result type is numeric (i.e. an integer, float, or complex number).

6.4.5 stringScalarExpr

Requirement 13 https://standards.isotc211.org/19123/-3/1/req/core/stringScalarExpr
A **stringScalarExpr shall** be a **scalarExpr** (see 7.4.1) whose result type is character string of length greater or equal to zero.

6.5 Coverage range value-changing expressions

6.5.1 inducedExpr

Requirement 14 https://standards.isotc211.org/19123/-3/1/req/core/inducedExprCases
An **inducedExpr shall** be either a **unaryInducedExpr** (see Subclause 7.5.2) or a **binaryInducedExpr** (see Subclause7.5.4) or a **rangeConstructorExpr** (see Subclause 7.5.5) or a **switchExpr** (see Subclause 7.5.5.2).

Induced operations allow the simultaneous application of a function originally working on a single value to all grid point values of a coverage.

NOTE 1 These operations can be expressed through a **coverageConstructorExpr**, however in a more verbose way.

Requirement 15 https://standards.isotc211.org/19123/-3/1/req/core/inducedExprComponents
In an **inducedExpr**, in case the range type contains more than one range component, the function **shall** be applied to each point simultaneously.

Requirement 16 https://standards.isotc211.org/19123/-3/1/req/core/inducedExpr
In an **inducedExpr** the result coverage **shall** have the same domain as the input coverage(s).

NOTE 2 In case of an n-ary induced operation, n>1, all input coverages need to share the same domain as a precondition.

NOTE 3 The result can have a different range type, see Subclause 6.9.5. The idea is that for each operation available on the range type, a corresponding coverage operation is provided (“induced from the range type operation”).

EXAMPLE Adding two RGB images will apply the “+” operation to each pixel, and within a pixel to each range field in turn.

6.5.2 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 17 https://standards.isotc211.org/19123/-3/1/req/core/unaryInducedExprCases
A **unaryInducedExpr shall** be either a **unaryArithmeticExpr**, or **trigonometricExpr**, or **exponentialExpr** (in which case it evaluates to a coverage with a numeric range type; see 7.5.2.1, 7.5.3, 7.5.3.1), a **booleanExpr** (in which case it evaluates to a Boolean expression; see 7.5.3.2), a **castExpr** (in which case it evaluates to a coverage with unchanged values, but another range type; see 7.5.3.3), or a **fieldExpr** (in which case a range field selection is performed; see 7.5.3.4).

6.5.2.1 unaryArithmeticExpr

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

Requirement 18 https://standards.isotc211.org/19123/-3/1/req/coreunaryArithmeticExpr
A **unaryArithmeticExpr shall** be defined as:

Let

C_1, C_2 be **coverageExpr**s with all range type components being numeric and additionally all range type components of C_1 being of type complex,
S_1, S_2 be **scalarExpr**s.

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_{re} = **re(**CC_1**)**
C_{im} = **im(**CC_1**)**

C_{plusSC} = S_1 **+** C_2
C_{minSC} = S_1 **-** C_2
C_{multSC} = S_1 ***** C_2
C_{divSC} = S_1 **/** C_2
C_{andSC} = S_1 **and** C_2
C_{orSC} = S_1 **or** C_2
C_{xorSC} = S_1 **xor** C_2
C_{eqSC} = S_1 **=** C_2
C_{ltSC} = S_1 **<** C_2
C_{gtSC} = S_1 **>** C_2
C_{leSC} = S_1 **<=** C_2
C_{geSC} = S_1 **>=** C_2
C_{neSC} = S_1 **!=** C_2

C_{plusCS} = C_1 **+** S_2
C_{mincS} = C_1 **-** S_2
C_{multCS} = C_1 ***** S_2
C_{divCS} = C_1 **/** S_2
C_{andCS} = C_1 **and** S_2
C_{orCS} = C_1 **or** S_2
C_{xorCS} = C_1 **xor** S_2
C_{eqCS} = C_1 **=** S_2
C_{ltCS} = C_1 **<**S_2
C_{gtCS} = C_1 **>**S_2
C_{leCS} = C_1 **<=** S_2
C_{geCS} = C_1 **>=** S_2
C_{neCS} = C_1 **!=** S_2

C_2 is defined as:

Coverage constituent
id(C_2) = "" (empty string)
crs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_{plus}) is given by Requirement 48 *rangeFieldType*(C_{minus}) is given by Requirement 48 *rangeFieldType*(C_{plusSC}) is given by Requirement 48 *rangeFieldType*(C_{sqrt}, r) = `double` if *rangeFieldType*(C_1, r) ≠ `complex` and $C_1.r \geq 0$, = `complex` otherwise, *rangeFieldType*(C_{abs}, r) = `unsigned int` if *rangeFieldType*(C_1, r) ∈ { `unsigned int`, `int` } = `float` if *rangeFieldType*(C_1, r) ∈ { `float`, `complex` } *rangeFieldType*(C_{plusSC}) is given by Requirement 48 *rangeFieldType*(C_{minSC}) is given by Requirement 48 *rangeFieldType*(C_{multSC}) is given by Requirement 48 *rangeFieldType*(C_{divSC}) is given by Requirement 48 *rangeFieldType*(C_{andSC}) = `boolean` *rangeFieldType*(C_{orSC}) = `boolean` *rangeFieldType*(C_{xorSC}) = `boolean` *rangeFieldType*(C_{eqSC}) = `boolean` *rangeFieldType*(C_{ltSC}) = `boolean` *rangeFieldType*(C_{gtSC}) = `boolean` *rangeFieldType*(C_{leSC}) = `boolean` *rangeFieldType*(C_{geSC}) = `boolean` *rangeFieldType*(C_{neSC}) = `boolean` *rangeFieldType*(C_{ovlSC}) = *rangeType*(C_2) *rangeFieldType*(C_{plusCS}, r) is given by Requirement 48 *rangeFieldType*(C_{minCS}, r) is given by Requirement 48 *rangeFieldType*(C_{multCS}, r) is given by Requirement 48 *rangeFieldType*(C_{divCS}, r) is given by Requirement 48 *rangeFieldType*(C_{andCS}, r) = `boolean` *rangeFieldType*(C_{orCS}, r) = `boolean` *rangeFieldType*(C_{xorCS}, r) = `boolean` *rangeFieldType*(C_{eqCS}, r) = `boolean` *rangeFieldType*(C_{ltCS}, r) = `boolean` *rangeFieldType*(C_{gtCS}, r) = `boolean` *rangeFieldType*(C_{leCS}, r) = `boolean` *rangeFieldType*(C_{geCS}, r) = `boolean` *rangeFieldType*(C_{neCS}, r) = `boolean` *rangeFieldType*(C_{ovlCS}, r) = `boolean`
`for all` $p \in$ *domain*(C_2): *value*(C_{plus}, p) = *value*(C_1, p),

Coverage constituent
$\mathit{value}(C_{\text{minus}}, p) = - \mathit{value}(C_1, p),$
$\mathit{value}(C_{\text{sqrt}}, p) = \texttt{sqrt}(\mathit{value}(C_1, p)),$
$\mathit{value}(C_{\text{abs}}, p) = \texttt{abs}(\mathit{value}(C_1, p)),$
$\mathit{value}(C_{\text{re}}, p) = \texttt{re}(\mathit{value}(C_1, p)),$
$\mathit{value}(C_{\text{im}}, p) = \texttt{im}(\mathit{value}(C_1, p)),$
$\mathit{value}(C_{\text{plusSC}}) = \mathit{value}(S_1) + \mathit{value}(C_2)$
$\mathit{value}(C_{\text{minSC}}) = \mathit{value}(S_1) - \mathit{value}(C_2)$
$\mathit{value}(C_{\text{multSC}}) = \mathit{value}(S_1) * \mathit{value}(C_2)$
$\mathit{value}(C_{\text{divSC}}) = \mathit{value}(S_1) / \mathit{value}(C_2)$
$\mathit{value}(C_{\text{andSC}}) = \mathit{value}(S_1) \texttt{ and } \mathit{value}(C_2)$
$\mathit{value}(C_{\text{orSC}}) = \mathit{value}(S_1) \texttt{ or } \mathit{value}(C_2)$
$\mathit{value}(C_{\text{xorSC}}) = \mathit{value}(S_1) \texttt{ xor } \mathit{value}(C_2)$
$\mathit{value}(C_{\text{eqSC}}) = \mathit{value}(S_1) == \mathit{value}(C_2)$
$\mathit{value}(C_{\text{ltSC}}) = \mathit{value}(S_1) < \mathit{value}(C_2)$
$\mathit{value}(C_{\text{gtSC}}) = \mathit{value}(S_1) > \mathit{value}(C_2)$
$\mathit{value}(C_{\text{leSC}}) = \mathit{value}(S_1) <= \mathit{value}(C_2)$
$\mathit{value}(C_{\text{geSC}}) = \mathit{value}(S_1) >= \mathit{value}(C_2)$
$\mathit{value}(C_{\text{neSC}}) = \mathit{value}(S_1) \,!= \mathit{value}(C_2)$
$\mathit{value}(C_{\text{ovlSC}}) = \mathit{value}(S_1) \texttt{ overlay } \mathit{value}(C_2)$
$\mathit{value}(C_{\text{plusC}})_{\text{S}} = \mathit{value}(C_1) + \mathit{value}(S_2)$
$\mathit{value}(C_{\text{mincS}}) = \mathit{value}(C_1) - \mathit{value}(S_2)$
$\mathit{value}(C_{\text{multCS}}) = \mathit{value}(C_1) * \mathit{value}(S_2)$
$\mathit{value}(C_{\text{divCS}}) = \mathit{value}(C_1) / \mathit{value}(S_2)$
$\mathit{value}(C_{\text{andCS}}) = \mathit{value}(C_1) \texttt{ and } \mathit{value}(S_2)$
$\mathit{value}(C_{\text{orCS}}) = \mathit{value}(C_1) \texttt{ or } \mathit{value}(S_2)$
$\mathit{value}(C_{\text{xorCS}}) = \mathit{value}(C_1) \texttt{ xor } \mathit{value}(S_2)$
$\mathit{value}(C_{\text{eqCS}}) = \mathit{value}(C_1) == \mathit{value}(S_2)$
$\mathit{value}(C_{\text{ltCS}}) = \mathit{value}(C_1) < \mathit{value}(S_2)$
$\mathit{value}(C_{\text{gtCS}}) = \mathit{value}(C_1) > \mathit{value}(S_2)$
$\mathit{value}(C_{\text{leCS}}) = \mathit{value}(C_1) <= \mathit{value}(S_2)$
$\mathit{value}(C_{\text{geCS}}) = \mathit{value}(C_1) >= \mathit{value}(S_2)$
$\mathit{value}(C_{\text{neCS}}) = \mathit{value}(C_1) \,!= \mathit{value}(S_2)$
$\mathit{value}(C_{\text{ovlCS}}) = \mathit{value}(C_1) \texttt{ overlay } \mathit{value}(S_2)$

EXAMPLE For two integer or float valued coverages `$c` and `$d`, 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 )
```

6.5.3 trigonometricExpr

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

Requirement 19 https://standards.isotc211.org/19123/-3/1/req/core/trigonometricExpr
A **trigonometricExpr shall** be defined as:

Let

C_1 be a **coverageExpr**

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:

Coverage constituent
id(C_2) = “” (empty string)
crs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
rangeFieldNames (C_2) = *rangeFieldNames* (C_1) for all fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = `complex` if *rangeFieldType*(C_1,r) = `complex` = `float` otherwise
for all $p \in$ *domain*(C_2): *value*(C_{sin},p) = `sin`(*value*(C_1,p)) *value*(C_{cos},p) = `cos`(*value*(C_1,p)) *value*(C_{tan},p) = `tan`(*value*(C_1,p)) *value*(C_{sinh},p) = `sinh`(*value*(C_1,p)) *value*(C_{cosh},p) = `cosh`(*value*(C_1,p)) *value*(C_{arcsin},p) = `arcsin`(*value*(C_1,p)) *value*(C_{arccos},p) = `arccos`(*value*(C_1,p)) *value*(C_{arctan},p) = `arctan`(*value*(C_1,p))

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

```
sin( $c )
```

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

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

6.5.3.1 exponentialExpr

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

Requirement 20 https://standards.isotc211.org/19123/-3/1/req/core/exponentialExpr
An **exponentialExpr shall** be defined as:

Let

C_1 be a **coverageExpr**,
c be a **floatConstant**or **complexConstant**

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, c **)**

C_2 is defined as:

Coverage constituent
id(C_2) = "" (empty string)
crs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
rangeFieldNames (C_2) = *rangeFieldNames* (C_1) for all fields $r \in$*rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = `complex` if *rangeFieldType*(C_1,r) = `complex` = `float` otherwise
for all $p \in$*domain*(C_2): *value*(C_{exp}, p) = `exp`(*value*(C_1,p)) *value*(C_{log} , p) = `log`(*value*(C_1,p)) *value*(C_{ln} , p) = `ln`(*value*(C_1,p)) *value*(C_{pow}, p) = *value*(C_1,p)^c

EXAMPLE The following expression derives the natural logarithm for all values of some all-positive coverage expression `$c`:

```
ln( $c )
```

6.5.3.2 booleanExpr

The **booleanExpr** element specifies a unary induced Boolean operation.

Requirement 21 https://standards.isotc211.org/19123/-3/1/req/core/booleanExpr
A **booleanExpr shall** be defined as:

Let

> C_1 be a **coverageExpr**,
> n be a positive integer number**.**

Then,

> for any **coverageExpr** C_2
> where
> C_2 = **`not`** C_1
> where n is an expression evaluating to a nonnegative integer value
>
> C_2 is defined as:

Coverage constituent
id(C_2) = “” (empty string)
crs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
rangeFieldNames (C_2) = *rangeFieldNames* (C_1) for all fields $r \in$ *rangeFieldNames*(C_2): *rangefieldType*(C_2,r) = `boolean`
for all $p \in$ *domain*(C_2): *value*(C_{not}, p) = `not`(*value*(C_1,p))

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

```
not $c
```

6.5.3.3 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, the conversion result can suffer from a loss of accuracy or overflow, up to being entirely wrong (such as when casting from long to short).

Requirement 22 https://standards.isotc211.org/19123/-3/1/req/core/castExpr
A **castExpr shall** be defined as:

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:

Coverage constituent
id(C_2) = “” (empty string)
crs(C_2) = *rs*(C_1)
domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
rangeFieldNames (C_2) = *rangeFieldNames* (C_1) for all fields $r \in$*rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = t
for all $p \in$*domain*(C_2): *value*(C_2, p) = (t) *value*(C_1,p)

EXAMPLE For some integer or float valued coverage the result range type of the following expression will be integer instead of float:

```
(integer) ( $c / 2 )
```

6.5.3.4 fieldExpr

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

NOTE Due to the current restriction to atomic range fields, the result of a field selection has atomic values too.

Requirement 23 https://standards.isotc211.org/19123/-3/1/req/core/fieldExpr
A **fieldExpr shall** be defined as:

Let

C_1 be a **coverageExpr**,
f be a **fieldName** appearing in *rangeFieldNames*(C_1),
i be an **integer** with $0 \leq i < |$*rangeFieldNames*(C_1)$|$.

Then,

for any **coverageExpr** C_2
where C_2 is one of:

$C_{2,f}$ = C_1 . f
$C_{2,I}$ = C_1 . i

C_2 is defined as:

Coverage constituent
id(C_2) = “” (empty string)
crs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
rangeFieldNames (C_2) = (f), the sequence containing only f *rangeFieldType*(C_2,f) = *rangeFieldType*(C_1,f)
for all $p \in$*domain*(C_2): *value*($C_{2,f}$,p) = *value*(C_1 . f,p) *value*($C_{2,i}$,p) = *value*(C_1 . g,p) where g is the i^{th} field in *rangeFieldNames*(C_1)

EXAMPLE Let `$c` refer to anexpression resulting in a coverage of with two bands, red and green. Then the following expression describes a single-field, integer-type coverage where each grid point value contains the ratio between red and green band, cast back to integer from the division result type float:

```
( integer ) $c.red / $c.green
```

Requirement 24 https://standards.isotc211.org/19123/-3/1/req/core/fieldExprShorthand
In a **fieldExpr** C. f where **|***rangeFieldNames*(C)|=1, the evaluation of C. f **shall** be identical to the evaluation of C.

EXAMPLE Let `$c` refer to a coverage expression with range component `red`, `$d` a single-component range type (say, a panchromatic satellite scene). Assuming both are compatible (as per induced expression definition) the following expression is valid:

```
$c.red - $d
```

6.5.4 binaryInducedExpr

The **binaryInducedExpr** element specifies a binary induced operation, i.e. an operation involving two coverage-valued arguments.

Requirement 25 https://standards.isotc211.org/19123/-3/1/req/core/binaryInducedExprNumber
In a **binaryInducedExpr**, both participating coverages **shall** be aligned in the following components:
- same native CRS;
- same domain;
- same number of range components;
- same interpolation for each axis.

Requirement 26 https://standards.isotc211.org/19123/-3/1/req/core/binaryInducedExpr
A **binaryInducedExpr shall** be defined as:

Let

C_1, C_2 be **coverageExpr**s,
N be 0 or some null value (to be defined by a concretization of this document)
where
crs(C_1) = *crs*(C_2),
domain(C_1,a) = *domain*(C_2,a),
rangeFieldNames(C_1) = *rangeFieldNames*(C_2),
rangeType(C_1,f) is cast-compatible with *rangeType*(C_2,f) or
rangeType(C_2,f) is cast-compatible with *rangeType*(C_1,f)
for all $f \in$*rangeFieldNames*(C_1).

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_3 is defined as:

Coverage constituent
id(C_3) = “” (empty string)
crs(C_3) = *crs*(C_1)
domain(C_3) = *domain*(C_1)
interpolation(C_3) = *interpolation*(C_1)
rangeFieldNames (C_3) = *rangeFieldNames* (C_1) for all $r \in$*rangeFieldNames*(C_3): *rangeFieldType*(C_{plusCC}, r) is given by Requirement 48 *rangeFieldType*(C_{minCC}, r) is given by Requirement 48 *rangeFieldType*(C_{multCC}, r) is given by Requirement 48 *rangeFieldType*(C_{divCC}, r) is given by Requirement 48 *rangeFieldType*(C_{andCC}, r) = `boolean` *rangeFieldType*(C_{orCC}, r) = `boolean`

Coverage constituent
$rangeFieldType(C_{xorCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{eqCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{ltCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{gtCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{leCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{geCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{neCC}, r) = \texttt{boolean}$ $rangeFieldType(C_{ovlCC}, r) = rangeFieldType(C_1, r)$
for all $p \in domain(C_3)$: $value(C_{plusCC}, p) = value(C_1, p) + value(C_2, p)$ $value(C_{minCC}, p) = value(C_1, p) - value(C_2, p)$ $value(C_{multCC}, p) = value(C_1, p) * value(C_2, p)$ $value(C_{divCC}, p) = value(C_1, p) / value(C_2, p)$ $value(C_{andCC}, p) = value(C_1, p) \texttt{ and } value(C_2, p)$ $value(C_{orCC}, p) = value(C_1, p) \texttt{ or } value(C_2, p)$ $value(C_{xorCC}, p) = value(C_1, p) \texttt{ xor } value(C_2, p)$ $value(C_{eqCC}, p) = value(C_1, p) = value(C_2, p)$ $value(C_{ltCC}, p) = value(C_1, p) < value(C_2, p)$ $value(C_{gtCC}, p) = value(C_1, p) > value(C_2, p)$ $value(C_{leCC}, p) = value(C_1, p) <= value(C_2, p)$ $value(C_{geCC}, p) = value(C_1, p) >= value(C_2, p)$ $value(C_{neCC}, p) = value(C_1, p) \,!= value(C_2, p)$ $value(C_{ovlCC}, p) = value(C_2, p)$ if $value(C_1, p)=N$ $value(C_1, p)$ otherwise

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
```

6.5.5 N-ary Induced operations

6.5.5.1 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 shall match wrt. domains and CRSs. An input coverage range field may be listed more than once.

Requirement 27 https://standards.isotc211.org/19123/-3/1/req/core/rangeConstructorExprNames
The names of the range fields generated by the operation **shall** be given by the names prefixed to each component expression.

Requirement 28 https://standards.isotc211.org/19123/-3/1/req/core/rangeConstructorExpr
A **rangeConstructorExpr shall** be defined as:

Let

n be an **integer** with $n \geq 1$,
$C_1, \ldots, C_n$ be **coverageExpr**s with $|rangeFieldNames(C_i)|=1$ (i.e., just a single range component),
$f_1, \ldots, f_n$ be **fieldName**s
where, for $1 \leq i,j \leq n$,

$crs(C_i) = crs(C_j)$,
$domain(C_i) = domain(C_j)$
$gridCrs(C_i) = gridCrs(C_j)$,
$interpolation(C_i) = interpolation(C_j)$.

Then,

for any **coverageExpr** C'
where C' is one of

C'_a = `{` f_1 `:` C_1 `;` … `;` f_n `:` C_n `}`
C'_b = **`struct`** `{` f_1 `:` C_1 `;` … `;` f_n `:` C_n `}`

C' is defined as:

Coverage constituent
$id(C')$ = “” (empty string)
$crs(C') = crs(C_1)$
$domain(C') = domain(C_1)$
$rangeFieldNames(C') = (f_1, \ldots, f_n)$ for all range fields f_i: $rangeFieldType(C', f_i) = rangeFieldType(C_i)$
for all $p \in domain(C')$: $value(C'.f_i, p) = value(C_i, p)$
for all range fields f_i: $interpolation(C') = interpolation(C_1)$

EXAMPLE 1 The expression below shows a false colour encoding by combining near-infrared, red and green bands into a 3-band image, which can potentially be visually interpreted as RGB:

```
struct {
  red:   $c.nir;
  green: $c.red;
  blue:  $c.green
}
```

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

```
struct {
  red:   $g.panchromatic;
  green: $g.panchromatic;
```

```
            blue:  $g.panchromatic
          }
```

6.5.5.2 switchExpr

The **switchExpr** provides a case distinction for choosing among a set of coverages that all share domain and range type. Conditions provided are evaluated sequentially, and the first *true* alternative is chosen if any; otherwise, the default alternative is chosen.

— If the result expressions return scalar values, the returned scalar value on a branch is used in places where the condition expression on that branch evaluates to *true*.

— If the result expressions return coverages, the values of the returned coverage on a branch are copied in the result coverage in all places where the condition coverage on that branch contains pixels with value *true*.

NOTE 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++. This implies that the conditions needs to be specified by order of generality, starting with the least general and ending with the default result, which is the most general one. A less general condition specified after a more general condition will be ignored, as the expression meeting the less general expression will have had met already the more general condition.

Requirement 29 https://standards.isotc211.org/19123/-3/1/req/core/switchExpr
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 `booleanExpr`s with a single Boolean range component,
$C_1, \ldots, C_n$ be `coverageExpr`s with a single Boolean range component,
$R, R_1, \ldots, R_{n+1}$ be `coverageExpr`s,

where, for $1 \leq i \leq n$,

$crs(C_1) = \ldots = crs(C_n) = crs(R_1) = \ldots = crs(R_{n+1})$,
$domain(C_1) = \ldots = domain(C_n) = domain(R_1) = \ldots = domain(R_{n+1})$,
$interpolation(C_1) = \ldots = interpolation(C_n) = interpolation(R_1) = \ldots = interpolation(R_{n+1})$,
$rangeType(R_1) = \ldots = rangeType(R_{n+1})$.

Then,

for any `coverageExpr` C'
where

```
    C' =  switch
          case C1 return R1
          ...
          case Cn return Rn
          default return Rn+1
```

C' is defined as:

Coverage constituent
$id(C')$ = “” (empty string)

crs(C') = *crs*(R_1)
domain(C') = *domain*(R_1)
interpolation(C') = *interpolation*(R_1)
rangeType(C') = rangeType(R_1)

for all $p \in$ *domain*(C'):
    value(C', p) = V
    where V =
        if *value*(C_1,p) then *value*(R_1,p)
    else if *value*(C_2,p) then *value*(R_2,p)
    ...
    else if *value*(C_n,p) then *value*(R_n,p)
                else *value*(R_{n+1},p)

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 a 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 if any cell were not above zero.

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

6.5.6 Coverage Domain-Changing Expressions

6.5.6.1 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, 7.5.6.2), slicing (i.e. cutting out a hyperplane from a coverage, 7.5.6.3), extending (7.5.6.3), and scaling (7.5.7) of a coverage expression.

Requirement 30 https://standards.isotc211.org/19123/-3/1/req/core/subsetExpr
A **subsetExpr shall** be either a **trimExpr** (7.5.6.2) or a **sliceExpr** (7.5.6.3) or an **extendExpr** (7.5.6.3) or a **scalingExpr** (7.5.7).

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** (see 7.5.3.4).

6.5.6.2 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.

Requirement 31 https://standards.isotc211.org/19123/-3/1/req/core/trimExprInside
In a **trimExpr** lower as well as upper limits **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 32 https://standards.isotc211.org/19123/-3/1/req/core/trimExpr
A **trimExpr shall** be defined as:

Let

> C_1 be a **coverageExpr**,
> n be an **integer** with $0 \leq n$,
> $(lo_1:hi_1),\ldots,(lo_n:hi_n)$ be **dimensionIntervalExpr**s with $lo_i \leq hi_i$ for $1 \leq i \leq n$.

Then,

> for any **coverageExpr** C_2
> where C_2 is one of
> $C_{\text{bracket}} = C_1 [p_1 , \ldots , p_n]$
> with
> p_i is one of
> $p_{\text{nat},I} = a_i (lo_i : hi_i)$
> $p_{\text{crs},I} = a_i : crs_i (lo_i : hi_i)$
>
> where each interval is within the coverage's bounds, as expressed by interval and axis (possibly reprojected from an optional CRS indicated).
>
> C_2 is defined as:

Coverage constituent
id(C_2) = "" (empty string)
crs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1) reduced to extent $(lo_i:hi_i)$ for any domain axis a_i (reprojected from crs_i into the coverage CRS if crs_i is present), and with domain extent properly adjusted for any index axis a_i present in the trim list
interpolation(C_2) = *interpolation*(C_1)
rangeType(C_2) = *rangeType*(C_1)
for all $p \in$ *domain*(C_2): *value*(C_2, p) = *value*(C_1, p)

EXAMPLE The following are syntactically valid, equivalent trim expressions:

```
$c[ Lon (-120: -80), Lat (-10: +10) ]
```

6.5.6.3 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.

Requirement 33 https://standards.isotc211.org/19123/-3/1/req/core/sliceExprCoordinatesInside
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 34 https://standards.isotc211.org/19123/-3/1/req/core/sliceExpr
A **sliceExpr shall** be defined as:

Let

C_1 be a **coverageExpr**,
n be an **integer** with $0 \leq n$,
$a_1, \ldots, a_n$ be pairwise distinct **axisName**s with $a_i \in axisNameSet(C_1)$ for $1 \leq i \leq n$,
$s_1, \ldots, s_n$ be **axisPointExpr**s for $1 \leq i \leq n$. which evaluate, according to normal arithmetic rules, to coordinate values.

Then,

for any **coverageExpr** C_2
where C_2 is one of
$C_{bracket}$ = C_1 **[** S_1 **,** … **,** S_n **]**
with
S_i is one of
$S_{nat,I}$ = a_i **(** s_i **)**
$S_{crs,I}$ = a_i **:** crs_i **(** s_i **)**

C_2 is defined as:

Coverage constituent
id(C_2) = "" (empty string)
crs(C_2) = *crs*(C_1) projected to the axes remaining
domain(C_2) = *domain*(C_1) reduced to the axes of *nativeCrs*(C_2)
interpolation(C_2) = *interpolation*(C_1)
rangeType(C_2) = *rangeType*(C_1)
for all $p \in domain(C_1)$: *value*(C_2, p) = *value*(C_1, p') where p' is the projection of p to *nativeCrs*(C_2)

EXAMPLE The following is a valid slice expression:

```
$c[ Date ( "2021-08-28" ) ]
```

6.5.6.4 extendExpr

The **extendExpr** element extends a coverage to the bounding box indicated. How the new grid points are filled with values is implementation-dependent (for example, null is an appropriate value).

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. Hence, the operation can extend or reduce the footprint in each axis individually.

NOTE 1 In this sense the **extendExpr** is a generalization of the **trimExpr**; nevertheless, it is best to use the **trimExpr** whenever the application wants to be sure that a proper subsetting has to take place.

Extension is only possible where the new coordinates can be extrapolated. This is the case for index and regular axes, and therefore no extension along an irregular axis is possible.

Requirement 35 https://standards.isotc211.org/19123/-3/1/req/core/extendExpr
An **extendExpr shall** be defined as:

Let

C_1 be a **coverageExpr**,
n be an **integer** with $0 \le n$,
$a_1, \ldots, a_n$ be pairwise distinct **axisName**s with $a_i \in$ *axisList*(*nativeCrs*(C_1)) for $1 \le i \le n$,
$crs_1, \ldots, crs_n$ be **crsName**s with $crs_i \in$ crsList(C_1) for $1 \le i \le n$,
(lo_1:hi_1),…,(lo_n:hi_n) be **dimensionIntervalExpr**s with $lo_i \le hi_i$ for $1 \le i \le n$,
N be 0 or NaN or some null value (to be defined by a concretization of this document).

Then,

for any **coverageExpr** C_2
where
C_2 = **extend** (C_1, { p_1, …, p_n })
with
p_i is one of
$p_{nat,I}$ = a_i (lo_i : hi_i)
$p_{crs,I}$ = a_i : crs_i (lo_i : hi_i)

C_2 is defined as:

Coverage constituent
id(C_2) = "" (empty string)
crs(C_2) = *crs*(C_1)

domain(C_2) = *domain*(C_1) adjusted to extent (lo_i:hi_i) for any domain axis a_i (reprojected from crs_i into the coverage nativeCRS if crs_i is present), and with domain extent properly adjusted for any axis a_i present in the extend list; axes not mentioned remain unchanged.
interpolation(C_2) = *interpolation*(C_1)
rangeType(C_2) = *rangeType*(C_1)
for all $p \in$ *domain*(C_2): *value*(C_2, p) = *value*(C_1,p) for $p \in$ *domain*(C_1) *value*(C_2, p) = N otherwise

NOTE 2 A concretization can restrict the CRSs available on the result, as not all CRSs necessarily are technically appropriate.

EXAMPLE The following is a valid *extend*() expression:

```
extend( $c, { x ( -200 : +200 ) } )
```

6.5.7 scaleExpr

The **scaleExpr** element reduces resolution of a grid coverage while leaving the geographic extent unchanged. The new target resolution is specified by a grid interval along each axis.

NOTE 1 Scaling regularly involves range interpolation, hence numerical effects have to be expected.

Requirement 36 https://standards.isotc211.org/19123/-3/1/req/core/scaleExpr1
A **scaleExpr shall** be defined as:

Let

C_1 be a **coverageExpr** with only index and regular grid axes,
m, n be **integer**s with $0 \leq m$ and $0 \leq n$,
$a_1, \ldots, a_m$ be pairwise distinct **axisName**s with $a_i \in$ *gridCrs*(C_1) for $1 \leq i \leq m$,
I_i be **intervalExpr**s for $1 \leq i \leq m$ which evaluate to pairs lo_i, hi_i with $lo_i \leq hi_i$.

Then,

For any **coverageExpr** C_2,
where
C_2 = **scale (C_1, { a_1 (I_1), …, a_m (I_m)})**

C_2 is defined as:

Coverage constituent
id(C_2) = “” (empty string)
rs(C_2) = *crs*(C_1)
domain(C_2) = *domain*(C_1)

interpolation(C_2) = *interpolation*(C_1)
rangeType(C_2) = *rangeType*(C_1)
for all $p \in$*domain*(C_2): *value*(C_2, p) is obtained by rescaling the coverage grid along dimensions a_i such that the coverage's extent along dimension a_i is set to (lo_i : hi_i), expressed in the coverage's grid CRS; all other dimensions remain unaffected. Whenever interpolation is needed the respective axis interpolation method of the coverage expression gets applied.

EXAMPLE The following expression performs x/y scaling of some coverage referenced by variable *$c* using the interpolation method of each coverage axis. Note that *$c* can have further axes, such as time, which would remain unaffected.

```
scale( $c, { x ( 100: 200), y ( 300: 400) } )
```

NOTE 2 In practice, a concretization will provide several variants of scaling for convenience.

6.6 Coverage Derivation Expressions

6.6.1 crsTransformExpr

The **crsTransformExpr** element performs reprojection of a coverage from its native CRS into another one; the dimension of the coverage as well as the axis types (such as regular vs. irregular) remains unchanged whereas axes and range values generally change. For the interpolation and resampling which is usually incurred, the interpolation method to be applied can be indicated optionally.

NOTE 1 This changes the range values (e.g. pixel radiometry).

NOTE 2 Some CRS combinations can be not supported.

Requirement 37 https://standards.isotc211.org/19123/-3/1/req/core/crsTransformExpr
A **crsTransformExpr shall** be defined as:

Let

C_1 be a **coverageExpr**,
c be a **crsName**.

Then,

for any **coverageExpr** C_2
where
C_2 = **crsTransform(** C_1, c **)**

C_2 is defined as:

Coverage constituent
id(C_2) = “” (empty string)
crs(C_2) = c

domain(C_2) = *domain*(C_1)
interpolation(C_2) = *interpolation*(C_1)
rangeFieldNames(C_2) = *rangeFieldNames*(C_1) for all range fields $r \in$ *rangeFieldNames*(C_2): *rangeFieldType*(C_2,r) = *rangeFieldType*(C_1,r)
for all $p \in$ *domain*(C_2): *value*(C_2,p) is obtained by reprojecting coverage C_1 from its CRS into CRS c. Interpolation will be applied as necessary.

EXAMPLE The following expression transforms coverage `$c` (which is assumed to be 2D with some not further specified CRS) into the CRS identified by EPSG:3035.

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

6.7 Coverage Aggregation Expressions

6.7.1 condenseExpr

Requirement 38 https://standards.isotc211.org/19123/-3/1/req/core/condenseExpr
A **condenseExpr shall** be either a **reduceExpr** (see 6.7.3) or a **generalCondenseExpr** (see 6.7.2).

This expression takes a coverage and summarizes its values using a summarization function. 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.

NOTE In practice, aggregation results can be null if aggregation encounters null values in the coverage expression. The handling of null values is is governed by the value set definition which is out of scope of this document. It depends on whether a concretization defines types with null values included. It is expected, though, that a concretization will define null value handling in a way that for every direct position evaluated, if any of the values participating is null, then the result for this direct position will be null.

6.7.2 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 **condenseOpType**s are the binary operations `+`, `*`, `max`, `min`, `and`, and `or`.

Requirement 39 https://standards.isotc211.org/19123/-3/1/req/core/generalCondenseExpr
A **generalCondenseExpr shall** be defined as:

Let

op be a **condenseOpType**,
n be some **integer** with $n \geq 0$,
d be some **integer** with $d > 0$,
$axis_i$ be **axisName**s for $1 \leq i \leq d$,
$name_i$ be pairwise distinct **variableName**s for $1 \leq i \leq d$ which, in the request on hand, are not used already as a variable in this expression's scope,

I_i be **intervalExpr**s for $1 \le i \le d$ which evaluate to pairs lo_i, hi_i with $lo_i \le hi_i$,
C_j be **coverageExpr**s for $1 \le j \le n$,
P be a **booleanExpr** possibly containing occurrences of $name_i$ and C_j,
V be a **scalarExpr** or **coverageExpr** possibly containing occurrences of $name_i$ and C_j,
N be a neutral element of type(V)
where
$1 \le i \le d$.

Then,

For any **scalarExpr** S
where S is one of

```
S′ =  condense op
      over name₁ axis₁ ( I₁ ),
           …,
           name_d  axis_d ( I_d )
      [ where P ]
      using V

S″ =  condense op
      over axis₁ ( I₁ ),
           …,
           axis_d ( I_d )
      [ where P ]
      using V
```

S is constructed as follows (for S'', substitute $name_i$ by $axis_i$):

```
S := N;
for all name₁ ∊ {lo₁,… ,hi₁}
  for all name₂ ∊ {lo₂,… ,hi₂}
    …
      for all name_d ∊ {lo_d,… ,hi_d}
        if (filtering expression P is present)
        then
          let predicate P′ be obtained from evaluating expression
          P by substituting all occurrences of name_i by its current
          value where name_i occurring in a coordinate position
          of C_j are coordinates in the CRS of C_j
        else
          P′ = true;
        fi
        if (P′)
        then
          let V′ be obtained from evaluating expression V
          by substituting all occurrences of name_i by its current
          value where name_i occurring in a coordinate position
          of C_j are coordinates in the CRS of C_j where
          possible extra dimensions in a coverageExpr are
          treated as in induced operations;
          S := S op value(V′)
        fi
      endfor
    …
    endfor
```

```
  endfor
return S
```

NOTE 1 Condensers are heavily used, among others, in two situations:

— to collapse Boolean-valued coverage expressions into scalar Boolean values so that they can be used in predicates;

— in conjunction with the **coverageConstructorExpr** (see 6.3.1.1) to phrase high-level imaging, signal processing, and statistical operations.

NOTE 2 The additional expressive power of **condenseExpr** over **reduceExpr** is twofold:

— a concretization can offer further summarisation functions, as long as these form a monoid, i.e. they are commutative and associative and have a neutral element;

— the **condenseExpr** gives explicit access to the coordinate values; this makes summarization considerably more powerful (see example below).

EXAMPLE 1 The following expression iterates over a 5000x5000 extent of image *$c* delivering the sum of all values encountered at the direct positions:

```
condense +
overx ( 0 : 4999 ), y ( 0 : 4999 )
using $c[ i(x) , j(y) ]
```

EXAMPLE 2 Iteration is possible also in native coordinates as the direct positions are uniquely identified:

```
condense +
overy ( 20 : 30 ), x ( 40 : 50 )
using $c[ Lat(y) , Lon(x) ]
```

EXAMPLE 3 A timeline diagram can be obtained through a 1D expression which aggregates over space while iterating over time:

```
coverage AverageTemperature
domain
  crs "OGC:DateTime" with t ( domain( $temperatureCube, Date )
)
range type t: float
range
  condense +
  over  lat ( domain( $temperatureCube, Lat ) ),
        lon ( domain( $temperatureCube, Lon ) )
  using $temperatureCube[ Lat(lat), (Lon(lon), Date( t ) ]
```

EXAMPLE 4 For a filter kernel *k*, the condenser summarizes not only over the grid point under inspection, but also some neighbourhood. The following applies a 3x3 filter kernel to band `b` of some coverage *$c* with extent x0…x1/y0…y1; note that the result image is defined to have an *x* and *y* dimension:

```
Coverage FilteredImage
domain
  crs "OGC:Index2D" with x (0 : 4999 ), y ( 0 : 4999 )
range type f: int
range
  condense +
  over i ( -1 : +1 ),
```



```
          j ( -1 : +1 )
    using $c[ x+i , y+j ] * k[ i, j ]
```

where k is a 3x3 matrix like:

1	2	1
0	0	0
-1	-2	-1

NOTE See **coverageConstantExpr** for a way to specify the k matrix.

6.7.3 reduceExpr

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

NOTE All these operations can be expressed through a **condenseExpr**, but in a more verbose way.

Requirement 40 https://standards.isotc211.org/19123/-3/1/req/core/reduceExpr
A **reduceExpr shall** be either an `add`, `avg`, `min`, `max`, `count`, `some`, or `all` operation as per Table 5.

NOTE Within Table 5, *$a* is assumed to evaluate to a coverage with a single numeric range field, *$b* to a coverage with a single Boolean range field.

Table 5 — reduceExpr definition via generalCondenseExpr

reduceExpr definition	Description

`add($a) =`
`condense +`
`over $p₁ (domain($a,D₁)),`
`…,`
`$p_d (domain($a,D₁)),`
`using $a[$p₁ , …, $p_d]`
	sum over all points in *$a*

`avg($a) =` `add($a) / | domain($a) |`	average of all points in *$a*

`min($a) =`
`condense min`
`over $p₁ (domain($a,D₁)),`
`…,`
`$p_d (domain($a,D₁))`
`using $a[ $p₁ , …, $p_d ]`
	minimum of all points in *$a*

`max($a) =`
`condense max`
`over $p₁ (domain($a,D₁)),`
`…,`
`$p_d (domain($a,D₁))`
`using $a[ $p₁ , …, $p_d]`
	maximum of all points in *$a*

`count($b) =`
`condense +`
`over $p₁ (domain($b,D₁)),`
`…,`
	number of points in *$b*

reduceExpr definition	Description

`$p_d (domain($b,D_1))`
`   where $b[ $p_1 , …, $p_d]`
`   using 1`	

`some($b) =`
`   condense or`
`   over  $p_1 (domain($b,D_1)),`
`         …,`
`         $p_d (domain($b,D_1))`
`   using $b[ $p_1 , …, $p_d ]`
	is there any point in *$b* with value true?

`all($b) =`
`   condense and`
`   over  $p_1 D_1(domain($b,D_1)),`
`         …,`
`         $p_d D_d(domain($b,D_1))`
`   using $b[ $p_1 , …, $p_d ]`
	do all points of *$b* have value true?

EXAMPLE The previous average temperature example can be expressed through a more compact range:

```
coverage AverageTemperature
domain
  crs “OGC:DateTime” with t ( domain( $temperatureCube, Date )
)
range type t: float
range
  avg( $temperatureCube[Date( t )]
```

6.8 Coverage Encode/Decode Expressions

6.8.1 encodeCoverageExpr

The **encodeCoverageExpr** element specifies encoding of a coverage-valued query result by means of a data format and possible extra encoding parameters.

Data format encodings are not in the scope of this document.

Requirement 41 https://standards.isotc211.org/19123/-3/1/req/core/encode
An **encodeCoverageExpr shall** be defined as:

Let

C be a **coverageExpr**,
f be a string
where
  f is a **stringConstant**,
  extraParams be a **stringConstant**.

Then,

for any **string** *S*
where *S* is one of

S_e = **encode** (*C*,*f*)
S_{ee} = **encode** (*C*,*f*, *extraParams*)

S is defined as that (binary or printable) byte string which encodes *C* into the data format specified by *formatName* and the optional *extraParams*.

Syntax and semantics of both *f* and the *extraParams* are not specified in this document. A set of suitable data formats is expected to be provided by a concretization of this language.

NOTE It is acceptable that some format encodings can lead to a loss of information, not allowing the reconstruction of a complete coverage or the reuse of it in a *decode*() operation.

EXAMPLE The following expression can retrieve coverage `$c` encoded in JPEG with a quality factor of 50 %:

```
encode( $c, "image/jpg", ".50" )
```

6.8.2 decodeCoverageExpr

A *decodeCoverageExpr* evaluates a byte stream passed as a parameter to a coverage by decoding the byte stream. This byte stream is required to represent a coverage encoding following CIS 1.1 [09-146r6][8] and its coverage encoding profiles.

NOTE Implementations will be able to recognize the encoding format used from analyzing the input byte stream, hence normally no format indication parameter is required. Generally, though, the *extraParams* syntax and semantics is data format and implementation dependent.

Requirement 42 https://standards.isotc211.org/19123/-3/1/req/core/decode
Syntax and semantics of a *decodeCoverageExpr* **shall** be given as follows.

Let

s be a *string*

where

s is a valid (binary or printable) representation of a complete coverage or a domain, range type, range, or metadata component of a coverage,
extraParams is a **stringConstant** containing decoding directives.

Then,

for any *decodeCoverageExprC*
where *C* is one of

C_e = decode(*s*)
C_{ee} = decode(*s*, *extraParams*)

C is defined as the decoded coverage or coverage component equivalent to *s* while applying the directives in *extraParams*.

In practice, this function can be used in several ways:

— to provide inline constants, encoded, for example, in XML or JSON;

— to provide complete input files, accompaniying the query, through positional parameters;

— to provide input coverages and other values by reference, such as through URIs.

EXAMPLE Assume a NetCDF file is passed as a single extra parameter in some concrete service. The service will decode the NetCDF byte stream and establish the corresponding coverage before further evaluation of the complete query:

```
decode( $1 )
```

6.9 Expression evaluation

This sublause defines additional rules for *ProcessCoverages* expression evaluation.

6.9.1 Evaluation sequence

Requirement 43 https://standards.isotc211.org/19123/-3/1/req/core/sequence
A **processingExpr shall** evaluate coverage expressions from left to right.

6.9.2 Nesting

Requirement 44 https://standards.isotc211.org/19123/-3/1/req/core/nesting
A **processingExpr shall** allow nesting all operators, constructors and functions arbitrarily, provided that each sub-expression's result type matches the required type at the position where the sub-expression occurs, and all semantics rules are fulfilled.

6.9.3 Parentheses

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

Requirement 45 https://standards.isotc211.org/19123/-3/1/req/core/parentheses
Parentheses enforcing evaluation sequence in a **processingExpr shall** be defined as:

Let

C_1 and C_2 be **coverageExpr**s.

Then,

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

C_2 is defined as yielding the same result as C_1 **.**

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

6.9.4 Operator precedence rules

Requirement 46 https://standards.isotc211.org/19123/-3/1/req/core/precedence
In case of ambiguities in the syntactical analysis of a request, operators **shall** have the following precedence (listed in descending strength of binding):

— Range field selection, trimming, slicing

— unary −

— unary arithmetic, trigonometric, and exponential functions

— binary `*`, `/`

— binary `+`, `-`

— binary <, <=, >, >=, != , =

— binary `and`

— binary `or`, `xor`

— :(interval constructor), condense, coverage, coverage constructor

— `Overlay`, `switch`

In all remaining cases evaluation **shall** be performed left to right.

6.9.5 Range type compatibility and extension

A range type t_1 is said to be **cast-compatible** with a range type t_2 if the following conditions hold:

— Both range types, t_1 and t_2, have the same number of field elements, say d;

— For each range field element position i with $1 \leq i \leq d$, the i^{th} range field type $f_{1,i}$ of t_1 is **cast-compatible** with the i^{th} range field type $f_{2,i}$ of t_2.

Cast compatibility is expected to be defined in detail in a concretization of this language.

Requirement 47 https://standards.isotc211.org/19123/-3/1/req/core/typeExtension
The type of each of the operands of an arithmetic operator (+, -, *, /) **shall** be a type that can be extended to a numeric numeric type, and the result type of anarithmetic expression shall be the common extended type of all of its operands as:
If the extended type is integer then integer arithmetic **shall** be performed.
If the extended type is float then floating-point arithmetic **shall** be performed.
If the extended type is complex then complex arithmetic **shall** be performed.
The result type **shall** be the smallest type allowing to represent the result without loss.

NOTE Explicit and implicit casts need to be used with caution, as unintended consequences can arise. Data can be lost when floating-point representations are converted to integral representations as the fractional components of the floating-point values will be truncated (rounded down). Conversely, converting from an integral representation to a floating-point one can also lose precision, since the floating-point type can potentially be unable to represent the integer exactly (for example, float possibly gets mapped to an IEEE 754 single precision type, which cannot represent the integer 16777217 exactly, while a 32-bit integer type can). This can lead to situations such as storing the same integer value into two variables of type int and type float which return false if compared for equality.

6.10 Evaluation response

If, for whatever reason, the query cannot be evaluated properly then an *error* is returned as the evaluation result. On an abstract level, an error is a possible result value not equal to any valid result.

Requirement 48 https://standards.isotc211.org/19123/-3/1/req/core/error
Whenever a coverage expression cannot be evaluated according to the rules specified in 6.1 and 6.8, evaluation **shall** respond with an error.

NOTE Concretizations of this specification will define some appropriate behaviour depending on the target environment, such as return codes, exceptions, etc. Even not all syntactically valid expressions will be semantically admissible in practice. Possible issues include: quota are exceeded, access restrictions apply.

EXAMPLE The following expressions will lead to an error (reasons: division by zero; illegal trigonometric argument):

```
$C / 0

arcsin( 2 )
```

The result of evaluating a **processCoveragesExpr** is one of the following:

Requirement 49 https://standards.isotc211.org/19123/-3/1/req/core/result
Depending on its result type, the normal result of evaluating a valid query **shall** consist of one of the following alternatives:

— a (possibly empty) list of coverages;

— a (possibly empty) list of scalars (where scalar summarizes all non-coverage type data, such as numbers, strings, URLs) or of records of scalars;

— an error.

Annex A
(normative)

Conformance Tests

A.1 Conformance Class

This document defines one conformance class, Coverage Processing, which constitutes the mandatory Core every standardization target shall support.

Standardization targets are specifications containing provisions for coverage processing. A specification claiming conformance to this document shall implement the Coverage Processing conformance class.

Conformance with this document shall be assessed using all conformance test cases specified in this annex.

A.2 Conformance Class Coverage Processing Core

Conformance test	**https://standards.isotc211.org/19123/-3/1/conf/core/allRequirements**
Reference	All normative statements in requirements class: *Coverage Processing*
Test purpose:	Verify that the specification under test conforms to all requirements of this conformance class
Test method:	Evaluate every requirement of this conformance class in turn; the overall test passes if every single test passes
Test type:	Basic

Annex B
(normative)

Expression Syntax

B.1 Overview

This annex summarizes the coverage processing expression syntax. It is described in W3C EBNF grammar syntax.[5]

NOTE 1 This is a machine-readable language not requiring formal translation into ISO-supported languages.

Tokens in single quotation marks represent literals which appear "as is" in a valid expression ("terminal symbols"). 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. The `processCoveragesExpr` nonterminal is the start of the production system.

Any number of whitespace characters (blank, tabulator, newline) may appear between tokens as long as parsing is unambiguous.

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

Meta symbols used are as:

— brackets ("`[…]`") denote optional elements which may occur or be left out;

— an asterisk after parentheses ("(…)`*`") denotes that an arbitrary number of repetitions of the parenthesis contents can be chosen, including none at all;

— a plus after parentheses ("(…)+") denotes that an arbitrary number of repetitions of the parenthesis contents can be chosen, at least one;

— a question mark after parentheses ("(…)`?`") denotes that zero or one of the parenthesis contents can be chosen;

— a vertical bar ("`|`") denotes alternatives from which exactly one shall be chosen;

— double slashes ("`//`") begin comments which continue until the end of the line. Comments are normative.

NOTE 2 The syntax as is remains ambiguous; the semantic rules listed in this document disambiguate the grammar.

B.2 Terminal symbols

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

A `variableName` shall adhere to the following regular expression: `$[a-zA-Z_][0-9a-zA-Z_]*`.

This regular expression describes a consecutive sequence of characters where the first character shall be either an alphabetical character or the “$” character and the remaining characters consist of decimal digits, upper case alphabetical characters, lower case alphabetical characters, underscore (“_”) and nothing else. The length of an identifier shall be at least 1.

A name shall adhere to the following regular expression: `([a-zA-Z_][0-9a-zA-Z_]*)|(“.+”)`.

NOTE This describes it to either be a consecutive sequence of digits and/or letters where the first character is a letter, or a non-empty string constant.

While this document does not make assumptions about particularities of atomic data types (such as short vs long integers, float vs double, and the associated bit lengths) the common basic data types Boolean, integer, float, and complex are assumed to be available (with complex syntactically being a composite expression, as usual).

A `booleanConstant` shall represent a logical truth value expressed as one of the literals “true” and “false” resp., whereby uppercase and lowercase characters shall not be distinguished.

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).

A `floatConstant` shall represent a floating point number in common decimal-point or exponential notation.

A `stringConstant` shall represent a character sequence enclosedin single or double quotes, with no mix of both in a single constant.

B.3 Processing Syntax

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

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

letBinding ::=
           variableName ':='  coverageExpr
         | scalarExpr
         | '[' intervalExpr ']'

processingExpr ::=
           encodeCoverageExpr
         | scalarExpr

formatName ::=
         stringConstant

extraParams ::=
         stringConstant

coverageExpr ::=
           coverageIdExpr
```



```
          | coverageConstructorExpr
          | coverageConstantExpr
          | getComponentExpr
          | inducedExpr
          | subsetExpr
          | crsTransformExpr
          | scaleExpr
          | decodeCoverageExpr

coverageIdExpr ::=
            coverageName

coverageConstructorExpr ::=
          'coverage' coverageName
          ( domainExpr )? ( rangeTypeExpr )? rangeSetExpr

domainExpr ::=
          'domain'
          'crs' nameOrString 'with'
          nameOrString axisDefExpr ( ',' nameOrString axisdefExpr )*
          ( interpolationExpr )?

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

interpolationMethod ::=
            none
          | name

axisDefExpr ::=
            'index' ( indexExpr ':' indexExpr )
          | 'regular' ( axisPointExpr ':' axisPointExpr )
            'resolution' axisPointExpr
          | 'irregular' ( axisPointExpr ( ',' axisPointExpr )* )

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

rangeComponent ::=
          name ':' rangeType

rangeType ::=
            'boolean'
          | ( 'unsigned' )? 'int'
          | 'float'
          | 'complex'

rangeSetExpr ::=
          'range' ( scalarExpr | rangeConstantExpr )

rangeConstantExpr ::=
          '<' constant ( ';' constant )* '>'

scalarExpr ::=
            getComponentExpr
```

```
          | booleanScalarExpr
          | numericScalarExpr
          | stringScalarExpr
          | '(' scalarExpr ')'

getComponentExpr ::=
            identifierExpr
          | crs '(' coverageExpr ')'        | getDomainExpr
          | interpolation '(' coverageExpr ')'

identifierExpr ::=
          | 'id' '(' coverageExpr ')'
          | 'name' '(' coverageExpr ')'

getDomainExpr ::=
            'domain' '(' coverageExpr ')'
          | 'domain' '(' coverageExpr ',' axisName ')'
          | 'domain' '(' coverageExpr ',' axisName ')' '.' 'lo'
          | 'domain' '(' coverageExpr ',' axisName ')' '.' 'hi'

booleanScalarExpr ::=
            booleanScalarExpr 'or' booleanScalarTerm
          | booleanScalarExpr 'xor' booleanScalarTerm
          | booleanScalarTerm

booleanScalarTerm ::=
            booleanScalarTerm 'and' booleanScalarFactor
          | booleanScalarFactor

booleanScalarFactor ::=
            numericScalarExpr compOp numericScalarExpr
          | stringScalarExpr compOp stringScalarExpr
          | not booleanScalarExpr
          | '(' booleanScalarExpr ')'
          | booleanConstant

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

numericScalarExpr ::=
            numericScalarExpr '+' numericScalarTerm
          | numericScalarExpr '-' numericScalarTerm
          | numericScalarTerm

numericScalarTerm ::=
            numericScalarTerm '*' numericScalarFactor
          | numericScalarTerm '/' numericScalarFactor
          | numericScalarFactor

numericScalarFactor ::=
            '(' numericScalarExpr ')'
```

```
          | '-' numericScalarFactor
          | 'round' '(' numericScalarExpr ')'
          | integerConstant
          | floatConstant
          | complexConstant
          | condenseExpr

stringScalarExpr ::=
            identifierExpr
          | stringConstant

inducedExpr ::=
            unaryInducedExpr
          | binaryInducedExpr
          | naryInducedExpr

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

unaryArithmeticExpr ::=
            '+' coverageAtom
          | '-' coverageAtom
          | 'sqrt' '(' coverageExpr ')'
          | 'abs' '(' coverageExpr ')'
          | 're' '(' coverageExpr ')'
          | 'im' '(' coverageExpr ')'

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

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

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

fieldExpr ::=
            coverageExpr '.' fieldName
          | coverageExpr '.' integerConstant
```

```
binaryInducedExpr ::=
           binaryInducedLogicExpr 'or' binaryInducedLogicTerm
         | binaryInducedLogicExpr 'xor' binaryInducedLogicTerm
         | binaryInducedLogicTerm

binaryInducedLogicTerm ::=
           binaryInducedLogicTerm 'and' binaryInducedLogicFactor
         | binaryInducedLogicFactor

binaryInducedLogicFactor ::=
           binaryInducedArithmExpr compOp binaryInducedArithmExpr
         | binaryInducedArithmExpr

binaryInducedArithmExpr ::=
           binaryInducedArithmExpr '+' binaryInducedArithmTerm
         | binaryInducedArithmExpr '-' binaryInducedArithmTerm
         | binaryInducedArithmTerm

binaryInducedArithmTerm ::=
           binaryInducedArithmTerm '*' binaryInducedArithmFactor
         | binaryInducedArithmTerm '/' binaryInducedArithmFactor
         | binaryInducedArithmFactor

binaryInducedArithmFactor ::=
           binaryInducedArithmFactor 'overlay' binaryInducedExpr
         | inducedExpr

naryInducedExpr ::=
           rangeConstructorExpr
         | switchExpr

rangeConstructorExpr ::=
         ( 'struct' )? '{' fieldName ':' scalarExpr
                          ( ';' fieldName ':' scalarExpr )* '}'

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

subsetExpr ::=
           trimExpr
         | sliceExpr
         | extendExpr
         | scalingExpr

trimExpr ::=
           coverageExpr '[' dimensionIntervalList ']'

dimensionIntervalExpr ::=
         dimensionIntervalExpr ( ',' dimensionIntervalExpr )*

dimensionIntervalExpr ::=
         axisExpr '(' axisPointExpr ':' axisPointExpr ')'
```



```
axisExpr ::=
         axisName ( ':' crsName )?

axisPointExpr ::=         axisName
         | floatConstant
         | stringConstant

sliceExpr ::=
         coverageExpr '[' axisPointElement ( ',' axisPointElement )* ']'

axisPointElement ::=
         axisExpr '(' axisPointExpr ')'

extendExpr ::=
         'extend' '(' coverageExpr ',' '{' dimensionIntervalList '}' ')'

scaleExpr ::=
         'scale' '(' coverageExpr ',' '{' dimensionIntervalList '}' ')'

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

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

decodeCoverageExpr ::=
         'decode' '(' stringConstant ( ',' extraParams )? ')'

condenseExpr ::=
           reduceExpr
         | generalCondenseExpr

generalCondenseExpr ::=
         'condense' condenseOpType
         'over' axisIterator ( ',' axisIterator )*
         ( 'where' booleanScalarExpr )?
         'using' scalarExpr

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

axisIterator ::=
         name [ axisName ] '(' intervalExpr ')'

intervalExpr ::=
         axisPointExpr ':' axisPointExpr

reduceExpr ::=
           'all' '(' coverageExpr ')'
         | 'some' '(' coverageExpr ')'
```

```
         | 'count' '(' coverageExpr ')'
         | 'add' '(' coverageExpr ')'
         | 'avg' '(' coverageExpr ')'
         | 'min' '(' coverageExpr ')'
         | 'max' '(' coverageExpr ')'

coverageName ::=
         nameOrString

crsName ::=
         nameOrString

axisName ::=
         nameOrString

fieldName ::=
         nameOrString

constant ::=
           stringConstant
         | booleanConstant
         | integerConstant
         | floatConstant
         | complexConstant

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

nameOrString ::=
           name
         | stringConstant
```

Annex C
(informative)

Syntax diagrams

Figures C.1 to C.70 provide graphical representations of the syntax (often called "syntax diagrams" or "railroad diagrams") for the reader's convenience. In case of deviation, the normative syntax in Annex B prevails.

NOTE 1 This is a machine language not requiring formal translation.

NOTE 2 Diagrams generated by RR - Railroad Diagram Generator.

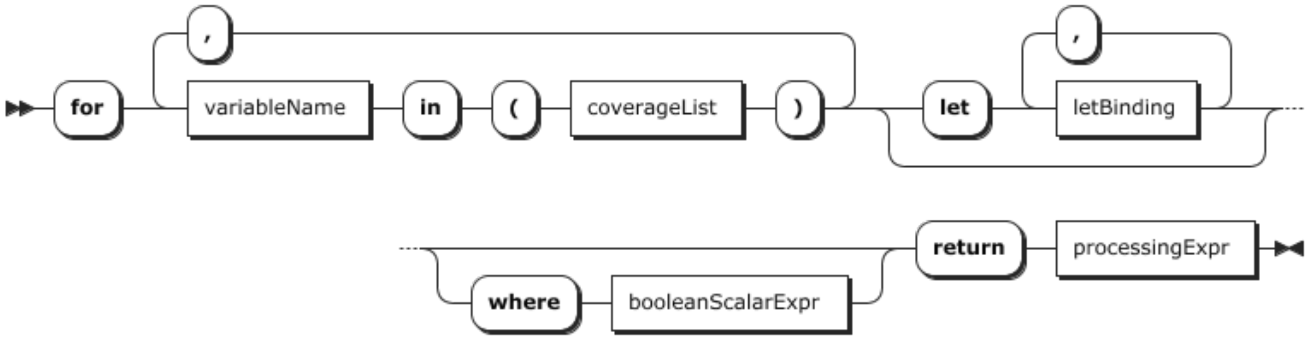

Figure C.1 — processCoveragesExpr

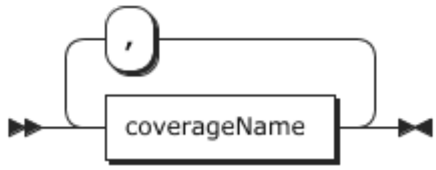

Figure C.2 — coverageList

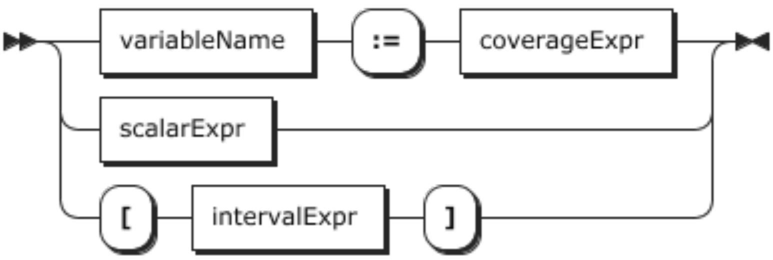

Figure C.3 — letBinding

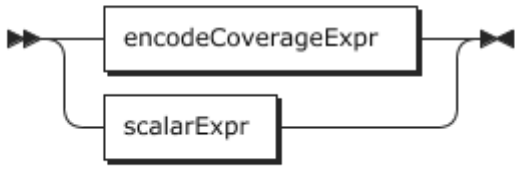

Figure C.4 — processingExpr

Figure C.5 — formatName

Figure C.6 — extraParams

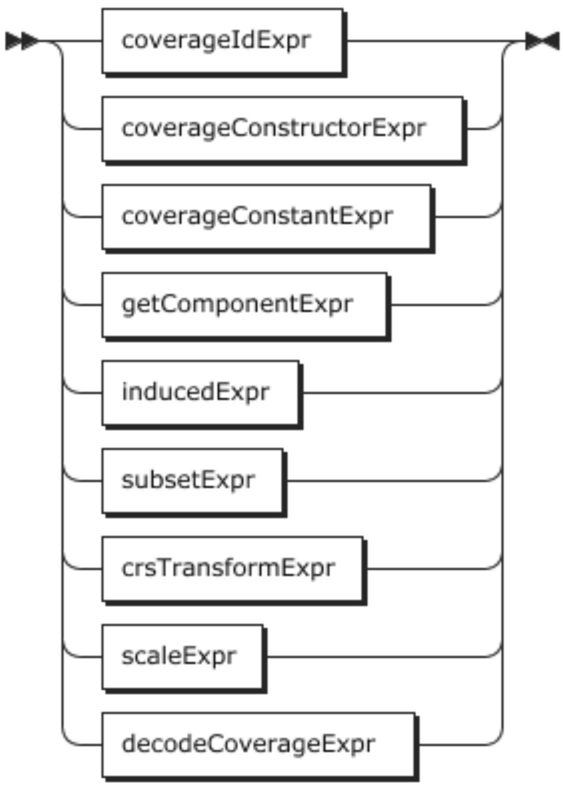

Figure C.7 — coverageExpr

Figure C.8 — coverageIdExpr

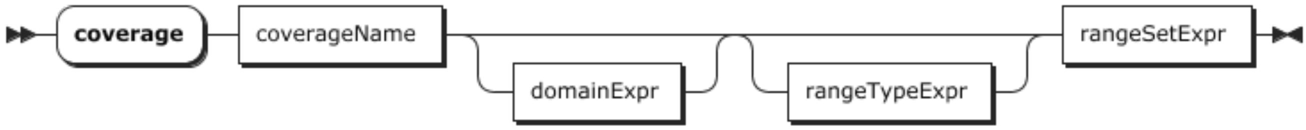

Figure C.9 — coverageConstructorExpr

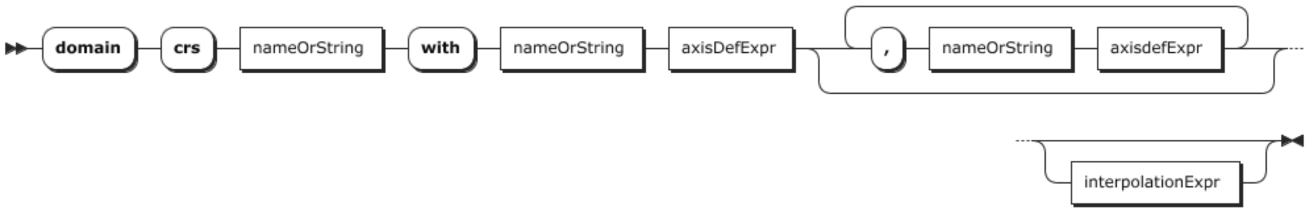

Figure C.10 — domainExpr

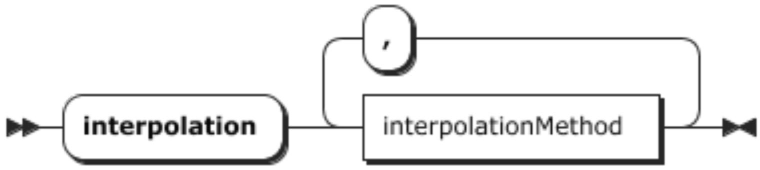

Figure C.11 — interpolationExpr

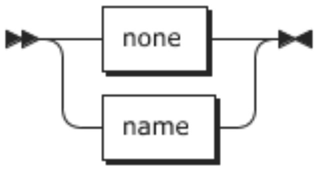

Figure C.12 — interpolationMethod

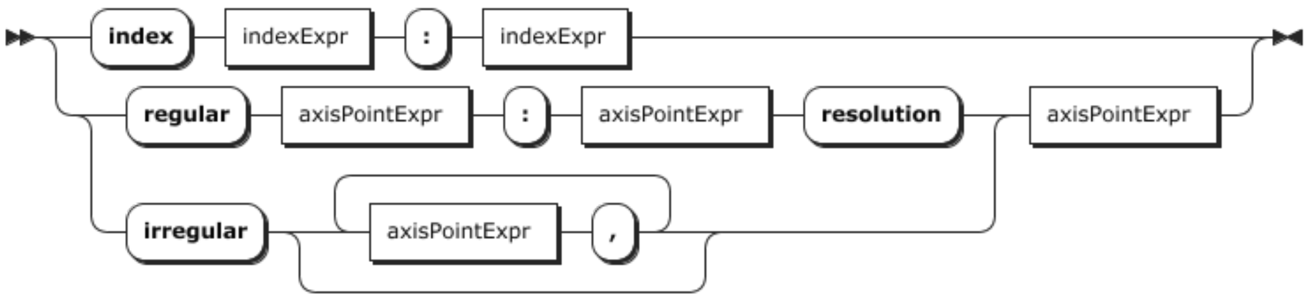

Figure C.13 — axisDefExpr

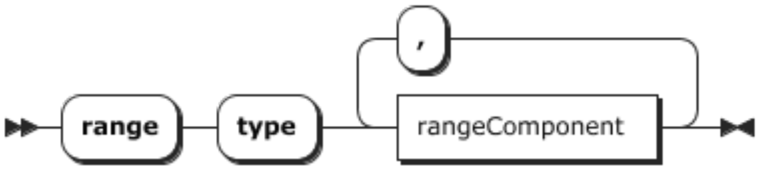

Figure C.14 — rangeTypeExpr

Figure C.15 — rangeComponent

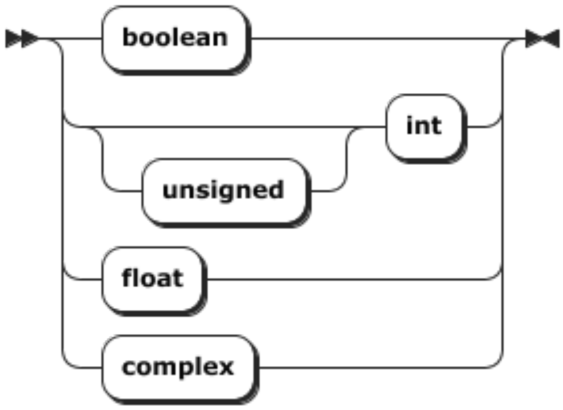

Figure C.16 — rangeType

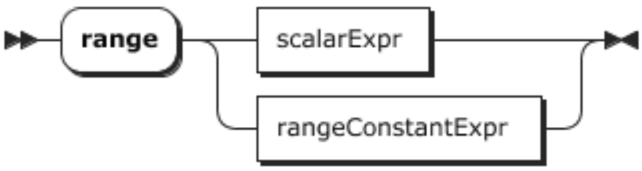

Figure C.17 — rangeSetExpr

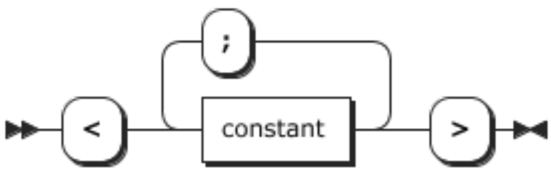

Figure C.18 — rangeConstantExpr

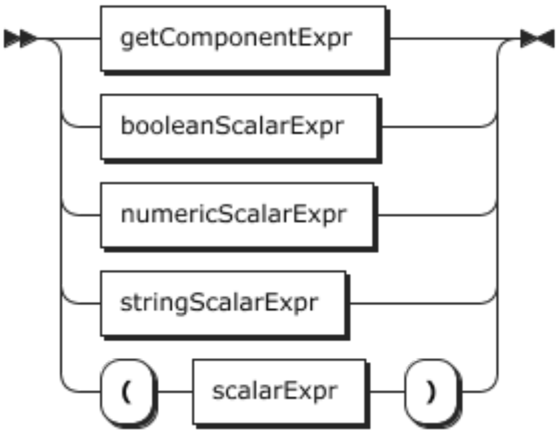

Figure C.19 — scalarExpr

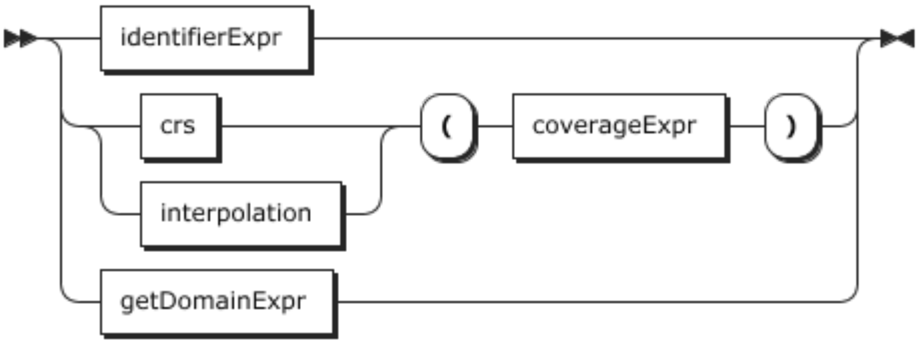

Figure C.20 — getComponentExpr

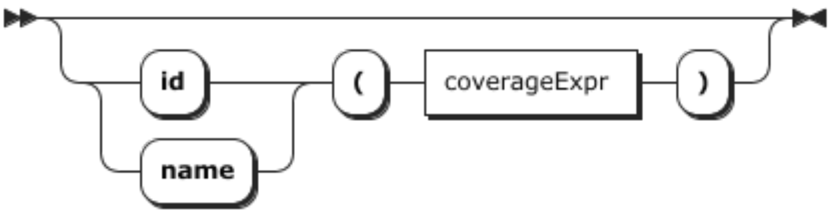

Figure C.21 — identifierExpr

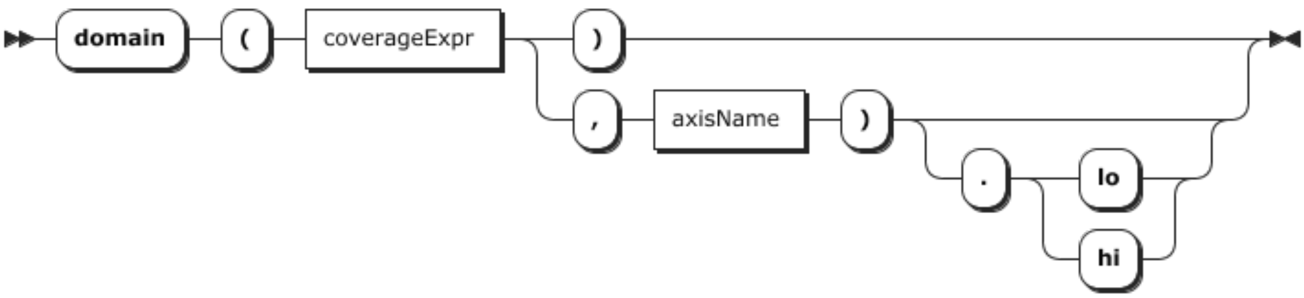

Figure C.22 — getDomainExpr

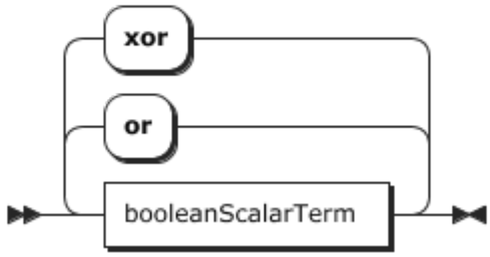

Figure C.23 — booleanScalarExpr

Figure C.24 — booleanScalarTerm

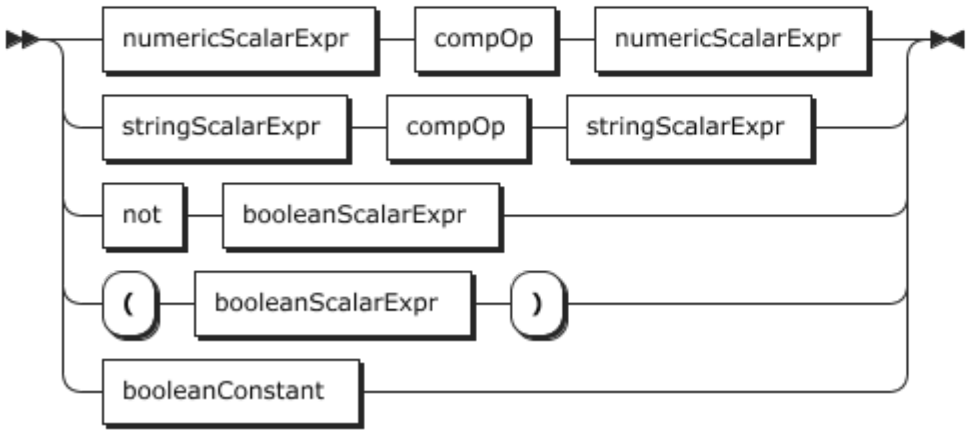

Figure C.25 — booleanScalarFactor

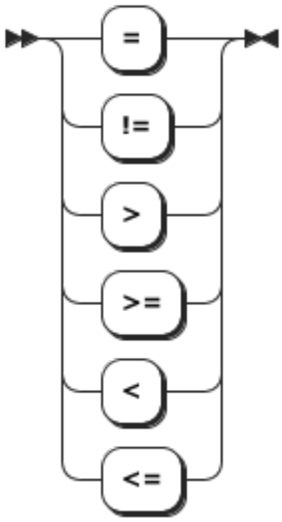

Figure C.26 — compOp

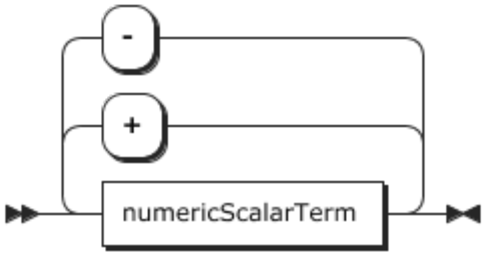

Figure C.27 — numericScalarExpr

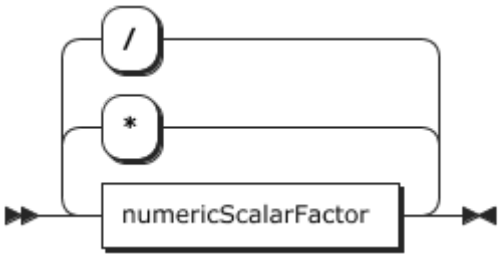

Figure C.28 — numericScalarTerm

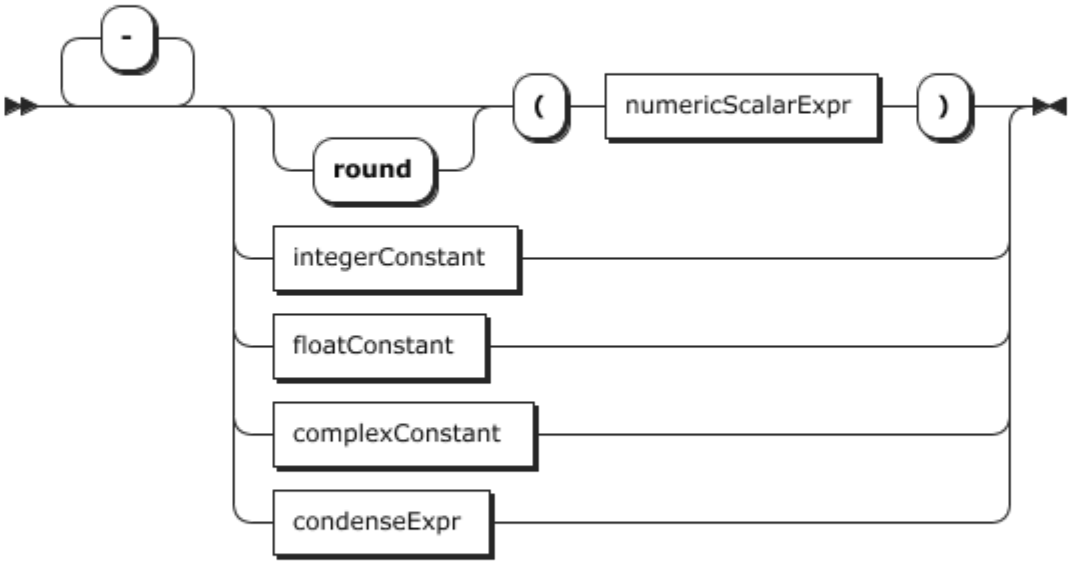

Figure C.29 — numericScalarFactor

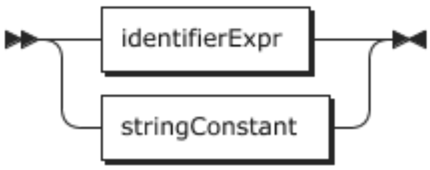

Figure C.30 — stringScalarExpr

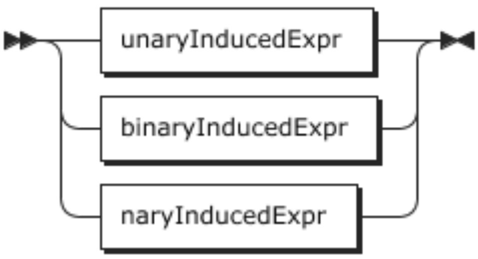

Figure C.31 — inducedExpr

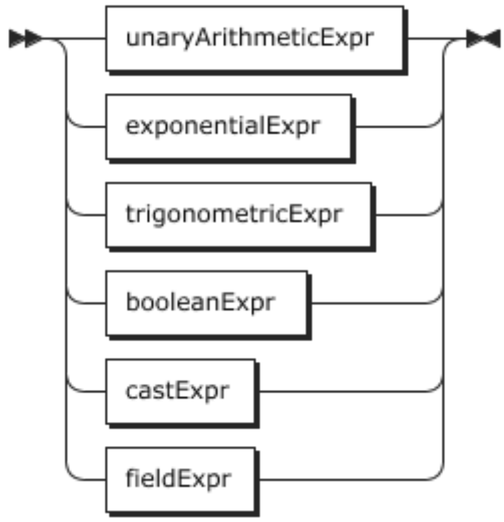

Figure C.32 — unaryInducedExpr

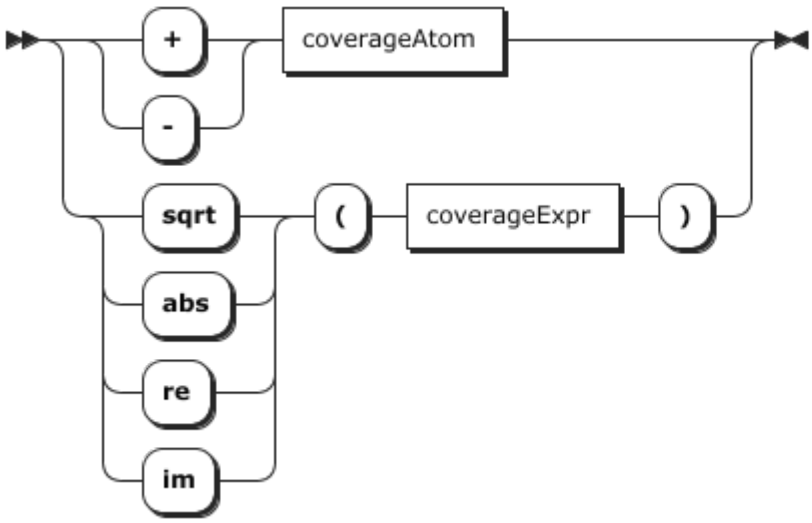

Figure C.33 — unaryArithmeticExpr

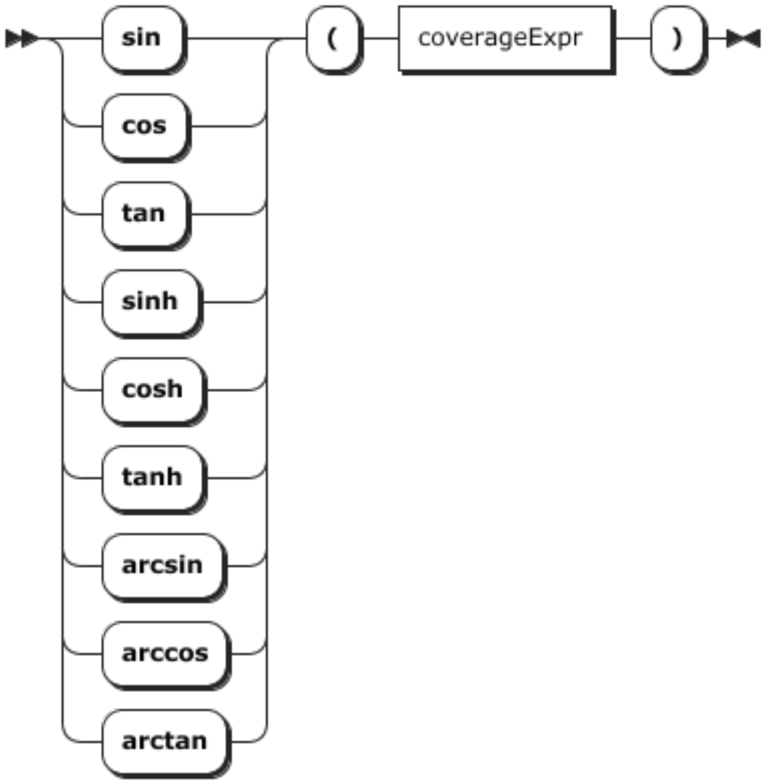

Figure C.34 — trigonometricExpr

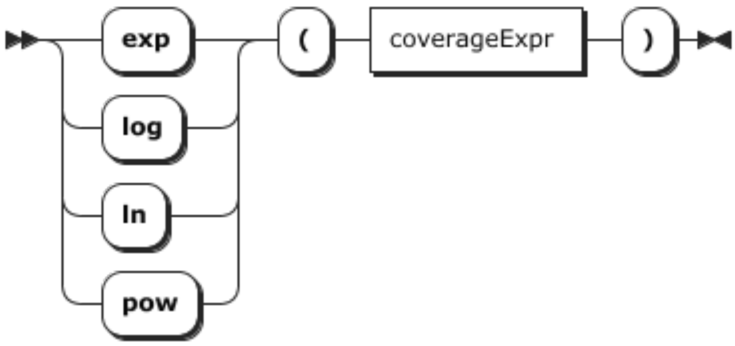

Figure C.35 — exponentialExpr

Figure C.36 — castExpr

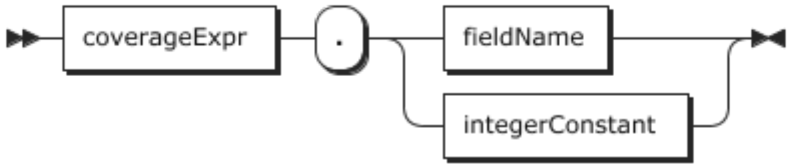

Figure C.37 — fieldExpr

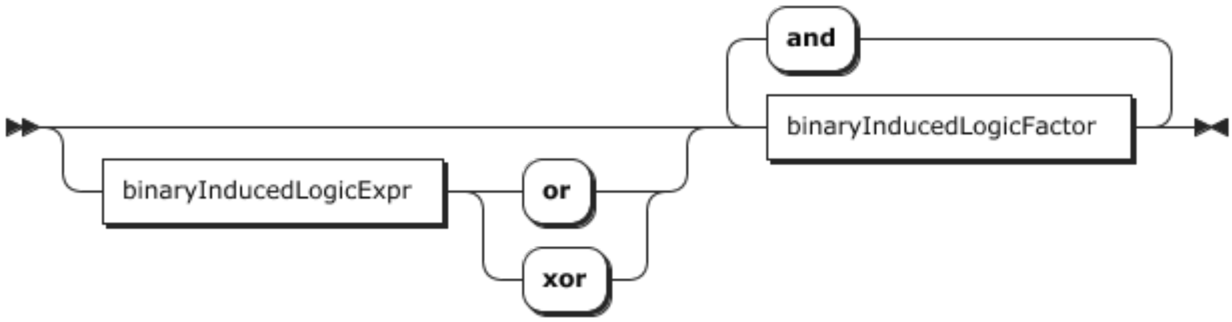

Figure C.38 — binaryInducedExpr

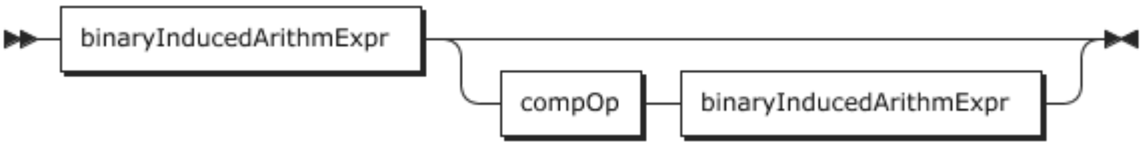

Figure C.39 — binaryInducedLogicFactor

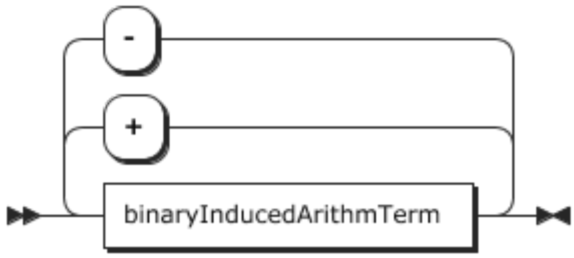

Figure C.40 — binaryInducedArithmExpr

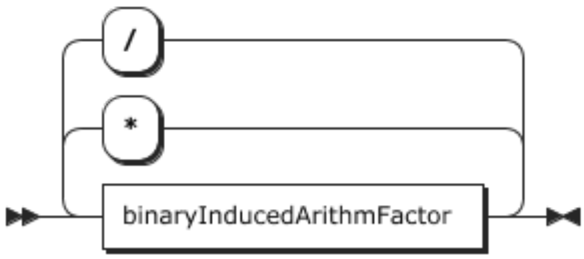

Figure C.41 — binaryInducedArithmTerm

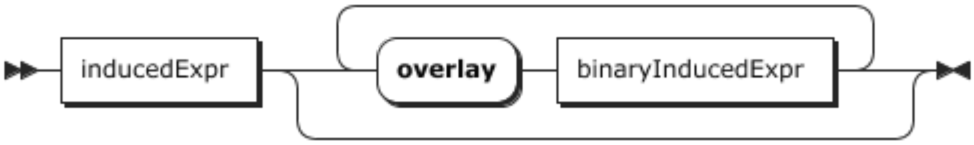

Figure C.42 — binaryInducedArithmFactor

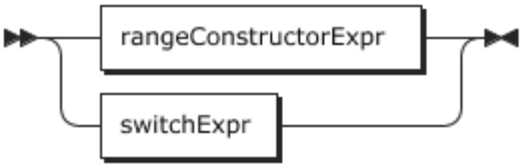

Figure C.43 — naryInducedExpr

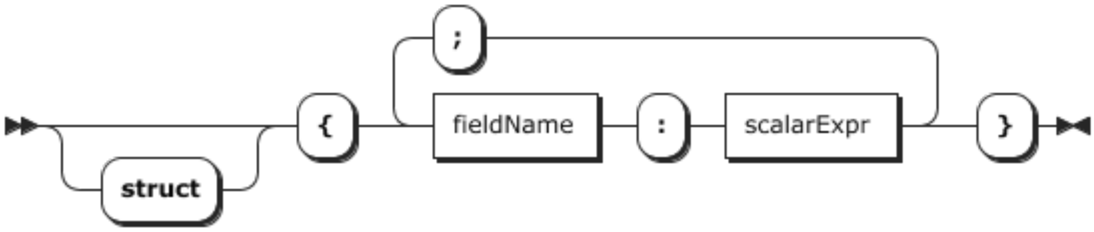

Figure C.44 — rangeConstructorExpr

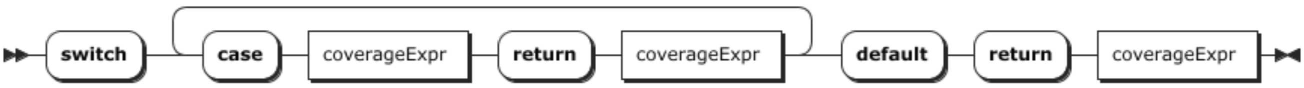

Figure C.45 — switchExpr

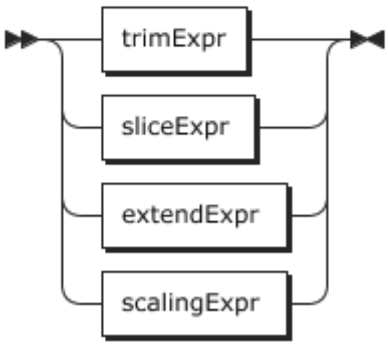

Figure C.46 — subsetExpr

Figure C.47 — trimExpr

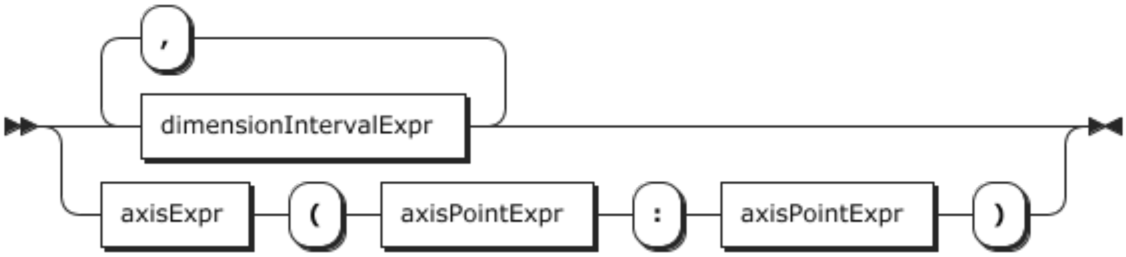

Figure C.48 — dimensionIntervalExpr

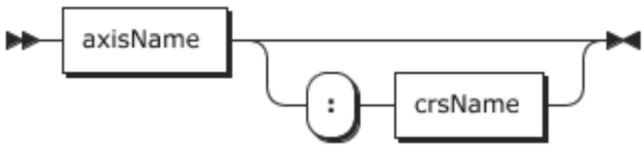

Figure C.49 — axisExpr

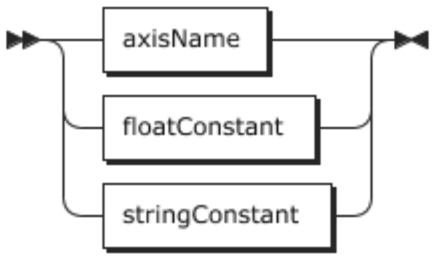

Figure C.50 — axisPointExpr

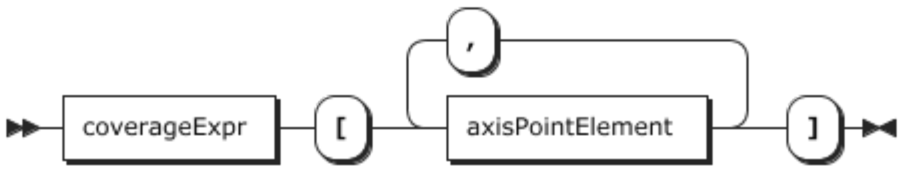

Figure C.51 — sliceExpr

Figure C.52 — axisPointElement

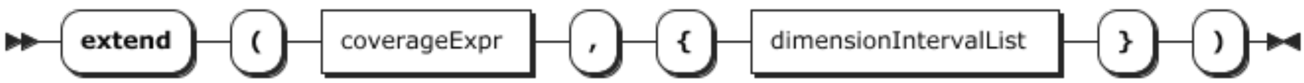

Figure C.53 — - extendExpr

Figure C.54 — scaleExpr

Figure C.55 — crsTransformExpr

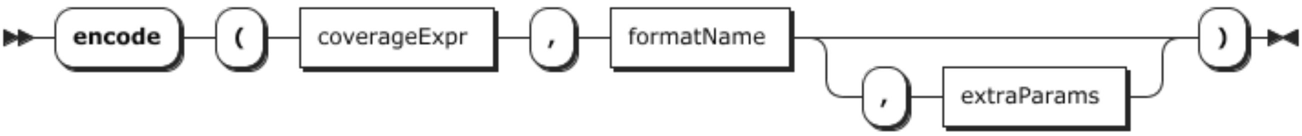

Figure C.56 — encodeCoverageExpr

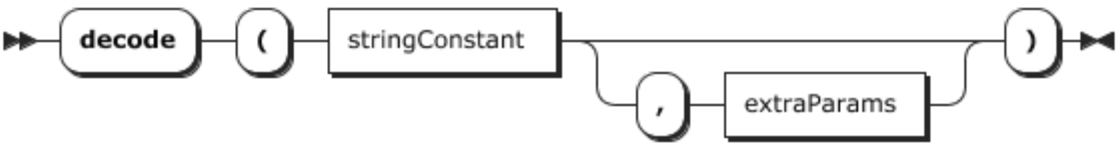

Figure C.57 — decodeCoverageExpr

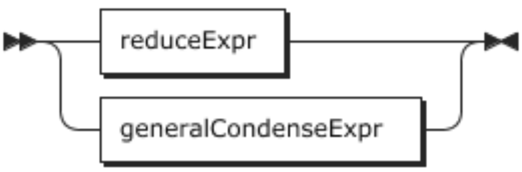

Figure C.58 — condenseExpr

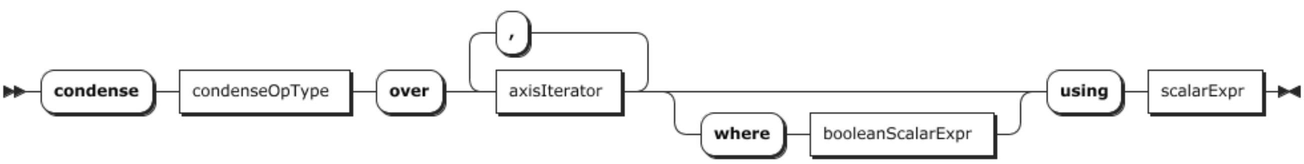

Figure C.59 — generalCondenseExpr

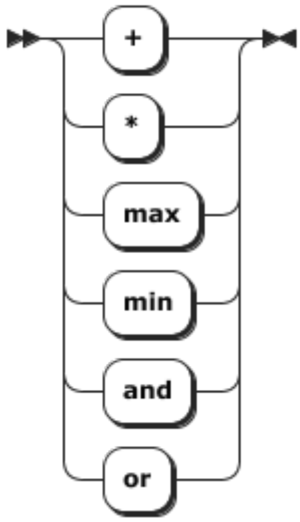

Figure C.60 — condenseOpType

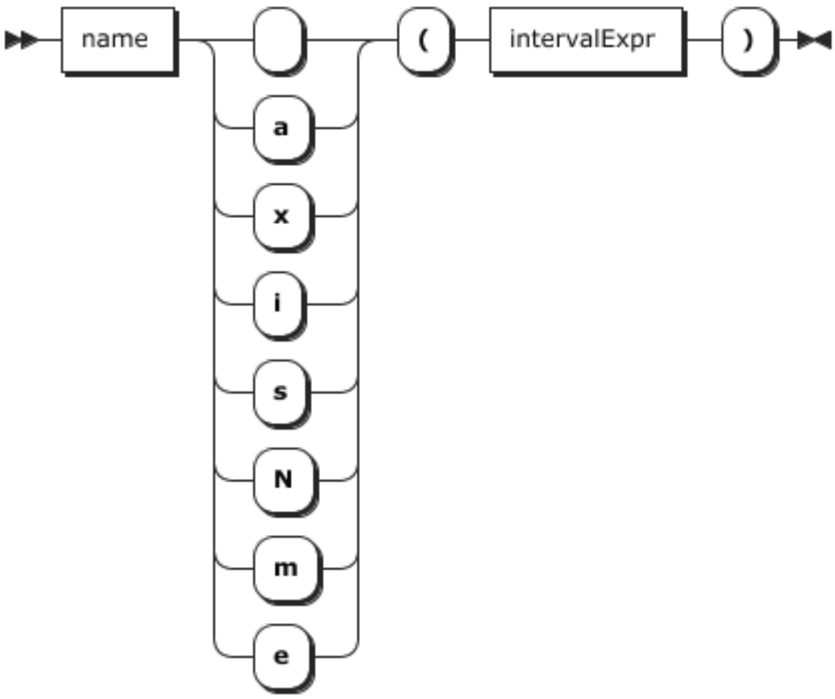

Figure C.61 — axisIterator

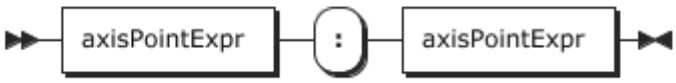

Figure C.62 — intervalExpr

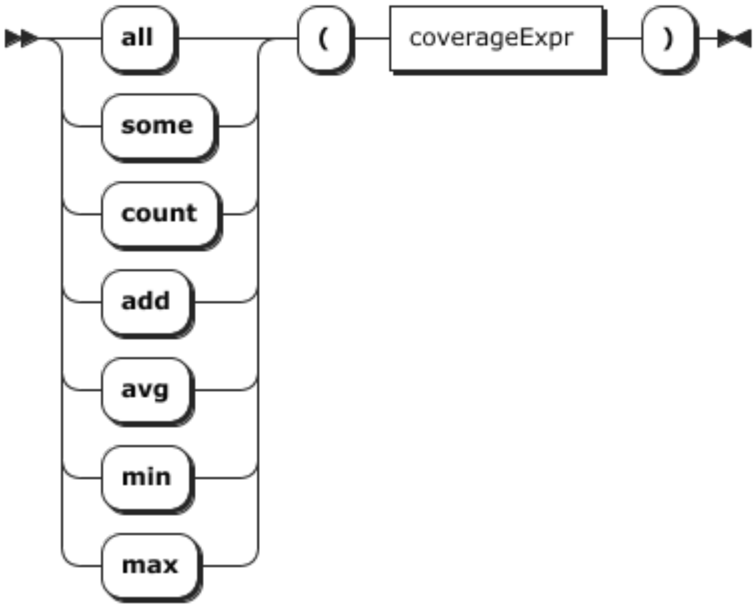

Figure C.63 — reduceExpr

Figure C.64 — coverageName

Figure C.65 — crsName

Figure C.66 — axisName

Figure C.67 — fieldName

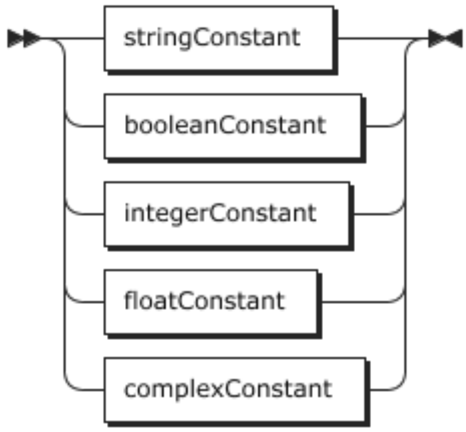

Figure C.68 — constant

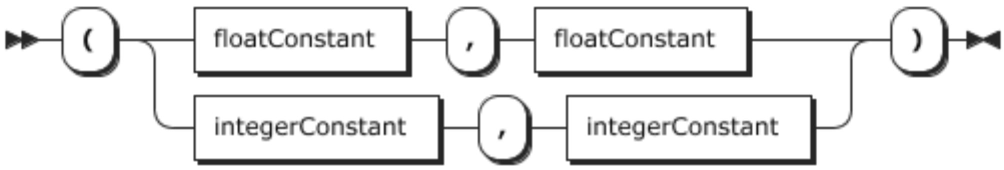

Figure C.69 — complexConstant

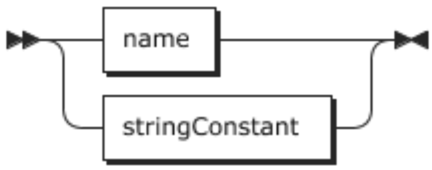

Figure C.70 — nameOrString

Annex D
(informative)

Sample service descriptions

D.1 Overview

This annex presents, as an example of the use of the coverage processing language, the specification of the OGC Web Coverage Service (WCS)[3] semantics through coverage expressions. WCS-Core and several of its extensions are modeled.

D.2 WCS-Core

WCS-Core defines access to a coverage, subsetting, and output format encoding in the *GetCoverage* request.

Extensions below often extend the *GetCoverage* request with additional parameters triggering the additional functionality in the server. Therefore, when such extension functionality is used the resulting ISO 19123-1 expression describing the semantics will be a functional merge of all individual WCS Core's and extensions' expressions involved.

Input parameters:

— {cov}
— {subset-axis1}, {subset-axis2}, ...
— {fmt} (default: coverage native format)

WCS *GetCoverage* request in GET/KVP syntax:

```
https://acme.com/wcs?SERVICE=WCS&VERSION=2.0&REQUEST=GetCoverage&
  COVERAGEID={cov}&
  SUBSET={subset-axis1}&SUBSET={subset-axis2}&...&
  FORMAT={fmt}
```

NOTE The SUBSET parameter gets broken down into a trim or slice on the axes addressed.

Semantics:

for $c **in** ({cov}) **return** encode({cov} {subset}, {fmt})

D.3 WCS-Range-Subsetting

WCS-Range-Subsetting is an optional WCS extension which allows extraction of range components (in various application domains also called "bands", "variables", etc.). Technically, an additional parameter extends the WCS-Core *GetCoverage* request.

Input parameters:

— {cov}
— {range-subset}

WCS *GetCoverage* request in GET/KVP syntax:

```
https://acme.com/wcs?SERVICE=WCS&VERSION=2.0&REQUEST=GetCoverage&
  COVERAGEID={cov}&
  RANGESUBSET={range-subset}
```

Semantics:

for $c **in** ({cov}) **return** encode({cov}. {range-subset}, {fmt})

D.4 WCS-Scaling

WCS-Scaling is an optional WCS extension which allows reducing the resolution of a grid coverage. Technically, additional parameters extend the WCS-Core *GetCoverage* request. Here, one of the several scaling variants is described:

Input parameters:

— {cov} (as per WCS-Core)
— {scale-factor}

WCS *GetCoverage* request in GET/KVP syntax:

```
https://acme.com/wcs?SERVICE=WCS&VERSION=2.0&REQUEST=GetCoverage&
  COVERAGEID={cov}&
  SCALEFACTOR={scale-factor}
```

Semantics:

for $c **in** ({cov}) **return** encode(scale({cov} {scale-factor}), {fmt})

D.5 WCS-CRS

WCS-CRS is an optional WCS extension which allows reprojection of a coverage into a different CRS (and formulate a subsetting request in a CRS different from the coverage's CRS. This is omitted here for simplicity). Technically, additional parameters extend the WCS-Core *GetCoverage* request.

Input parameters:

— {cov} (as per WCS-Core)
— {output-crs} CRS into which coverage is transformed
— {format} encoding format in which result is returned

WCS *GetCoverage* request in GET/KVP syntax:

```
https://acme.com/wcs?SERVICE=WCS&VERSION=2.0&REQUEST=GetCoverage&
  COVERAGEID={cov}&
  OUTPUTCRS={output-crs}
```

Semantics:

for $c **in** ({cov}) **return** encode(crsTransform({cov}, {output-crs}), {format})

D.6 WCS-Processing

WCS-Processing is an optional WCS extension which allows sending an OGC WCPS request to a server and obtain the evaluation result. WCPS is based on the OGC Coverage Implementation Schema (CIS) model which is identical to ISO 19123-2, a concretization of the ISO 19123-1 data model. Technically, an additional request type is added to WCS named *ProcessCoverages*. For the overlapping part of both languages and assuming the ISO 19123-2 coverage model, translation is 1:1.

Input parameters:

— {wcps-expression}

WCS *ProcessCoverage* request in GET/KVP syntax:

```
https://acme.com/wcs?SERVICE=WCS&VERSION=2.0&REQUEST=ProcessCoverage&
  QUERY={wcps-expression}
```

Semantics:

{wcps-expression}

Bibliography

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

[2] Baumann, P.: OGC Web Coverage Processing Service (WCPS) Language Interface Standard. OGC document 08-068r3 [online]. Available from: https://docs.ogc.org/is/08-068r3/08-068r3.html

[3] P. Baumann: Towards a Model-Driven Datacube Analytics Language. Proc. IEEE Big Spatial Data Workshop, December 17, 2021, [online]. Available from: http://localhost/public_html/Website-IUB/iu-bremen.de_pbaumann//Papers/2021/IEEE-BigSpatialData_WCPS.pdf

[4] Baumann, P.: OGC Web Coverage Service (WCS) Interface Standard – Core. OGC document 17-089r1 [online]. Available from: http://docs.opengeospatial.org/is/17-089r1/17-089r1.html

[5] ISO/IEC 19123:2022, Geographic information —Schema for coverage geometry and functions — Part 1: Fundamentals

[6] W3C: XQuery 1.0: An XML Query Language (Second Edition) [online]. Available from: https://www.w3.org/TR/2010/REC-xquery-20101214

[7] ISO/IEC 19123-2:2019, Geographic information — Schema for coverage geometry and functions — Part 2: Coverage implementation schema

[8] OGC, OGC Coverage Implementation Schema [online]. Available from: https://docs.ogc.org/is/09-146r6/09-146r6.html

ICS 35.240.70

Price based on 77 pages