**ISO 19123-3:2022**

ISO TC 211/WG 6

Date: 2022-11-15

**Geographic information — Schema for coverage geometry and functions — Part 3: Processing Fundamentals**

DIS stage

© ISO 2022

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO’s member body in the country of the requester.

ISO copyright office

CP 401 • Ch. de Blandonnet 8

CH-1214 Vernier, Geneva

Phone: +41 22 749 01 11

Email: copyright@iso.org

Website: www.iso.org

Published in Switzerland

O:\\Documents\\TC211\\ - ISO\_19123-1 (Ed 1)\\90.20\\170\\ISO\_19123-1\_(E).docINTERNATIONAL STANDARD© ISO 2020– All rights reservedISO 19123-1:20xx(E) 63Part 1: FundamentalsGeographic information — Schema for coverage geometry and functionsInformation géographique — Schéma de la géométrie et des fonctions de couverture – Partie 1 : Principes de baseGeographic information — Schema for coverage geometry and functions -- Part 3: Processing FundamentalsE2020-05-26(90.92) RevisionISOISO    International Standard 2022ISO 19123ISO 19123-1ISO 19123-1 SIS Geographic information/Geomatics  211 2Heading 2Heading 1     0 CR6STD Version 2.090.92   4             ISO TC 211

ISO 19123-1:20xx(E)

Secretariat: SIS

DIS edition  
2022-10-xx

Geographic information — Schema for coverage geometry and functions -- Part 3: Processing Fundamentals

*Information géographique — Schéma de la géométrie et des fonctions de couverture – Partie 1 : Principes de base*

**FDIS Draft**

**for project 19123-3  
based on the OGC WCPS 1.1 standard**

Contents Page

Foreword vi

Introduction vii

1 Scope 1

2 Normative references 1

3 Terms, definitions, abbreviated terms and notation 1

3.1 Terms and definitions 1

3.2 Abbreviated terms 2

4 Conformance 2

4.1 Notation 2

5 2

5.1 Interoperability and Conformance Testing 2

5.2 Organization 2

6 Coverage model 3

6.1 Overview 3

6.2 Coverage model 3

6.3 Coverage identifier 4

6.4 Domain 4

6.4.1 Direct Position 4

6.4.2 Grid 4

6.5 Interpolation 6

6.6 Range values 7

6.7 Range type 7

6.8 Coverage probing functions synopsis 8

7 Coverage processing language 9

7.1 Syntax and Semantics Definition Style 9

7.1.1 Expression Syntax 9

7.1.2 Expression Semantics 9

7.2 Coverage Processing Expressions 10

7.2.1 processCoveragesExpr 10

7.2.2 processingExpr 12

7.2.3 coverageExpr 12

7.2.4 coverageIdExpr 12

7.3 Coverage-Generating Expressions 13

7.3.1 coverageConstructorExpr 13

7.3.2 Examples 16

7.4 Coverage Extraction Expressions 17

7.4.1 scalarExpr 17

7.4.2 getComponentExpr 18

7.4.3 booleanScalarExpr 19

7.4.4 numericScalarExpr 19

7.4.5 stringScalarExpr 19

7.5 Coverage range value-changing expressions 19

7.5.1 inducedExpr 19

7.5.2 unaryInducedExpr 19

7.5.3 trigonometricExpr 22

7.5.4 binaryInducedExpr 27

7.5.5 N-ary Induced operations 29

7.5.6 Coverage Domain-Changing Expressions 32

7.5.7 scaleExpr 35

7.6 Coverage Derivation Expressions 36

7.6.1 crsTransformExpr 36

7.7 Coverage Aggregation Expressions 37

7.7.1 condenseExpr 37

7.7.2 generalCondenseExpr 37

7.7.3 reduceExpr 40

7.8 Coverage Encode/Decode Expressions 42

7.8.1 encodeCoverageExpr 42

7.8.2 decodeCoverageExpr 42

7.9 Expression evaluation 43

7.9.1 Evaluation sequence 43

7.9.2 Nesting 43

7.9.3 Parentheses 43

7.9.4 Operator precedence rules 44

7.9.5 Range type compatibility and extension 44

7.10 Evaluation response 45

Annex A (normative) Conformance Tests 46

A.1 Conformance Class 46

A.2 Conformance Class Coverage Processing Core 46

Annex B (normative) Expression Syntax 47

B.1 Overview 47

B.2 Terminal Symbols 47

B.3 19123-3Processing Syntax 48

Annex C (non-normative) Syntax Diagrams 55

Annex D (non-normative) Exemplary Service Descriptions 70

D.1 Overview 70

D.2 WCS-Core 70

D.3 WCS-Range-Subsetting 70

D.4 WCS-Scaling 71

D.5 WCS-CRS 71

D.6 WCS-Processing 72

Bibliography 73

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](https://www.iso.org/directives-and-policies.html)).

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](https://www.iso.org/iso-standards-and-patents.html)).

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](https://www.iso.org/foreword-supplementary-information.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), under participation of the IEEE GRSS Earth Science Informatics Technical Committee, and derived from the Open Geospatial Consortium (OGC) standard WCPS 1.1 with permission.

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](https://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 OGC Web Coverage Service (WCS) [3].

Throughout the 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.

# 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 version , regular and irregular multi-dimensional grids are supported, for axes that can carry spatial, temporal, or any other semantics. Future versions will additionally support further axis types as well as further coverage types from 19123-1, in particular: point clouds and meshes.

# 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: Coverage Fundamentals*

# Terms, definitions, abbreviated terms and notation

## Terms and definitions

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

For the purposes of this document, the following terms and definitions apply.

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

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

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

3.1.1

probing function

<coverage> function extracting information from the coverage

# Conformance

## 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 |

## Interoperability and Conformance Testing

This document being an abstract standard 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 is 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.

## Organization

Table 2 — Conformance classes

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

# Coverage model

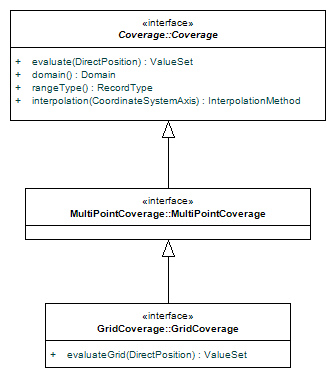
## Overview

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.

## 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?*”



1. – Coverage and GridCoverage (ISO 19123-1)

Note Coverage in 19123-1 defines an interface which describes such an object’s behavior, 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 some 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 document’s language semantics.

Note 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.

## Coverage identifier

Coverages in this document have an identifier which is used in a query to address a coverage to derive from. Therefore, this identifier must 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 so that querying the object in the new context potentially is not possible any longer.

Note In a concrete service, coverages available typically would 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* )

## Domain

### Direct Position

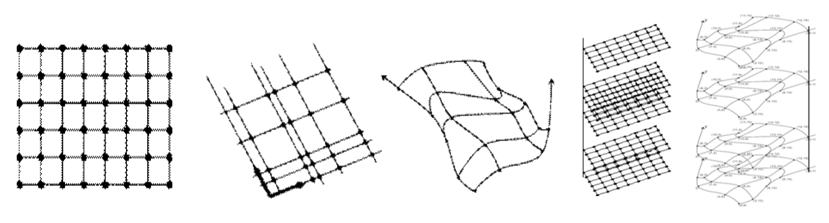
A coverage offers values for particular positions in its domain; these are called “direct positions”; further values can possibly be derived through interpolation, depending on whether and what type of interpolation a coverage allows.

For some direct position *p* = (*p1*,…,*pd*) from a domain whose *d*-dimensional CRS contains axes (*a1*,…,*ad*) we write *p*[*ai*] for accessing the coordinate tuple component corresponding with axis *ai*:

*p*[*ai*] = *pi*

### 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 obviously less neighbours are available. Figure 2 shows some regular and irregular grid examples.



1. – Sample regular and irregular grid structures (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 19123-1 are currently supported by this document:

* A **Cartesian** (“index”) axis 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 geo­referencing. Definition and interpretation of CRSs is based on ISO 19111:2019.

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

*crs*(*C*)

Auxiliary probing function *axisList*()extracts the ordered list of axes (*a1*,…,*ad*) from a *d*-dimensional CRS:

*axisList*( *crs* )

Note As per 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 some nonempty, totally ordered set of values which can be numeric or, in the general case, strings (such as “2020-08-05T”).

For some 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 bound, 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* = ⊥  
 else *x* is given by: *x*[*ax*] = *p*[*ax*] for all *ax* ∈ *domain*( *C* ) \ {*a*}, and *x*[*a*] = max( *x*’ | *x*’ ∈ *domain*( *C*, *a* ) and *x*’ < *p*[*a*] )

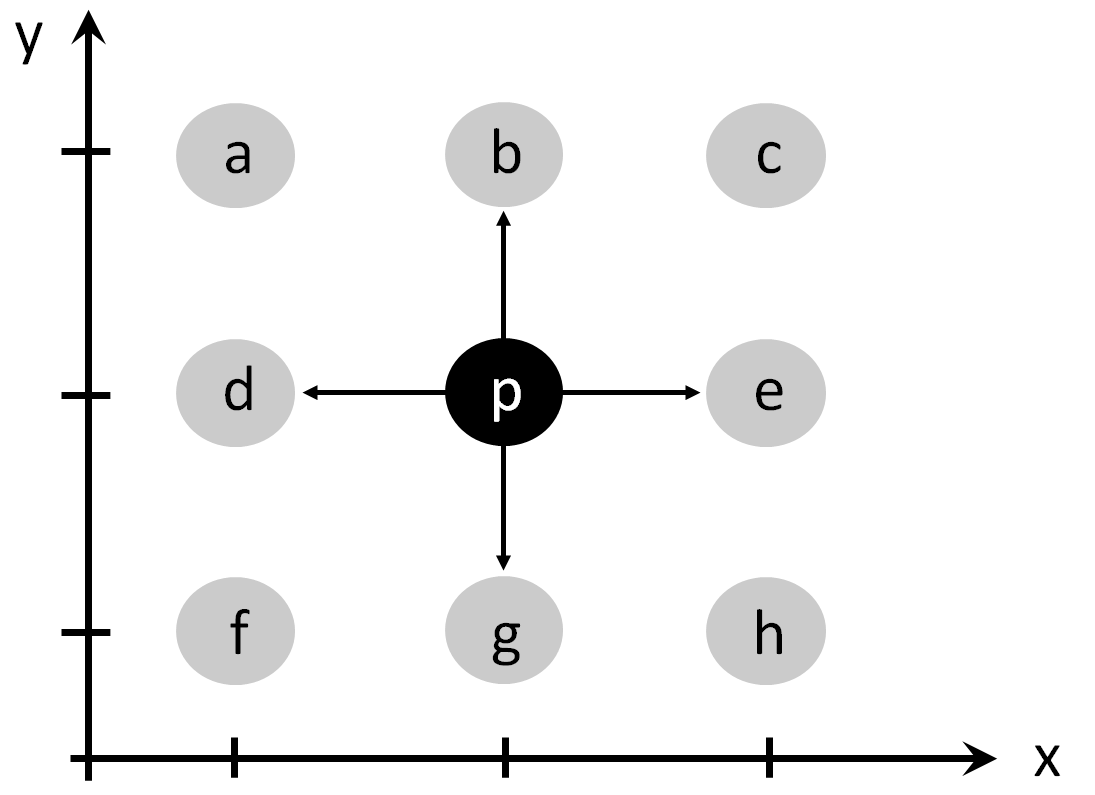
*succ*( *D*, *a, p* ) = *x* where   
 if *p*[*a*] = *D*[*a*].*hi* *domain*(*C*,*a*).*hi* then *x* = ⊥  
 else *x* is given by: *x*[*ax*] = *p*[*ax*] for all *ax* ∈ *domain*( *C* ) \ {*a*}, and *x*[*a*] = min( *x*’ | *x*’ ∈ *domain*( *C*, *a* ) and *x*’ > *p*[*a*] )

Example In , 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 reader’s convenience basic arithmetic functions are assumed on this grid navigation, defined recursively such as sketched below:

*add*( *D*, *x*, *p1*, *p2* ) = *add*( *D*, *x*, *succ*( *p1*), *pred*( *p2* ) )

*subtract(D*, *x*, *p1*, *p2* ) = *subtract( D*, *x*, *succ*( *p1* ), *pred*( *p2* ) )



1. – Sample grid neighbourhood

## 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 In 19123-1 every coverage has exactly one interpolation method associated (for all axes or per axis). In practice, coverages may 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 19123-1 and 19123-3 this ambiguity is not desirable.

For the purpose of this document a special interpolation method noneis assumed as defined, e.g., in 19123-1 Annex B. Noneindicates that no interpolation is possible along the axis under consideration.

Note 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.

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

## Range values

The range value at some direct position p can be obtained with function *evaluate*C(p) which, for some given coverage *C*, returns the value associated with p∈*domain*(C) expressed in the coverage’s CRS.

The corresponding probing function is:

*value*( C, p ) = *evaluate*C( p ) for some direct position p∈*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.

## 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 some 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 standard may add its own data types as long as these are coherent with this standard 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.

## Coverage probing functions synopsis

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

Table 3 —Coverage probing functions synopsis

|  |  |  |
| --- | --- | --- |
| Coverage  characteristic | Probing function  for some coverage *C*,  based on 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* ) = (*a1*,…,*ad*) 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 Clause | 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,...*) ∈*rangeType*( C ) | The range type record is described by a list describing its components in sequence; for the purpose of this standard only component name and its data type are considered. |
| Range field name list | *rangeFieldNames*( C ) = (f1, …, fn) where  *rangeType*( C) = ( (f1;t1,…), …, (fn:tn,…) ),  with field names fi and types ti | 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 standard, their values are to be defined else­where (e.g., implementation dependent) |
| Range values | *value*(C,p)  = *evaluate*C(p),p∈*domain*(C)  with *evaluate*() as per 19123-1 | Range values of the coverage at some direct position (or some position inbetween, interpolation permitting) |
| 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 |

# Coverage processing language

This clause establishes 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 Subclause 6.2). Each international standard claiming to support this specification shall provide the coverage processing operations as specified in the following subclauses. A sample application is provided in (informative) Annex D.

Note 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, no mix of both.

## Syntax and Semantics Definition Style

### 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 subsections the language elements are detailed. The complete syntax is listed in normative 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.

1. **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 (non-normative) Annex C for visualization of the grammar.

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

### 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.

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

EXAMPLE The coverage expression following 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.

## Coverage Processing Expressions

### 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.

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

Let

v1, … vn be n pairwise different **iteratorVar**s (n≥1),  
L1, … Ln be n **coverageList**s (n≥1),  
b be a **booleanScalarExpr** possibly containing occurrences of one or more vi (1≤i≤n),  
P be a **processingExpr** possibly containing occurrences of vi (1≤i≤n).

Then,

m,n≥1 be natural numbers,  
v1, … cn, be n **iteratorVar**s,  
c1, … cm, be n pairwise different **variableName**s,  
e1, … em, be n+m optional **coverageExpr**s or **scalarExpr**s or bracket-enclosed **intervalExpr**s, which may contain occurrences of v1, … cn and c1, … cm,  
c be a **coverageExpr** or **scalarExpr**,  
where everyci is defined before used in an expression.

Then,

for any **processCoveragesExpr** E  
where  
 E = for v1 in ( L1),  
 v2 in ( L2),  
 … ,  
 vn in ( Ln)  
 [ letc1 := 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 viin 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 Coverage identifiers may 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 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 The following statement defines a constant of name $timeAxis with value “date”.

**let** $timeAxis := "date"

Note 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 (beware!) a mixture of the two.

Named constants cannot be updated – 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 (cf. XPath literature). This rule might 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.

### processingExpr

1. **https://standards.isotc211.org/19123/-3/1/req/core/processingExpr**  
   A **processingExpr** element **shall** be either a **encodeCoverageExpr** (see Subclause6.8.1) or a **scalarExpr** (see Subclause ).

### coverageExpr

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

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

### 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 Subclause 6.2).

1. **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 C1 available.

Then,

for any **coverageExpr** C2,   
where  
 C2 = id

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = *id(*C1) |
| *crs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeType*(C2) = *rangeType*(C1) |
| for all p∈*domain*(C2):  *value*(C2,p) = *value*(C1,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

## Coverage-Generating Expressions

### coverageConstructorExpr

The **coverageConstructorExpr** element creates a d-dimensional grid coverage for some d≥1 by defining the coverage’s domain, range type and range through expressions. This allows deriving entirely new shapes, dimensions, and values (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 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.

1. **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,  
ai be pairwise distinct **variableName**s for 1≤i≤d,  
axisi be pairwise distinct **axisName**s for 1≤i≤d,  
iei,1, iei,2 be integer-valued **indexExpr**s for 1≤i≤d with iei,1 ≤ iei,2**,**  
cei,1, cei,2 be **axisPointExpr**s for 1≤i≤d, which are valid coordinates for axis i as per CRS c with cei,1 ≤ cei,2**,**resi be **axisPointExpr**s with res1<…<resdfor 1≤i≤d valid for the ith axis as per c,xei,1,… be **axisPointExpr**s for 1≤i≤d, which are valid coordinates for axis axisi as per CRS c with xei,1<xei,2<…,  
im1,…, imm be (not necessarily distinct) **interpolationMethod**s for 1≤i≤m with m>0.

Where

D is a **domainExpr**  
with   
 D = domain  
 crs c with  
 axis1 axisdef1 [ **interpolation** im1 ],  
 … ,  
 axisd axisdefd [ **interpolation** imd ]

And

axisdefi is one of  
 axisdefi,index = index **(** iei,1 **:** iei,2 **)**  
 axisdefi,regular = regular **(** cei,1 **:** cei,2 **)** resolution resi  
 axisdefi,irregular = irregular**(** xei,1 **,** … **,** xei,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 axisiare 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 may use the axis names defined in the CRS, or may 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,  
f1,…, fn be **fieldNames**,  
t1,…, tn 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 axisi as defined in D and range component identifiers fj as defined in T,  
c1, …, cm 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 evaluatingr |
| *domain*(C) = domain extent resulting from evaluating D if present, otherwise the domain extent resulting from evaluatingr |
| *interpolation*(C) = ( x1,…, xd ) where xi = imi where imi is indicated, otherwisexi = none. |
| rangeType(C) = ( (f1,t1), …, (fn,tn) ) if T is present, otherwise the range type resulting from evaluatingr ; if no field names are provided (such as with R2) then the range field names are implementation-dependent. |
| for all p∈*domain*(C) and **scalarExpr**r:  *value*(C,p) = range value resulting from evaluating r, with possible occurrences of ai 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∈*domain*(C) and **rangeConstantExpr** <c1,…, cm*>*:  *value*(C, p) is determined by assigning each value ci 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.

### 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

Below follows a complete coverage constructor representing a 3-D 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 like this:

**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) ]

## Coverage Extraction Expressions

### scalarExpr

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

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

### 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).

1. **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 may 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 )

### booleanScalarExpr

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

### numericScalarExpr

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

### stringScalarExpr

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

## Coverage range value-changing expressions

### inducedExpr

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

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

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

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

Note 2 The result mayhave 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.

### 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.

1. **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 Subclauses , , ), a **booleanExpr** (in which case it eval­uates to a Boolean expression; see Subclause ), a **castExpr** (in which case it evaluates to a coverage with unchanged values, but another range type; see Subclause ), or a **field­Expr** (in which case a range field selection is performed; see Sub­clause ).

#### unaryArithmeticExpr

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

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

Let

C1, C2 be **coverageExpr**s with all range type components being numeric and additionally all range type components of C1 being of type complex,  
S1, S2 be **scalarExpr**s.

Then,

for any **coverageExpr** C2  
where C2 is one of   
 Cplus = + C1  
 Cminus = - C1  
 Csqrt = sqrt(C1)  
 Cabs = abs(C1)  
 Cre = re(CC1)  
 Cim = im(CC1)

CplusSC = S1 + C2  
 CminSC = S1 - C2  
 CmultSC = S1 \* C2  
 CdivSC = S1 / C2  
 CandSC = S1 and C2  
 CorSC = S1 or C2  
 CxorSC = S1 xor C2  
 CeqSC = S1 = C2  
 CltSC = S1 < C2  
 CgtSC = S1 > C2  
 CleSC  = S1 <= C2  
 CgeSC = S1 >= C2  
 CneSC = S1 != C2

CplusCS = C1 + S2  
 CmincS = C1 - S2  
 CmultCS = C1 \* S2  
 CdivCS = C1 / S2  
 CandCS = C1 and S2  
 CorCS = C1 or S2  
 CxorCS = C1 xor S2  
 CeqCS = C1 = S2  
 CltCS  = C1 <S2  
 CgtCS = C1 >S2  
 CleCS = C1 <= S2  
 CgeCS = C1 >= S2  
 CneCS = C1 != S2

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| for all range fields r∈*rangeFieldNames*(C2):  *rangeFieldType*( Cplus ) is given by Requirement 48  *rangeFieldType*( Cminus ) is given by Requirement 48  *rangeFieldType*( CplusSC ) is given by Requirement 48 *rangeFieldType*( Csqrt,r )  = double if *rangeFieldType*(C1,r) ≠ complex and C1.r≥0,  *=* complex otherwise,  *rangeFieldType*(Cabs,r)   = unsigned int if *rangeFieldType*(C1,r) ∈{ unsigned int, int }= float if *rangeFieldType*(C1,r) ∈ { float, complex }  *rangeFieldType*( CplusSC) is given by   *rangeFieldType*( CminSC) is given by   *rangeFieldType*( CmultSC) is given by   *rangeFieldType*( CdivSC) is given by   *rangeFieldType*( CandSC) = boolean  *rangeFieldType*( CorSC) = boolean  *rangeFieldType*( CxorSC) = boolean  *rangeFieldType*( CeqSC) = boolean  *rangeFieldType*( CltSC) = boolean  *rangeFieldType*( CgtSC) = boolean  *rangeFieldType*( CleSC) = boolean  *rangeFieldType*( CgeSC) = boolean  *rangeFieldType*( CneSC) = boolean  *rangeFieldType*( CovlSC) = *rangeType*(C2)  *rangeFieldType*( CplusCS, r) is given by   *rangeFieldType*( CminCS, r) is given by   *rangeFieldType*( CmultCS, r) is given by   *rangeFieldType*( CdivCS, r) is given by   *rangeFieldType*( CandCS, r) = boolean  *rangeFieldType*( CorCS, r) = boolean  *rangeFieldType*( CxorCS, r) = boolean  *rangeFieldType*( CeqCS, r) = boolean  *rangeFieldType*( CltCS, r) = boolean  *rangeFieldType*( CgtCS, r) = boolean  *rangeFieldType*( CleCS, r) = boolean  *rangeFieldType*( CgeCS, r) = boolean  *rangeFieldType*( CneCS, r) = boolean  *rangeFieldType*( CovlCS, r) = boolean |
| for all p∈*domain*(C2):  *value*( Cplus, p ) = *value*( C1, p ),  *value*( Cminus, p ) = - *value*( C1, p ),  *value*( Csqrt, p ) = sqrt( *value*( C1, p ) ),  *value*( Cabs, p ) = abs( *value*( C1, p ) ),  *value*( Cre, p ) = re( *value*( C1, p ) ),  *value*( Cim, p ) = im( *value*( C1, p ) ),  *value*( CplusSC ) = *value*( S1 ) + *value*( C2 )  *value*( CminSC ) = *value*( S1 ) - *value*( C2 )  *value*( CmultSC ) = *value*( S1 ) \* *value*( C2 )  *value*( CdivSC ) = *value*( S1 ) / *value*( C2 )  *value*( CandSC ) = *value*( S1 ) and *value*( C2 )  *value*( CorSC ) = *value*( S1 ) or *value*( C2 )  *value*( CxorSC ) = *value*( S1 ) xor *value*( C2 )  *value*( CeqSC ) = *value*( S1 ) == *value*( C2 )  *value*( CltSC ) = *value*( S1 ) < *value*( C2 )  *value*( CgtSC ) = *value*( S1 ) > *value*( C2 )  *value*( CleSC ) = *value*( S1 ) <= *value*( C2 )  *value*( CgeSC ) = *value*( S1 ) >= *value*( C2 )  *value*( CneSC ) = *value*( S1 ) != *value*( C2 )  *value*( CovlSC ) = *value*( S1 ) overlay *value*( C2 )  *value*( CplusC)S = *value*( C1) + *value*( S2)  *value*( CmincS) = *value*( C1) - *value*( S2)  *value*( CmultCS) = *value*( C1) \* *value*( S2)  *value*( CdivCS) = *value*( C1) / *value*( S2)  *value*( CandCS) = *value*( C1) and *value*( S2)  *value*( CorCS) = *value*( C1) or *value*( S2)  *value*( CxorCS) = *value*( C1) xor *value*( S2)  *value*( CeqCS) = *value*( C1) == *value*( S2)  *value*( CltCS ) = *value*( C1) < *value*( S2)  *value*( CgtCS) = *value*( C1) > *value*( S2)  *value*( CleCS) = *value*( C1) <= *value*( S2)  *value*( CgeCS) = *value*( C1) >= *value*( S2)  *value*( CneCS) = *value*( C1) != *value*( S2)  *value*( CovlCS) = *value*( C1) overlay *value*( S2) |

EXAMPLE For two integer or float valued coverages $c and $dthe 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 )

### trigonometricExpr

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

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

Let

C1 be a **coverageExpr**

Then,

for any **coverageExpr**C2  
where C2 is one of  
 Csin = sin( C1 )  
 Ccos = cos( C1 )  
 Ctan = tan( C1 )  
 Csinh = sinh( C1 )  
 Ccosh = cosh( C1 )  
 Carcsin = arcsin( C1 )  
 Carccos = arccos( C1 )  
 Carctan = arctan( C1 )

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeFieldNames* (C2 ) = *rangeFieldNames* (C1)  for all fields r∈*rangeFieldNames*(C2):  *rangeFieldType*(C2,r)   = complex if *rangeFieldType*(C1,r) = complex  = float otherwise |
| for all p∈*domain*(C2):  *value*(Csin,p) = sin( *value*(C1,p) )  *value*(Ccos,p) = cos( *value*(C1,p) )  *value*(Ctan,p) = tan( *value*(C1,p) )  *value*(Csinh,p) = sinh( *value*(C1,p) )  *value*(Ccosh,p) = cosh( *value*(C1,p) )  *value*(Carcsin,p) = arcsin( *value*(C1,p) )  *value*(Carccos,p) = arccos( *value*(C1,p) )  *value*(Carctan,p) = arctan( *value*(C1,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 )

#### exponentialExpr

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

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

Let

C1 be a **coverageExpr**,  
c be a **floatConstant**or **complexConstant**

Then,

for any **coverageExpr**C2  
where C2 is one of  
 Cexp = exp( C1 )  
 Clog = log( C1 )  
 Cln = ln( C1 )  
 Cpow  = pow( C1, c )

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeFieldNames* (C2 ) = *rangeFieldNames* (C1)  for all fields r∈*rangeFieldNames*(C2):  *rangeFieldType*(C2,r)   = complex if *rangeFieldType*(C1,r) = complex  = float otherwise |
| for all p∈*domain*(C2):  *value*( Cexp, p ) = exp( *value*(C1,p) )  *value*( Clog , p ) = log( *value*(C1,p) )  *value*( Cln , p ) = ln( *value*(C1,p) )  *value*( Cpow, p ) = *value*(C1,p)c |

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

ln( $c )

#### booleanExpr

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

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

Let

C1 be a **coverageExpr**,  
n be a positive integer number**.**

Then,

for any **coverageExpr**C2  
where  
 C2 = not C1  
where n is an expression evaluating to a nonnegative integer value

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeFieldNames* (C2 ) = *rangeFieldNames* (C1)  for all fields r∈*rangeFieldNames*(C2):  *rangefieldType*(C2,r) = boolean |
| for all p∈*domain*(C2):  *value*( Cnot , p ) = not( *value*(C1,p) ) |

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

not $c

#### 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).

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

Let

C1 be a **coverageExpr**,  
t be a range field type name.

Then,

for any **coverageExpr**C2  
where  
 C2 = ( t ) C1

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *rs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeFieldNames* (C2 ) = *rangeFieldNames* (C1)  for all fields r∈*rangeFieldNames*(C2):  *rangeFieldType*(C2,r) = t |
| for all p∈*domain*(C2):  *value*( C2 , p ) = (t) *value*(C1,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 )

#### 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.

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

Let

C1 be a **coverageExpr**,  
f be a **fieldName** appearing in *rangeFieldNames*(C1),  
i be an **integer** with 0≤i<|*rangeFieldNames*(C1)|.

Then,

for any **coverageExpr**C2  
where C2 is one of:  
 C2,f = C1 . f  
 C2,I = C1 . i

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeFieldNames* (C2 ) = ( f ), the sequence containing only f  *rangeFieldType*(C2,f) = *rangeFieldType*(C1,f) |
| for all p∈*domain*(C2):  *value*(C2,f,p) = *value*(C1.f,p)  *value*(C2,i,p) = *value*(C1.g,p)   where g is the ith field in *rangeFieldNames*(C1) |

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

1. **https://standards.isotc211.org/19123/-3/1/req/core/fieldExprShorthand**  
   In a **fieldExpr** C.fwhere **|***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

### binaryInducedExpr

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

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

Let

C1, C2 be **coverageExpr**s,  
N be 0 or some null value (to be defined by a concretization of this document)   
where  
 *crs*(C1) = *crs*(C2),  
 *domain*(C1,a) = *domain*(C2,a),  
 *rangeFieldNames*(C1) = *rangeFieldNames*(C2),  
 *rangeType*(C1,f) is cast-compatible with *rangeType*(C2,f) or   
 *rangeType*(C2,f) is cast-compatible with *rangeType*(C1,f)  
 for all f∈*rangeFieldNames*(C1).

Then,

for any **coverageExpr**C3  
where C3 is one of  
 CplusCC = C1 + C2  
 CminCC = C1 - C2  
 CmultCC = C1 \* C2  
 CdivCC = C1 / C2  
 CandCC = C1 and C2  
 CorCC = C1 or C2  
 CxorCC = C1 xor C2  
 CeqCC = C1 = C2  
 CltCC  = C1 < C2  
 CgtCC = C1 > C2  
 CleCC = C1 <= C2  
 CgeCC = C1 >= C2  
 CneCC = C1 != C2  
 CovlCC = C1 overlay C2

C3 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C3) = “” (empty string) |
| *crs*(C3) = *crs*(C1) |
| *domain*(C3) = *domain*(C1) |
| *interpolation*(C3) = *interpolation*(C1) |
| *rangeFieldNames* (C3 ) = *rangeFieldNames* (C1)  for all r∈*rangeFieldNames*(C3):  *rangeFieldType*( CplusCC, r ) is given by Requirement 48  *rangeFieldType*( CminCC, r ) is given by Requirement 48  *rangeFieldType*( CmultCC, r ) is given by Requirement 48  *rangeFieldType*( CdivCC, r ) is given by Requirement 48  *rangeFieldType*( CandCC, r ) = boolean  *rangeFieldType*( CorCC, r ) = boolean  *rangeFieldType*( CxorCC, r ) = boolean  *rangeFieldType*( CeqCC, r ) = boolean  *rangeFieldType*( CltCC, r ) = boolean  *rangeFieldType*( CgtCC, r ) = boolean  *rangeFieldType*( CleCC, r ) = boolean  *rangeFieldType*( CgeCC, r ) = boolean  *rangeFieldType*( CneCC, r ) = boolean  *rangeFieldType*( CovlCC, r ) = *rangeFieldType*( C1, r ) |
| for all p∈*domain*(C3):  *value*( CplusCC, p ) = *value*(C1, p) + *value*(C2, p)  *value*( CminCC, p ) = *value*(C1, p) - *value*(C2, p)  *value*( CmultCC, p ) = *value*(C1, p) \* *value*(C2, p)  *value*( CdivCC, p ) = *value*(C1, p) / *value*(C2, p)  *value*( CandCC, p ) = *value*(C1, p) and *value*(C2, p)  *value*( CorCC, p ) = *value*(C1, p) or *value*(C2, p)  *value*( CxorCC, p ) = *value*(C1, p) xor *value*(C2, p)  *value*( CeqCC, p ) = *value*(C1, p) = *value*(C2, p)  *value*( CltCC, p ) = *value*(C1, p) < *value*(C2, p)  *value*( CgtCC, p ) = *value*(C1, p) > *value*(C2, p)  *value*( CleCC, p ) = *value*(C1, p) <= *value*(C2, p)  *value*( CgeCC, p ) = *value*(C1, p) >= *value*(C2, p)  *value*( CneCC, p ) = *value*(C1, p) != *value*(C2, p)  *value*( CovlCC, p ) = *value*(C2, p) if *value*(C1, p)=N  *value*(C1, 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

### N-ary Induced operations

#### 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 maybe listed more than once.

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

Let

n be an **integer** with n≥1,  
C1, …, Cn be **coverageExpr**s with |*rangeFieldNames*(Ci)|=1 (i.e., just a single range component),   
f1, …, fn be **fieldName**s  
where, for 1≤i,j≤n,  
 *crs*(Ci) = *crs*(Cj),   
 *domain*(Ci) = *domain*(Cj)  
 *gridCrs*(Ci) = *gridCrs*(Cj),   
 *interpolation*(Ci) = *interpolation*(Cj).

Then,

for any **coverageExpr**C’  
where C’ is one of   
 C’a = { f1 : C1 ; … ; fn : Cn }  
 C’b = struct { f1 : C1 ; … ; fn : Cn }

C’ is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C’) = “” (empty string) |
| *crs*(C’) = *crs*(C1) |
| *domain*(C’) = *domain*(C1) |
| *rangeFieldNames*(C’) = (f1, …, fn )  for all range fields fi:  *rangeFieldType*(C’,fi) = *rangeFieldType*(Ci) |
| for all p∈*domain*(C’):  *value*(C’.fi,p) = *value*(Ci,p) |
| for all range fields fi:  *interpolation*(C’) = *interpolation*(C1) |

EXAMPLE 1: The expression below does a false color encoding by combining near-infrared, red, and green bands into a 3-band image, which might 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   
}

#### 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.

1. **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 withn≥1,  
b1, …, bn be booleanExprs with a single Boolean range component,  
C1, …, Cn be coverageExprs with a single Boolean range component,  
R, R1, …, Rn+1 be coverageExprs,

where, for 1≤i≤n,

*crs*(C1) = … = *crs*(Cn) = *crs*(R1) = … = *crs*(Rn+1),  
*domain*(C1) = … = *domain*(Cn) = *domain*(R1) = … = *domain(*Rn+1),  
*interpolation(*C1) = … = *interpolation(*Cn) = *interpolation(*R1) = … = *interpolation(*Rn+1),  
*rangeType*(R1) = … = *rangeType*(Rn+1).

Then,

for any coverageExprC’  
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*(R1) |
| *domain(*C’) = *domain(*R1) |
| *interpolation*(C’) = *interpolation*(R1) |
| *rangeType*(*C’*) = rangeType(R1) |
| for all p∈*domain(*C’):  *value*( C’, p ) = V  where V =  if *value*( C1,p ) then *value*( R1,p )  else if *value*( C2,p ) then *value*( R2,p )  …  else if *value*( Cn,p ) then *value*( Rn,p )  else *value*( Rn+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 log of all positive values in $c, and assigns 0 to the remaining ones. This way it avoids an exception that would otherwise be thrown should any cell not be above zero.

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

### Coverage Domain-Changing Expressions

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

1. **https://standards.isotc211.org/19123/-3/1/req/core/subsetExpr**  
   A **subsetExpr shall** be either a **trimExpr** (Subclause 7.5.6.2) or a **sliceExpr** (Subclause 7.5.6.3) or an **extendExpr** (Subclause 7.5.6.3) or a **scaling­Expr** (Subclause 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**(cf. Subclause 7.5.3.4).

#### 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.

1. **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.

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

Let

C1 be a **coverageExpr**,  
n be an **integer** with 0≤n,  
(lo1:hi1),…,(lon:hin) be **dimensionIntervalExpr**s with loi≤hii for 1≤i≤n.

Then,

for any **coverageExpr**C2  
where C2 is one of  
 Cbracket = C1[p1, …, pn ]  
with   
 pi is one of  
 pnat,I = ai ( loi : hii )  
 pcrs,I = ai **:** crsi ( loi : hii )

where each interval is within the coverage’s bounds, as expressed by interval and axis (possibly reprojected from an optional CRS indicated)

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain(*C2) = *domain(*C1) reduced to extent (loi:hii) for any domain axis ai (reprojected from crsi into the coverage CRS if crsi is present), and with domain extent properly adjusted for any index axis ai present in the trim list |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeType*(C2 ) = *rangeType*(C1) |
| for all p∈*domain(*C2):  *value*(C2, p) = *value*(C1, p) |

EXAMPLE The following are syntactically valid, equivalent trim expressions:

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

#### 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.

1. **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.

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

Let

C1 be a **coverageExpr**,  
n be an **integer** with 0≤n,  
a1,…,an be pairwise distinct **axisName**s with ai ∈*axisNameSet*(C1) for 1≤i≤n,  
s1,…,sn be **axisPointExpr**s for 1≤i≤n. which evaluate, according to normal arithmetic rules, to coordinate values.

Then,

for any **coverageExpr**C2  
where C2 is one of  
 Cbracket = C1[S1, …, Sn ]  
with   
 Si is one of  
 Snat,I = ai ( si )  
 Scrs,I = ai **:** crsi ( si )

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| id(C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) projected to the axes remaining |
| *domain(*C2) = *domain(*C1) reduced to the axes of *nativeCrs*(C2) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeType*(C2) = *rangeType*(C1) |
| for all p∈*domain(*C1) :  *value*(C2, p) = *value*(C1,p’) where p’is the projection of p to *nativeCrs*(C2) |

EXAMPLE The following is a valid slice expression:

$c[ Date ( “2021-08-28” ) ]

#### 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 In this sense the **extendExpr** is a generalization of the **trimExpr**; still 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.

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

Let

C1 be a **coverageExpr**,  
n be an **integer** with 0≤n,  
a1,…,an be pairwise distinct **axisName**s with ai ∈*axisList*(*nativeCrs*(C1)) for 1≤i≤n,  
crs1,…,crsn be **crsName**s with crsi ∈crsList(C1) for 1≤i≤n,  
(lo1:hi1),…,(lon:hin) be **dimensionIntervalExpr**s with loi≤hii for 1≤i≤n,  
N be 0 or NaN or some null value (to be defined by a concretization of this document).

Then,

for any **coverageExpr**C2  
where  
 C2 = extend ( C1, {p1, …, pn } )  
with   
 pi is one of  
 pnat,I = ai ( loi : hii )  
 pcrs,I = ai : crsi ( loi : hii )

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = *crs*(C1) |
| *domain(*C2) = *domain(*C1) adjusted to extent (loi:hii) for any domain axis ai (reprojected from crsi into the coverage nativeCRS if crsi is present), and with domain extent properly adjusted for any axis ai present in the extend list; axes not mentioned remain unchanged. |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeType*(C2 ) = *rangeType*(C1) |
| for all p∈*domain*(C2):  *value*( C2, p ) = *value*(C1,p) for p∈*domain*(C1)   *value*( C2, p ) = N otherwise |

Note 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 ) } )

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

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

Let

C1 be a **coverageExpr** with only index and regular grid axes,  
m, n be **integer**s with 0≤m and 0≤n,  
a1,…,am be pairwise distinct **axisName**s with ai∈*gridCrs*(C1) for 1≤i≤m,  
Ii be **intervalExpr**s for 1≤i≤m which evaluate to pairs loi, hii with loi≤hii.

Then,

For any **coverageExpr** C2,  
where  
 C2 = scale ( C1, { a1 ( I1 ), …, am ( Im )} **)**

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *rs*(C2) = *crs*(C1) |
| *domain*(C2) = *domain*(C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeType*(C2) = *rangeType*(C1) |
| for all p∈*domain*(C2):  *value*(C2, p ) is obtained by rescaling the coverage grid along dimensions ai such that the coverage’s extent along dimension ai is set to (loi:hii), 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 might have further axes, such as time, which would remain unaffected.

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

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

## Coverage Derivation Expressions

### 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 usually is 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 may not be supported.

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

Let

C1 be a **coverageExpr**,  
c be a **crsName**.

Then,

for any **coverageExpr**C2  
where  
 C2 = crsTransform( C1, c **)**

C2 is defined as:

|  |
| --- |
| **Coverage constituent** |
| *id(*C2) = “” (empty string) |
| *crs*(C2) = c |
| *domain(*C2) = *domain(*C1) |
| *interpolation*(C2) = *interpolation*(C1) |
| *rangeFieldNames*(C2) = *rangeFieldNames*(C1)  for all range fields r∈*rangeFieldNames*(C2):  *rangeFieldType*(C2,r) = *rangeFieldType*(C1,r) |
| for all p∈*domain(*C2):  *value*(C2,p) is obtained by reprojecting coverage C1 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” )

## Coverage Aggregation Expressions

### condenseExpr

1. **https://standards.isotc211.org/19123/-3/1/req/core/condenseExpr**  
   A **condenseExpr shall** be either a **reduceExpr** (see Subclause6.7.3) or a **general­Condense­Expr** (see Subclause 6.7.2).

This expression takes a coverage and summarizes its values using some 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. Handling of null values is is governed by the value set definition which is out of scope of this document. Rather, 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.

### 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**,** andor.

1. **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≥0,  
d be some **integer** with d>0,  
axisi be **axisName**s for 1≤i≤d,  
namei be pairwise distinct **variableName**s for 1≤i≤d which, in the request on hand, are not used already as a variable in this expression’s scope,  
Ii be **intervalExpr**s for 1≤i≤d which evaluate to pairs loi, hii with loi≤hii,  
Cj be **coverageExpr**s for 1≤j≤n,  
P be a **booleanExpr** possibly containing occurrences of namei and Cj,  
V be a **scalarExpr** or **coverageExpr** possibly containing occurrences of namei and Cj,  
N be a neutral element of type(V)  
where  
 1≤i≤d**.**

Then,

For any **scalarExpr**S  
where S is one of  
 S’ = condense op  
 over name1 axis1 ( I1 ),  
 …,  
 named axisd ( Id )  
 [ whereP ]  
 using V

S” = condense op  
 over axis1 ( I1 ),  
 …,  
 axisd ( Id )  
 [ whereP ]  
 using V

S is constructed as follows (for S”, substitute namei by axisi):

S := N;  
for all name1 ∈ {lo1,… ,hi1}  
 for all name2 ∈ {lo2,… ,hi2}  
 …  
 for all named ∈ {lod,… ,hid}  
 if (filtering expression P is present)  
 then  
 let predicate P’ be obtained from evaluating expression   
 P by substituting all occurrences of namei by its current  
 value where namei occurring in a coordinate position  
 of Cj are coordinates in the CRS of Cj  
 else  
 P’ = *true*;  
 fi  
 if (P’)  
 then  
 let V’ be obtained from evaluating expression V  
 by substituting all occurrences of namei by its current   
 value where namei occurring in a coordinate position  
 of Cj are coordinates in the CRS of Cj 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 these 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 Subclause 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 summarisation 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** +  
**over**x ( 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** +  
**over**y ( 20 : 30 ), x ( 40 : 50 )  
**using** $c[ Lat(y) , Lon(x) ]

EXAMPLE 3 A timeline diagram can be obtained through a 1-D 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 summarises 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.

### 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**, however in a more verbose way.

1. **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.

Table 5 — reduceExpr definition via generalCondenseExpr

|  |  |  |  |
| --- | --- | --- | --- |
| **reduceExpr definition (**$a is assumed to evaluate to a coverage with a single numeric range field, $b to a coverage with a single Boolean range field. | | | Description |
| add($a) =  condense +  over $p**1** (domain($a,D1)),  …,  $pd (domain($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)),  …,  $pd (domain($a,D1))  using $a[ $p1 , …, $p**d** ] | minimum of all points in $a | |
| max($a) =  condense max  over $p1 (domain($a,D1)),  …,  $pd (domain($a,D1))  using $a[ $p1 , …, $p**d**] | maximum of all points in $a | |
| count($b) =  condense +  over $p1 (domain($b,D1)),  …,  $pd (domain($b,D1))  where $b[ $p1 , …, $p**d**]  using 1 | number of points in $b | |
| some($b) =  condense or  over $p1 (domain($b,D1)),  …,  $pd (domain($b,D1))  using $b[ $p1 , …, $p**d** ] | is there any point in $b with value true? | |
| all($b) =  condense and  over $p1 D1(domain($b,D1)),  …,  $pd Dd(domain($b,D1))  using $b[ $p1 , …, $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 )]

## Coverage Encode/Decode Expressions

### 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.

1. **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  
 Se = encode ( C,f )  
 See = 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 to reconstruct a complete coverage or reusing it in a *decode*() operation.

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

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

### decodeCoverageExpr

A decodeCoverageExpr evaluates a byte stream passed as parameter to a coverage by decoding the byte stream. This byte stream must represent a coverage encoding following CIS 1.1 [09-146r6] 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.

1. **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  
whereC is one of  
 Ce = decode( s )  
 Cee = decode( s, extraParams )

C is defined as the decoded coverage or coverage component equivalent toswhile applying the directives in extraParams.

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

* To provide inline constants, encoded, e.g., 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 decodes the NetCDF byte stream and establishes the corresponding coverage before further evaluation of the complete query:

decode( $1 )

## Expression evaluation

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

### Evaluation sequence

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

### Nesting

1. **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.

### Parentheses

A **processingExpr**may containparentheses to enforce a particular evaluation sequence.

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

Let

C1 and C2 be **coverageExpr**s.

Then,

For any **coverageExpr** C2where  
 C2 = ( C1 )

C2 is defined as yielding the same result as C1.

EXAMPLE $c \* ( $c > 0 )

### Operator precedence rules

1. **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 done left to right.

### Range type compatibility and extension

A range type t1 is said to be **cast-compatible** with a range type t2 iff the following conditions hold:

* Both range types,t1 and t2, have the same number of field elements, say d;
* For each range field element positioni with 1≤i≤d, theith range field type f1,i of t1 is **cast-compatible** with theith range field type f2,i of t2.

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

1. **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 loose 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.

## Evaluation response

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

1. **https://standards.isotc211.org/19123/-3/1/req/core/error**  
   Whenever a coverage expression cannot be evaluated according to the rules specified in Clauses 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:

1. **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.

1. (normative)  
     
   Conformance Tests
   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 Annex A (normative) of this standard.

* 1. 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 |

1. (normative)  
     
   Expression Syntax
   1. Overview

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

Note This is a machine readable language not requiring formal translation into ISO supported languages.

Tokens in single quotes 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 The syntax as is remains ambiguous; the semantic rules listed in this document disambiguate the grammar.

* 1. 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 upper and lower case 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.

* 1. 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 ')'

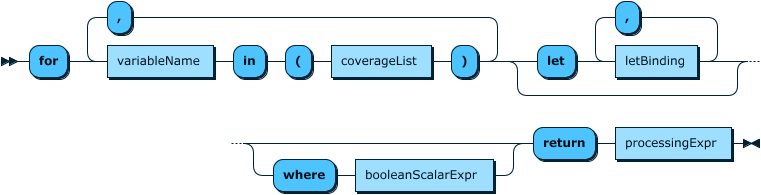
nameOrString ::=  
 name  
 | stringConstant

1. (non-normative)  
     
   Syntax diagrams

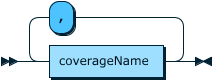
The following graphical representation of the syntax (often called “syntax diagrams” or “railroad diagrams”) is provided for the reader’s convenience. In case of deviation the normative syntax in Annex B prevails.

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

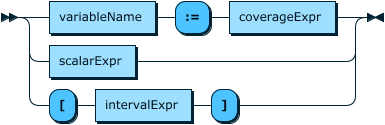
Note 2 Diagrams generated by [RR - Railroad Diagram Generator](https://bottlecaps.de/rr/ui).



1. **- processCoveragesExpr**



1. **- coverageList**



1. **- letBinding**



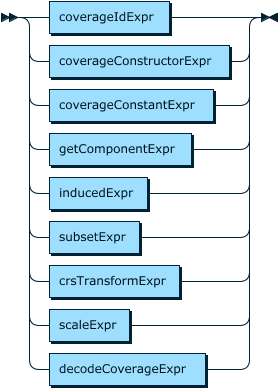
1. **- processingExpr**

C:\Documents and Settings\Administrator\Desktop\diagram\formatName.png

1. **- formatName**

C:\Documents and Settings\Administrator\Desktop\diagram\extraParams.png

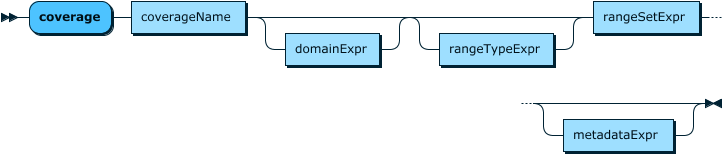
1. **- extraParams**



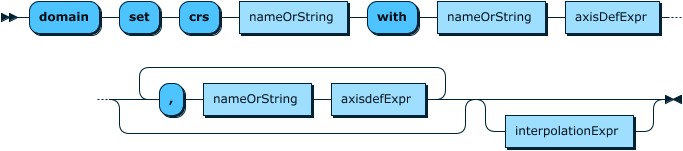
1. **- coverageExpr**

C:\Documents and Settings\Administrator\Desktop\diagram\coverageIdExpr.png

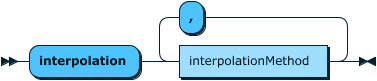
1. **- coverageIdExpr**



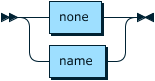
1. **- coverageConstructorExpr**



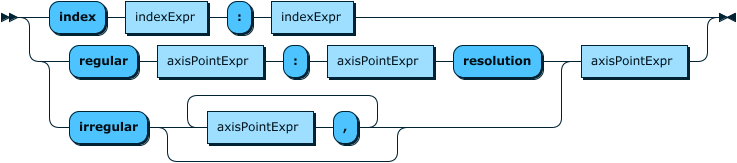
1. **- domainExpr**



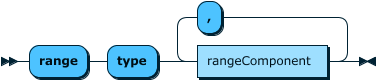
1. **- interpolationExpr**



1. **- interpolationMethod**



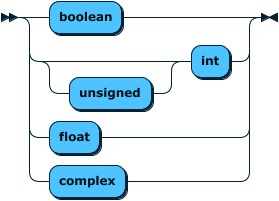
1. **- axisDefExpr**

****

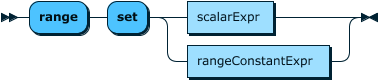
1. **- rangeTypeExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\rangeComponent.png**

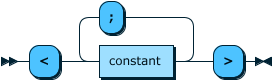
1. **- rangeComponent**

****

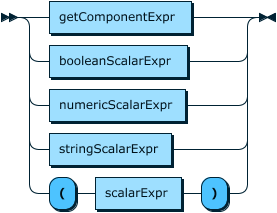
1. **- rangeType**

****

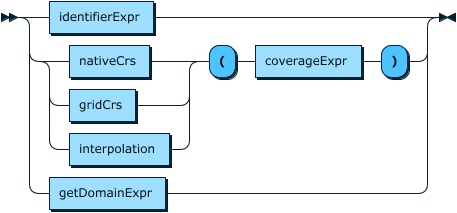
1. **- rangeSetExpr**

****

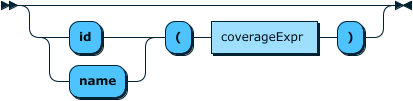
1. **- rangeConstantExpr**

****

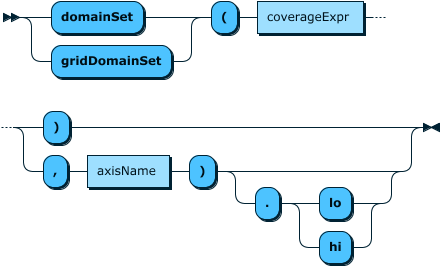
1. **- scalarExpr**

****

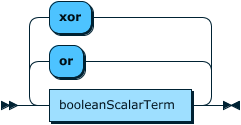
1. **- getComponentExpr**

****

1. **- identifierExpr**

****

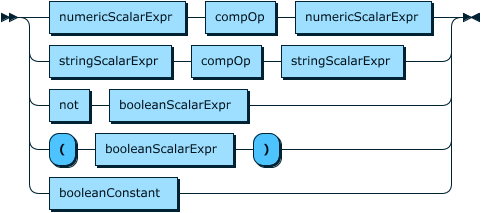
1. **- getDomainExpr**

****

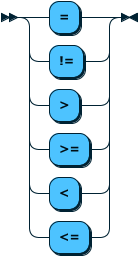
1. **- booleanScalarExpr**

****

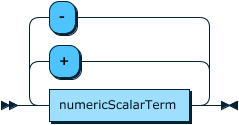
1. **- booleanScalarTerm**

****

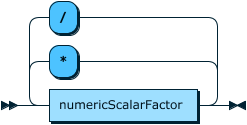
1. **- booleanScalarFactor**

****

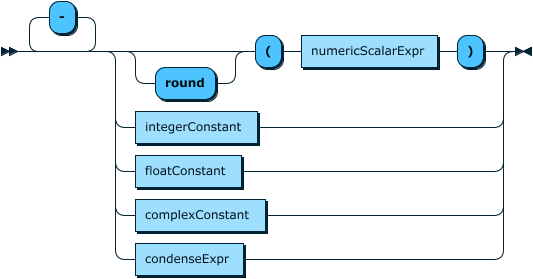
1. **- compOp**

****

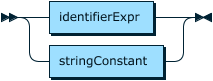
1. **- numericScalarExpr**

****

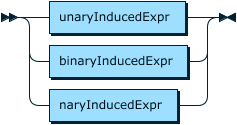
1. **- numericScalarTerm**

****

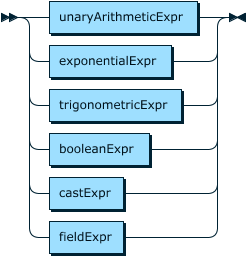
1. **- numericScalarFactor**

****

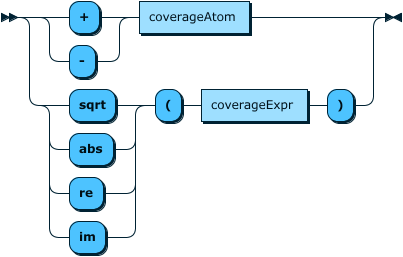
1. **- stringScalarExpr**

****

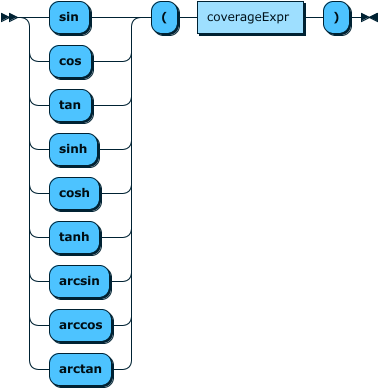
1. **- inducedExpr**

****

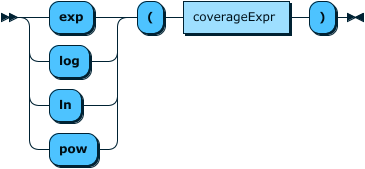
1. **- unaryInducedExpr**

****

1. **- unaryArithmeticExpr**

****

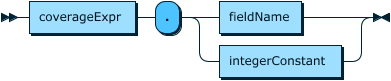
1. **- trigonometricExpr**

****

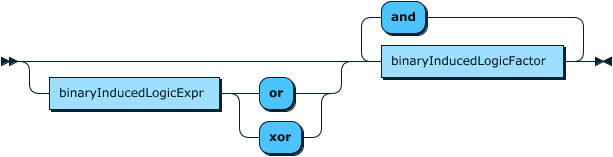
1. **- exponentialExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\castExpr.png**

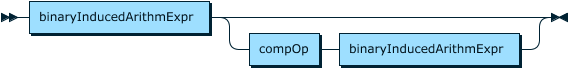
1. **- castExpr**

****

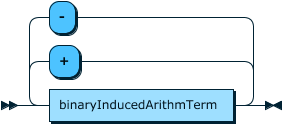
1. **- fieldExpr**

****

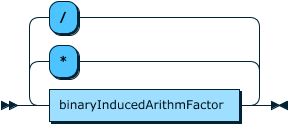
1. **- binaryInducedExpr**

****

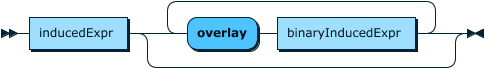
1. **- binaryInducedLogicFactor**

****

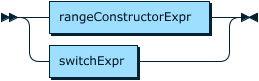
1. **- binaryInducedArithmExpr**

****

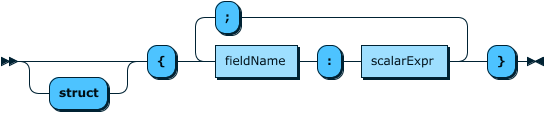
1. **- binaryInducedArithmTerm**

****

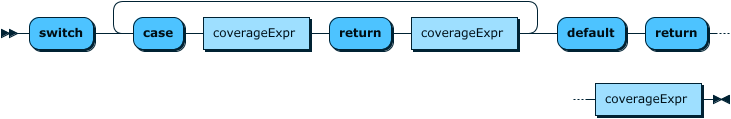
1. **- binaryInducedArithmFactor**

****

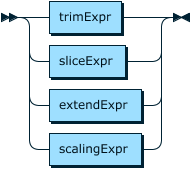
1. **- naryInducedExpr**

****

1. **- rangeConstructorExpr**

****

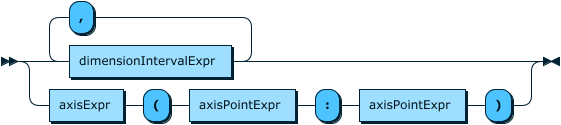
1. **- switchExpr**

****

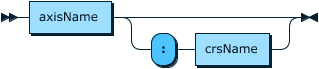
1. **- subsetExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\trimExpr.png**

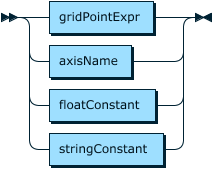
1. **- trimExpr**

****

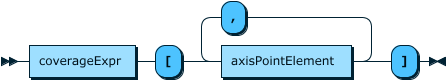
1. **- dimensionIntervalExpr**

****

1. **- axisExpr**

****

1. **- axisPointExpr**

****

1. **- sliceExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\axisPointElement.png**

1. **- axisPointElement**

**C:\Documents and Settings\Administrator\Desktop\diagram\extendExpr.png**

1. **- extendExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\scaleExpr.png**

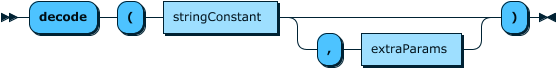
1. **- scaleExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\crsTransformExpr.png**

1. **- crsTransformExpr**

****

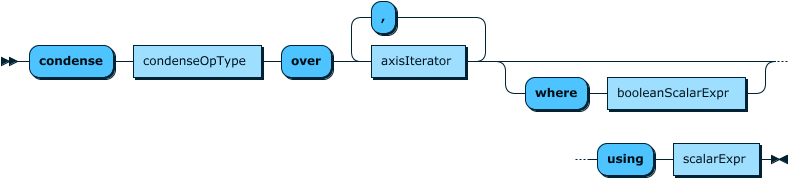
1. **- encodeCoverageExpr**

****

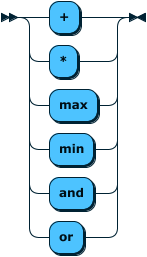
1. **- decodeCoverageExpr**

****

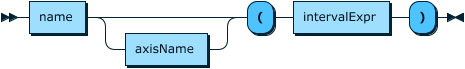
1. **- condenseExpr**

****

1. **- generalCondenseExpr**

****

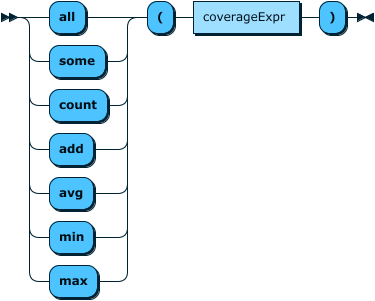
1. **- condenseOpType**

****

1. **- axisIterator**

**C:\Documents and Settings\Administrator\Desktop\diagram\intervalExpr.png**

1. **- intervalExpr**

****

**- reduceExpr**

**C:\Documents and Settings\Administrator\Desktop\diagram\coverageName.png**

1. **- coverageName**

**C:\Documents and Settings\Administrator\Desktop\diagram\crsName.png**

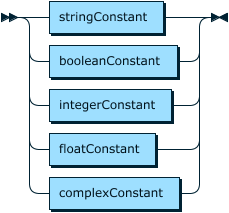
1. **- crsName**

**C:\Documents and Settings\Administrator\Desktop\diagram\axisName.png**

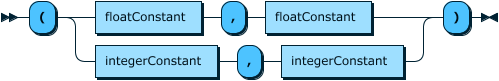
1. **- axisName**

**C:\Documents and Settings\Administrator\Desktop\diagram\fieldName.png**

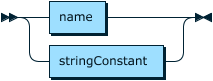
1. **- fieldName**

****

1. **- constant**

****

1. **- complexConstant**

****

1. **- nameOrString**
2. (non-normative)  
     
   Sample service descriptions
   1. Overview

This Annex presents, as an example of using 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.

* 1. 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 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} )

* 1. 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} )

* 1. 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} )

* 1. 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} )

* 1. 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 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, <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, <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, <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). <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