

A Declarative Semantics for SNOMED CT Expression Constraints

May 1, 2015

Contents

1	Data Types	3
1.1	Atomic Data Types	3
1.2	Composite Data Types	4
2	The Substrate	5
2.1	Substrate Components	5
2.1.1	Quad	5
2.1.2	conceptReference	5
2.2	Substrate	6
2.2.1	Substrate Definition	6
2.2.2	Strict Substrate	9
2.2.3	Permissive Substrate	9
3	Interpretation of Expression Constraints	11
3.1	Interpretation Output	11
3.2	expressionConstraint	11
3.2.1	unrefinedExpressionConstraint	12
3.2.2	refinedExpressionConstraint	12
3.2.3	simpleExpressionConstraint	13
3.2.4	compoundExpressionConstraint	14
3.2.5	conjunctionExpressionConstraint	15
3.2.6	disjunctionExpressionConstraint	15
3.2.7	exclusionExpressionConstraint	16
3.2.8	subExpressionConstraint	16
3.3	refinement	17
3.3.1	conjunctionRefinementSet	18
3.3.2	disjunctionRefinementSet	19
3.3.3	subRefinement	19
3.4	attributeSet	20
3.4.1	conjunctionAttributeSet	20
3.4.2	disjunctionAttributeSet	21

3.4.3	subAttributeSet	22
3.5	attribute	22
3.5.1	attribute cardinality	24
3.5.2	expressionConstraintValue	26
3.6	attributeGroup	26
3.6.1	attribute group cardinality	27
3.6.2	attributeSet inside a groups	28
3.6.3	conjunctionAttributeSet inside a group	29
3.6.4	disjunctionAttributeSet inside a group	29
3.6.5	subAttributeSet inside a group	29
3.6.6	attribute inside a group	30
3.6.7	cardinality inside a group	30
4	Substrate Interpretations	32
4.1	constraintOperator	32
4.2	attributeOperator	33
4.3	expressionComparisonOperator	34
4.4	numericComparisonOperator	34
4.5	stringComparisonOperator	35
4.6	focusConcept	36
4.6.1	memberOf	37
5	Types	38
5.1	Types	38
5.1.1	Quads_or_Error	38
5.1.2	setIdGroups	38
5.2	Result transformations	38
A	Optional elements	41
B	Generic cardinality evaluation	41
C	Generic sequence function	41

1 Data Types

1.1 Atomic Data Types

This section identifies the atomic data types that are assumed for the rest of this specification. These types are defined as lexical components in the specification.

```
sctId = digitNonZero 5*17( digit )
term = 1*nonwsNonPipe *( 1*SP 1*nonwsNonPipe )
numericValue = QM stringValue QM / "#" numericValue ( decimalValue / integerValue)
stringValue = QM 1*(anyNonEscapedChar / escapedChar) QM
integerValue = ( ["-" / "+" ] digitNonZero *digit ) / zero
decimalValue = integerValue "." 1*digit
nonNegativeIntegerValue = (digitNonZero *digit ) / zero
```

- **SCTID** – A *SNOMED CT* id is used to represent an attribute id or a concept id. The initial digit may not be zero. The smallest number of digits is six, and the maximum is 18.
- **TERM** – The term must be the term from a *SNOMED CT* description that is associated with the concept identified by the preceding concept identifier. For example, the term could be the preferred description, or the preferred description associated with a particular translation. The term may include valid UTF-8 characters except for the pipe "|" character. The term begins with the first non-whitespace character following the starting "|" character and ends with the last non-whitespace character preceding the next "|" character.
- **STRINGVALUE** – A string value includes one or more of any printable ASCII characters enclosed in quotation marks. Quotes and backslash characters within the string must be preceded by the escape character ("\").
- **Z** – built-in Z equivalent of **INTEGERVALUE** . An integer may be positive, negative or zero. Positive integers optionally start with a plus sign ("+"), followed by a non-zero digit followed by zero to many additional digits. Negative integers begin with a minus sign ("-") followed by a non-zero digit and zero to many additional digits.
- **DECIMALVALUE** – A decimal value starts with an integer. This is followed by a decimal point and one to many digits.
- **N** – built-in Z equivalent of **NONNEGATIVEINTEGERVALUE**
- **GROUPID** – a role group identifier. (Not explicated in the specification, but needed)

[*sctId*, *term*, *decimalValue*, *stringValue*, *groupId*]

We specifically define some well known identifiers: the *is_a* attribute, the *zero_group* and *attribute_concept*, and *refset_concept* the parents of all attributes and all refsets respectively.

	<i>is_a</i> : <i>sctId</i>
	<i>zero_group</i> : <i>groupId</i>
	<i>attribute_concept</i> : <i>sctId</i>
	<i>refset_concept</i> : <i>sctId</i>

1.2 Composite Data Types

concreteValue is in anticipation of the enhanced specification that includes target strings, integers or decimals.

- **concreteValue** – a string, integer or decimal literal
- **target** – the target of a relationship that is either an *sctId* or a *concreteValue*

concreteValue ::=
 cv_string⟨⟨*stringValue*⟩⟩ | *cv_integer*⟨⟨ \mathbb{Z} ⟩⟩ | *cv_decimal*⟨⟨*decimalValue*⟩⟩
target ::= *t_sctid*⟨⟨*sctId*⟩⟩ | *t_concrete*⟨⟨*concreteValue*⟩⟩

2 The Substrate

A substrate represents the context of an interpretation.

2.1 Substrate Components

2.1.1 Quad

Relationships in the substrate are represented a 4 element tuples or “quads” which consist of a source, attribute, target and role group identifier. The *is_a* attribute may only appear in the zero group, and the target of an *is_a* attribute must be a *sctId* (not a *concreteValue*)

<i>Quad</i>
<i>s</i> : <i>sctId</i>
<i>a</i> : <i>sctId</i>
<i>t</i> : <i>target</i>
<i>g</i> : <i>groupId</i>
$a = is_a \Rightarrow (g = zero_group \wedge t \in \text{ran } t_sctid)$

2.1.2 conceptReference

The root of the expression constraint language is concept references:

- **CONCEPTREFERENCE** – A *conceptReference* is represented by a *ConceptId* optionally followed by a term enclosed by a pair of “|” characters. Whitespace before or after the *ConceptId* is ignored as is any whitespace between the initial “|” characters and the first non-whitespace character in the term or between the last non-whitespace character and before second “|” character. **TERM** is ignored for the purposes of interpretation.
- **CONCEPTID** – The *ConceptId* must be a valid SNOMED CT identifier for a concept. The initial digit may not be zero. The smallest number of digits is six, and the maximum is 18.
- **ATTRIBUTEName** – The attribute name is the name of an attribute (or relationship type) to which a value is applied to refine the meaning of a containing expression. The attribute name is represented in the same way as other concept references. If the attribute name is not known then a wild card may be used to represent any attribute concept in the given substrate.

<pre>conceptReference = conceptId [ws " " ws term ws " "] conceptId = sctId attributeName = conceptReference / wildCard</pre>

$conceptReference == conceptId \times term[0..1]$
 $conceptId == sctId$
 $attributeName ::= ancr\langle\langle conceptReference \rangle\rangle \mid anwc$

2.2 Substrate

2.2.1 Substrate Definition

A substrate consists of:

- **concepts** The set of *sctIds* (concepts) that are considered valid in the context of the substrate.
- **relationships** A set of relationship quads (source, attribute, target, group)
- **parentsOf** A function from an *sctId* to its asserted and inferred parents
- **equivalent_concepts** A function from an *sctId* to the set of *sctId*'s that have been determined to be equivalent to it.
- **refsets** The reference sets within the context of the substrate whose members are members are concept identifiers (i.e. are in *concepts*). While not formally spelled out in this specification, it is assumed that the typical reference set function would be returning a subset of the *refsetId* / *referencedComponentId* tuples represented in one or more RF2 Refset Distribution tables.

The following functions can be computed from the basic set above

- **childrenOf** The inverse of the *parentsOf* function
- **descendants** The transitive closure of the *childrenOf* function
- **ancestors** The transitive closure of the *parentsOf* function

A substrate also implements three functions:

- **i_conceptReference** The interpretation of a concept reference. This function can return a (possibly empty) set of *sctId*'s or an error.
- **i_attributeName** The interpretation of an attribute name. This function can return a (possibly empty) set of *sctId*'s or an error.
- **i_refsetId** The interpretation of a refset identifier. This function can return a (possibly empty) set of *sctId*'s or an error.

The formal definition of substrate follows. The expressions below assert that:

1. All *relationship* sources, attributes and *sctId* targets are in *concepts*.
2. There is a *parentsOf* entry for every substrate *concept*.
3. Every *sctId* that can be returned by the *parentsOf* function is a *concept*.
4. Every *is_a* relationship entry is represented in the *parentsOf* function.
Note that there can be additional entries represented in the *parentsOf* function that aren't in the relationships table.
5. There is an *equivalent_concepts* assertion for every substrate *concept*.
6. The *equivalent_concepts* function is reflexive (i.e. every concept is equivalent to itself).
7. All equivalent concepts are in *concepts*.

8. If two concepts (c_1 and c_2) are equivalent, then they:
 - Have the same parents
 - Appear the subject, attribute and object of the same set of relationships
 - Appear in the domain of the same set of refsets
 - Both appear in the range of any refset that one appears in
9. Every refset is a substrate *concept*
10. Every member of a refset is a substrate *concept*
11. *childrenOf* is the inverse of *parentsOf*, where any concept that doesn't appear a parent has no (the emptyset) children.
12. *descendants* is the transitive closure of the *childrenOf* function
13. *ancestors* is the transitive closure of the *parentsOf* function
14. No concept can be its own ancestor (or, by inference, descendant)
15. The *i_conceptReference*, *i_attributeName* and *i_refsetId* functions are defined for all possible *conceptReferences* and *attributeNames* (because they are complete functions).
16. All *sctId*'s that are produced by the The *i_conceptReference*, *i_attributeName* and *i_refsetId* functions are substrate *concepts*.

Substrate

$concepts : \mathbb{P} \text{ sctId}$
 $relationships : \mathbb{P} \text{ Quad}$
 $parentsOf : \text{ sctId} \rightarrow \mathbb{P} \text{ sctId}$
 $equivalent_concepts : \text{ sctId} \rightarrow \mathbb{P} \text{ sctId}$
 $refsets : \text{ sctId} \rightarrow \mathbb{P} \text{ sctId}$

 $childrenOf : \text{ sctId} \rightarrow \mathbb{P} \text{ sctId}$
 $descendants : \text{ sctId} \rightarrow \mathbb{P} \text{ sctId}$
 $ancestors : \text{ sctId} \rightarrow \mathbb{P} \text{ sctId}$

 $groups : \mathbb{P} \text{ groupId}$
 $subjects : \mathbb{P} \text{ sctId}$
 $targets : \mathbb{P} \text{ target}$

 $i_conceptReference : \text{ conceptReference} \rightarrow \text{ Sctids_or_Error}$
 $i_attributeName : \text{ attributeName} \rightarrow \text{ Sctids_or_Error}$
 $i_refsetId : \text{ sctId} \rightarrow \text{ Sctids_or_Error}$

$\forall rel : relationships \bullet rel.s \in concepts \wedge rel.a \in concepts \wedge$
 $(rel.t \in \text{ran } t_sctid \Rightarrow t_sctid \sim rel.t \in concepts)$

$\text{dom } parentsOf = concepts$
 $\bigcup(\text{ran } parentsOf) \subseteq concepts$

$\forall r : relationships \bullet r.a = is_a \Rightarrow (t_sctid \sim r.t) \in parentsOf \ r.s$

$\text{dom } equivalent_concepts = concepts$
 $\bigcup(\text{ran } equivalent_concepts) \subseteq concepts$

$\forall c : concepts \bullet c \in equivalent_concepts \ c$

$\forall c1, c2 : concepts \mid c2 \in (equivalent_concepts \ c1) \bullet$

$parentsOf \ c1 = parentsOf \ c2 \wedge$

$\{r : relationships \mid r.s = c1\} = \{r : relationships \mid r.s = c2\} \wedge$

$\{r : relationships \mid r.a = c1\} = \{r : relationships \mid r.a = c2\} \wedge$

$\{r : relationships \mid t_sctid \sim r.t = c1\} = \{r : relationships \mid t_sctid \sim r.t = c2\} \wedge$

$c1 \in \text{dom } refsets \Leftrightarrow c2 \in \text{dom } refsets \wedge$

$c1 \in \text{dom } refsets \Rightarrow refsets \ c1 = refsets \ c2 \wedge$

$(\forall rsd : \text{ran } refsets \bullet c1 \in rsd \Leftrightarrow c2 \in rsd)$

$\text{dom } refsets \subseteq concepts$

$\bigcup(\text{ran } refsets) \subseteq concepts$

$\text{dom } childrenOf = concepts$

$\forall s, t : concepts \bullet t \in parentsOf \ s \Leftrightarrow s \in childrenOf \ t$

$\forall c : concepts \mid c \notin \bigcup(\text{ran } childrenOf) \bullet childrenOf \ c = \emptyset$

$\forall s : concepts \bullet$

$descendants \ s = childrenOf \ s \cup \bigcup\{t : childrenOf \ s \bullet descendants \ t\}$

$\forall t : concepts \bullet$

$ancestors \ t = parentsOf \ t \cup \bigcup\{s : parentsOf \ t \bullet ancestors \ s\}$

$\forall t : concepts \bullet t \notin ancestors \ t$

$groups = \{r : relationships \bullet r.g\}$

$subjects = \{r : relationships \bullet r.s\}$

$targets = \{r : relationships \bullet r.t\}$

$\forall cr_interp : \text{ran } i_conceptReference \mid cr_interp \in \text{ran } ok \bullet$
 $result_sctids \ cr_interp \subseteq concepts$

$\forall att_interp : \text{ran } i_attributeName \mid att_interp \in \text{ran } ok \bullet$
 $result_sctids \ att_interp \subseteq concepts$

$\forall refset_interp : \text{ran } i_refsetId \mid refset_interp \in \text{ran } ok \bullet$
 $result_sctids \ refset_interp \subseteq concepts$

Implementations may choose to implement “strict” substrates, where additional rules apply or “permissive” substrates where rules are relaxed.

2.2.2 Strict Substrate

A **strict_substrate** is a substrate where:

- If a `conceptReference` is not in the substrate concepts it returns an error, otherwise the set of equivalent concepts
- If an attribute name is a `conceptReference` and it is not in the substrate concepts or is not a descendant of the `attribute_concept` it returns an error, otherwise the set of equivalent attributes. If an attribute name is a wild card, it returns all descendants of `attribute_concept`
- If a concept reference that is a target of a `memberOf` function is not in the substrate concepts or is not a descendant of the `refset_concept` it returns an error, otherwise the set of equivalent refset identifiers

<i>strict_substrate</i>
<div style="border-bottom: 1px solid black; margin-bottom: 10px;"> <i>Substrate</i> </div> <div> $\forall cr : \text{conceptReference} \bullet i_conceptReference\ cr =$ if $first\ cr \notin concepts$ then $error\ unknownConceptReference$ else $ok\ (equivalent_concepts\ (first\ cr))$ $\forall an : attributeName \bullet i_attributeName\ an =$ if $ancr \sim an \in conceptReference$ then $(let\ rslt == i_conceptReference\ (ancr \sim an) \bullet$ if $rslt \in ran\ error$ then $rslt$ else if $result_sctids\ rslt \subseteq (descendants\ attribute_concept)$ then $rslt$ else $error\ unknownAttributeId)$ else $ok\ (descendants\ attribute_concept)$ $\forall rsid : sctId \bullet i_refsetId\ rsid =$ if $rsid \in descendants\ refset_concept$ then $ok\ \{rsid\}$ else $error\ unknownRefsetId$ </div>

2.2.3 Permissive Substrate

A **permissive_substrate** never raises an error condition. concept references that are not in the substrate are “discarded” (i.e. map to the empty set). Any valid `conceptReference` may be used in the position of an attribute or a refset id. The `attributeName` wild card returns the set of all possible concepts.

permissive_substrate

Substrate

```

 $\forall cr : \text{conceptReference} \bullet i\_conceptReference\ cr =$ 
  if first cr  $\notin$  concepts then ok  $\emptyset$ 
  else ok (equivalent_concepts (first cr))

 $\forall an : \text{attributeName} \bullet i\_attributeName\ an =$ 
if ancr $\sim an \in \text{conceptReference}$  then
  i_conceptReference (ancr $\sim an$ )
else
  ok concepts

 $\forall rsid : \text{sctId} \bullet i\_refsetId\ rsid =$ 
  if rsid  $\notin$  concepts then ok  $\emptyset$ 
  else ok (equivalent_concepts (rsid))
```

3 Interpretation of Expression Constraints

An `EXPRESSIONCONSTRAINT` is interpreted in the context of a *Substrate* and returns a set of *sctIds* or an error indicator.

3.1 Interpretation Output

The result of applying a query against a substrate is either a (possibly empty) set of *sctId*'s or an *ERROR*. An *ERROR* occurs when:

- The interpretation of a *conceptId* is not a substrate *concept*
- The interpretation of a relationship attribute is not a substrate *attributeId*
- The interpretation of a reset is not a substrate *refsetId*

$$ERROR ::= unknownConceptReference \mid unknownAttributeId \mid unknownRefsetId$$

$$Sctids_or_Error ::= ok\langle\mathbb{P} \textit{sctId}\rangle \mid error\langle\langle ERROR \rangle\rangle$$

Each interpretation that follows begins with a simplified version of the language construct in the specification. It then formally specifies the constructs that are used in the interpretation, followed by the interpretation itself. We start with the definition of *expressionConstraint*, which, once interpreted, returns either a set of *sctIds* or an error condition.

3.2 expressionConstraint

An expression constraint is either a refined expression constraint or an unrefined expression constraint.

$$\textit{expressionConstraint} = \textit{ws} \left(\textit{refinedExpressionConstraint} / \textit{unrefinedExpressionConstraint} \right)$$

$$\textit{ws}$$

The interpretation of an `EXPRESSIONCONSTRAINT` is the interpretation of either the `REFINEDEXPRESSIONCONSTRAINT` or the `UNREFINEDEXPRESSIONCONSTRAINT`.

$$\begin{aligned} \textit{expressionConstraint} ::= & \\ & \textit{expcons_refined}\langle\langle \textit{refinedExpressionConstraint} \rangle\rangle \mid \\ & \textit{expcons_unrefined}\langle\langle \textit{unrefinedExpressionConstraint} \rangle\rangle \end{aligned}$$

$i_expressionConstraint :$ $Substrate \rightarrow expressionConstraint \rightarrow Sctids_or_Error$
$\forall ss : Substrate; ec : expressionConstraint \bullet i_expressionConstraint ss ec =$ if $ec \in \text{ran } expcons_refined$ then $i_refinedExpressionConstraint ss (expcons_refined \sim ec)$ else $i_unrefinedExpressionConstraint ss (expcons_unrefined \sim ec)$

3.2.1 unrefinedExpressionConstraint

An unrefined expression constraint is either compound or simple.

$unrefinedExpressionConstraint = compoundExpressionConstraint / simpleExpressionConstraint$

The interpretation of an `UNREFINEDEXPRESSIONCONSTRAINT` is either the interpretation of a `COMPOUNDEXPRESSIONCONSTRAINT` or a `SIMPLEEXPRESSIONCONSTRAINT`

$unrefinedExpressionConstraint ::=$
 $unrefined_compound \langle\langle compoundExpressionConstraint \rangle\rangle \mid$
 $unrefined_simple \langle\langle simpleExpressionConstraint \rangle\rangle$

$i_unrefinedExpressionConstraint :$ $Substrate \rightarrow unrefinedExpressionConstraint \rightarrow Sctids_or_Error$
$\forall ss : Substrate; uec : unrefinedExpressionConstraint \bullet$ $i_unrefinedExpressionConstraint ss uec =$ if $uec \in \text{ran } unrefined_compound$ then $i_compoundExpressionConstraint ss (unrefined_compound \sim uec)$ else $i_simpleExpressionConstraint ss (unrefined_simple \sim uec)$

3.2.2 refinedExpressionConstraint

A refined expression constraint starts with an unrefined expression constraint and adds a refinement. Refined expression constraints may optionally be placed in brackets.

```

refinedExpressionConstraint = unrefinedExpressionConstraint ws ":" ws refinement / "(" ws
refinedExpressionConstraint ws ")"

```

The interpretation of `REFINEDEXPRESSIONCONSTRAINT` is the intersection of the interpretation of the `UNREFINEDEXPRESSIONCONSTRAINT` and the `REFINEMENT`, both of which return a set of `sctId`'s or an error. The second production defines `REFINEDEXPRESSIONCONSTRAINT` in terms of itself and has no effect on the results.

Note: The second declaration below (*refinedExpressionConstraint'*) is used to avoid circular definitions.

$$\begin{aligned}
 \text{refinedExpressionConstraint} &== \\
 &\quad \text{unrefinedExpressionConstraint} \times \text{refinement} \\
 &\quad [\text{refinedExpressionConstraint}']
 \end{aligned}$$

```

i_refinedExpressionConstraint :
  Substrate → refinedExpressionConstraint → Sctids_or_Error

∀ ss : Substrate; rec : refinedExpressionConstraint •
  (let unref_interp == (i_unrefinedExpressionConstraint ss (first rec)) •
   i_refinedExpressionConstraint ss rec =
     intersect unref_interp (i_refinement ss (second rec)))

```

Duplicate signature for recursive definitions

```

i_refinedExpressionConstraint' :
  Substrate → refinedExpressionConstraint' → Sctids_or_Error

```

3.2.3 simpleExpressionConstraint

A simple expression constraint includes exactly one focus concept, optionally preceded by a constraint operator.

```

simpleExpressionConstraint = [constraintOperator ws] focusConcept

```

The interpretation of `SIMPLEEXPRESSIONCONSTRAINT` is the application of an optional constraint operator to the interpretation of `FOCUSCONCEPT`, which returns a set of `sctId`'s or an error. The interpretation of an error is the error.

$$\text{simpleExpressionConstraint} == \text{constraintOperator}[0..1] \times \text{focusConcept}$$

$i_simpleExpressionConstraint :$ $Substrate \rightarrow simpleExpressionConstraint \rightarrow Sctids_or_Error$
$\forall ss : Substrate; sec : simpleExpressionConstraint \bullet$ $i_simpleExpressionConstraint ss sec =$ $i_constraintOperator ss (first sec) (i_focusConcept ss (second sec))$

3.2.4 compoundExpressionConstraint

A compound expression constraint contains two or more simple expression constraints joined by either a conjunction, disjunction or exclusion. Compound expression constraints may optionally be placed in brackets.

compoundExpressionConstraint = conjunctionExpressionConstraint / disjunctionExpressionConstraint / exclusionExpressionConstraint / "(" ws compoundExpressionConstraint ws ")"

The interpretation of a *compoundExpressionConstraint* is the interpretation the conjunction, disjunction or exclusion of nested compound constraint.

compoundExpressionConstraint ::=

$$\begin{aligned}
& compound_conj \langle\langle conjunctionExpressionConstraint \rangle\rangle \mid \\
& compound_disj \langle\langle disjunctionExpressionConstraint \rangle\rangle \mid \\
& compound_excl \langle\langle exclusionExpressionConstraint \rangle\rangle \mid \\
& compound_nested \langle\langle compoundExpressionConstraint' \rangle\rangle \\
& [compoundExpressionConstraint']
\end{aligned}$$

$i_compoundExpressionConstraint :$ $Substrate \rightarrow compoundExpressionConstraint \rightarrow Sctids_or_Error$
$\forall ss : Substrate; cec : compoundExpressionConstraint \bullet$ $i_compoundExpressionConstraint ss cec =$ <p>if $cec \in \text{ran } compound_conj$ then</p> $i_conjunctionExpressionConstraint ss (compound_conj \sim cec)$ <p>else if $cec \in \text{ran } compound_disj$ then</p> $i_disjunctionExpressionConstraint ss (compound_disj \sim cec)$ <p>else if $cec \in \text{ran } compound_excl$ then</p> $i_exclusionExpressionConstraint ss (compound_excl \sim cec)$ <p>else</p> $i_compoundExpressionConstraint' ss (compound_nested \sim cec)$

$i_compoundExpressionConstraint' :$ $Substrate \rightarrow compoundExpressionConstraint' \rightarrow Sctids_or_Error$
--

3.2.5 conjunctionExpressionConstraint

A conjunction expression constraint combines two or more expression constraints with a conjunction ("and") operator. More than one conjunction may be used without brackets. However any compound expression constraint (using a different binary operator) that appears within a conjunction expression constraint must be enclosed by brackets.

conjunctionExpressionConstraint = subExpressionConstraint 1*(ws conjunction ws subExpressionConstraint)

CONJUNCTIONEXPRESSIONCONSTRAINT is interpreted as the intersection of the interpretations of the SUBEXPRESSIONCONSTRAINTS .

$$\text{conjunctionExpressionConstraint} == \text{subExpressionConstraint} \times \text{seq}_1(\text{subExpressionConstraint})$$

Apply the intersection operator to the interpretation of each subExpressionConstraint

i_conjunctionExpressionConstraint :
Substrate → conjunctionExpressionConstraint → Sctids_or_Error

$\forall ss : \text{Substrate}; cecr : \text{conjunctionExpressionConstraint} \bullet$
i_conjunctionExpressionConstraint ss cecr =
applyToSequence ss i_subExpressionConstraint intersect cecr

3.2.6 disjunctionExpressionConstraint

A disjunction expression constraint combines two or more expression constraints with a disjunction ("or") operator. More than one disjunction may be used without brackets. However any compound expression constraint (using a different binary operator) that appears within a disjunction expression constraint must be enclosed by brackets.

disjunctionExpressionConstraint = subExpressionConstraint 1*(ws disjunction ws subExpressionConstraint)

$$\text{disjunctionExpressionConstraint} == \text{subExpressionConstraint} \times \text{seq}_1(\text{subExpressionConstraint})$$

`DISJUNCTIONEXPRESSIONCONSTRAINT` is interpreted as the union of the interpretations of the `SUBEXPRESSIONCONSTRAINTS` .

$i_disjunctionExpressionConstraint :$ $Substrate \rightarrow disjunctionExpressionConstraint \rightarrow Sctids_or_Error$
$\forall ss : Substrate; decr : disjunctionExpressionConstraint \bullet$ $i_disjunctionExpressionConstraint ss decr =$ $applyToSequence ss i_subExpressionConstraint union decr$

3.2.7 exclusionExpressionConstraint

An exclusion expression constraint combines two expression constraints with an exclusion ("minus") operator. A single exclusion operator may be used without brackets. However when the operands of the exclusion expression constraint are compound, these compound expression constraints must be enclosed by brackets.

`exclusionExpressionConstraint = subExpressionConstraint ws exclusion ws subExpressionConstraint`

`EXCLUSIONESPRESSIONCONSTRAINT` is interpreted as the interpretation of the `SUBEXPRESSIONCONSTRAINT` minus the interpretation of the second.

$exclusionExpressionConstraint ==$
 $subExpressionConstraint \times subExpressionConstraint$

$i_exclusionExpressionConstraint :$ $Substrate \rightarrow exclusionExpressionConstraint \rightarrow Sctids_or_Error$
$\forall ss : Substrate; ecr : exclusionExpressionConstraint \bullet$ $(\text{let } first_sec == (i_subExpressionConstraint ss (first ecr));$ $second_sec == (i_subExpressionConstraint ss (second ecr)) \bullet$ $i_exclusionExpressionConstraint ss ecr = minus first_sec second_sec)$

3.2.8 subExpressionConstraint

A subexpression constraint, which appears within a compound expression constraint, must either be simple, or a bracketed compound or refined expression constraint.


```
subExpressionConstraint = simpleExpressionConstraint / "(" ws (compoundExpressionCon-
straint / refinedExpressionConstraint) ws ")"
```

The interpretation of `SUBEXPRESSIONCONSTRAINT` is the interpretation the `SIMPLEEXPRESSIONCONSTRAINT` `COMPOUNDEXPRESSIONCONSTRAINT` or `REFINEDEXPRESSIONCONSTRAINT`

```
subExpressionConstraint ::=
  subExpr_simple⟨⟨simpleExpressionConstraint⟩⟩ |
  subExpr_compound⟨⟨compoundExpressionConstraint'⟩⟩ |
  subExpr_refined⟨⟨refinedExpressionConstraint'⟩⟩
```

```
i_subExpressionConstraint :
  Substrate → subExpressionConstraint → Sctids_or_Error

∀ ss : Substrate; sec : subExpressionConstraint •
i_subExpressionConstraint ss sec =
if sec ∈ ran subExpr_simple then
  i_simpleExpressionConstraint ss (subExpr_simple~sec)
else if sec ∈ ran subExpr_compound then
  i_compoundExpressionConstraint' ss (subExpr_compound~sec)
else
  i_refinedExpressionConstraint' ss (subExpr_refined~sec)
```

3.3 refinement

A refinement contains all the grouped and ungrouped attributes that refine the set of clinical meanings satisfied by the expression constraint. Refinements may represent the conjunction or disjunction of two smaller refinements, and may optionally be placed in brackets. Where both conjunction and disjunction are used, brackets are mandatory to disambiguate the intended meaning.

```
refinement = subRefinement ws [conjunctionRefinementSet / disjunctionRefinementSet]
```

The interpretation of `REFINEMENT` is one of:

- The intersection of the interpretation of the `SUBREFINEMENT` and the `CONJUNCTIONREFINEMENTSET`
- The union of the interpretation of the `SUBREFINEMENT` and the `DISJUNCTIONREFINEMENTSET`
- The interpretation of `SUBREFINEMENT` if neither conjunction nor disjunction set is present

```
refinement == subRefinement × refinementConjunctionOrDisjunction[0 .. 1]
[refinement']
```

```

refinementConjunctionOrDisjunction ::=
  refine_conjset⟨⟨conjunctionRefinementSet⟩⟩ |
  refine_disjset⟨⟨disjunctionRefinementSet⟩⟩

```

$i_refinement : Substrate \rightarrow refinement \rightarrow Sctids_or_Error$
$\forall ss : Substrate; rfnment : refinement \bullet$ $i_refinement\ ss\ rfnment =$ $(let\ lhs == i_subRefinement\ ss\ (first\ rfnment);\ rhs == second\ rfnment \bullet$ if $\#rhs = 0$ then lhs else if $(head\ rhs) \in \text{ran}\ refine_conjset$ then $intersect\ lhs\ (i_conjunctionRefinementSet\ ss\ (refine_conjset^\sim(head\ rhs)))$ else $union\ lhs\ (i_disjunctionRefinementSet\ ss\ (refine_disjset^\sim(head\ rhs)))$

$i_refinement' : Substrate \rightarrow refinement' \rightarrow Sctids_or_Error$
--

3.3.1 conjunctionRefinementSet

<p>A conjunction refinement set consists of one or more conjunction operators, each followed by a subRefinement.</p>
--

$conjunctionRefinementSet = 1^*(ws\ conjunction\ ws\ subRefinement)$
--

$conjunctionRefinementSet == seq_1\ subRefinement$

The interpretation of a CONJUNCTIONREFINEMENTSET is the intersection of the interpretation of the SUBREFINEMENTS .

$i_conjunctionRefinementSet :$ $Substrate \rightarrow conjunctionRefinementSet \rightarrow Sctids_or_Error$
$\forall ss : Substrate; conjset : conjunctionRefinementSet \bullet$ $i_conjunctionRefinementSet\ ss\ conjset =$ if $tail\ conjset = \langle \rangle$ then $i_subRefinement\ ss\ (head\ conjset)$ else $intersect\ (i_subRefinement\ ss\ (head\ conjset))\ (i_conjunctionRefinementSet\ ss\ (tail\ conjset))$

3.3.2 disjunctionRefinementSet

disjunctionRefinementSet = 1*(ws disjunction ws subRefinement)

disjunctionRefinementSet == seq₁ *subRefinement*

The interpretation of a **DISJUNCTIONREFINEMENTSET** is the union of the interpretation of the **SUBREFINEMENTS** .

<i>i_disjunctionRefinementSet</i> : <i>Substrate</i> → <i>disjunctionRefinementSet</i> → <i>Sctids_or_Error</i>
$\forall ss : \text{Substrate}; \text{disjset} : \text{disjunctionRefinementSet} \bullet$ <i>i_disjunctionRefinementSet</i> <i>ss</i> <i>disjset</i> = if <i>tail disjset</i> = $\langle \rangle$ then <i>i_subRefinement</i> <i>ss</i> (<i>head disjset</i>) else <i>union</i> (<i>i_subRefinement</i> <i>ss</i> (<i>head disjset</i>)) (<i>i_disjunctionRefinementSet</i> <i>ss</i> (<i>tail disjset</i>))

3.3.3 subRefinement

A *subRefinement* is either an attribute set, an attribute group or a bracketed refinement.

subRefinement = attributeSet / attributeGroup / "(" ws refinement ws ")"

The interpretation of a **SUBREFINEMENT** is the interpretation of the corresponding **ATTRIBUTESET** , **ATTRIBUTEGROUP** OR **REFINEMENT** .

subRefinement ::=

subrefine_attset $\langle\langle$ *attributeSet* $\rangle\rangle$ |

subrefine_attgroup $\langle\langle$ *attributeGroup* $\rangle\rangle$ |

subrefine_refinement $\langle\langle$ *refinement'* $\rangle\rangle$

<i>i_subRefinement</i> : <i>Substrate</i> → <i>subRefinement</i> → <i>Sctids_or_Error</i>
$\forall ss : \text{Substrate}; \text{subrefine} : \text{subRefinement} \bullet$ <i>i_subRefinement</i> <i>ss</i> <i>subrefine</i> = if <i>subrefine</i> ∈ ran <i>subrefine_attset</i> then <i>i_attributeSet</i> <i>ss</i> (<i>subrefine_attset</i> ~ <i>subrefine</i>) else if <i>subrefine</i> ∈ ran <i>subrefine_attgroup</i> then <i>i_attributeGroup</i> <i>ss</i> (<i>subrefine_attgroup</i> ~ <i>subrefine</i>) else <i>i_refinement'</i> <i>ss</i> (<i>subrefine_refinement</i> ~ <i>subrefine</i>)

3.4 attributeSet

An attribute set contains one or more attribute name-value pairs separated by a conjunction or disjunction operator. An attribute set may optionally be placed in brackets.

attributeSet = subAttributeSet ws [conjunctionAttributeSet / disjunctionAttributeSet]

The interpretation of an `ATTRIBUTESET` is one of:

- The intersection of the interpretation of the `SUBATTRIBUTESET` and the `CONJUNCTIONATTRIBUTESET`
- The union of the interpretation of the `SUBATTRIBUTESET` and the `DISJUNCTIONATTRIBUTESET`
- The interpretation of `SUBATTRIBUTESET` if neither conjunction nor disjunction set is present

$attributeSet == subAttributeSet \times conjunctionOrDisjunctionAttributeSet[0..1]$
 $[attributeSet']$

$conjunctionOrDisjunctionAttributeSet ::=$
 $attset_conjattset \langle\langle conjunctionAttributeSet \rangle\rangle \mid$
 $attset_disjattset \langle\langle disjunctionAttributeSet \rangle\rangle$

$i_attributeSet :$
 $Substrate \rightarrow attributeSet \rightarrow Sctids_or_Error$

$\forall ss : Substrate; attset : attributeSet \bullet$
 $i_attributeSet ss attset =$
 $(let\ lhs == i_subAttributeSet ss (first attset); rhs == second attset \bullet$
if $\#rhs = 0$ **then**
 lhs
else if $head\ rhs \in ran\ attset_conjattset$ **then**
 $intersect\ lhs\ (i_conjunctionAttributeSet\ ss\ (attset_conjattset \sim (head\ rhs)))$
else
 $union\ lhs\ (i_disjunctionAttributeSet\ ss\ (attset_conjattset \sim (head\ rhs)))$

$i_attributeSet' :$
 $Substrate \rightarrow attributeSet' \rightarrow Sctids_or_Error$

3.4.1 conjunctionAttributeSet

A conjunction attribute set consists of one or more conjunction operators, each followed by a subAttributeSet.

conjunctionAttributeSet = 1*(ws conjunction ws subAttributeSet)

The interpretation of a CONJUNCTIONATTRIBUTESET is the intersection of the interpretations of its SUBATTRIBUTESETS .

$conjunctionAttributeSet == seq_1 subAttributeSet$

$i_conjunctionAttributeSet :$ $Substrate \rightarrow conjunctionAttributeSet \rightarrow Sctids_or_Error$
$\forall ss : Substrate; conjset : conjunctionAttributeSet \bullet$ $i_conjunctionAttributeSet ss conjset =$ if $tail conjset = \langle \rangle$ then $i_subAttributeSet ss (head conjset)$ else $intersect (i_subAttributeSet ss (head conjset)) (i_conjunctionAttributeSet ss (tail conjset))$

3.4.2 disjunctionAttributeSet

A disjunction attribute set consists of one or more disjunction operators, each followed by a subAttributeSet.

disjunctionAttributeSet = 1*(ws disjunction ws subAttributeSet)

The interpretation of a DISJUNCTIONATTRIBUTESET is the union of the interpretations of its SUBATTRIBUTESETS .

$disjunctionAttributeSet == seq_1 subAttributeSet$

$i_disjunctionAttributeSet :$ $Substrate \rightarrow disjunctionAttributeSet \rightarrow Sctids_or_Error$
$\forall ss : Substrate; disjset : disjunctionAttributeSet \bullet$ $i_disjunctionAttributeSet ss disjset =$ if $tail disjset = \langle \rangle$ then $i_subAttributeSet ss (head disjset)$ else $union (i_subAttributeSet ss (head disjset)) (i_disjunctionAttributeSet ss (tail disjset))$

3.4.3 subAttributeSet

A subAttributeSet is either an attribute or a bracketed attribute set.

`subAttributeSet = attribute / "(" ws attributeSet ws ")"`

The interpretation of a `SUBATTRIBUTESET` is either the interpretation of the `ATTRIBUTE` or the interpretation of the `ATTRIBUTESET`.

$subAttributeSet ::=$
 $subaset_attribute \langle\langle attribute \rangle\rangle \mid$
 $subaset_attset \langle\langle attributeSet' \rangle\rangle$

$i_subAttributeSet :$
 $Substrate \rightarrow subAttributeSet \rightarrow Sctids_or_Error$

$\forall ss : Substrate; subaset : subAttributeSet \bullet$
 $i_subAttributeSet ss subaset =$
if $subaset \in \text{ran } subaset_attribute$ **then**
 $i_attribute ss (subaset_attribute \sim subaset)$
else
 $i_attributeSet' ss (subaset_attset \sim subaset)$

3.5 attribute

An attribute is a name-value pair expressing a single refinement of the containing expression constraint. Either the attribute value must satisfy (or not) the given expression constraint, the attribute value is compared with a given numeric value (integer or decimal) using a numeric comparison operator, or the attribute value must be equal to (or not equal to) the given string value. The attribute may optionally be preceded by a cardinality constraint, a reverse flag and/or an attribute operator.

`attribute = [cardinality ws] [reverseFlag ws] [attributeOperator ws] attributeName ws (expressionComparisonOperator ws expressionConstraintValue / numericComparisonOperator ws numericValue / stringComparisonOperator ws stringValue numericValue = "#" (decimalValue / integerValue))`

Conceptually, the interpretation of `ATTRIBUTE` consists of the following steps:

1. Interpret `ATTRIBUTENAME`

2. Interpret `ATTRIBUTEOPERATOR` against the result of the previous step
3. Interpret one of:
 - The `EXPRESSIONCOMPARISONOPERATOR` applied to the `REVERSEFLAG` , the interpretation of `ATTRIBUTEOPERATOR` and `EXPRESSIONCONSTRAINTVALUE` .
 - The `NUMERICCOMPARISONOPERATOR` applied to the interpretation of `ATTRIBUTEOPERATOR` and `NUMERICVALUE` .
 - The `STRINGCOMPARISONOPERATOR` applied to the interpretation of `ATTRIBUTEOPERATOR` and `STRINGVALUE` .
4. Interpret `CARDINALITY` against the result of the previous step

Note that the `REVERSEFLAG` only applies in the case of `EXPRESSIONCOMPARISONOPERATOR`

. Not sure what to do if it is present in the other two cases...

The schema below provides names the various components of an attribute.

<i>attribute</i>
<i>card</i> : <i>cardinality</i> [0 .. 1]
<i>rf</i> : <i>reverseFlag</i> [0 .. 1]
<i>attOper</i> : <i>constraintOperator</i> [0 .. 1]
<i>name</i> : <i>attributeName</i>
<i>opValue</i> : <i>attributeOperatorValue</i>

attributeOperatorValue ::=
 attrib_expr⟨⟨*expressionComparisonOperator* × *expressionConstraintValue*⟩⟩ |
 attrib_num⟨⟨*numericComparisonOperator* × *numericValue*⟩⟩ |
 attrib_str⟨⟨*stringComparisonOperator* × *stringValue*⟩⟩
numericValue ::= *nv_decimal*⟨⟨*decimalValue*⟩⟩ | *nv_integer*⟨⟨ \mathbb{N} ⟩⟩
 [*reverseFlag*]

$i_attribute :$ $Substrate \rightarrow attribute \rightarrow Sctids_or_Error$
$\forall ss : Substrate; att : attribute \bullet$ $i_attribute\ ss\ att =$ $(let\ att_sctids == i_attributeOperator\ ss\ att.attOper\ (ss.i_attributeName\ att.name) \bullet$ if $att_sctids \in \text{ran } error$ then att_sctids else if $att.opValue \in \text{ran } attrib_expr$ then $(let\ expr_interp ==$ $i_expressionComparisonOperator\ ss\ att.rf\ (result_sctids\ att_sctids)\ (attrib_expr \sim att.opValue) \bullet$ $i_att_cardinality\ ss\ att.card\ expr_interp)$ else if $att.opValue \in \text{ran } attrib_num$ then $(let\ num_interp ==$ $i_numericComparisonOperator\ ss\ att.rf\ (result_sctids\ att_sctids)\ (attrib_num \sim att.opValue) \bullet$ $i_att_cardinality\ ss\ att.card\ num_interp)$ else $(let\ str_interp ==$ $i_stringComparisonOperator\ ss\ att.rf\ (result_sctids\ att_sctids)\ (attrib_str \sim att.opValue) \bullet$ $i_att_cardinality\ ss\ att.card\ str_interp)$

3.5.1 attribute cardinality

The cardinality of an attribute represents a constraint on the minimum and maximum number of instances of the given attribute on each concept. The cardinality is enclosed in square brackets with the minimum cardinality appearing first, followed by two dots and then the maximum cardinality. The minimum cardinality must always be less than or equal to the maximum cardinality. A maximum cardinality of 'many' indicates that there is no limit on the number of times the attribute may appear on each concept.

$cardinality = "[\text{nonNegativeIntegerValue to (nonNegativeIntegerValue / many)}]"$

$unlimitedNat ::= num\langle\mathbb{N}\rangle \mid many$
 $cardinality == \mathbb{N} \times unlimitedNat$

The interpretation of $optional[cardinality]$ in the context of an attribute is:

1. $error$ if there is an error in the supplied list of quads
2. $i_required_cardinality\ 1\ many$ if the cardinality isn't supplied

3. *i_required_cardinality cardinality* if the minimum cardinality is > 0
4. *i_optional_cardinality ss cardinality* if the minimum cardinality is $= 0$

$i_att_cardinality :$ $Substrate \rightarrow cardinality[0..1] \rightarrow Quads_or_Error \rightarrow Sctids_or_Error$
$\forall ss : Substrate; ocard : cardinality[0..1]; gore : Quads_or_Error \bullet$ $i_att_cardinality ss ocard gore =$ $\text{if } gore \in \text{ran } qerror \text{ then}$ $error(qerror \sim gore)$ $\text{else if } \#ocard = 0 \text{ then}$ $ok(i_required_cardinality 1 many gore)$ $\text{else if } first(head ocard) > 0 \text{ then}$ $ok(i_required_cardinality (first(head ocard)) (second(head ocard)) gore)$ else $ok(i_optional_cardinality ss (second(head ocard)) gore)$

i_required_cardinality returns the set of subject or target sctids that meet the cardinality requirements

$i_required_cardinality :$ $\mathbb{N} \rightarrow unlimitedNat \rightarrow Quads_or_Error \rightarrow \mathbb{P} sctId$
$\forall min : \mathbb{N}; max : unlimitedNat; gore : Quads_or_Error \bullet$ $i_required_cardinality min max gore =$ $\text{if } quad_direction gore = source_direction \text{ then}$ $\{subj : \{q : quads_for gore \bullet q.s\} \mid$ $evalCardinality[Quad] min max \{q : quads_for gore \mid q.s = subj\} \neq \emptyset\}$ else $\{targ : \{q : quads_for gore \bullet t_sctid \sim q.t\} \mid$ $evalCardinality[Quad] min max \{q : quads_for gore \mid (t_sctid \sim q.t) = targ\} \neq \emptyset\}$

i_optional_cardinality returns all of the concepts in the substrate that don't fail the cardinality test. The failing concepts are equivalent to the set of all subject (or target) concepts in the set of quads minus the set of concepts that pass.

$i_optional_cardinality :$ $Substrate \rightarrow unlimitedNat \rightarrow Quads_or_Error \rightarrow \mathbb{P} sctId$
$\forall ss : Substrate; max : unlimitedNat; gore : Quads_or_Error \bullet$ $i_optional_cardinality ss max gore =$ $\text{if } quad_direction gore = source_direction \text{ then}$ $ss.concepts \setminus (\{q : quads_for gore \bullet q.s\} \setminus (i_required_cardinality 0 max gore))$ else $ss.concepts \setminus (\{q : quads_for gore \bullet t_sctid \sim q.t\} \setminus (i_required_cardinality 0 max gore))$

3.5.2 expressionConstraintValue

An expression constraint value is either a simple expression constraint, or is a refined or compound expression constraint enclosed in brackets.

`expressionConstraintValue = simpleExpressionConstraint / "(" ws (refinedExpressionConstraint / compoundExpressionConstraint) ws ")"`

The interpretation of `EXPRESSIONCONSTRAINTVALUE` is the interpretation of the corresponding `SIMPLEEXPRESSIONCONSTRAINT`, `REFINEDEXPRESSIONCONSTRAINT` or `COMPOUNDEXPRESSIONCONSTRAINT`

expressionConstraintValue ::=
expression_simple $\langle\langle$ *simpleExpressionConstraint* $\rangle\rangle$ |
expression_refined $\langle\langle$ *refinedExpressionConstraint'* $\rangle\rangle$ |
expression_compound $\langle\langle$ *compoundExpressionConstraint* $\rangle\rangle$

i_expressionConstraintValue :
Substrate \rightarrow *expressionConstraintValue* \rightarrow *Sctids_or_Error*

$\forall ss : \text{Substrate}; \text{ecv} : \text{expressionConstraintValue} \bullet$
i_expressionConstraintValue *ss* *ecv* =
if *ecv* \in *ran* *expression_simple* **then**
 i_simpleExpressionConstraint *ss* (*expression_simple* \sim *ecv*)
else if *ecv* \in *ran* *expression_refined* **then**
 i_refinedExpressionConstraint' *ss* (*expression_refined* \sim *ecv*)
else
 i_compoundExpressionConstraint *ss* (*expression_compound* \sim *ecv*)

3.6 attributeGroup

An attribute group contains a collection of attributes that operate together as part of the refinement of the containing expression constraint. An attribute group may optionally be preceded by a cardinality. An attribute group cardinality indicates the minimum and maximum number of attribute groups that must satisfy the given attributeSet constraint for the expression constraint to be satisfied.

`attributeGroup = [cardinality ws] "{" ws attributeSet ws "}"`

attributeGroup == *cardinality*[0..1] \times *attributeSet*

The difference between the interpretation of an `ATTRIBUTEGROUP` and an `ATTRIBUTESET` is that, within an `ATTRIBUTEGROUP` all of the qualifying `SCIDS` must belong to the same group¹. Outside, a `SCID` would qualify an conjunction if one group passed the first match and a different one the second. *Within* a group, however, conjunctions and disjunctions only count if they both apply in the same group. This means that we have to re-interpret the meaning of `ATTRIBUTESET` on down in terms of *IDGroups* rather than *Quads*

The interpretation of an attribute group is the application of a cardinality constraint to all of the *scIds* that pass the grouped attribute set.

$i_attributeGroup : Substrate \rightarrow attributeGroup \rightarrow Sctids_or_Error$
$\forall ss : Substrate; ag : attributeGroup \bullet$ $i_attributeGroup ss ag =$ $i_group_cardinality ss (first ag) (i_groupAttributeSet ss (second ag))$

3.6.1 attribute group cardinality

The interpretation of `CARDINALITY` within the context of an attribute group is as follows:

- If an error has been encountered anywhere in the process, it should be propagated.
- If the cardinality is omitted, it is interpreted as $[1 \dots *]$
- If the minimum cardinality is $>$ then return the set of *sctids* with that appear at least *min* and at most *max* groups
- If the minimum cardinality is 0 then return every *sctid* that doesn't fail the max cardinality

$i_group_cardinality :$ $Substrate \rightarrow cardinality[0 \dots 1] \rightarrow sctIdGroups_or_Error \rightarrow Sctids_or_Error$
$\forall ss : Substrate; ocard : cardinality[0 \dots 1]; idg : sctIdGroups_or_Error \bullet$ $i_group_cardinality ss ocard idg =$ $\text{if } idg \in \text{ran } gerror \text{ then}$ $error(gerror \sim idg)$ $\text{else (let } gvals == group_value \sim idg \bullet$ $\text{if } \#ocard = 0 \text{ then}$ $ok(i_required_group_cardinality 1 many gvals)$ $\text{else if } first(head ocard) > 0 \text{ then}$ $ok(i_required_group_cardinality (first(head ocard)) (second(head ocard)) gvals)$ else $ok(i_optional_group_cardinality ss (second(head ocard)) gvals)$

¹The zero group is a special case, where each quad in a zero group is viewed as its own unique group

i_required_group_cardinality returns the set *sctids* that pass in at least *min* groups and at most *max* groups

$i_required_group_cardinality :$ $\mathbb{N} \rightarrow unlimitedNat \rightarrow \mathbb{P} sctIdGroup \rightarrow \mathbb{P} sctId$
$\forall min : \mathbb{N}; max : unlimitedNat; groups : \mathbb{P} sctIdGroup \bullet$ $i_required_group_cardinality min max groups =$ $\{sctid : \{s : groups \bullet first s\} \mid$ $evalCardinality[sctIdGroup] min max \{g : groups \mid first g = sctid\} \neq \emptyset\}$

i_optional_group_cardinality returns the set of *sctids* that pass in at most *max* groups, which is equivalent to the set of all possible *sctIds* (i.e. *ss.concepts*) minus the set of *sctids* that pass in more than *max* groups.

$i_optional_group_cardinality :$ $Substrate \rightarrow unlimitedNat \rightarrow \mathbb{P} sctIdGroup \rightarrow \mathbb{P} sctId$
$\forall ss : Substrate; max : unlimitedNat; groups : \mathbb{P} sctIdGroup \bullet$ $i_optional_group_cardinality ss max groups =$ $ss.concepts \setminus$ $(\{s : groups \bullet first s\} \setminus (i_required_group_cardinality 0 max groups))$

3.6.2 attributeSet inside a groups

The interpretation of an **ATTRIBUTESET** a group is the interpretation of the **SUBATTRIBUTESET** optionally intersected or union with additional conjunction or disjunction attribute sets.

$i_groupAttributeSet :$ $Substrate \rightarrow attributeSet \rightarrow sctIdGroups_or_Error$
$\forall ss : Substrate; attset : attributeSet \bullet$ $i_groupAttributeSet ss attset =$ $(\text{let } lhs == i_groupSubAttributeSet ss (first attset); rhs == second attset \bullet$ $\text{if } \#rhs = 0 \text{ then}$ lhs $\text{else if } head rhs \in \text{ran } attset_conjattset \text{ then}$ $gintersect lhs (i_groupConjunctionAttributeSet ss (attset_conjattset \sim (head rhs)))$ else $gunion lhs (i_groupDisjunctionAttributeSet ss (attset_conjattset \sim (head rhs))))$

$i_groupAttributeSet' :$ $Substrate \rightarrow attributeSet' \rightarrow sctIdGroups_or_Error$
--

3.6.3 conjunctionAttributeSet inside a group

Apply the intersect operator to the interpretation of each subAttributeSet in a group context. This is the same as the non-grouped *subAttributeSet* with the exception that the inputs to and result of the function is *sctId* / *groupId* tuples rather than

$i_groupConjunctionAttributeSet :$ $Substrate \rightarrow conjunctionAttributeSet \rightarrow sctIdGroups_or_Error$
$\forall ss : Substrate; conjset : conjunctionAttributeSet \bullet$ $i_groupConjunctionAttributeSet ss conjset =$ $\text{if } tail\ conjset = \langle \rangle \text{ then}$ $i_groupSubAttributeSet ss (head\ conjset)$ else $gintersect\ (i_groupSubAttributeSet\ ss\ (head\ conjset))\ (i_groupConjunctionAttributeSet\ ss\ (tail\ conjset))$

3.6.4 disjunctionAttributeSet inside a group

Apply the union operator to the interpretation of each subAttributeSet in a group context. This is the same as the non-grouped *subAttributeSet* with the exception that the inputs to and result of the function is *sctId* / *groupId* tuples rather than just *sctIds*.

$i_groupDisjunctionAttributeSet :$ $Substrate \rightarrow disjunctionAttributeSet \rightarrow sctIdGroups_or_Error$
$\forall ss : Substrate; disjset : disjunctionAttributeSet \bullet$ $i_groupDisjunctionAttributeSet ss disjset =$ $\text{if } tail\ disjset = \langle \rangle \text{ then}$ $i_groupSubAttributeSet ss (head\ disjset)$ else $gunion\ (i_groupSubAttributeSet\ ss\ (head\ disjset))\ (i_groupDisjunctionAttributeSet\ ss\ (tail\ disjset))$

3.6.5 subAttributeSet inside a group

This is the same as the non-grouped *subAttributeSet* with the exception that the result of the function is *sctId* / *groupId* tuples rather than just *sctIds*.

$i_groupSubAttributeSet :$ $Substrate \rightarrow subAttributeSet \rightarrow sctIdGroups_or_Error$
$\forall ss : Substrate; subaset : subAttributeSet \bullet$ $i_groupSubAttributeSet ss subaset =$ $\text{if } subaset \in \text{ran } subaset_attribute$ $\text{ then } i_groupAttribute\ ss\ (subaset_attribute \sim subaset)$ $\text{ else } i_groupAttributeSet'\ ss\ (subaset_attset \sim subaset)$

3.6.6 attribute inside a group

The interpretation of an attribute within the context of a group is the set of *sctIds* and the group or groups in which they were valid. It is possible for the same *sctId* to valid in the interpretation of a given attribute for more than one group. The union and intersection operators operate on the *sctId* / *groupId* tuples (ne. *sctIdGroup*).

<i>i_groupAttribute</i> : <i>Substrate</i> → <i>attribute</i> → <i>sctIdGroups_or_Error</i>
$\forall ss : \text{Substrate}; att : \text{attribute} \bullet$ <i>i_groupAttribute</i> <i>ss att</i> = (let <i>att_sctids</i> == <i>i_attributeOperator</i> <i>ss att.attOper</i> (<i>ss.i_attributeName att.name</i>) • if <i>att_sctids</i> ∈ ran <i>error</i> then <i>gerror</i> (<i>error</i> ~ <i>att_sctids</i>) else if <i>att.opValue</i> ∈ ran <i>attrib_expr</i> then (let <i>expr_interp</i> == <i>i_expressionComparisonOperator</i> <i>ss att.rf</i> (<i>result_sctids att_sctids</i>) (<i>attrib_expr</i> ~ <i>att.opValue</i>) • <i>i_att_group_cardinality</i> <i>ss att.card</i> <i>expr_interp</i>) else if <i>att.opValue</i> ∈ ran <i>attrib_num</i> then (let <i>num_interp</i> == <i>i_numericComparisonOperator</i> <i>ss att.rf</i> (<i>result_sctids att_sctids</i>) (<i>attrib_num</i> ~ <i>att.opValue</i>) • <i>i_att_group_cardinality</i> <i>ss att.card</i> <i>num_interp</i>) else (let <i>str_interp</i> == <i>i_stringComparisonOperator</i> <i>ss att.rf</i> (<i>result_sctids att_sctids</i>) (<i>attrib_str</i> ~ <i>att.opValue</i>) • <i>i_att_group_cardinality</i> <i>ss att.card</i> <i>str_interp</i>))

3.6.7 cardinality inside a group

Cardinality is interpreted differently within the context of a a group. In particular:

- Zero groups are treated as "singletons" – every quad with a zero group is treated as a group unto itself.
- A *sctid* can pass in the context of more than one group. Intersections, unions, etc. apply to *sctid*'s in the *same* group.

i_att_group_cardinality takes an optional cardinality and a set of quads and returns a list *sctids* and the groups in which they passed the cardinality constraint. The cardinality of each zero group is evaluated separately, but only a single entry is created in the passing list if *any* of the singleton groups passes.

$i_att_group_cardinality :$ $Substrate \rightarrow cardinality[0..1] \rightarrow Quads_or_Error \rightarrow sctIdGroups_or_Error$
$\forall ss : Substrate; ocard : cardinality[0..1]; gore : Quads_or_Error \bullet$ $i_att_group_cardinality ss ocard gore =$ if $gore \in \text{ran } qerror$ then $qerror(qerror \sim gore)$ else if $\#ocard = 0$ then $group_value (i_required_att_group_cardinality 1 many gore)$ else if $first (head ocard) > 0$ then $group_value (i_required_att_group_cardinality (first (head ocard)) (second (head ocard)) gore)$ else $group_value (i_optional_att_group_cardinality ss (second (head ocard)) gore)$

$i_required_att_group_cardinality$ returns the set of subject or target sctids combined with the group(s) in which they qualified

$i_required_att_group_cardinality :$ $\mathbb{N} \rightarrow unlimitedNat \rightarrow Quads_or_Error \rightarrow \mathbb{P} sctIdGroup$
$\forall min : \mathbb{N}; max : unlimitedNat; gore : Quads_or_Error \bullet$ $i_required_att_group_cardinality min max gore =$ if $quad_direction gore = source_direction$ then $\{ subgroup : \{ q : quads_for gore \mid q.g \neq zero_group \bullet (q.s, q.g) \} \mid$ $evalCardinality[Quad] min max \{ q : quads_for gore \mid$ $q.s = first subgroup \wedge q.g = second subgroup \} \neq \emptyset \}$ \cup $\{ q : quads_for gore \mid q.g = zero_group \wedge$ $evalCardinality[Quad] min max \{ q \} \neq \emptyset \bullet (q.s, q.g) \}$ else $\{ targgroup : \{ q : quads_for gore \mid$ $q.g \neq zero_group \wedge q.t \in \text{ran } t_sctid \bullet (t_sctid \sim q.t, q.g) \} \mid$ $evalCardinality[Quad] min max \{ q : quads_for gore \mid$ $t_sctid \sim q.t = first targgroup \wedge$ $q.g = second targgroup \} \neq \emptyset \}$ \cup $\{ q : quads_for gore \mid q.g = zero_group \wedge$ $evalCardinality[Quad] min max \{ q \} \neq \emptyset \bullet (t_sctid \sim q.t, q.g) \}$

$i_optional_att_group_cardinality$ returns the set of subject or target sctids along with all of the corresponding groups that *fail* the cardinality test. The failing list is equivalent to the set of all possible sctid / non-zero groups in the substrate minus the set of sctid / non-zero groups that fail the maximum

cardinality. The set of failures is determined by removing the set of passing tuples from the total possible set in the supplied quads.

$i_optional_att_group_cardinality :$ $Substrate \rightarrow unlimitedNat \rightarrow Quads_or_Error \rightarrow \mathbb{P} \text{ setIdGroup}$
$\forall ss : Substrate; max : unlimitedNat; gore : Quads_or_Error \bullet$ $i_optional_att_group_cardinality ss max gore =$ $\text{if } quad_direction\ gore = source_direction \text{ then}$ $\{rel : ss.relationships \mid rel.g \neq zero_group \bullet (rel.s, rel.g)\} \setminus$ $\{q : quads_for\ gore \mid q.g \neq zero_group \bullet (q.s, q.g)\} \cup$ $i_required_att_group_cardinality\ 0\ max\ gore$ else $\{rel : ss.relationships \mid rel.g \neq zero_group \wedge rel.t \in \text{ran } t_scid \bullet (t_scid \sim rel.t, rel.g)\} \setminus$ $\{q : quads_for\ gore \mid q.g \neq zero_group \bullet (t_scid \sim q.t, q.g)\} \cup$ $i_required_att_group_cardinality\ 0\ max\ gore$

4 Substrate Interpretations

This section defines the interpretations that are realized against the substrate.

4.1 constraintOperator

A constraint operator is either 'descendantOrSelfOf', 'descendantOf', 'ancestorOrSelfOf', or 'ancestorOf'.

constraintOperator = descendantOrSelfOf / descendantOf / ancestorOrSelfOf / ancestorOf

The interpretation of *constraintOperator*[0 .. 1] is one of:

- The input if an operator isn't supplied or if it contains an error
- The union of descendants of all members of the input, as provided by the substrate if the operator is *descendantOf*
- The union of descendants plus the input if the operator is *descendantOrSelfOf*
- The union of ancestors of all members of the input, as provided by the substrate if the operator is *ancestorOf*
- The union of ancestors plus the input if the operator is *ancestorsOrSelfOf*

Note that it is possible for some or all of the members of the input to be included in the *descendantOf* or *ancestorOf* operations as the descendants of one input scid may include another input scid.

The *completeFun* function assures that an operation on any identifier that isn't in the domain of the substrate *descendants* or *ancestors* function returns the empty set (\emptyset).

constraintOperator ::=
descendantOrSelfOf | *descendantOf* | *ancestorOrSelfOf* | *ancestorOf*

<p><i>i_constraintOperator</i> : <i>Substrate</i> \rightarrow <i>constraintOperator</i>[0 .. 1] \rightarrow <i>Sctids_or_Error</i> \rightarrow <i>Sctids_or_Error</i></p> <p><i>completeFun</i> : (<i>sctId</i> \leftrightarrow \mathbb{P} <i>sctId</i>) \rightarrow <i>sctId</i> \rightarrow \mathbb{P} <i>sctId</i></p> <hr/> <p>$\forall ss : \text{Substrate}; oco : \text{constraintOperator}[0 \dots 1]; input : \text{Sctids_or_Error} \bullet$ <i>i_constraintOperator</i> <i>ss</i> <i>oco</i> <i>input</i> = if <i>error</i> \sim <i>input</i> \in <i>ERROR</i> \vee $\#oco = 0$ then <i>input</i> else if <i>head</i> <i>oco</i> = <i>descendantOrSelfOf</i> then <i>ok</i>($\bigcup \{id : \text{result_sctids } input \bullet$ <i>completeFun</i> <i>ss</i>.<i>descendants</i> <i>id</i> $\} \cup \text{result_sctids } input$) else if <i>head</i> <i>oco</i> = <i>descendantOf</i> then <i>ok</i>($\bigcup \{id : \text{result_sctids } input \bullet$ <i>completeFun</i> <i>ss</i>.<i>descendants</i> <i>id</i> $\}$) else if <i>head</i> <i>oco</i> = <i>ancestorOrSelfOf</i> then <i>ok</i>($\bigcup \{id : \text{result_sctids } input \bullet$ <i>completeFun</i> <i>ss</i>.<i>ancestors</i> <i>id</i> $\} \cup \text{result_sctids } input$) else <i>ok</i>($\bigcup \{id : \text{result_sctids } input$ \bullet <i>completeFun</i> <i>ss</i>.<i>ancestors</i> <i>id</i> $\}$)</p> <p>$\forall f : (\text{sctId} \leftrightarrow \mathbb{P} \text{sctId}); id : \text{sctId} \bullet \text{completeFun } f \text{ } id =$ if <i>id</i> \in <i>dom</i> <i>f</i> then <i>f</i> <i>id</i> else \emptyset</p>
--

4.2 attributeOperator

An *attribute operator* indicates that instead of just matching the named attribute with the given attribute value, any descendants of or any descendants or self of the named attribute may match the given attribute value.

attributeOperator = *descendantOrSelfOf* / *descendantOf*

The interpretation of *attributeOperator* is the same as the interpretation of *constraintOperator*

attributeOperator == *constraintOperator*
i_attributeOperator == *i_constraintOperator*

4.3 expressionComparisonOperator

Attributes whose value is a concept may be compared to an expression constraint using either equals ("=") or not equals ("!="). In the full syntax " i_c " and "not =" (case insensitive) are also valid ways to represent not equals.

expressionComparisonOperator = "=" / "!="

The interpretation of `EXPRESSIONCOMPARISONOPERATOR` is the set of all quads in the substrate having a target (or subject if the direction is reverse) that are (or are not) in the interpretation of the `EXPRESSIONCONSTRAINTVALUE`

$expressionComparisonOperator ::= eco_eq \mid eco_neq$

i_expressionComparisonOperator :

$Substrate \rightarrow reverseFlag[0..1] \rightarrow \mathbb{P} sctId \rightarrow$
 $(expressionComparisonOperator \times expressionConstraintValue) \rightarrow Quads_or_Error$

$\forall ss : Substrate; rf : reverseFlag[0..1]; atts : \mathbb{P} sctId;$
 $ec : (expressionComparisonOperator \times expressionConstraintValue) \bullet$
 $i_expressionComparisonOperator\ ss\ rf\ atts\ ec =$
(let $ecv == i_expressionConstraintValue\ ss\ (second\ ec) \bullet$
 if $ecv \in \text{ran } error$ **then**
 $qerror(error \sim ecv)$
 else if $\#reverseFlag = 0 \wedge (first\ ec = eco_eq)$ **then**
 $quad_value(\{q : ss.relationships \mid$
 $q.t \in \text{ran } t_sctid \wedge (t_sctid \sim q.t) \in (result_sctids\ ecv)\}, source_direction)$
 else if $\#reverseFlag > 0 \wedge (first\ ec = eco_eq)$ **then**
 $quad_value(\{q : ss.relationships \mid$
 $q.s \in (result_sctids\ ecv)\}, targets_direction)$
 else if $\#reverseFlag = 0 \wedge (first\ ec = eco_neq)$ **then**
 $quad_value(\{q : ss.relationships \mid$
 $q.t \in \text{ran } t_sctid \wedge (t_sctid \sim q.t) \notin (result_sctids\ ecv)\}, source_direction)$
 else
 $quad_value(\{q : ss.relationships \mid$
 $q.s \notin (result_sctids\ ecv)\}, targets_direction))$

4.4 numericComparisonOperator

Attributes whose value is numeric (i.e. integer or decimal) may be compared to a specific concrete value using a variety of comparison operators, including

equals ("="), less than ("<"), less than or equals ("<="), greater than (">"), greater than or equals (">=") and not equals ("!="). In the full syntax "<>" and "not =" (case insensitive) are also valid ways to represent not equals.

numericComparisonOperator = "=" / "!=" / "<=" / "<" / ">=" / ">"

The interpretation of `NUMERICCOMPARISONOPERATOR` is the set of all quads in the substrate having a *concreteValue* target of type *cv_integer* or *cv_decimal* that (or do not meet) the supplied criteria. As concrete values cannot occur as subjects, this function returns the empty set if the reverse flag is supplied

numericComparisonOperator ::= *nco_eq* | *nco_neq* | *nco_gt* | *nco_ge* | *nco_lt* | *nco_le*

```

i_numericComparisonOperator :
  Substrate → reverseFlag[0..1] →  $\mathbb{P}$  sctId →
  (numericComparisonOperator × numericValue) → Quads_or_Error

∀ ss : Substrate; rf : reverseFlag[0..1]; atts :  $\mathbb{P}$  sctId;
  ncv : (numericComparisonOperator × numericValue) •
  i_numericComparisonOperator ss rf atts ncv =
  if #rf > 0 then
    quad_value(∅, targets_direction)
  else
    quad_value({ q : ss.relationships | q.a ∈ atts ∧ q.t ∈ ran t_concrete ∧
      numericComparison (t_concrete ~ q.t) (first ncv) = (second ncv) }, source_direction)

```

Numeric comparison function. Compares a *concreteValue* with a *numericValue*.

TODO: Need to specify the rules for comparing integers and decimal numbers as well as decimal/decimal. As an example, does "5.00" = "5.0"

```

numericComparison :
  concreteValue → numericComparisonOperator → numericValue

```

4.5 stringComparisonOperator

Attributes whose value is numeric may be compared to an expression constraint using either equals ("=") or not equals ("!="). In the full syntax "*i*" and "not =" (case insensitive) are also valid ways to represent not equals.

$$\text{stringComparisonOperator} = "=" \text{ / } "!="$$

$$\text{stringComparisonOperator} ::= \text{sco_eq} \mid \text{sco_neq}$$

$$\begin{aligned} & i_stringComparisonOperator : \\ & \quad Substrate \rightarrow reverseFlag[0..1] \rightarrow \mathbb{P} \text{ sctId} \rightarrow \\ & \quad (stringComparisonOperator \times stringValue) \rightarrow Quads_or_Error \\ & \forall ss : Substrate; rf : reverseFlag[0..1]; atts : \mathbb{P} \text{ sctId}; \\ & \quad scv : (stringComparisonOperator \times stringValue) \bullet \\ & i_stringComparisonOperator \ ss \ rf \ atts \ scv = \\ & \text{if } \#rf > 0 \text{ then} \\ & \quad quad_value(\emptyset, targets_direction) \\ & \text{else} \\ & \quad quad_value(\{q : ss.relationships \mid q.a \in atts \wedge q.t \in \text{ran } t_concrete \wedge \\ & \quad stringComparison(t_concrete \sim q.t) (first \ scv) = (second \ scv)\}, source_direction) \end{aligned}$$

String comparison function. Compares a *concreteValue* with a *stringValue*.

$$\begin{aligned} & stringComparison : \\ & \quad concreteValue \rightarrow stringComparisonOperator \rightarrow stringValue \end{aligned}$$

4.6 focusConcept

A *focus concept* is a *concept reference* or *wild card*, which is optionally preceded by a *member of function*. A *memberOf* function should be used only when the *conceptReference* refers to a *reference set concept*, or a *wild card* is used. A *wild card* represents any concept in the given substrate. In the brief syntax, a *wildcard* is represented using the “*” symbol. In the full syntax, the text “ANY” (case insensitive) is also allowed.

$$\text{focusConcept} = [\text{memberOf } ws] (\text{conceptReference} \text{ / } \text{wildCard})$$

$$\begin{aligned} crOrWildCard & ::= cr \langle\langle \text{conceptReference} \rangle\rangle \mid wc \\ focusConcept & ::= \\ & \quad focusConcept_m \langle\langle crOrWildCard \rangle\rangle \mid \\ & \quad focusConcept_c \langle\langle crOrWildCard \rangle\rangle \end{aligned}$$

Interpretation: If **MEMBEROF** is present the interpretation of **FOCUSCONCEPT** is union the interpretation of **MEMBEROF** applied to each element in the interpretation of **CONCEPTREFERENCE**. Note that the interpretation of **WILDCARD** is

different in the context of `MEMBEROF` than it is outside. `MEMBEROF WILDCARD` is interpreted as the interpretation of the domain of the *refset* function (i.e. all of the refsets in the substrate), where, without, it is interpreted as *all* of the concepts in the system.

If `MEMBEROF` isn't specified, the interpretation is the substrate interpretations of `CONCEPTREFERENCE` itself

$i_focusConcept : Substrate \rightarrow focusConcept \rightarrow Sctids_or_Error$
$\forall ss : Substrate; fc : focusConcept \bullet$ $i_focusConcept\ ss\ fc =$ $\text{if } focusConcept_c \sim fc \in crOrWildCard \text{ then}$ $\quad \text{if } cr \sim (focusConcept_c \sim fc) \in conceptReference \text{ then}$ $\quad \quad ss.i_conceptReference\ (cr \sim (focusConcept_c \sim fc))$ $\quad \text{else}$ $\quad \quad ok\ ss.concepts$ else $\quad i_memberOf\ ss\ (focusConcept_m \sim fc)$

4.6.1 memberOf

`MEMBEROF` returns the union of the application of the substrate *refset* function to a wild card or the the supplied reference set identifiers. An error is returned if (a) *refsetids* already has an error or (b) the substrate interpretation of a given *refsetId* returns an error.

$i_memberOf : Substrate \rightarrow crOrWildCard \rightarrow Sctids_or_Error$
$\forall ss : Substrate; crorc : crOrWildCard \bullet$ $i_memberOf\ ss\ crorc =$ $(\text{let } refsetids ==$ $\quad \text{if } cr \sim crorc \in conceptReference$ $\quad \text{then}$ $\quad \quad ss.i_conceptReference\ (cr \sim crorc)$ $\quad \text{else}$ $\quad \quad ok\ (ss.descendants\ refset_concept) \bullet$ $\quad \text{if } refsetids \in \text{ran error}$ $\quad \text{then}$ $\quad \quad refsetids$ $\quad \text{else}$ $\quad \quad bigunion\{sctid : result_sctids\ refsetids \bullet ss.i_refsetId\ sctid\})$

5 Types

This section carries various type transformations and error checking functions

5.1 Types

5.1.1 Quads_or_Error

Quads_or_Error is a collection of *Quads* or an error condition. If it is a collection of *Quads*, it also carries a direction indicator that determines whether it is the source or targets that carry the matching elements. Note:

- *source_direction* – matches were performed on attribute/target. Return the source *sctIds*.
- *targets_direction* – matches were performed on attribute/source (**REVERSEFLAG** was present). Return the target *sctIds* or, eventually, concrete values

$$\begin{aligned} direction &::= source_direction \mid targets_direction \\ Quads_or_Error &::= quad_value\langle\mathbb{P} Quad \times direction\rangle \mid qerror\langle ERROR \rangle \end{aligned}$$

5.1.2 sctIdGroups

A *sctIdGroup* is the combination of an *sctId* and a non-zero *groupId*. The set of all possible *sctIdGroups* in a given substrate would be the union of all relationship subjects combined with the non-zero relationship groups in which they appear in the substrate relationships.

sctIdGroups_or_Error represents a set of *sctIdGroups* or an error condition

$$\begin{aligned} sctIdGroup &== sctId \times groupId \\ sctIdGroups_or_Error &::= group_value\langle\mathbb{P} sctIdGroup\rangle \mid \\ &\quad gerror\langle ERROR \rangle \end{aligned}$$

5.2 Result transformations

- **result_sctids** – the set of *sctIds* in *Sctids_or_Error* or the empty set if there is an error
- **quads_for** – the set of quads in a *Quads_or_Error* or an empty set if there is an error
- **quad_direction** – the direction of a *Quads_or_Error* result. Undefined if error

$result_sctids : Sctids_or_Error \rightarrow \mathbb{P} \text{ sctId}$ $quads_for : Quads_or_Error \rightarrow \mathbb{P} \text{ Quad}$ $quad_direction : Quads_or_Error \rightarrow direction$
$\forall r : Sctids_or_Error \bullet result_sctids\ r =$ if $r \in \text{ran } error$ then \emptyset else $ok \sim r$
$\forall q : Quads_or_Error \bullet quads_for\ q =$ if $q \in \text{ran } qerror$ then \emptyset else $first\ (quad_value \sim q)$
$\forall q : Quads_or_Error \bullet quad_direction\ q =$ $second\ (quad_value \sim q)$

Definition of the various functions that are performed on the result type.

- **firstError** – aggregate one or more *Sctids_or_Error* types, at least one of which carries an error and merge them into a single *Sctid_or_Error* instance propagating at least one of the errors (Not fully defined)
- **gfirstError** – convert two *sctIdGroups_or_Error* into a single *sctIdGroups_or_Error* propagating at least one of the errors.
- **union** – return the union of two *Sctids_or_Error* types, propagating errors if they exist, else returning the union of the sctId sets.
- **intersect** – return the intersection of two *Sctids_or_Error* types, propagating errors if they exist, else returning the intersection of the sctId sets.
- **minus** – return the difference of one *Sctids_or_Error* type and a second, propagating errors if they exist, else returning the set of sctId's in the first set that aren't in the second.
- **bigunion** – return the union of a set of *Sctids_or_Error* types, propagating errors if they exist, else returning the union of all of the sctId sets.
- **bigintersect** – return the intersection a set of *Sctids_or_Error* types, propagating errors if they exist, else returning the intersection of all of the sctId sets.

$firstError : \mathbb{P} Sctids_or_Error \rightarrow Sctids_or_Error$ $gfirstError : \mathbb{P} sctIdGroups_or_Error \rightarrow sctIdGroups_or_Error$ $union, intersect, minus : Sctids_or_Error \rightarrow Sctids_or_Error \rightarrow Sctids_or_Error$ $bigunion, bigintersect : \mathbb{P} Sctids_or_Error \rightarrow Sctids_or_Error$ $gunion, gintersect, gminus : sctIdGroups_or_Error \rightarrow sctIdGroups_or_Error \rightarrow sctIdGroups_or_Error$
$\forall x, y : Sctids_or_Error \bullet union\ x\ y =$ if $x \in \text{ran } error \vee y \in \text{ran } error$ then $firstError\ \{x, y\}$ else $ok\ ((ok \sim x) \cup (ok \sim y))$ $\forall x, y : Sctids_or_Error \bullet intersect\ x\ y =$ if $x \in \text{ran } error \vee y \in \text{ran } error$ then $firstError\ \{x, y\}$ else $ok\ ((ok \sim x) \cap (ok \sim y))$ $\forall x, y : Sctids_or_Error \bullet minus\ x\ y =$ if $x \in \text{ran } error \vee y \in \text{ran } error$ then $firstError\ \{x, y\}$ else $ok\ ((ok \sim x) \setminus (ok \sim y))$ $\forall rs : \mathbb{P} Sctids_or_Error \bullet bigunion\ rs =$ if $\exists r : rs \bullet r \in \text{ran } error$ then $firstError\ rs$ else $ok\ (\bigcup \{r : rs \bullet result_sctids\ r\})$ $\forall rs : \mathbb{P} Sctids_or_Error \bullet bigintersect\ rs =$ if $\exists r : rs \bullet r \in \text{ran } error$ then $firstError\ rs$ else $ok\ (\bigcap \{r : rs \bullet result_sctids\ r\})$ $\forall x, y : sctIdGroups_or_Error \bullet gintersect\ x\ y =$ if $x \in \text{ran } gerror \vee y \in \text{ran } gerror$ then $gfirstError\ \{x, y\}$ else $group_value((group_value \sim x) \cap (group_value \sim y))$ $\forall x, y : sctIdGroups_or_Error \bullet gunion\ x\ y =$ if $x \in \text{ran } gerror \vee y \in \text{ran } gerror$ then $gfirstError\ \{x, y\}$ else $group_value((group_value \sim x) \cup (group_value \sim y))$ $\forall x, y : sctIdGroups_or_Error \bullet gminus\ x\ y =$ if $x \in \text{ran } gerror \vee y \in \text{ran } gerror$ then $gfirstError\ \{x, y\}$ else $group_value((group_value \sim x) \setminus (group_value \sim y))$

A Optional elements

Representing optional elements of type T . Representing it as a sequence allows us to determine absence by $\#T = 0$ and the value by $head\ T$.

$$T[0..1] == \{s : seq\ T \mid \#s \leq 1\}$$

B Generic cardinality evaluation

evalCardinality Evaluate the cardinality of an arbitrary set of type T . If the number of elements in the set of T is greater than min and:

- max is a number and the number of elements in T is less than max
- max is *many*

Then return the the set. Otherwise return the empty set.

$[T]$	$evalCardinality : \mathbb{N} \rightarrow unlimitedNat \rightarrow \mathbb{P}\ T \rightarrow \mathbb{P}\ T$
$\forall min : \mathbb{N}; max : unlimitedNat; t : \mathbb{P}\ T \bullet$	$evalCardinality\ min\ max\ t =$
	if $(\#t \geq min) \wedge (max = many \vee \#t \leq (num \sim max))$
	then
	t
	else
	\emptyset

C Generic sequence function

A generic function that takes:

- A substrate
- A function that takes a substrate, a sequence of type T and returns *Sctids_or_Error* (example: *i_subExpressionConstraint*)
- An operator that takes two *Sctids_or_Error* and returns a combination (example: *union*)
- A tuple of the form “ $T \times seq_1\ T$ ”

And returns *Sctids_or_Error*

In the formalization below, *first seq_e* refers to the left hand side of the $T \times seq_1\ T$ and *second seq_e* to the right hand side. *head(second seq_e)* refers to the first element in the sequence and *tail(second seq_e)* refers to the remaining elements in the sequence, which may be empty ($\langle \rangle$).

$[T]$	$ \begin{aligned} & \text{applyToSequence} : \text{Substrate} \rightarrow (\text{Substrate} \rightarrow T \rightarrow \text{Sctids_or_Error}) \rightarrow \\ & \quad (\text{Sctids_or_Error} \rightarrow \text{Sctids_or_Error} \rightarrow \text{Sctids_or_Error}) \rightarrow \\ & \quad (T \times \text{seq}_1 T) \rightarrow \text{Sctids_or_Error} \\ & \forall ss : \text{Substrate}; f : (\text{Substrate} \rightarrow T \rightarrow \text{Sctids_or_Error}); \\ & \quad op : (\text{Sctids_or_Error} \rightarrow \text{Sctids_or_Error} \rightarrow \text{Sctids_or_Error}); \\ & \quad seq_e : (T \times \text{seq}_1 T) \bullet \\ & \text{applyToSequence } ss \ f \ op \ seq_e = \\ & \text{if } tail(second \ seq_e) = \langle \rangle \text{ then} \\ & \quad op \ (f \ ss \ (first \ seq_e)) \ (f \ ss \ (head \ (second \ seq_e))) \\ & \text{else} \\ & \quad op \ (f \ ss \ (first \ seq_e)) \ (\text{applyToSequence } ss \ f \ op \ (head \ (second \ seq_e), tail \ (second \ seq_e))) \end{aligned} $
-------	---