XULE Language Syntax for XBRL - V1

Version 1.0
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>5</td>
</tr>
<tr>
<td>XULE Syntax</td>
<td>5</td>
</tr>
<tr>
<td>Factset Filtering</td>
<td>5</td>
</tr>
<tr>
<td>Taxonomy Navigation</td>
<td>5</td>
</tr>
<tr>
<td>Importance of Types</td>
<td>5</td>
</tr>
<tr>
<td>Concepts, QNames and Local Names</td>
<td>6</td>
</tr>
<tr>
<td>Data Model</td>
<td>6</td>
</tr>
<tr>
<td>XULE Processing Model</td>
<td>6</td>
</tr>
<tr>
<td>Iterations</td>
<td>6</td>
</tr>
<tr>
<td>Evaluating Facts</td>
<td>7</td>
</tr>
<tr>
<td>Factsets</td>
<td>7</td>
</tr>
<tr>
<td>Factset without Dimensions</td>
<td>7</td>
</tr>
<tr>
<td>Factset Filters</td>
<td>7</td>
</tr>
<tr>
<td>Concept Filter</td>
<td>8</td>
</tr>
<tr>
<td>Period Filter</td>
<td>10</td>
</tr>
<tr>
<td>Unit Filter</td>
<td>13</td>
</tr>
<tr>
<td>Entity Filter</td>
<td>13</td>
</tr>
<tr>
<td>Dimension Filter</td>
<td>14</td>
</tr>
<tr>
<td>Aspect Filter Operators</td>
<td>15</td>
</tr>
<tr>
<td>Non-Equivalence (Complement) Operator</td>
<td>16</td>
</tr>
<tr>
<td>In Operator</td>
<td>16</td>
</tr>
<tr>
<td>Combining Aspect Filters</td>
<td>16</td>
</tr>
<tr>
<td>Where Filters</td>
<td>16</td>
</tr>
<tr>
<td>Fact Property Notation</td>
<td>17</td>
</tr>
<tr>
<td>Aspect Alias</td>
<td>18</td>
</tr>
<tr>
<td>Unknown Aspects</td>
<td>19</td>
</tr>
<tr>
<td>Implicit Matching</td>
<td>19</td>
</tr>
<tr>
<td>Covering</td>
<td>20</td>
</tr>
<tr>
<td>Nested Alignment (Alignment Windows)</td>
<td>21</td>
</tr>
<tr>
<td>Nil Values</td>
<td>23</td>
</tr>
<tr>
<td>Nil Values in an Expression</td>
<td>24</td>
</tr>
<tr>
<td>None Values and Iteration</td>
<td>24</td>
</tr>
<tr>
<td>Handling of None</td>
<td>25</td>
</tr>
<tr>
<td>Skipping an Iteration</td>
<td>25</td>
</tr>
<tr>
<td>Factset Grammar Syntax</td>
<td>26</td>
</tr>
<tr>
<td>Defining Factsets as Variables</td>
<td>27</td>
</tr>
<tr>
<td>Cube as a Filter</td>
<td>27</td>
</tr>
</tbody>
</table>
Concept Object
  Concept Equality 50
  Concept Properties 50
Reference Object
  Reference Properties 52
Parts Object 53
Label Object
  Label Object Properties 54
Type Object
  Type Object Properties 54
Cube Object
  Cube Properties 55
Dimension Object 55
Members Object (Not Yet Implemented) 56
Taxonomy (DTS) Object
  Taxonomy Properties 57
Network Set Object 59
Network Object
  Network Properties 59
Role Object
  Role Properties 60
Relationship Object
  Relationship Properties 61

Properties and Functions 62
  Numerical Properties & Functions 62
  String Functions and Properties 63
  Generic Properties 65
  Date Properties & Functions 65
  Aggregation Functions 68
  Statistical Functions 69
  Existence Functions 69
  Unit Functions 70
  DTS Functions 71
  Range Function 71
  Data Functions 72
  Information Functions 73
  Custom Functions 73

Defining Namespaces 74

Assertion Types 74
  Satisfied Types 74
Overview

The XULE syntax is a domain specific language used to define queries and assertions over an XBRL instance or taxonomy. The XULE language described in this document can be used with an Arelle Plugin for processing DQC rules, as well as Altova’s XMLSpy and RaptorXML+XBRL Server. The DQC plugin processes these rules. Rules can be defined to use XBRL instance documents and/or XBRL Taxonomies (including extension taxonomies) as inputs to the rule.

XULE Syntax

The XULE syntax has two distinct components. The first is factset filtering and the second is taxonomy navigation.

Factset Filtering

The first component is factset filtering. A factset is a XULE term that defines the facts in an instance document and their associated decimals, units and contexts. Every fact has aspects associated with it that defines what the fact is, how it is disaggregated and the period at which the fact was measured at or the duration it occurred over. Factset filtering is the action of extracting data from an XBRL instance based on these aspects of the fact. XULE allows a user to put these values into a set, list or dictionary and manipulate the filtered data. Because all of the data is in sets, basic set manipulation can be performed on the data such as a union or intersection or complement. A rule can filter a factset not only using the aspects of a fact, but also the properties of the aspect. For example, this allows a user to return all monetary concepts in an instance, or all fact values with a debit balance. XULE also permits the evaluation of expressions between factsets. For example a set of Liabilities can be deducted from a set of Assets. Obviously, a user of the data would align it before doing such an operation so that Liabilities for 2016 are deducted from Assets for 2016. XULE handles this alignment to ensure that the evaluation of expressions forces the factsets to be aligned. XULE also allows any aspect of a fact to be taken out of alignment or to be aligned. This is covered in detail in this document.

Taxonomy Navigation

XULE provides a syntax that allows the navigation of XBRL networks across many taxonomies. This means XULE can compare relationships between taxonomies by combining taxonomy navigation with set manipulation features. For example, a rule can compare the structure of the company extension taxonomy against the US GAAP taxonomy. The resulting taxonomy relationship sets can then be combined with a factset to determine where values have been used. These navigation features are explained in detail in the Navigation section of this document.

Importance of Types

XULE uses a number of different types and objects. It is important to understand the differences between them to use the language effectively. Any value defined is of a particular type. When defining variables you do not have to define what the types are, the variable will just use the type of the value assigned to it.
Concepts, QNames and Local Names

The relationship between concepts, qnames and local name is an important one. When returning data from an instance or a taxonomy the rule will refer to a concept such as Assets. This concept however can be represented in three different ways.

1. The concept Assets
2. The qname Assets
3. The local name Assets

The concept Assets is an object with many properties. The concept Assets has a label, a definition, a reference, a namespace a balance type a period type etc. The concept Assets is always associated with a taxonomy. All of these properties are available if you have a variable which represents the concept Assets.

The qname Assets represents the namespace and local name of Assets which would be something like us-gaap:Assets. If you get the qname of Assets you can access the namespace and local-name but not its label, balance period type etc. To get the details of a concept you have to look up the concept Assets. This can be done by passing the qname to the function concept(). I.e. taxonomy().concept(Assets) will give you the concept object as described above. This will give you the concept object for the current instance. To get the concept Assets for the us-gaap taxonomy you need to define the following: taxonomy($url_to_US-GAAP).concept(Assets) where the taxonomy function defines the taxonomy to use. If no value is provided then it is the current instance.

The local name assets is just a string with the value “Assets”. This would typically be used in a message. It can be accessed from the concept as follows: $Assets.name.local-name where $Assets is the concept Assets.

Values returned from an instance document maintain their types. A fact value which has a datatype of date for example will have a date type when processed by xule. Fact values are not treated as strings so functions do not have to be applied to elements that have a date type.

Data Model

XULE follows the XBRL data model - defining objects, methods and properties that can be used to understand an XBRL filing or Taxonomy. The model operates at a semantic level - no operations are dependent on any underlying xml, json or csv syntax, although the model does allow access to the syntactic components using functions. This document defines all of the object classes and properties of those classes. An understanding of the XBRL data model is helpful when writing rules in XULE.

XULE Processing Model

Iterations

Factsets and “for loops” create iterations. These can be thought of as methods to make the rule run multiple times. The XULE processing model can run multiple times for a given rule, in some cases returning a result as a message or returning no message. The rule starts with a single iteration, when a factset is encountered iterations are added for each value of the factset.
Iterations are also created when a for loop is encountered. An iteration is created for each loop of the for expression. For example if the value of assets is tested to determine that it is less than zero. If assets is reported for 3 periods the rule will test that assets is less than zero for all three periods or 3 iterations. If assets is also reported in multiple currencies each of these currency disclosures will be tested by the same rule. To do this the processor looks at the number of times a value appears in the instance and executes the rule for each occurrence of the fact. If the fact does not exist in the instance then a single iteration will check for its existence and once determined it does not exist the rule will complete without producing a message.

Evaluating Facts

Factsets

A factset in XULE is used to represent a group of XBRL fact values. The information in a factset not only includes the values of a fact but all the “aspects” (defining characteristics such as units, time period, etc.) of that fact. This allows the properties of a specific fact in a factset to be queried. The factset is a representation of the information in an XBRL instance. The factset contains no information about the Discoverable Taxonomy Set (DTS) associated with the instance. For example, you cannot determine which calculations a fact participates in from the factset and any of the factset operatives. These have to be accessed from the DTS.

The curly bracket notation is used to explicitly state in the XULE language that something is a factset. The @ is used to indicate an aspect and all facts that have this aspect will be returned. The @ can stand by itself as shown in the following examples. Curly brackets are optional.

\{ @ \} or {} or @
This will return all fact values in an instance

Factset without Dimensions

Dimensions are used in XBRL to disaggregate values, for example, the concept “Revenue” can be disaggregated by geography using Western Region and Southern Region. If a subset of facts with no dimensions is required, a square bracket notation can be used to return this subset. The square bracket and curly bracket notation are used independently - each is a container representing a factset.

\[ @ \] or [
This will return all non dimensional fact values in an instance.

Curly brackets are used to return all facts. [ ] Square brackets are used to return only facts with no dimensions.

Factset Filters

The factset body is comprised of filters that allow filtering of facts in an instance. Without filters, all XBRL facts available in the instance would be returned by the Xule processor. The filters work by explicitly stating the aspects that are required. The following are aspect filters can be used:
In addition, the value of a fact can be filtered on using a **where** filter, which is defined in greater detail below. The where filter allows more refined filtering of the factset already filtered by aspects, specifically where a fact value is constrained, such as “**where** the value is less than zero”. The where filter is used here as there is no aspect for the facts numerical value.

To indicate that an aspect filter is being used in a factset, XULE needs to know that it is an aspect. The **@** sign is used to indicate aspects in a factset. For example:

```plaintext
{ @concept = Assets }
```

*This will return all the values of Assets in a given instance.*

**Concept Filter**

As discussed previously, the facts can be restricted by filtering the concept by name as follows:

```plaintext
{ @concept = Assets }
```

This can also be defined in the short form:

```plaintext
{ Assets }
```

The concept filter is a default and is the only filter where the aspect name does not have to be provided. This is a three-fold convenience, as 1) concept is the most widely used aspect filter, 2) concept names can be very long, and 3) this structure improves readability.

The concept aspect can be combined with the following properties to filter the data returned:

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Xule Syntax</th>
<th>Filter by</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>Default or name</td>
<td>QName of fact element. May have multiple QName choices or an expression. Concept can also be filtered by a concept object. \n  <code>{@concept = Assets}</code></td>
</tr>
<tr>
<td><strong>Local Name</strong></td>
<td>local-name</td>
<td>Local name of fact element. \n  <code>{@concept.local-name = ‘Assets’}</code></td>
</tr>
<tr>
<td><strong>Period type</strong></td>
<td>period-type</td>
<td>Concept schema declared period type, for example, instant or duration. \n  <code>{@concept.period-type = instant}</code> \n  <code>{@concept.period-type = duration}</code></td>
</tr>
<tr>
<td>Balance</td>
<td>balance</td>
<td>Concept schema declared balance, for example, credit, debit, <em>, none. (</em> will return all concepts with either a balance of credit or debit)</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.balance = debit}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.balance = credit}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.balance = none}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.balance = *}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom attribute</th>
<th>attribute(name)</th>
<th>Concept schema element declaration custom attribute value, for example, Value, <em>, none (</em> will return all concepts with any attribute value associated with the concept)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{concept.attribute(someattr) = *}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Type</th>
<th>data-type</th>
<th>Concept schema declared data type.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>base-type</td>
<td>Base XBRL type that the data type is derived from.</td>
</tr>
<tr>
<td></td>
<td>data-type-ancestorry</td>
<td>Returns a list of types in order of the type ancestry.</td>
</tr>
<tr>
<td></td>
<td>enumerations</td>
<td>Returns a set of enumerated values</td>
</tr>
<tr>
<td></td>
<td>has enumerations</td>
<td>true or false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.has enumerations = true}</td>
</tr>
<tr>
<td></td>
<td>is-monetary</td>
<td>Values can be true or false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.is-monetary = true}</td>
</tr>
<tr>
<td></td>
<td>is-numeric</td>
<td>Values can be true or false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{concept.is-numeric = true}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Substitution group</th>
<th>substitution</th>
<th>Concept schema declared substitution group.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>substitution-ancestorry</td>
<td>Returns a list of qnames of substitution groups in order of the ancestry.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Namespace</th>
<th>namespace-uri</th>
<th>URI of concept namespace.</th>
</tr>
</thead>
</table>

{concept.period-type = instant concept.balance = debit}

This will return all values in an instance for all concepts that have a period type equal to instant and have a debit balance. Because the statement is enclosed in curly brackets, it will return all values, including
dimensionally qualified values. This will return all the values for Assets, Land, Accounts Receivable Cash, etc., if they exist in the instance document.

{concept.base-type = xbrli:stringItemType @concept.data-type != us-types:zoneStatusItemType}

This will return all values that are string items and are not a zoneStatusItemType.

{concept.is-numeric = true}

This will return all values that are numeric.

{concept.is-monetary = true}

This will return all values that are monetary.

{concept } or { concept = * }

This will return all values in an instance as all facts have a concept. It will take the concept dimension out of alignment. This is discussed later in the document - see Nested Alignment.

The concept filter can also operate on a set of element names using the ‘in’ keyword.

{concept in list(Assets, Land, Cash, AccountsReceivable)}

This will return all values Assets, Land, AccountsReceivable and Cash, if they exist in the instance document.

The concept filter can also operate on a converse set of element names using the ‘not in’ keyword.

{concept not in list(Assets, Land, Cash, AccountsReceivable)}

This will return all values that are not Assets, Land, AccountsReceivable and Cash, if they exist in the instance document.

Period Filter

Facts can be restricted by filtering the period aspect with the following properties:

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Xule Syntax</th>
<th>Filter by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>default</td>
<td>A period expression i.e. forever, date('2016-12-31'), duration('2016-01-01','2016-12-31'), *</td>
</tr>
<tr>
<td>Period start</td>
<td>start</td>
<td>Date and time values to match start for a durational period. If the fact is an instant, the period.end or period.start will return the same date value.</td>
</tr>
<tr>
<td>Period end</td>
<td>end</td>
<td>Date and time values to match end for a durational period. If the fact is an instant, the period.end or period.start will return the same date value.</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Period duration</td>
<td>days</td>
<td>Specify duration of time in days to return values that match that period.</td>
</tr>
<tr>
<td>Period year</td>
<td>year</td>
<td>Year of the end date of a fact. Any fact with a matching year is filtered on. i.e., period.year = &quot;2017&quot;</td>
</tr>
<tr>
<td>Period fiscal year</td>
<td>fiscal-year</td>
<td>Year of the end date of a fact, after being adjusted for fiscal alignment. Fiscal alignment is based on the balance date. If a balance date is not provided, 12/31 is used. Any fact with a matching year is filtered on, i.e., period.fiscal-year = &quot;2017&quot;</td>
</tr>
<tr>
<td>Period is year</td>
<td>is-year</td>
<td>Boolean result of true or false. If true, then only years are filtered on.</td>
</tr>
<tr>
<td>Period is quarter</td>
<td>is-quarter</td>
<td>Boolean values of true or false. If true, then only quarters are filtered on, i.e., period.is-quarter = true</td>
</tr>
<tr>
<td></td>
<td>is-quarter(1..4)</td>
<td>By passing a value of 1 through 4, then that specific quarter will be returned based on a calendar year, for example is-quarter(1) will return a fact for the 1st quarter</td>
</tr>
<tr>
<td></td>
<td>is-fiscal-quarter</td>
<td>Boolean values of true or false. If true, then only fiscal quarters are filtered on, i.e., period.is-fiscal-quarter = true</td>
</tr>
<tr>
<td></td>
<td>is-fiscal-quarter</td>
<td>By passing a value of 1 through 4 then that specific quarter will be returned based on a fiscal year.</td>
</tr>
<tr>
<td>Period is half year</td>
<td>is-half-year</td>
<td>Boolean values of true or false. If true then only half-years are filtered on, i.e., period.is-quarter = true</td>
</tr>
<tr>
<td></td>
<td>is-half-year(1..2)</td>
<td>By passing a value of 1 through 2 then that specific half year will be returned based on a calendar year.</td>
</tr>
<tr>
<td>Period is quarter</td>
<td>is-fiscal-half-year</td>
<td>Boolean values of true or false. If true, then only half-years are filtered on, i.e., period.is-quarter = true</td>
</tr>
<tr>
<td></td>
<td>is-fiscal-half-year(1..2)</td>
<td>By passing a value of 1 through 2, then that specific half year will be returned based on a calendar year.</td>
</tr>
</tbody>
</table>

The following examples show how the period filter can be used:

```
{@period = forever }
This will return all facts with a forever duration.

{@period = date('2016-12-31') }
This will return all fact values with an instant of 2016-12-31

{@period in list(duration('2016-01-01', '2016-12-31'), date('2016-12-31')) } }
This will return all fact values with an instant of 2016-12-31 and duration of 2016-01-01 to 2016-12-31.
```
{@period.start = date('2016-01-01')} }  
This will return all fact values with a start date of 2016-01-01

{@period.days = 90} }  
This return all duration facts that are 90 days in length.

{@period }  
This return all facts in the instance but will mean any calculations done on the result will take the period out of aspect alignment - see Nested Alignment.

{@period.is-year = true}  
This will return all facts representing a calendar year.

{@period.year = '2017'}  
This will return all facts that end in the calendar year 2017.

{@period.year in list('2017','2016','2015')}  
This will return all facts that end in the calendar year 2017, 2016, 2015.

{@period.fiscal-year = '2017'} }  
This will return all facts that end in the fiscal year 2017.

{@period.is-fiscal-quarter(1) = true}  
This will return Returns all fact values that are fiscal first quarter values

{@period.is-quarter = true}  
This will return all fact values that are calendar quarters.

{@period.is-quarter(2) = true}  
This will return all fact values that are calendar second quarter values.

{@period.is-half-year = true}  
This will return all fact values that are in the second half of the calendar year.

{@period.is-half-year(2) = true}  
This will return all fact values that are in the second half of the calendar year.

{@period.is-fiscal-half-year(2) = true}  
This will return all fact values that are in the second half of the fiscal year.

{@period.is-3Q-cum = true}  
This will return all fact values that are a culmination of 3 quarters.

The periods filter also supports automatic type casting where the type is known to be a date type based on the filter request. For example, the following expressions are equivalent:

Period Start:
Period Start:

```
@period.start = date('2016-12-31')  \-With explicit date type cast
@period.start = '2016-12-31'        \-Without type cast
```

Period End:

```
@period.end = date('2016-12-31')  \-With explicit date type cast
@period.end = '2016-12-31'        \-Without type cast
```

Unit Filter

Facts can be restricted by filtering the unit with the following filters:

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Xule Syntax</th>
<th>Filter by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit</td>
<td>default</td>
<td>A unit object. For single measure units the qname of the measure.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For multi-measure units, a combination of units or qnames with '/' and '*'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operators to compose the multi-measure unit.</td>
</tr>
</tbody>
</table>

The following examples show how the unit filter can be used:

```
{@unit = xbrli:pure}
This will return all facts with a unit of pure.
```

```
{@unit = unit(xbrli:pure)}
This will return all facts with a unit of pure. This example is using the unit() function.
```

```
{@unit = unit(iso4217:USD, xbrli:shares)}
This will return all facts with a unit of USD/Share.
```

```
{@unit}
This will return all facts that do not have a unit and take unit out of alignment - see Nested Alignment.
```

```
{@unit = *}
This will return only those facts with a unit, and will take the unit out of aspect alignment - see Nested Alignment.
```

Entity Filter

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Xule Syntax</th>
<th>Filter by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>default</td>
<td>entity(scheme, identifier), *</td>
</tr>
<tr>
<td>Scheme</td>
<td>scheme</td>
<td>Pass the uri of the scheme. I.e. <code>@entity.scheme = 'xxx'</code></td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>Identifier</td>
<td>id</td>
<td>Pass the id of the entity I.e. <code>@entity.id = 'xxx'</code></td>
</tr>
</tbody>
</table>

**Examples**

```plaintext
{@entity.scheme = 'http://www.sec.gov/CIK'}
This will return all facts with an entity using the SEC scheme.

{@entity.id='0000320193'}
This will return all facts with an entity using the identifier 0000320193.

{@entity = *}
This will return all facts for all entities in the instance.
```

**Dimension Filter**

<table>
<thead>
<tr>
<th>Filter Name</th>
<th>Xule Syntax</th>
<th>Filter by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension</td>
<td>(aspect name) Qname</td>
<td>Value of the dimension which is either a qname, a type, *, none</td>
</tr>
<tr>
<td>No additional dimensions</td>
<td>[ ], dimensions() != *</td>
<td>Limits the facts to those that do not have the dimension aspects specified in the factset.</td>
</tr>
</tbody>
</table>

The following examples show how the dimension filter can be used:

```plaintext
{@dei:LegalEntityAxis = *}
This will return all the fact values with the legal entity axis, except for values with no dimensions (default values). It also returns fact values that have other dimensions in addition to the legal entity axis because the filter uses curly brackets.

[@dei:LegalEntityAxis = *]
This will return all the fact values with only the legal entity axis. It does not return default values because the filter uses square brackets. It also will not return fact values that have other dimensions.

{@dei:LegalEntityAxis}
This will return all the fact values including default values in the instance, but takes the legal entity axis out of alignment - see Nested Alignment.
```
[@dei:LegalEntityAxis]
This will return all the fact values with the legal entity axis. It also returns default values. It will not return fact values that also have other dimensions.

[@dei:LegalEntityAxis ! = *] is the same as [@] or []

This will return all values with no dimension aspects.

{@dei:LegalEntityAxis != *}
This will return all facts that do not have a legal entity axis.

{@dei:LegalEntityAxis = none}
This will return all facts that do not have a legal entity axis. This is the same as the above.

If a typed dimension is used and the value of the member on the typed dimension is nil, this is returned if the none keyword is used.

To return typed dimensions with a value of nil requires the use of a where clause.

{where taxonomy().concept(RevenueRemainingPerformanceObligationExpectedTimingOfSatisfactionStartDateAxis) in $fact.dimensions-typed and ($fact.dimensions-typed)[taxonomy().concept(RevenueRemainingPerformanceObligationExpectedTimingOfSatisfactionStartDateAxis)] = none}

In addition, multiple dimensional filters can be defined in a factset.

{@concept = Assets @dei:LegalEntityAxis=* @GeographyAxis in list(NY, CA)}

This will return all facts for Assets that have a legal entity and geography for NY or CA defined.

Aspect Filter Operators
The following operators may be used with any aspect filter.

Equivalence Operator
Filters define equivalence using the = operator. For example, to get a list of facts that are assets, use:

@concept = Assets
Non-Equivalence (Complement) Operator
Filters define a complement using the != operator (“is not”). For example, to get a list of facts that are not assets, use:

```plaintext
@concept != Assets
```

In Operator
Filters define equivalence to an item in a set using the in operator. This works the same as “in” in SQL. The in operator can be used with a single value (like the = operator) or with an array of items.

```plaintext
@concept in list(Assets, Liabilities)
```

This will return the facts defined with the concept assets or liabilities.

Combining Aspect Filters
When used together, aspect filters are separated with a space.

```plaintext
{ @concept = Assets @dei:LegalEntityAxis = * @period = date('2014-12-31') @unit = unit(iso4217:USD) }
```

In the example above, the expression will return those facts with a concept of assets, a year end of 2014 where the asset value is disaggregated by a legal entity, and a value measured in USD.

If the user wanted the variable to include both facts in USD and facts that are in Euro they cannot add another @unit to the expression because a fact cannot be in USD and EURO at the same time. Use a list for multiple units:

```plaintext
{ @concept = Assets @dei:LegalEntityAxis = * @period = date('2014-12-31') @unit in list(iso4217:USD, iso4217:EUR) }
```

The in operator can be followed by a list that must be in parenthesis. This can be done with any of the filters.

There is no restriction on including the same aspect filter multiple times in a factset expression. Each additional filter represents an AND operator and not an OR operator. Although the following expression is valid, it will yield no results:

```plaintext
{ @concept=Assets @concept=Liabilities }
```

This expression returns all those values that are both an asset and a liability. A value can only be one or the other so the results of the expression will always be empty.

Where Filters
The where filter allows filtering by attributes of the fact or any attribute of an aspect. Although it is more effective to filter by the aspect; the where filter allows further refinement of results after all aspect filters have been applied.

---

1 = is used in the same way that “in” would be used in other languages.
The **where** filter can be used to filter facts on variables defined by navigating the DTS of the filing, a base taxonomy, or derived from another factset. Variables defined in other parts of the expression can then be passed into the **where** clause.

The **where** filter can be used to filter facts where a function is applied to filter values that return a boolean result such as `is_base(concept)`. This function indicates if the concept is in a set of predefined namespaces.

To evaluate factset results with the **where** filter, use `$fact` to refer to those results. For example, if we wanted to pull all negative values from a filing, that is expressed as follows:

```xquery
{@ where $fact < 0}
```

Alternatively, the same facts can be pulled without the **where** clause (recall that the single `@` aspect represents all facts in the instance) by using the following:

```xquery
{@} < 0
```

To select all facts where: the unit is not a pure type; is less than zero, and; is accurate to 6 decimal places:

```xquery
{@unit ! = xbrli:pure where $fact < 0 and $fact.decimals == -6}
```

In this case, the **where** clause must be used. In the previous case it was not necessary to use the **where** clause as the fact set could be evaluated in a collapsed form without the **where** clause. This is because the properties of the value (such as decimals) cannot be expressed using an aspect filter.

This example shows the **where** clause used with a boolean function `is_base()`

```xquery
{@unit ! = xbrli:pure where (is_base($fact.concept))}
```

The **where** filter supports boolean and numeric comparison operators. The operators supported by the **where** clause are listed in Appendix 1.

**Fact Property Notation**

The aspects and properties of the `$fact` variable are accessed with dot notation. The properties match the aspect filter names:

<table>
<thead>
<tr>
<th>Aspect &amp; Fact Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fact.concept</td>
<td>Returns the concept of the fact.</td>
</tr>
<tr>
<td>$fact.period</td>
<td>Returns the period of the fact.</td>
</tr>
<tr>
<td>$fact.unit</td>
<td>Returns the unit of the fact.</td>
</tr>
<tr>
<td>$fact.entity</td>
<td>Returns the entity of the fact.</td>
</tr>
</tbody>
</table>
$\text{fact.decimals}$

Returns the decimals of a fact.

$\text{fact.dimension(qname of dimension)}$

Returns the explicit member of the fact as a concept for the specified dimension or returns a value for a typed dimension.

$\text{fact.dimensions()}$

Returns a dictionary of key value pairs of dimension keys and member values.

$\text{fact.dimensions-explicit()}$

$\text{fact.dimensions-typed()}$

Returns the id associated with a fact in the instance. If there is no id associated with the fact then a value of none is returned.

$\text{fact.inline-scale}$

Returns the id associated with a fact in the instance. If there is no id associated with the fact then a value of none is returned.

The following examples show how the property notation can be used:

\[
\begin{align*}
\{ & \text{@concept = Assets @dei:LegalEntityAxis =* where $\text{fact} > 0$ and} \\
& \text{$\text{fact.dimension(dei:LegalEntityAxis).name} == \text{ParentCompanyMember}$} \}
\end{align*}
\]

This will return all facts with the ParentMember and legal entity axis.

\[
\begin{align*}
\{ & \text{@concept = Assets where $\text{fact.dimensions-explicit().values.name.contains(ParentMember)}$} \\
& \text{}}\}
\end{align*}
\]

This will return all facts with the ParentMember without having to know that the fact is on the legal entity dimension.\(^2\)

Aspects Alias

XULE allows the setting of aliases for any aspect. This makes it easier to handle expressions in the where clause as the names can be made shorter and easier to read. Typically, a dimension will be given an alias which represents all members of the dimension. An alias can be defined by using the as expression similar to SQL.

The following expression uses an alias named “$\text{lea}$” to represent a set of members on the legal entity axis.

\[
\begin{align*}
\{ & \text{@concept = Assets @dei:LegalEntityAxis =* as $\text{lea}$} \}
\end{align*}
\]

The alias can be used in the where expression:

\[
\begin{align*}
\{ & \text{@concept = Assets @dei:LegalEntityAxis =* as $\text{lea}$ where $\text{fact} > 0$ and} \\
& \text{$\text{lea} == \text{ParentCompanyMember}$} \}
\end{align*}
\]

---

\(^2\) See Dictionary Operators and Properties dictionary operators later in the document.
This could also be written as follows:

```xule
{@concept = Assets \@dei:LegalEntityAxis as $lea where $fact > 0 and $lea == ParentCompanyMember}
```

This last example may be less efficient with some processors, however, since all asset values are returned - including facts that are not on the legal entity axis - before being excluded by the where clause.

**Unknown Aspects**

Use the `dimensions()` property to return a list of all dimensions and associated members of a fact. If the dimension is known, it can be used as a property of the fact value using the `dimension()` property.

**Examples**

```xule
{@concept = Assets where $fact.dimensions().values.name.contains(ParentMember)}
```

*This will return all facts for assets where the member is called parent member irrespective of the dimension.*

```xule
{@ where $fact.dimension(dei:LegalEntityAxis).name == ParentCompanyMember}
```

*This will return facts with a legal entity axis and a member equal to ParentCompanyMember*

**Implicit Matching**

In the following example a user wants to calculate shareholders equity for two of its legal entities SnapsCo and WidgetsCo by deducting Liabilities from Assets. In this case Widgets Co has assets of $100 and liabilities of $80. SnapsCo has Assets of $80 and Liabilities of $70. The shareholders equity of WidgetsCo should resolve to $20 and $10 for SnapsCo to $10.

In this case we are deducting one factset of values from another factset of values. Using the aspect filters to extract the values we need to do the calculation.

```xule
{@concept = Assets @dei:LegalEntityAxis in list(WidgetsCO, SnapsCO)} -{@concept = Liabilities @dei:LegalEntityAxis in list(WidgetsCO, SnapsCO)}
```

However the filters also determine alignment of facts. If an aspect is not defined it is assumed that a calculation on a set of facts will align. We expect if we have multiple periods we will do the calculation for each period and not across periods. By selecting the legal entities we take them out of alignment.

This will create the following factsets:

1. Assets(WidgetsCO) - Liabilities(WidgetsCo) = She => $20
1. Assets(WidgetsC0) - Liabilities(WidgetsC0) = She(WidgetsC0) => 20
2. Assets(SnapsC0) - Liabilities(SnapsC0) = She(SnapsC0) => 10

When the user actually only wanted this:

1. Assets(WidgetsC0) - Liabilities(WidgetsC0) = She => $30
2. Assets(SnapsC0) - Liabilities(WidgetsC0) = She => $-10
3. Assets(SnapsC0) - Liabilities(SnapsC0) = She => $10

Use double @@ to force alignment in a filter's definition. For example, to calculate shareholders equity for multiple entities, model the following expression:

\[
\{ \text{concept} = \text{Assets} \, \! @ \! \text{@@dei:LegalEntityAxis} = (\text{WidgetsCO}, \text{SnapsCO}) \} - \{ \text{concept} = \text{Liabilities} \, \! @ \! \text{@@dei:LegalEntityAxis} = (\text{WidgetsCO}, \text{SnapsCO}) \}
\]

The double @@ filters on the values and keeps them in aspect alignment. To avoid the Cartesian product the double @@ sign is used on the aspect to be matched. This is the same as an uncovered aspect in XBRL formula.

If an aspect is left out of the factset expression, that aspect defaults to implicit matching and is matched automatically. For example, these two expressions are identical:

\[
\{ \text{concept} = \text{Assets} \} \\
\{ \text{concept} = \text{Assets} \, \! @ \! \text{@@period@@unit@@entity@@dei:LegalEntityAxis} \}
\]

In fact, a @@ expression is unnecessary if the aspect after the @@ does not express a filtered value.

Covering

Covering excludes an aspect from implicit matching when performing an operation on facts as shown in the example above. The syntax @@ is used to align an aspect and the single @ is used to cover (or un-align) an aspect.

In some cases a user may want to unalign or cover all aspects of a fact, but is unaware of what the actual aspects to cover on the fact are. In these cases, the key word covered is used to un-align or cover all aspects of the fact. This is best explained with an example.

Examples

\{ covered @Assets \}
This will return all the values for assets, but will have no observable effect. It will have an impact when an aggregation function such as count() is used. See examples below.

\{ @concept = Assets \, \! < \! 100000 \, \! and \, \{ covered @concept = dei:EntityFilerCategory \} == 'Large Accelerated Filer' \}
This will return a boolean result of true if assets are less than 100,000 and the company is a large accelerated filer. Because assets could have dimensional aspects and be reported for many periods, it would not align with the EntityFilerCategory. The EntityFilerCategory only exists in a single period and
has no dimensional aspects. By covering the entityFilerCategory factset, it can be lined up and evaluated against each value for assets irrespective of its aspects.

\[
\text{count(list({\texttt{covered @concept = Assets}}))}
\]

This will return an aggregate count of all facts using the Assets concept in the instance.

\[
\text{count(list({@concept = Assets})})
\]

This will return a count of 1 for every fact using the Assets concept in the instance. A count of 2 is returned if a fact is duplicated. Aggregation functions only aggregate values with the same dimensions.

In some situations the user wants to cover all the dimensions but leave values in alignment for periods, units or concepts. Normally the dimensions are taken out of alignment with the @. However if you want to cover all dimensions it is not feasible to make a large list of all possible dimensions in the filing. The key word ‘\texttt{covered-dims}’ can be used to cover all dimensions in the returned factset.

\[
\text{exists({\texttt{covered-dims @concept = Assets}}) and {@concept = Liabilities where $fact < 0}}
\]

A variable which represents a factset cannot be covered. The underlying factset defining the variable must be covered.

**Nested Alignment (Alignment Windows)**

In some cases, multiple factsets need to be aligned using different aspects when adding, subtracting, multiplying and dividing. Generally, adjusting alignments is done with @@. However, when addition and subtraction operations are required between factsets with different aspects, then the facts with the same aspect alignment need to be grouped in a nested structure.

For example, a company wants to calculate their net monthly payment for electricity service. The actual net monthly payment is expressed as follows:

\[
{\texttt{@actualMonthlyPayment}} - {\texttt{@actualMonthlyReimbursement}}
\]

This will return the net payment. All periods, units and dimensions align - i.e. the actual reimbursement for the month will be deducted from the actual payment for the same month and not for a different month.

The company also has a contractual monthly rate through an agreement with the electric utility, which is reported with the concept monthly payment. The monthly payment is a fixed amount with a period aspect value of forever. I.e. It is a monthly contract rate that is agreed for the term of the agreement. To determine the monthly difference between what they actually paid and what they were obligated to pay, the company deducts the actual payment and reimbursement from the contracted monthly rate. You might think this could be represented as follows:

\[
{\texttt{@contractedMonthlyPayment}} - {\texttt{@actualMonthlyPayment}} - {\texttt{@actualMonthlyReimbursement}}
\]
This calculation would produce incorrect results because the contracted monthly payment has no period aspect and the actual payment and reimbursement do have a quantifiable monthly period. As a result, the calculation would not be performed. To get the correct value, the calculation needs to be done without regard to the period. To achieve this the period is taken out of alignment for the contracted monthly payment as shown below.

\[ \{\text{contractedMonthlyPayment} \circ \text{period}\} - \{\text{actualMonthlyPayment}\} - \{\text{actualMonthlyReimbursement}\} \]

It's also necessary to take the actual payment and reimbursement out of period alignment so they can be compared with the contracted monthly payment. Furthermore, it's also necessary for the last two elements to be aligned by period with each other, so the monthly payment and reimbursement are subtracted using the same period. To accomplish this, the following relationship is grouped together in parenthesis as a new factset:

\[ \{\{\text{actualMonthlyPayment}\} - \{\text{actualMonthlyReimbursement}\}\} \]

This new group can now be compared to the contracted monthly payment as long as the period is taken out of alignment:

\[ \{\text{period} \{\text{actualMonthlyPayment}\} - \{\text{actualMonthlyReimbursement}\}\} \]

The full calculation is expressed as follows:

\[ \{\text{contractedMonthlyPayment} \circ \text{period}\} - \{\text{period} \{\text{actualMonthlyPayment}\} - \{\text{actualMonthlyReimbursement}\}\} \]

If an instance has a contracted rate of $200 and paid the following amounts:

<table>
<thead>
<tr>
<th>Month</th>
<th>actualMonthlyPayment</th>
<th>actualMonthlyReimbursement</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>210</td>
<td>12</td>
<td>198</td>
</tr>
<tr>
<td>Feb</td>
<td>205</td>
<td>11</td>
<td>194</td>
</tr>
<tr>
<td>Mar</td>
<td>212</td>
<td>10</td>
<td>202</td>
</tr>
<tr>
<td>Apr</td>
<td>210</td>
<td>6</td>
<td>204</td>
</tr>
</tbody>
</table>

Applying the full calculation from above produces the following results:

<table>
<thead>
<tr>
<th>Month</th>
<th>Actual Monthly Payment</th>
<th>Actual Monthly Reimbursement</th>
<th>Netted Payment</th>
<th>Contracted Amount $200</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>210</td>
<td>12</td>
<td>198</td>
<td>200</td>
<td>2</td>
</tr>
<tr>
<td>Feb</td>
<td>205</td>
<td>11</td>
<td>194</td>
<td>200</td>
<td>6</td>
</tr>
</tbody>
</table>
Note that the value of $200 binds with every period even though there is only one value reported of $200. The value of $200 aligns with every period because the `@period` associated with the concept forces it to bind with all other values. In addition, the Netted Payment value for every period must also be taken out of period alignment so it can bind with the contracted amount of $200. The inner factset remains in alignment with the period and the outer factset is not in alignment with the `@period`.

\[
\{\text{@period} \{\text{actualMonthlyPayment}\} - \{\text{actualMonthlyReimbursement}\}\}
\]

The first calculation is nested within the outer factset. Nesting factsets allows aspects to be removed depending on how many aspects need to be removed and the different combinations they have.

In the following example an auditor wants to test that the `DefinedBenefitPlanFundedPercentage` is calculated properly by dividing the Plan assets by the Plan obligations. The assertion is as follows:

\[
\frac{\text{DefinedBenefitPlanFairValueOfPlanAssets}}{\text{DefinedBenefitPlanBenefitObligation}} = \text{DefinedBenefitPlanFundedPercentage}
\]

This has some complications as the plan assets and obligations are measured in monetary units and the funded percentage has no units as it is a percentage or is a pure number. This means the units need to be taken out of alignment but it’s necessary to divide plan assets by plan obligations with the same units. To do this, use the nested structure can be used:

\[
\{\text{@unit}\{\text{DefinedBenefitPlanFairValueOfPlanAssets}\} / \{\text{DefinedBenefitPlanBenefitObligation}\}\}
\]

Now the resulting calculation can be compared to the funded percentage concept of Defined Benefit Plan Funded Percentage.

\[
\{\text{DefinedBenefitPlanFundedPercentage} \text{ @unit}\} != \{\text{unit} \\
\quad \{\text{DefinedBenefitPlanFairValueOfPlanAssets}\} / \{\text{DefinedBenefitPlanBenefitObligation}\}\}
\]

This will return cases where the funded percentage is not the same as the calculated value.

**Nil Values**

XULE by default includes nil values in the factset. A xule processor can be instructed to exclude nil values from the factset when processing is initiated. In addition, a factset can be defined to
exclude nil values by using the keyword nonils. When a value of a fact is returned for a nil item it is returned with a value of “none”.

\{covered nonils @DefinedBenefitPlanFund}nedPercentage\}

This will exclude nil facts from the factset. The “nonils” keyword overrides the processor settings. In addition, a the keyword nils can be set to include nils. This is used if the processor is set to not handle nils.

Nil Values in an Expression
In the case of nonils the processor ignores nil values. However, in some cases nil values may need to be defined in an expression. The XULE processor treats nil values as a ‘none’, and is treated the same way as none would be in an expression. In some cases nil values in the instance need to be treated as if it has a value of zero. If two value are compared such as nil >= 0 this will return a value of none. To treat a nil as a zero value this must be explicitly defined in the expression. This is done using the nildefault keyword.

\{nildefault @Assets} != \{nildefault @LiabilitiesAndStockholdersEquity\}

This expression will treat any nil values returned as if they had a value of zero. In the case above if Assets has a value of nil and Liabilities and equity has a value of zero then the expression will resolve to false. If the nildefault is left off then the expression will result in a boolean of true.

The value returned for nil when the nildefault keyword is used will differ depending on the type of the concept. If the concept is a numeric type then a value of zero is returned. If the concept is a non-numeric then an empty string is returned.

None Values and Iteration
A value of none occurs when a rule returns a value of none. For example a rule that checks equivalence such as the following:

@Liabilities + @Equity != @LiabilitiesAndEquity

If the value of Equity does not exist in the instance the processor will return a value of none. However this does not mean that the rule would terminate with no result. It would continue to evaluate for every instance (iteration) where Liabilities and LiabilitiesAndEquity appear with the same aspects. In this case Equity with a value of none would be treated as if it had a value of 0. If both Liabilities, and Equity were none and LiabilitiesAndEquity had a value then the processor would be skipped as addition and subtraction of nones skips the iteration.

\exists(@LiabilitiesAndEquity) and \exists(@Assets)

In this case the exists(@LiabilitiesAndEquity) resolves to a value of none and Assets exists resolves to true. A value of true combined with a value of none with AND resolves to a skip instruction. This means that the iteration will skip and will return nothing.
Handling of None

If a `none` value is returned and compared to other values then the following rules apply:

**Addition and Subtraction:** A none value can be added or subtracted from another value and is treated as if it has a zero value. (Unless overridden by a specific operator\(^3\)). If both are none then the iteration is skipped.

If the value is none plus a string value the empty none is treated as an empty string. I.e. `None + “hello” = “hello”`.

**Adding and subtracting None and Skip:** The following rules apply:

```
none + skip = skip
skip + none = skip.
```

**Multiplication and Division:** A none value can be multiplied and divided by another value and a none value in these cases will **skip** to the next iteration.

**Greater/Less than:** A comparison to a none value using greater than or less than then will return none when compared to another value.

**Equal:** If both values are None then will return a value of true. If one is a none value and the other is any value then the processor will return a value of false.

**Booleans:** If a None value is included in a boolean then the following occurs.

1. NONE and TRUE: will **skip** the iteration
2. NONE and FALSE: will return FALSE
3. NONE and NONE: will **skip** the iteration
4. NONE or TRUE: will return TRUE
5. NONE or FALSE: will **skip** the iteration
6. NONE or NONE: will **skip** the iteration

**Exists:** The exists function will return a value of true if a set or a list contains only a value of None.

**Skipping an Iteration**

An iteration can be skipped if certain conditions are defined in the rule. This could be done in an if statement for example:

```xquery
if (exists({covered @dei:DocumentType}))
    skip
else
    true
```

This will check if the document type is reported. If it is then the processor will skip the iteration without returning any value.

---

\(^3\) See operator section of this document to control bindings if a value is none.
Factset Grammar Syntax

The following diagram shows the syntax of the factset and the available options.

The factset body is comprised of the following:

**FactsetBody:**

- `covered`  
- `nils`  
- `nonils`  
- `where`  
- `Expr`

The aspect filter has the following options available:

**AspectFilter:**

- `@`  
- `@@`  
- `AspectName`  
- `=`  
- `AspectExpr`  
- `as`  
- `NCname`

An aspect name can be comprised of any or all of the following:
Defining Factsets as Variables

When results in a factset are converted to a variable, it maintains all the properties and attributes of the factset from which it was derived. In an example above, a user calculated shareholders equity by deducting Liabilities from Assets. The shareholders equity variable would contain all the facts for all the periods and dimensions that resulted from the calculation. The variable is defined as follows:

\[
\$she = \{ @concept = \text{Assets} \quad @\text{dei:LegalEntityAxis} = (\text{WidgetsCO}, \text{SnapsCO}) \} - \\
\{ @concept = \text{Liabilities} \quad @\text{dei:LegalEntityAxis} = (\text{WidgetsCO}, \text{SnapsCO}) \};
\]

Variable endings can optionally be expressed with a semicolon for readability.

Cube as a Filter

A factset can be defined by returning the facts based on a set of aspects, and filtering results with a \textit{where} clause. In many filing regimes however, facts are reported using an XBRL hypercube as a template. These cubes establish a multidimensional grid on which facts need to be reported. These cubes can also exclude facts that should not be reported in the cube. Rather than defining all the concepts, axes and members associated with a cube using an aspect filter, a cube filter can be defined to do this. This is defined as follows:

\[
\{ @\text{cube} = \text{taxonomy().cube(StatementTable, BalanceSheet)} \}
\]

\textit{This will return all the values associated with the cube StatementTable. In the US-GAAP Taxonomy, this returns all facts on the face financial statements.}

Cubes can also be specified by the hypercube concept name or drs role of the cube.

\[
\{ @\text{cube.name} = \text{StatementTable} \}
\]

\textit{This will return all the values associated with the any cube named StatementTable. In the US-GAAP Taxonomy, this returns all facts on the face financial statements.}
In the example above, a cube may exist in multiple roles and the facts associated with a cube in a different role may be different. To restrict values returned to those of a specific cube in a specific role then the where clause is used.

Likewise, cubes be specified by the drs role\(^4\).

\[
\{ \text{@cube.drs-role = BalanceSheet} \}
\]

*This will return all the values associated with the any cube in drs role BalanceSheet. In the US-GAAP Taxonomy, this returns all facts on the face financial statements.*

Although @cube looks like an aspect of a factset, ‘cube’ is not an aspect. Using @cube has no effect on the alignment of the returned facts. @cube is only used to filter facts for a factset.

\[
\{ \text{@cube.name = StatementTable @cube.drs-role = BalanceSheet} \}
\]

\[ OR \]

\[
\{ \text{where exists($fact.cube(StatementTable, BalanceSheet))} \}
\]

*This will return all the values associated with the cube StatementTable in the balance sheet extended link.*

\[
\{ \text{@cube != none} \}
\]

*This will return all the values associated with any cube.*

\[
\{ \text{@cube = none} \}
\]

*This will return all the values not in a cube.*

### Navigation

Navigation is used to traverse the relationships in a taxonomy. A navigation returns a set. The items in the set are determined by what is provided in the navigation. In its simplest form, navigation requires a direction.

**navigate descendants**

*This will return all the descendent concepts across all networks in the instance taxonomy.*

**navigate parent-child descendants**

*This will return all descendant concepts in the presentation parent-child relationships of the instance taxonomy.*

### Arcrole

The navigation can be limited to specified arcroles.

---

\(^4\) See [DRS Role](#).
navigate parent-child descendants
This will return all the descendant concepts in the presentation parent-child relationships in the instance taxonomy.

The arcrole is specified using the last path component of the arcrole uri or the full uri of the arcrole as a string. These two navigations operations are equivalent:

- navigate parent-child descendants

The allowable arcroles include those defined in XBRL specifications and any arcrole defined in the taxonomy currently being navigated.

When the last path component of the arcrole is used, the last path component must be unique within the taxonomy.

Direction
The direction indicates the path of the navigation. The allowable directions are:
- descendants
- children
- ancestors
- parents
- siblings
- previous-siblings
- following-siblings

For descendants and ancestors, the number of levels to navigate can be specified after the direction.

navigate parent-child descendants 2
This will navigate to the grand children of the root concepts.

When no level is specified, the navigation will traverse to all levels.

By default, navigation returns the target concepts of the relationships exclusive of the starting concepts in the returned list. To include starting concepts, use ‘include start’ after the direction.

navigate parent-child descendants include start
This will return all concepts in all parent-child relationships, including the root concepts.

navigate parent-child descendants 2 include start
This will return the root, child and grandchild concepts.

Role
An extended link role may be specified by the keyword ‘role’ followed by the role. The role is specified with the last component path. When the last path component of the roles used, the last path component must be unique within the taxonomy. These two operations are equivalent:
• **navigate** parent-child descendants role
  `http://www.abc.com/role/ConsolidatedBalanceSheets`

**OR**

• **navigate** parent-child descendants role ConsolidatedBalanceSheets

This will return all the target concepts in the presentation of the balance sheet only. Note that a short name can be used. This should not use quotes.

Any role defined in the taxonomy can be referenced including roles defined in a generic graph. The standard roles that can be used are as follows:

- parent-child
- summation-item
- dimension-domain
- dimension-default
- domain-member
- hypercube-dimension
- all
- general-special
- essence-alias

In addition to these roles there are custom arc especially for navigating dimension described in the section on navigating dimensions.

**Starting and Ending Navigation**

By default, the navigation starts at the roots. To start at a particular concept, add the keyword `from` and the concept or concept name.

```plaintext
navigate parent-child descendants 2 from Assets
```

This will return all the child and grandchild concepts starting from Assets.

An ending concept may also be specified by adding the keyword `to`.

```plaintext
navigate parent-child descendants from Assets to OtherAssetsCurrent
```

This will return all the concepts that are descendants of Assets but will stop at OtherAssetsCurrent. The results will only include those concepts that are in the path between Assets and OtherAssetsCurrent. Concepts in paths that do not end with OtherAssetsCurrent will not be included in the result.

If an ending concept is provided with the keyword `to` then the relationships between the starting concept or root concept to the `to` concept will be returned. If a tree or graph is navigated and the `to` concept is never reached then no results will be returned.

**Stopping Navigation**

Navigation of a tree or graph can be stopped when certain conditions on the relationship are encountered. This is done using the keywords `stop when` followed by an expression that
resolves to true. The expression will evaluate an attribute of the relationship such as when the
target-name equals a certain concept or the weight is negative one. This enables you to
navigate a tree and stop navigating down a given branch when a condition is met on the
relationship. The ‘stop when’ keyword differs from the ‘to’ keyword in that all relationships that
evaluate to false will be returned. Using the ‘to’ concept will only return relationships if the ‘to’
concept actually exists in the tree or graph.

The relationship that evaluates to true when using ‘stop when’ will be returned as part of the
result. For example, if you navigate a calculation tree and stop navigation when you reach the
target concept “Net Income”, the relationship with the target concept of “Net Income” will be
returned. This relationship could then be removed using the ‘where’ clause discussed below.

navigate parent-child descendants from IncomeStatementAbstract stop
when $relationship.target.name == GrossProfit where
$relationship.target.name != GrossProfit returns target

This will return all the descendants concepts of IncomeStatementAbstract in the presentation linkbase of
the filing, but will exclude any children of GrossProfit and because of the where clause will also exclude
the concept GrossProfit.

Taxonomy

By default, navigation occurs in the taxonomy of the instance document. A different taxonomy
may be specified by using the keyword ‘taxonomy’ followed by the taxonomy.

navigate parent-child descendants from Assets taxonomy
taxonomy(‘http://xbrl.fasb.org/us-gaap/2016/entire/us-gaap-entryPoint-
std-2016-01-31.xsd’)

This will return all the descendants of Assets in the US GAAP taxonomy.

Filtering Results

Navigation ultimately returns relationship information based on the relationship found during the
traversal. A ‘where’ expression can be used to filter the found relationships. For each found
relationship, the ‘where’ expression is evaluated. When the result of evaluating the ‘where’
expression for a relationship is true, the relationship is included in the result.

A special variable $relationship is available in the ‘where’ expression to refer to the relationship
being filtered.

navigate parent-child descendants from Assets where not
$relationship.target.is-abstract

This will return all non-abstract descendants of Assets.
Return Options

The default result of navigation returns a set of the target concepts of the relationships that are found in the navigation.

The 'returns' keyword can be used to specify additional components of the relationship to return.

```
navigate parent-child descendants from Assets returns (source)
```

This will return a list of the source concepts of the relationships.

The components that may be returned are:

- **source**
  The source **concept** of the relationship

- **source-name**
  The **QName** of the source concept of the relationship

- **target**
  The target **concept** of the relationship

- **target-name**
  The **QName** of the target concept of the relationship

- **order**
  The value of the **order** attribute on the relationship

- **weight**
  The value of the **weight** attribute on the relationship

- **preferred-label**
  The label object for the label indicated by the preferred label for the target concept. This includes uri, description, lang and text properties..

- **preferred-label-role**
  The role object for the preferred label. This includes the uri, description and used on properties.

- **relationship**
  The **relationship**

- **role**
  The extended link **role** of the network

- **role-uri**
  The extended link **role uri** of the network

- **role-description**
  The **description** of the role of the network

- **arcrole**
  The **arcrole** of the network

- **arcrole-uri**
  The **arcrole uri** of the network

- **arcrole-description**
  The **description** of the arcrole of the network

- **arcrole-cycles-allowed**
  The cycles allowed attribute of the arcrole definition. One of: 'any', 'undirected', 'none'

- **link-name**
  The **QName** of the extended link element

- **arc-name**
  The **QName** of the arc element

- **network**
  The network
- **cycle** An indicator if the relationship starts a cycle in the navigation
- **navigation-order** The calculated sibling order of the relationship target. This is not the order on the relationship but is calculated during the navigation
- **navigation-depth** The depth of the relationship target concept from the starting concept
- **result-order** The order of the result within the full result list
- **arc attribute** Specified by the QName of the attribute. The value of the arc attribute. Unlike the other return components, this is not a keyword “arc attribute”, but the actual qname of the arc attribute is used in the “returns” statement. For example:
  ```
  returns (source-name, target-name, ex:specialAttribute)
  “ex:specialAttribute” is the qname of the attribute on the arc.
  ```
- **dimension-type** The purpose of the target concept in dimensional navigation. See Dimensional Navigation.
- **dimension-sub-type** The more specific purpose of the target concept in dimensional navigation. See Dimensional Navigation.
- **drs-role** The initial role of the dimensional relationship set. See Dimensional Navigation.

Multiple components may be returned by composing them in a list. In this case the navigation will return a list of lists when executed. The inner list will be the values corresponding to the specified components.

```
navigate parent-child descendants from Assets returns (target, preferred-label)
```

*This will return a list of the relationships. Each item in the list will be a list with two values, the target concept and the preferred label for the relationship.*

The ‘**include start**’ keyword creates an extra result for each starting concept returned as the target. This affects the way the following return components are returned:

- **source** Returned as None
- **source-name** Returned as None
- **target** The start concept
- **target-name** The QName of the start concept
- **order** Returned as None
- **weight** Returned as None
- **preferred-label** Returned as None
- relationship Returned as None
- cycle False
- navigation-order The calculated sibling order of the relationship target. This is not the order on the relationship but is calculated during the navigation
- navigation-depth 0
- result-order The order of the result within the full result list
- arc-attribute Returned as None
- dimension-type The dimension type of the start concept
- dimension-sub-type The dimension subtype of the start concept

Returning a dictionary

The default result value is a list. When multiple return components are returned, it can be more useful to have the result returned as a dictionary. The `as dictionary` keyword is used to structure returned results as key-value pairs.

```plaintext
navigate parent-child descendants from Assets returns (source, target, role) as dictionary
```

This will return a list of dictionaries. Each dictionary will have three entries with the keys of “source”, “target” and “role”.

<table>
<thead>
<tr>
<th>Dictionary 1</th>
<th>Dictionary 2</th>
<th>Dictionary 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Value</td>
<td>Key</td>
</tr>
<tr>
<td>source</td>
<td>Assets</td>
<td>source</td>
</tr>
<tr>
<td>target</td>
<td>AssetsCurrent</td>
<td>target</td>
</tr>
<tr>
<td>role</td>
<td>balanceSheet</td>
<td>role</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dictionary 4</th>
<th>Dictionary 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>Value</td>
</tr>
<tr>
<td>source</td>
<td>AssetCurrent</td>
</tr>
<tr>
<td>target</td>
<td>Cash</td>
</tr>
<tr>
<td>role</td>
<td>balanceSheet</td>
</tr>
</tbody>
</table>
Returning a list - duplicate results

Navigation normally returns the results in a set or a list. Since sets cannot contain duplicates, the navigation result is deduplicated. To include duplicate values in the return, use the 'list' keyword.

```
navigate parent-child descendants from Assets returns list
This will return a list target concepts that are descendant from Assets. If a concept is in more than one branch, it will be included multiple times in the result.
```

The order of the result is based on depth first traversal of the navigation. The order of the starting concepts is undefined, but is deterministic. The order of siblings is determined by the order attribute of the sibling relationships. When sibling relationships have the same order, then the order is undefined, but is deterministic.

Returning networks

The normal result of navigation is a flat list. The results can be organized by networks by using the 'networks' keyword.

```
navigate parent-child descendants returns by network

This will return the following dictionary of lists:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network One</td>
<td>(B, C)</td>
</tr>
<tr>
<td>Network Two</td>
<td>(Y, Z)</td>
</tr>
</tbody>
</table>
```

```
navigate parent-child descendants include start returns by network (target, role) as dictionary
This will return a dictionary of networks. The value for each network is a list of dictionaries of each relationship.
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network One</td>
<td>List of dictionaries</td>
</tr>
<tr>
<td></td>
<td>Dictionary 1</td>
</tr>
<tr>
<td></td>
<td>Dictionary 2</td>
</tr>
<tr>
<td></td>
<td>Key</td>
</tr>
</tbody>
</table>
$a = \text{navigate parent-child descendants include start returns by network (target, role) as dictionary}
$Network_uri = \text{set(for \$network in a.keys())}
\begin{array}{ll}
\text{if} (a[$network][1][‘target’] ) == “B” )
& a[$network][1][‘role’].uri \text{ OR } a[$network].role.uri
\text{else}
& \text{none}
\end{array}
This will return all the role URI’s that include the concept B as a target.

Returning paths
Default navigation returns a list of results. Alternatively, the results can be organized by the path of the navigation. This is indicated by the keyword ‘paths’. A path result uses a double list. The outer list contains a list for each path of traversal. The inner list contains each result in the order of the traversal.

\begin{itemize}
\item \text{navigate parent-child descendants include start from A returns paths}
\end{itemize}
This will return the following lists:
- \( (A,B,D) \)
- \( (A,B,E) \)
- \( (A,C,F) \)
- \( (A,C,G) \)
navigate parent-child descendants include start from A returns paths (source, target, order)
This will return the following lists:
- ((None, A, None),(A,B,1),(B,D,1))
- ((None, A, None),(A,B,1),(B,E,2))
- ((None, A, None),(A,C,2),(C,F,1))
- ((None, A, None),(A,C,2),(C,G,2))

The following example demonstrates how paths can be used to calculate the effective weight between two elements.

```
product(navigate summation-item descendants include start from ProfitLoss to Revenues returns paths (weight))
```

Dimensional Navigation

Dimension navigation is used to navigate dimensional relationship sets (DRS). The DRS includes relationships from multiple arc roles and extended link roles to form a model of the cubes defined in the taxonomy. The following diagram shows how the dimension arc roles are composed into the DRS.

Dimensional navigation will traverse the multiple arc roles that make up a DRS. For the purpose of dimensional navigation, the hypercube is the root of the structure. This is slightly different than the standard XBRL dimension model which treats the primary item as the root of the structure. Note that the ‘all’ arc role in the XBRL dimension model is from the primary item to the hypercube. In dimensional navigation, this is flipped, and the all arc role is treated as from the hypercube to the primary item.
To indicate dimension navigation, add the ‘dimensions’ keyword.

```
navigate dimensions descendants from dei:LegalEntityAxis
This will return all the dimension members of the Legal Entity Axis (dimension). If the
dei:LegalEntityAxis dimension is used in multiple cubes with a different set of members, this will
traverse each version of the dimension.
```

Navigation can be constrained to a single cube by using the ‘cube’ keyword.

```
navigate dimensions descendants from dei:LegalEntityAxis cube us-gaap:StatementTable
This will return all the dimension members of the Legal Entity Axis (dimension) in the us-
gaap:StatementTable.
```

Arc roles can be used to limit the results of the navigation to only those relationships with the
specified arc role.

```
navigate dimensions dimension-domain descendants from us-
gaap:StatementTable
This returns the dimension and the domain concepts that are in the us-gaap:StatementTable.
```

**Pseudo arc roles**
In addition to the arc roles that are used to define dimensional relationships, these additional
pseudo arc roles can be used:

<table>
<thead>
<tr>
<th>pseudo arc role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypercube-primary</td>
<td>Relationships between a cube concept and the primary item concept. Similar to the ‘all’ arc role to define a relationship between a primary item and a cube, but in opposite direction.</td>
</tr>
<tr>
<td>dimension-member</td>
<td>Domain-member relationships that stem from a dimension concept via a dimension-domain relationship. Only include member concepts that are members of a dimension (not a primary item).</td>
</tr>
<tr>
<td>primary-member</td>
<td>Domain-member relationships that stem from the primary concept of a cube. Only include member concepts that are members of a primary item (not a dimension).</td>
</tr>
</tbody>
</table>

Any of the standard dimension arc roles, except for all and not-all, may be used for dimensional navigation. In dimensional navigation, the domain-member arc role applies to both members of a dimension and the members of the primary item.

When using an arc role or pseudo arc role the navigation will still traverse the DRS from the
starting concepts regardless of the arc role specified. Only the relationships from the specified arc role will be returned.
navigate dimensions dimension-member descendants from us-gaap:StatementTable
This will return all the members of any dimension of the us-gaap:StatementTable. Note that the traversal starts from the hypercube concept on the hypercube-dimension relationship and then to the domain members via the dimension-domain relationships. Only the domain-member relationships are returned.

DRS role
Dimensional navigation can traverse more than one extended link role. The original role used on the 'all' relationships between the primary item concept and the hypercube concept is the drs-role. This remains the same when navigating a DRS even though the extended link role may change.

To limit dimensional navigation to only one drs role the 'drs-role' keyword can be used.

navigate dimensions descendants from us-gaap:StatementTable drs-role BalanceSheet
This returns the dimension and the domain concepts that are in the us-gaap:StatementTable.

Like extended link roles, the drs-role can be specified using the uri of the role in quotes or just the last path component. When the last path component of the drs-role is used, the last path component must be unique within the taxonomy, otherwise it is an error.

Dimension return components
In addition to the non-dimensional navigation return components, dimensional navigation can use the following return components:

<table>
<thead>
<tr>
<th>Return component</th>
<th>Description</th>
</tr>
</thead>
</table>
| dimension-type   | The dimension-type return component identifies the dimensional purpose of the target concept. The values are:  
  - hypercube  
  - primary-member  
  - dimension  
  - dimension-member |
| dimension-sub-type | The dimension-sub-type return component identifies the specific dimensional purpose for dimensions and members. The values are:  
  For dimension-type = dimension  
    - explicit - explicit dimension concepts  
    - typed - typed dimension concepts  
  For dimension-type = dimension-member  
    - default - the default member of an explicit dimension  
  For dimension-type = primary-member  
    - primary - the primary item |
| **drs-role** | The extended link role of the primary to hypercube relationship ('all' relationship). In a dimensional relationship set, the role can be different on different relationships. The drs-role is constant for the all the relationships that make up the dimensional model of a cube. |
| **usable** | For members, identifies the value of the usable attribute. If there is no usable attribute it defaults to true. |

**Alternative to dimensional navigation**

Beside dimensional navigation, dimensional information can be accessed with dimension functions.

- Return the dimensions associated with a given member: `dimensions($member)`
- Return the members on a dimension on a role member: `member($dimension)`
- Return all the dimensions in a taxonomy: `taxonomy().dimensions`
- Return all the hypercubes in a taxonomy: `taxonomy().cubes`

**Navigation Expression**

```
navigate dimensions {arc role} {direction} {levels} include start from {starting concepts} to {ending concepts} stop when {expression} role {roles} drs-role {DRS roles} linkbase {linkbase element name} cube {hypercube concept} taxonomy {taxonomy} where {where expression} returns by network list |set paths set {return components} as list|dictionary
```

Note, the only required component of navigation is the direction.

```
navigate descendants
```

*This will return all the concepts that participate in any relationship, excluding the root concepts.*
This will return all the concepts that participate in any relationship.

Filtering Collections

The filter expression is used to filter a collection such as a set, or a list. A filter returns a set or a list depending on the collection type passed to it. The items in the collection are determined by what is provided in the filter expression. In its most simplest form the filter expression requires a collection to filter.

\[
\text{filter}\ a
\]

This returns all the items in the set.

The filter expression uses a \texttt{where} clause to filter the list or set on a condition. The \texttt{where} clause uses the \texttt{$item} variable to represent the current value in the set.

\[
\text{filter}\ \text{set}(1,2,3)\ \text{where}\ \$item > 1
\]

This returns a set of (2,3)

The values returned by the filter can also be defined. For example if a set of of qname concepts is in a set these can be turned into local names using the filter expression:

\[
\text{filter}\ \$\text{networkQname}\ \text{returns}\ \$item.\text{local-name}
\]

This returns a set of local names from the set of qnames defined in the variable \texttt{$\text{networkQname}}.

This alleviates the need to create a for loop to iterate through the set. The syntax of the filter is as follows:

\[
\text{Filter:}
\]

The returns expression can also incorporate strings to build up text based output.

\[
(\text{filter}\ \$\text{sub-periods}\ \text{returns}\ "\t" + \$item.\text{period.string} + "\t" + \$item.\text{string}).\text{join}("\n")
\]

This returns a string value showing the period of the item and the item separated by tabs and ended with a carriage return.
Conditional and Iterative Statements

Iterative Statements (Loops)

In XULE, a for loop can be used to iterate through a set of values. There are a number of XULE objects that need to use the for loop to access values. These include the following:

- Relationships
- References
- Reference Parts
- Labels
- Networks
- Concepts
- Or any other set or list.

The for loop has the following structure:

```
for (variable in set)
    Repetend
```

Where repetend is the thing to be repeated. For loops in XULE do not define how many times the loop should execute. The loop will run until it gets to the end of the set or list. The for loop has parentheses to indicate the set or object to loop through. The set can be entered as a variable or as an expression. For example

```
for ($c1 in taxonomy().concepts)
    $c1.name
```

This for loop evaluates the taxonomy().concepts to a list of all concepts in the taxonomy. The for loop then iterates through each concept and returns the qname of each concept in the taxonomy. The loop will be repeated for every concept in the taxonomy.

The results of a for loop can also be returned as another set by defining a variable. For example

```
$string_name_of_concepts = set(for ($c1 in taxonomy().concepts)
    $c1.name.local-name)
```

This for loop evaluates the taxonomy().concepts to a list of all concepts in the taxonomy. The for loop then iterates through each concept and returns the local name of each concept in the taxonomy and adds it to a set called $string_name_of_concepts.

Conditional Statements (If-else statements)

The if-else statement has the form

```
if ( <condition> ) <statement1> else <statement2>
```

The <condition> is a boolean expression. Both the <statement1> and <statement 2> need to be included in the if-else statement.
If the else statement is to do nothing then the term none should be used. The else statement is required to indicate the end of the if condition.

The condition must be encapsulated in parentheses. The statement blocks should not be encapsulated with curly brackets.

Typically an if statement is going to be included in a for loop. As the for loop iterates through a set of values the if statement is used to filter items out of the set based on the if condition. In a factset filter the where clause is used to perform the same function.

The following example shows an if statement not in a for loop

```xule
if (@dei:AmendmentFlag == true and count(list({covered @dei:AmendmentDescription})) == 0))
    true
else
    false
```

This statement returns a true condition if the amendment flag in the default is set to true and there is no amendment description anywhere in the filing. If either of these conditions are false then the else statement returns false.

**Setting Variables**

Variables are defined in XULE by defining a variable using the $ symbol. Values are assigned to the variable using a single equals “=”.

To define a variable of “a” with a value of 10 the following syntax is used:

```xule
$a = 10
```

This assigns a value of 10 to the variable $a.

All variables when defined and when used must have a $ sign to indicate that they are a variable. The type of the variable is not defined. A variable will inherit the type of the value assigned to it.

**Order of Evaluation**

The same variable can be defined multiple times. For example:

```xule
$a = 10;
$a = 20;
```

This assigns a value of 10 to the variable $a and subsequently assigns a value of 20 to a second variable called $a. If you output the value of $a a value of 20 is returned.

Variables can be set in a number of ways and it may not be clear which value is assigned to a variable with the same name. Variables can be defined by a constant, as an argument to a function or direct assignment to a variable.

The following example explains. The following function called test() is defined:
function test($a)
    $a = 30;
    $a

This function will always return a value of 30.

A constant of $a is defined as part of the rules with a value of 40

constant $a = 40;

The test function is called and is passed a value of 20.

    $a = test(20) + $a

The resulting value of $a will be 70. (30 + 40)

Constants can be used in a function but the value will be superseded by the argument to the function and this will be superseded by direct variable assignment in the function.

If the test function was defined as follows:

function test($a)
    $a

The resulting value of $a in test(20) + $a will be 60. (20 + 40)

If the test function was defined as follows:

function test($b)
    $a

The resulting value of $a in test(20) + $a will be 80. (40 + 40)

Collections (Sets, Lists and Dictionaries)

A collection is a generic term used to refer to sets, lists and dictionaries. Each of which facilitates the collection of data. XULE supports sets, lists and dictionaries and operations between them.

Sets

A set is an unordered collection that cannot include duplicate items. Sets can be used to remove duplicates and to test for membership in a set. Sets also allow the use of mathematical operations like union, intersection, difference, and symmetric difference.

Any duplicate values assigned to a set are removed. If a factset is assigned to a set with the values covered then any values of the fact that are the same will be removed even if they represent different element names and dimensions. For this reason, groups of factsets should generally be assigned to a list and not a set.
## Set & List Operators and Properties

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Property</th>
<th>Sets</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of 2 sets</td>
<td>+</td>
<td>union()</td>
<td>$A = \text{set}(a,b,c); $B = \text{set}(c,d,e);$</td>
<td>$A + B$; $A</td>
<td>B$; $A.\text{union}(B)$</td>
</tr>
<tr>
<td>Intersection of 2 sets</td>
<td>&amp;, intersect</td>
<td>intersect()</td>
<td>$A = \text{set}(a,b,c); $B = \text{set}(c,d,e);$</td>
<td>$A &amp; B$; $A.\text{intersects}(B)$</td>
<td>set(c)</td>
</tr>
<tr>
<td>Difference of 2 sets</td>
<td>-</td>
<td>difference()</td>
<td>$A = \text{set}(a,b,c); $B = \text{set}(c,d,e);$</td>
<td>$A - B$; $A.\text{difference}(B)$</td>
<td>set(a,b)</td>
</tr>
<tr>
<td>Symmetric difference of 2 sets</td>
<td>^</td>
<td>symmetric-difference()</td>
<td>$A = \text{set}(a,b,c); $B = \text{set}(c,d,e);$</td>
<td>$A^B$; $A.\text{symmetric-difference}(B)$</td>
<td>set(a,b,d,e)</td>
</tr>
<tr>
<td>Test if item is in set</td>
<td>in</td>
<td>in</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>c in $A</td>
<td>true</td>
</tr>
<tr>
<td>Test if set contains item</td>
<td></td>
<td>contains()</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>$A.\text{contains}(c)$</td>
<td>true</td>
</tr>
<tr>
<td>Test if item not in set</td>
<td>not in</td>
<td>not in</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>b not in $A</td>
<td>false</td>
</tr>
<tr>
<td>Get length of a set</td>
<td></td>
<td>length()</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>$A.\text{length}$</td>
<td>3</td>
</tr>
<tr>
<td>Convert list to a set</td>
<td></td>
<td>to-set()</td>
<td>$A = \text{list}(a,b,c,b);$</td>
<td>$A.\text{to-set}$</td>
<td>set(a,b,c)</td>
</tr>
<tr>
<td>Convert set to a list</td>
<td></td>
<td>to-list()</td>
<td>$A = \text{set}(a,b,c,d);$</td>
<td>$A.\text{to-list}$</td>
<td>list(a,b,c,d)</td>
</tr>
<tr>
<td>Convert a set to a string</td>
<td></td>
<td>join()</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>$A.\text{join}(,')$</td>
<td>&quot;a,b,c,&quot;</td>
</tr>
<tr>
<td>Test if a set is a subset</td>
<td>&lt;=</td>
<td>is-subset()</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>$A.\text{is-subset}(B)$</td>
<td>true</td>
</tr>
<tr>
<td>Test if a set is a superset</td>
<td>&gt;=</td>
<td>is-superset()</td>
<td>$A = \text{set}(a,b,c);$</td>
<td>$B.\text{is-superset}(A)$</td>
<td>true</td>
</tr>
<tr>
<td>Return value of an index</td>
<td>[ ]</td>
<td>index()</td>
<td>$B = \text{list}(a,b,c,d,e);$</td>
<td>$B.\text{index}(1)$</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>Converts a set or a list to a dictionary</td>
<td></td>
<td>to-dict()</td>
<td>$A = \text{set(list('AAxis','AMember'),list('BAxis','BMem')}$</td>
<td>$A.\text{to-dict}$</td>
<td>dict(list('AAxis','AMember'))</td>
</tr>
</tbody>
</table>

5 The index starts at 1 rather than 0.
Converts a list, set or dictionary to a json string format.

```
A.to-json()
```

$A.to-json

Converts a list, list of lists, or dictionary to a csv format.

```
A.to-csv(separator, header flag, quote strings)
```

$A.to-csv("","",true,true)

Sorts a list or set, uses the argument ‘desc’ to sort descending and the ‘asc’ argument to sort ascending. If no parameter is provided, then the default sort is ascending. If a set is sorted it returns a list.

```
A.sort()
```

$A = list(a,c,b);

$A.sort(desc)

$A.sort(asc)

list(c,b,a)

list(a,b,c)

---

### Dictionaries

A dictionary is an unordered set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A dictionary can be created as follows:

**Example**

```
dict(list('AAxis','AMember'),list('BAxis','BMember'))
```

*This uses the comma to separate the key from the value.*

```
set(list('AAxis','AMember'), list('BAxis','BMember')).to-dict
```

*This creates a dictionary using a list in a set and converting it to a dictionary*

A dictionary cannot be created using curly brackets (as in python) as this is reserved for the factset definition.

### Dictionary Operators and Properties

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Property</th>
<th>Dict</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert a dictionary</td>
<td></td>
<td>join()</td>
<td>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</td>
<td>$A.join(';', '=');</td>
<td>“AAxis=AMember, BAxis=BMember”</td>
</tr>
<tr>
<td>to a string</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return the number</td>
<td></td>
<td>length()</td>
<td>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</td>
<td>$A.length</td>
<td>2</td>
</tr>
</tbody>
</table>
of key-value pairs in a dictionary.

<table>
<thead>
<tr>
<th>Return value of a key</th>
<th>dict(list('AAxis','AMember'), list('BAxis','BMember'));</th>
<th>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</th>
<th>$A['AAxis']</th>
<th>'AMember'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return a set of the key values in dictionary, can add value to get matching keys</td>
<td>keys()</td>
<td>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</td>
<td>$A.keys</td>
<td>set('AAxis', 'BAxis')</td>
</tr>
<tr>
<td></td>
<td>keys(value)</td>
<td>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</td>
<td>$A.keys('BMember')</td>
<td>set('BAxis')</td>
</tr>
<tr>
<td>Test if key is in dictionary</td>
<td>in has-key()</td>
<td>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</td>
<td>$A.has-key('AAxis')</td>
<td>true</td>
</tr>
<tr>
<td>Return a list of the values from key-value pairs</td>
<td>values()</td>
<td>$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));</td>
<td>$A.values</td>
<td>set('AMember', 'BMember')</td>
</tr>
</tbody>
</table>

Note that separate dictionaries cannot be joined using a union in the same manner that a set can be.

**Instance Objects**

The following objects are derived from an XBRL Instance.

**Fact Object**

The fact object is returned when defining a factset. Each fact in the factset has properties that can be queried to get more information about the fact.

**Fact Properties**

In addition to the fact properties listed below the value properties\(^6\) for numerical, string and date values can also be used on the fact object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimals()</td>
<td>The decimal value of the fact value</td>
<td><code>{@Revenues where $fact.decimals == -6}</code></td>
</tr>
<tr>
<td>concept()</td>
<td>The concept of the fact value</td>
<td><code>{@ where $fact.concept.name == Revenues}</code></td>
</tr>
</tbody>
</table>

\(^6\) Listed later in the document.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>period()</td>
<td>The period of the fact. The period object supports properties of start, end, days</td>
<td>{@Revenues where $fact.period.days &gt; 100 and $fact.period.start &gt; date('2014-12-31')}</td>
</tr>
<tr>
<td>unit()</td>
<td>The unit of measure of the fact.</td>
<td>{@ where $fact.unit = unit(xbrli:pure)} Returns all facts that are pure.</td>
</tr>
<tr>
<td>entity()</td>
<td>Returns the entity of the fact. The properties schema and id can be added as additional properties.</td>
<td>{@ where $fact.entity.id = '00000000001'} Returns all facts for entity 00000000001</td>
</tr>
<tr>
<td>dimension(qname of dimension)</td>
<td>Returns the member of the fact for the specified dimension</td>
<td>{@ where $fact.dimension(dei:LegalEntityAxis).name == ABC}</td>
</tr>
<tr>
<td>dimensions()</td>
<td>Returns a dictionary of key values pairs of dimension keys and member values.</td>
<td>{Revenues}.dimensions()</td>
</tr>
<tr>
<td>dimensions-explicit()</td>
<td>Returns a dictionary of key values pairs of explicit dimension keys and member values.</td>
<td>{Revenues}.dimensions-explicit()</td>
</tr>
<tr>
<td>dimensions-typed()</td>
<td>Returns a dictionary of key values pairs of typed dimension keys and member values.</td>
<td>{Revenues}.dimensions-typed()</td>
</tr>
<tr>
<td>id</td>
<td>Returns the id of the fact. If the fact does not have an id, none is returned.</td>
<td>{Revenues}.id</td>
</tr>
</tbody>
</table>

**Inline XBRL Properties**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>inline-is-hidden()</td>
<td>Returns a Boolean if the fact is hidden in an inline document.</td>
<td>{ where $fact.inline-is-hidden == true} Returns all hidden facts</td>
</tr>
<tr>
<td>inline-scale()</td>
<td>Returns the scale of a fact in an inline xbrl document.</td>
<td>{ where $fact.inline-scale == 6} Returns all facts that have a scale of 6.</td>
</tr>
<tr>
<td>inline-format()</td>
<td>Returns the format qname associated with the fact in an inline XBRL document.</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>inline-display-value()</td>
<td>Returns the display value associated with the fact in an inline XBRL document.</td>
<td></td>
</tr>
<tr>
<td>inline-negated()</td>
<td>Returns a Boolean if the fact has a sign in an inline document.</td>
<td></td>
</tr>
</tbody>
</table>

### Period Object

The period object is used to describe the period information associated with a fact value in the instance. Every fact in XBRL has period information associated with it.

#### Period Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>days()</td>
<td>Returns the number of days in a given duration period.</td>
<td>{[@Revenues where $fact.period.days &gt; 100]}</td>
</tr>
<tr>
<td>end()</td>
<td>Returns the end date of a durational period or the date of an instant</td>
<td>{[@Assets where $fact.period.end &gt; date('2014-12-31')] }</td>
</tr>
<tr>
<td>start()</td>
<td>Returns the start date of a durational period</td>
<td>{[@Revenues where $fact.period.start &gt; &quot;2014-12-31&quot;] }</td>
</tr>
</tbody>
</table>

### Unit Object

The unit object is used to describe the unit information associated with a fact value in the instance. Every numerical fact will have unit information associated with it.

#### Unit Properties
<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>numerator()</td>
<td>Returns the xbrl measure of the numerator as a qname. If the unit only has a measure and no division the property returns the measure.</td>
<td><em>where</em> $\text{fact.unit.numerator} == \text{unit(iso4217:USD)}$&lt;br&gt;Tests if the numerator of the fact is USD</td>
</tr>
<tr>
<td>denominator()</td>
<td>Returns the xbrl measure of the denominator as a qname. If there is no denominator an empty value is returned.</td>
<td><em>where</em> $\text{fact.unit.denominator} == \text{“iso4217:USD”}$&lt;br&gt;Tests if the denominator of the fact is USD</td>
</tr>
<tr>
<td>id</td>
<td>Returns the id of the unit used in the instance.</td>
<td>$\text{fact.unit.id}$</td>
</tr>
<tr>
<td>utr()</td>
<td>Returns the symbol of the unit from the units registry if a value exists.</td>
<td>$\text{fact.unit.utr(\text{symbol})}$&lt;br&gt;Retruns the symbol of the fact. Such as $</td>
</tr>
</tbody>
</table>

**Taxonomy Objects**

**Concept Object**

**Concept Equality**

Concepts are compared based on qname. A concept used in two different dt's are considered to be the same concept for equality as the qnames are compared.

**Concept Properties**

Xule includes the following properties for a concept object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute(name)</td>
<td>Returns the value of a custom attribute based on the name provided to the function.</td>
<td>$\text{fact.concept.attribute(abc)}$</td>
</tr>
<tr>
<td>balance()</td>
<td>Returns the balance attribute of a fact. This can be either debit or credit or none.</td>
<td>$\text{fact.concept.balance}, {\text{@concept.balance = debit}}$&lt;br&gt;<em>For Assets will return debit</em></td>
</tr>
<tr>
<td>base-type</td>
<td>Returns the base XBRL type of a concept. For concepts that use a derived type, this will be the XBRL type that the type is originally.</td>
<td>$\text{fact.concept.base-type.name}$&lt;br&gt;<em>For BasisOfAccount (which has a data type of nonnum:textBlockItemType) will return</em></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| derived from. | Returns the type of a concept. | $\text{fact.concept.data-type.name}$
|          | $\text{For Assets will return the type object for xbrli:stringItemType.}$ |         |
| data-type |          |         |
| enumerations | Returns a set of enumerated values allowed for the concept. | $\text{concept.enumerations}$ |
| has-enumerations | Returns a true of false if the concept has enumerations in the datatype | $\text{concept.has-enumerations}$ |
| is-abstract() | Returns true if the concept has an abstract value of true. This attribute can only be on the concept object. | $\text{concept.is-abstract,}$
<p>|          | $\text{[@concept.is-abstract = false]}$ | $\text{For Assets will return false}$ |
| is-monetary() | Returns Boolean result if the concept has that type. | $\text{$\text{fact.concept.is-monetary()}$}$ |
|          | $\text{For Assets will return true}$ |         |
| is-numeric() | Returns Boolean result if the concept has that type. | $\text{$\text{fact.concept.is-numeric()}$}$ |
|          | $\text{For Assets will return true}$ |         |
| is-type(type) | Returns Boolean result if the concept has that type. The type is provided as a qname. | $\text{$\text{fact.concept.is-type(xbrli:monetaryItemType)}$}$ |
|          | $\text{For Assets will return true}$ |         |
| label(label role, language) | For a concept the label property can be used to return the label associated with a concept. The two parameters are optional. If no parameter is provided a label object is returned. This property will return a label object. To get to the text, role and language of a label use .text, .role, .lang, respectively. | $\text{taxonomy().concept(Assets).label.text}$ |
|          | $\text{For Assets will return the string of Assets}$ |         |
| local-name() | Returns the local name of the concept name. | $\text{$\text{fact.concept.name.local-name}$}$ |
|          | $\text{For Assets will return the string Assets}$ |         |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>name()</td>
<td>Returns the qname of the concept. This includes the local name and URI.</td>
<td>$fact.concept.name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Assets will return us-gaap:Assets</td>
</tr>
<tr>
<td>namespace-uri()</td>
<td>Returns the uri of the concept name.</td>
<td>$fact.concept.name.namespace-uri</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Assets will return us-gaap</td>
</tr>
<tr>
<td>period-type()</td>
<td>Returns the period type, instant or duration.</td>
<td>$fact.concept.period-type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Assets will return instant</td>
</tr>
<tr>
<td>references(refer-</td>
<td>Returns the references associated with a concept for a given dts.</td>
<td>taxonomy().concept(Assets).references(&quot;<a href="http://www.xbrl.org/2003/role/presentationRef">http://www.xbrl.org/2003/role/presentationRef</a>&quot;)</td>
</tr>
<tr>
<td>ence-role)</td>
<td></td>
<td>Returns a set of reference objects for Assets</td>
</tr>
<tr>
<td>relationships()</td>
<td>Returns all relationships associated with the concept.</td>
<td></td>
</tr>
<tr>
<td>source-relationships()</td>
<td>Returns the relationships where the concept is the source.</td>
<td></td>
</tr>
<tr>
<td>substitution()</td>
<td>Returns the substitution group of the concept.</td>
<td>$fact.concept.substitution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Assets will return xbrli:item</td>
</tr>
<tr>
<td>target-relationships()</td>
<td>Returns the relationships where the concept is the target.</td>
<td></td>
</tr>
</tbody>
</table>

Reference Object

The reference object has all the detailed parts of a given reference. These parts can be accessed from the reference object using reference properties.

Reference Properties

Xule includes the following properties for a reference object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>part-by-name(part</td>
<td>Returns the reference part for a reference based on part name.</td>
<td>reference.part-by-name(cod:Topic).part-value</td>
</tr>
<tr>
<td>qname)</td>
<td></td>
<td>Returns the reference part value with a qname of cod:Topic</td>
</tr>
<tr>
<td>parts()</td>
<td>Returns a set of parts for a concept.</td>
<td>reference.parts</td>
</tr>
<tr>
<td>Name</td>
<td>Definition</td>
<td>Examples</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>part-value</td>
<td>Returns the value associated with a reference part</td>
<td>reference().parts.part-value</td>
</tr>
<tr>
<td>name()</td>
<td>Returns the qname of the reference part.</td>
<td>reference().parts.name</td>
</tr>
<tr>
<td>namespace()</td>
<td>Returns the namespace of the reference part</td>
<td>reference().parts.namespace</td>
</tr>
<tr>
<td>local-name()</td>
<td>Returns the local name of the reference part</td>
<td>reference().parts.local-name</td>
</tr>
<tr>
<td>order()</td>
<td>Returns the order of the part</td>
<td>reference().parts.order</td>
</tr>
</tbody>
</table>

**Reference Example**

```
$con = $us-gaap.concept(Assets);

    for ($ref in $con.references('http://www.xbrl.org/2003/role/presentationRef'))
        list("reference\n",
            for ($p in $ref.parts)
                $p.name.string + " - " + $p.part-value + "\n"
        ).join(""
    )
```

*This will return a list of all the presentation references for the concept Assets in the us gaap taxonomy and adds some formatting*

*I.e. the first item.*

*reference*
ref:Publisher - FASB
ref:Name - Accounting Standards Codification
codification-part:Topic - 942
codification-part:SubTopic - 210
ref:Section - S99
ref:Paragraph - 1
ref:Subparagraph - (SX 210.9-03(11))
Note that the variable $us-gaap is set as a constant to the US GAAP entry point with references of: constant $us-gaap = taxonomy('http://xbrl.fasb.org/us-gaap/2017/entire/us-gaap-entryPoint-all-2017-01-31.xsd')

Label Object
A label object represents a dictionary of labels. A concept can have multiple labels and the label object can be accessed to query those labels. Each individual label has the following properties.

Label Object Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>text()</td>
<td>Text of a label</td>
<td>label.text</td>
</tr>
<tr>
<td>lang()</td>
<td>The language of a label</td>
<td>label.lang</td>
</tr>
<tr>
<td>role()</td>
<td>The role of the label</td>
<td>label.role</td>
</tr>
</tbody>
</table>

If the label object is referenced in an output message and multiple values exist for the object, then the processor will return the standard label as the default. The documentation label will only be returned if no other label is available.

Type Object
A type object represents a data type of a concept.

Type Object Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>name()</td>
<td>QName of the type, for simple types</td>
<td>concept.data-type.name</td>
</tr>
<tr>
<td>enumerations()</td>
<td>Returns a set of allowed values for the type</td>
<td>concept.data-type.enumerations</td>
</tr>
<tr>
<td>has_enumerations()</td>
<td>Returns a boolean if the type is restricted to a list of enumerated values.</td>
<td>concept.data-type.has_enumerations</td>
</tr>
</tbody>
</table>
Cube Object

The cube object reflects the cube concept in an extended link role, the associated axis, primary concepts members, facts, defaults and domains.

Cube Properties

XULE includes the following properties for a cube object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>cube-concept</td>
<td>Returns the hypercube concept of the cube</td>
<td>cube.cube-concept()</td>
</tr>
<tr>
<td>drs-role()</td>
<td>Returns an extended link role object of the role that the specified cube is included in.</td>
<td>cube.drs-role().uri == “BalanceSheet”</td>
</tr>
<tr>
<td>dimensions()</td>
<td>Returns the dimensions of a cube as a dimensions object</td>
<td>cube.dimensions</td>
</tr>
<tr>
<td>primary-concepts()</td>
<td>Returns the primary concepts of a cube.</td>
<td>cube.primary-concepts</td>
</tr>
<tr>
<td>members()</td>
<td>Returns the primary concepts of a cube.</td>
<td>cube.members</td>
</tr>
<tr>
<td>closed()</td>
<td>Returns a boolean result of true if the cube is closed.</td>
<td>cube.closed</td>
</tr>
<tr>
<td>facts()</td>
<td>Returns all the facts associated with a cube.</td>
<td>cube.facts</td>
</tr>
</tbody>
</table>

Dimension Object

The dimension object is different than the dimension concept. The dimension object is a dimension on a cube in a given linkrole with members, domains and defaults.

Dimension Properties

XULE includes the following properties for a dimension object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th><strong>dimension-type()</strong></th>
<th>Returns typed dimensions.</th>
<th><strong>dimension.dimension-type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Returns a string value of typed or explicit.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>members()</strong></th>
<th>Returns the members on a dimension</th>
<th><strong>dimension.members</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>default()</strong></th>
<th>Returns the default of a given dimension.</th>
<th><strong>dimension.default</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>concept()</strong></th>
<th>Gets the qname concept for the dimension</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>domains()</strong></th>
<th>Returns the domains of a given dimension</th>
<th><strong>dimension.domains</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>useable-members()</strong></th>
<th>Returns the useable members on a dimension.</th>
<th><strong>dimension.useable-members</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>nonuseable-members()</strong></th>
<th>Returns the non-useable members on a dimension.</th>
<th><strong>dimension.non-useable-members</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>cube</strong></th>
<th>Return the cube object that the dimension is on.</th>
</tr>
</thead>
</table>

---

**Members Object (Not Yet Implemented)**

<table>
<thead>
<tr>
<th><strong>concept()</strong></th>
<th>Returns the concept associated with a member.</th>
<th><strong>members.concept</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>dimensions()</strong></th>
<th>Returns the dimensions associated with a member</th>
<th><strong>members.dimensions</strong></th>
</tr>
</thead>
</table>

---

**Taxonomy (DTS) Object**

XULE automatically provides access to the taxonomy of the instance via the `taxonomy()` function. A taxonomy is primarily used to find concepts and navigate relationships.

```plaintext
taxonomy().concept(Assets)
```

*Return the Assets qname from the taxonomy of the instance.*
Additional taxonomies can be accessed by providing the entry point documents to the 
taxonomy() function.

```
```

*This loads the US GAAP elements taxonomy.*

**Taxonomy Properties**

Xule includes the following properties for a taxonomy or DTS object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepts()</td>
<td>Returns a set of qnames representing every concept in a taxonomy or a network. Can be used on a taxonomy type or a network type.</td>
<td><code>taxonomy().concepts</code> Returns all the concepts in the instance taxonomy. (Whether they have values or not, or are included in a tree or not.)</td>
</tr>
<tr>
<td>concept(QName)</td>
<td>Returns the concept in a taxonomy based on the concept qname.</td>
<td><code>taxonomy().concept(Assets)</code> Return the Assets qname from the taxonomy of the instance.</td>
</tr>
<tr>
<td>cube(Concept or QName, Role)</td>
<td>Returns a cube from the taxonomy. The first argument is the concept or qname of the cube concept. The role is the drs-role.</td>
<td><code>taxonomy().cube(StatementTable, ShareholdersEquity)</code> Returns the cube object</td>
</tr>
<tr>
<td>cubes()</td>
<td>Returns all the cubes in the taxonomy as a set.</td>
<td><code>taxonomy().cubes</code> Returns all the cubes in a taxonomy</td>
</tr>
<tr>
<td>effective-weight(QName, Qname)</td>
<td>Returns the effective weight between two concepts aggregated across all calculation networks.</td>
<td><code>taxonomy().effective-weight(NetCashProvidedByUsedInOperatingActivities,IncomeLossFromEquityMethodInvestments)</code> Return the effective weight between 2 concepts. This operates over all networks. If the weight is not the same across networks it returns 0. The values can be -1, 1 or 0.</td>
</tr>
<tr>
<td>effective-weight-network(QName, QName, Role)</td>
<td>Returns the effective weight between two concepts in a given calculation network. Returns a set</td>
<td><code>taxonomy().effective-weight-network(NetCashProvidedByUsedInOperatingActivities,IncomeLossFromEquityMethodInvestments)</code> Return the effective weight between 2 concepts in a given calculation network. Returns a set</td>
</tr>
</tbody>
</table>
of lists containing the network and effective weight between the 2 concepts. The third parameter (role) is optional.

| Dimensions() | Return all the dimensions in a taxonomy that are dimension objects. | taxonomy().dimensions Returns all the dimensions in a taxonomy |
| Dimension(QName) | | |
| Explicit-dimensions() | Return all the dimensions in a taxonomy that are explicit dimension objects. | taxonomy().explicit-dimensions Returns all the dimensions in a taxonomy |
| Typed-dimensions() | Return all the dimensions in a taxonomy that are typed dimension objects. | taxonomy().explicit-dimensions Returns all the dimensions in a taxonomy |
| Networks(arcrole, extended link role) | Returns a set of network objects from the taxonomy. Allows the parameters of arc role and extended link role. Both of the parameters are optional. The extended link role can use a short name but should not be in quotes. | taxonomy().networks() Returns all the networks in the taxonomy. |
|  |  | taxonomy().networks(parent-child) Returns all the parent-child networks in the taxonomy. |

Return the effective weight between 2 concepts by network. This example operates over all networks.

taxonomy().effective-weight-network(NetCashProvidedByUsedInOperatingActivities,IncomeLossFromEquityMethodInvestments, StatementCashFlow)

Return the effective weight between 2 concepts by network. This example operates over all networks.
Network Set Object

XULE provides access to the networks comprising a taxonomy of the instance via the networks() function. Networks are all the networks in the taxonomy. Specific networks can be returned by specifying the arcrole and or extended link role. Both are optional.

\[\text{taxonomy().networks(parent-child,} \ 'http://www.abc.com/role/ConsolidatedBalanceSheets')\]

*Returns the parent-child network for the consolidated Balance Sheet in the taxonomy.*

Network Object

The network object represents a single network in the taxonomy, identified by its extended link role, arc-role, arc element name, and extended link name.

Network Properties

Xule includes the following properties for a Network() object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>arc-role()</td>
<td>Returns the arc-role of the network.</td>
<td>Network().arc-role</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Could return summation-item, parent-child etc.</td>
</tr>
<tr>
<td>concept-names()</td>
<td>Returns a set of qnames representing every concept in a network.</td>
<td>network().concept-names</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns all the qnames in a network. (Whether they have values or not, or are included in a tree or not.)</td>
</tr>
<tr>
<td>concepts()</td>
<td>Returns a set of every concept in a network including target and source</td>
<td>network().concepts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns all the concepts in a network.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>source-concepts()</td>
<td>Returns the source concepts from a network.</td>
<td></td>
</tr>
<tr>
<td>target-concepts()</td>
<td>Returns the target concepts in a network.</td>
<td></td>
</tr>
<tr>
<td>relationships()</td>
<td>Returns a set of relationships that can be looped to get the relationship object</td>
<td>network().relationships</td>
</tr>
<tr>
<td>role()</td>
<td>Returns the role of the network. The role as 3 properties of uri, description and tree (linkbase) applicable to. (Is the extended link role)</td>
<td>network.role.uri</td>
</tr>
<tr>
<td></td>
<td></td>
<td>network().role.description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>network().role.used-on</td>
</tr>
<tr>
<td>roots()</td>
<td>Returns a set of qnames representing the root concepts of a network</td>
<td>network().roots</td>
</tr>
</tbody>
</table>

### Role Object

### Role Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>uri()</td>
<td>Returns the uri of a role.</td>
<td>network.role.uri</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns the uri of the role</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>network.arcrole.uri</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns the uri of the arcrole</strong></td>
</tr>
<tr>
<td>description()</td>
<td>Returns the description of the role.</td>
<td>network.role.description</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns the description of the role</strong></td>
</tr>
</tbody>
</table>
Relationship Object

This object is used to define a relationship between two concepts. It is not used for a relationship between a concept and a resource.

The relationship object is usually obtained by using the navigate function in XULE. However it can also be obtained by looping through the relationships object.

Relationship Properties

XULE includes the following properties for a Relationship() object.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>source()</td>
<td>The source concept of the relationship</td>
<td>relationship().source</td>
</tr>
<tr>
<td>source-name()</td>
<td>The QName of the source concept of the relationship</td>
<td>relationship().source-name</td>
</tr>
<tr>
<td>target()</td>
<td>The target concept of the relationship</td>
<td>relationship().target</td>
</tr>
<tr>
<td>target-name()</td>
<td>The QName of the target concept of the relationship</td>
<td>relationship().target-name</td>
</tr>
<tr>
<td>order()</td>
<td>The value of the order attribute on the relationship</td>
<td>relationship().order</td>
</tr>
<tr>
<td>weight()</td>
<td>The value of the weight</td>
<td>relationship().weight</td>
</tr>
<tr>
<td>Attribute on the Relationship</td>
<td>Description</td>
<td>Syntax</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>preferred-label()</td>
<td>The value of the preferred Label attribute on the relationship</td>
<td>relationship().preferred-label</td>
</tr>
<tr>
<td>role()</td>
<td>The extended link role of the network</td>
<td>relationship().role</td>
</tr>
<tr>
<td>arcrole()</td>
<td>The arcrole of the network</td>
<td>relationship().arcrole</td>
</tr>
<tr>
<td>arcrole-uri()</td>
<td>The arcrole uri of the network</td>
<td>relationship().arcrole-uri</td>
</tr>
<tr>
<td>arcrole-description()</td>
<td>The description of the arcrole of the network</td>
<td>relationship().arcrole-description</td>
</tr>
<tr>
<td>link-name()</td>
<td>The QName of the extended link element</td>
<td>relationship().link-name</td>
</tr>
<tr>
<td>arc-name()</td>
<td>The QName of the arc element</td>
<td>relationship().arc-name</td>
</tr>
<tr>
<td>network()</td>
<td>The network</td>
<td>relationship().network</td>
</tr>
</tbody>
</table>

### Properties and Functions

Properties and functions can be used interchangeably, either as a property of an object or as functions that can be passed a parameter of a value or a object.

#### Numerical Properties & Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
</table>
| abs() | Returns the absolute value of a numerical value. Can be used on an integer, float, decimal and fact type. | Property: {@Assets}.abs  
Function: abs({@Assets})  

Returns the value of assets as an absolute value. |
| log10() | Returns the log of a number. Can be used on an integer, float, and decimal type. | Property: {@Assets}.log10  
Function: log10({@Assets})  

Returns the log10 value of assets. |
| power() | Returns the power of a number. Can be used on an integer, float, and decimal type. If you want to | Property: 4.power(2).  

Returns a value of 16. |
<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>signum()</td>
<td>Returns a value of -1 if the number is negative and a value of positive 1 if positive. If the value is zero it returns 0. Can be used on an integer, float, decimal and fact type.</td>
<td>Property: `{@Assets}.signum Function: signum({@Assets})</td>
</tr>
<tr>
<td>trunc(number, places)</td>
<td>Truncates the decimal places on a number. Places defaults to zero if not provided.</td>
<td>Property: `{@Assets}.trunc(2) Function: trunc({@Assets},2) truncates the value of assets to two decimal places</td>
</tr>
<tr>
<td>round(number, places)</td>
<td>Rounds a number to the decimal places indicated. Rounds to the nearest even.</td>
<td>Property: `{@Assets}.round(2) Function: round({@Assets},2) Rounds the value of assets to two decimal places</td>
</tr>
<tr>
<td>mod(numerator, divisor)</td>
<td>Returns the mod of a numerator and a divisor.</td>
<td>Property: `{@Assets}.mod({@Liabilities}) Function: mod({@Assets},{@Liabilities})</td>
</tr>
</tbody>
</table>

### String Functions and Properties

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains()</td>
<td>Returns a boolean result if the provided value is contained in the provided string.</td>
<td>“<a href="http://some/role/for/cashflow/%22.lower-case.contains(%22cashflow">http://some/role/for/cashflow/&quot;.lower-case.contains(&quot;cashflow</a>&quot;) Returns a boolean of true</td>
</tr>
<tr>
<td>index-of()</td>
<td>The index-of(string) returns an integer of the position of the first instance of a given string in another string.</td>
<td>“Hello Mr. Stains”.index-of(’llo’) = 3 Returns a value of 3</td>
</tr>
<tr>
<td>string()</td>
<td>Converts an integer, float or decimal to a string</td>
<td>3.string = “3” Returns a string value of “3”</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>last-index-of()</td>
<td>The last-index-of(string) returns an integer of the position of the last instance of a given string in another string.</td>
<td>&quot;Hello Mr Stains&quot;.last-index-of(‘o’) = 5  &lt;br&gt; Returns a value of 5</td>
</tr>
<tr>
<td>length()</td>
<td>Returns the length as an integer of a string.</td>
<td>&quot;Hello Mr. Stains&quot;.length() = 16  &lt;br&gt; Returns a value of 16</td>
</tr>
<tr>
<td>lower-case()</td>
<td>Returns a string as lowercase characters.</td>
<td>&quot;<a href="http://some/role/for/cashflow/%22.lower-case()">http://some/role/for/cashflow/&quot;.lower-case()</a></td>
</tr>
<tr>
<td>split()</td>
<td>Split a string into a list based on a defined delimiter character. If an empty string is used as the separator, the returned list will contain 1 item with the entire string in it.</td>
<td>&quot;Hello Mr Stains&quot;.split(‘ ‘) = list(“Hello”, “Mr”, ”Stains”)</td>
</tr>
<tr>
<td>number()</td>
<td>Converts a string to a number. If the string has a period it is converted to a decimal. If it has inf then it is converted to a floating number otherwise the string is converted to an integer.</td>
<td>&quot;3.4&quot;.number = 3.4  &lt;br&gt; Returns a decimal number value of 3.4</td>
</tr>
<tr>
<td>starts-with()</td>
<td>Returns a value of true or false if the string starts with a given string.</td>
<td>&quot;Value of Derivatives&quot;.starts-with(‘Value’) = true  &lt;br&gt; Returns a value of true</td>
</tr>
<tr>
<td>ends-with()</td>
<td>Returns a value of true or false if the string ends with a given string.</td>
<td>&quot;Value of Derivatives&quot;.ends-with(‘ves’)</td>
</tr>
<tr>
<td>substring()</td>
<td>The substring(int beginIndex, int endIndex) returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex. If the last</td>
<td>&quot;Hello Mr. Stains&quot;.substring(1,5)  &lt;br&gt; Returns the string of “Hello”</td>
</tr>
</tbody>
</table>
parameter is left off it continues to the end of the string.

to-qname() | Converts a string to a qname. The string can include a prefix which is resolved with the namespace declarations in the rule set. If the namespace cannot be resolved, the property will raise an error.

| ‘Abc’.to-qname |
| Returns a qname with local name ‘abc’ and the default namespace defined in the rule set. |
| ‘us-gaap:Assets’.to-qname |
| Returns a qname with the namespace defined in the rule set for prefix ‘us-gaap’ and local name ‘Assets’. |

upper-case() | Returns a string as lowercase characters.

| “cashflow”.upper-case() |
| Returns the string of “CASHFLOW” |

Generic Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>is-fact</td>
<td>Returns true or false if a value represents a fact object.</td>
<td>{@Assets}.is-fact</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.is-fact</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns false.</td>
</tr>
</tbody>
</table>

Date Properties & Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>date(string)</td>
<td>Pass this function a point in time string in the format yyyy-mm-dd to produce an instant date that can be compared to a fact. NOTE that a date fact is already a date type and should not be converted to a date.</td>
<td>date(“2017-12-31”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR “2017-12-31”.date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Converts the string to a date.</td>
</tr>
</tbody>
</table>

<p>| day() | Returns the day (number) from a given date. | day(date(‘2017-12-31’)) |
|       | Returns a value of 31. |</p>
<table>
<thead>
<tr>
<th>Function/Property</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration(start-date, end-date)</td>
<td>Pass this function a start and end date in the format yyyy-mm-dd to produce a duration period that can be compared to a fact.</td>
<td>Function: <code>{@period = duration('2016-01-01', '2016-12-31')}</code> Converts two string dates to a duration for comparing or filtering a fact.</td>
</tr>
<tr>
<td>fiscal-year()</td>
<td>Returns the fiscal year (number) from a given date.</td>
<td>Property: <code>{@period.fiscal-year = '2017'}</code> The property filters on the fiscal year of the fact based on the balance date of the filing.</td>
</tr>
<tr>
<td>forever()</td>
<td>Generates a period equal to the forever period</td>
<td>Function: <code>{@period = forever }</code> Returns all periods that have a period of forever.</td>
</tr>
<tr>
<td>month()</td>
<td>Returns the month (number) from a given date.</td>
<td>Function: <code>month(date(&quot;2017-12-31&quot;))</code> Returns a value of 12.</td>
</tr>
<tr>
<td>is-fiscal-half-year(integer)</td>
<td>Returns a boolean value of true if the period of the fact represents a specific half year. The property requires a parameter representing a specific half year of either 1 or 2. The property is only applicable to fact values</td>
<td>Property: <code>{@period.is-fiscal-half-year(1) = true}</code> Returns all fact values that are in the first half of the year.</td>
</tr>
<tr>
<td>is-fiscal-quarter(integer)</td>
<td>Returns a Boolean value of true if the period of the fact represents a specific fiscal quarter. The property requires a parameter representing a specific quarter of either 1,2,3,or 4. The property is only applicable to fact values</td>
<td>Property: <code>{@period.is-fiscal-quarter(1) = true}</code> Returns all fact values that are in the first quarter</td>
</tr>
<tr>
<td>is-half-year()</td>
<td>Returns a Boolean value of true if the period of the fact represents a half year. The property can also take a parameter representing a specific calendar half-year of either 1 or 2. The property is only applicable to</td>
<td>Property: <code>{@period.is-half-year = true}</code> Returns all fact values that are half years</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Property: <code>{@period.is-half-year(1) = true}</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Property/Function</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>fact values</td>
<td>Returns all fact values that are in the first half of the year</td>
<td></td>
</tr>
<tr>
<td>is-fiscal-year</td>
<td>Filters on periods that are actual fiscal years.</td>
<td></td>
</tr>
<tr>
<td>is-quarter()</td>
<td>Returns a Boolean value of true if the period of the fact represents a quarter. The property can also take a parameter representing a specific quarter of either 1, 2, 3, or 4. The property is only applicable to fact values.</td>
<td>Property: {@period.is-quarter = true} Returns all fact values that are quarters Property: {@period.is-quarter(1) = true} Returns all fact values that are in the first quarter</td>
</tr>
<tr>
<td>is-year</td>
<td>Returns a value of true for all facts that have a period that is a year in length.</td>
<td>Property: {@period.is-year = true} Returns all facts that are a year in length</td>
</tr>
<tr>
<td>is-3Q-cum</td>
<td>Returns a value of true for all facts that have a period that is three quarters in length.</td>
<td>Property: {@period.is-3Q-cum = true} Returns all facts that are a three quarters in length</td>
</tr>
<tr>
<td>is-month-day</td>
<td>Checks the value of the element with a datatype of gMonthDayItemType is valid.</td>
<td>Property: {@CurrentFiscalYearEndDate where $fact.is-month-day != true} Identifies if the value of CurrentFiscalYearEndDate is valid.</td>
</tr>
<tr>
<td>is-leap-year</td>
<td>Identifies if a year(number) is a leap year</td>
<td>Property: year(date(“2017-12-31”)).is-leap-year</td>
</tr>
<tr>
<td>days()</td>
<td>Is a property of a period.</td>
<td>Property: {@period.days = 90} Returns all facts that are 90 days in length</td>
</tr>
<tr>
<td>time-span()</td>
<td>Allows a span of time to be defined for example time-span(“P90D”). This uses the XML duration format to return the number of days in the period</td>
<td>function:$document_period_end_date + (time-span(&quot;P4D&quot;)) Adds 4 days to the value of document period end date.</td>
</tr>
<tr>
<td>year()</td>
<td>Returns the year (number) from a fact or the year from a given date.</td>
<td>Property: {@period.year = ‘2017’} Function: year(date(“2017-12-31”))</td>
</tr>
</tbody>
</table>
Aggregation Functions

Aggregation functions only work on sets and lists. The values returned from a factset need to be expressed as a list or set before the aggregation functions below can be used.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>average(set</td>
<td>list)</td>
<td>Will return the average of values in a set or a list.</td>
</tr>
<tr>
<td>count()</td>
<td>Will return the count of values in a set or a list. Operates on any type.</td>
<td>Function: \texttt{count(set(a,b,c,d,e))} This will return a value of 4</td>
</tr>
<tr>
<td>max()</td>
<td>Returns the maximum value of a set or a list</td>
<td>Function: \texttt{max(list({@concept = PlanAssets @DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))} This will return the largest value of the Plan Assets from all plans.</td>
</tr>
<tr>
<td>min()</td>
<td>Returns the minimum value in a set or list.</td>
<td>Function: \texttt{min(list( {@concept = PlanAssets @DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))} This will return the lowest value of the Plan Assets from all plans.</td>
</tr>
<tr>
<td>sum()</td>
<td>Returns the sum of a set or a list, this operates on any numerical and string type. Strings are concatenated.</td>
<td>Function: \texttt{sum(list( {@concept = PlanAssets @DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))} This will sum the value of the Plan Assets for every Plan member.</td>
</tr>
</tbody>
</table>
| stdev()      | Returns the standard deviation of a                                        | Function: \texttt{stdev(list( {@concept =} | XULE Language Syntax v1.0 | October 2019
| PlanAssets @DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *))
| This will return the stdev of the Plan Assets for every Plan member.

**prod()**

Returns the product of a set or a list.

*Function:* prod(list({@concept = PlanAssets @DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))

This will return the product of Plan Assets for every Plan member.

| all() |
| Returns true if all values in a list or a set are true |
| any() |
| Returns true if any value is true in a list or a set. |
| first() |
| Returns the first value found in a set or in a list |
| last() |
| Will return the last value in a set or a list |

### Statistical Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>corr(y, x)</td>
<td>correlation coefficient</td>
<td></td>
</tr>
<tr>
<td>regr_r2(y, x)</td>
<td>square of the correlation coefficient</td>
<td></td>
</tr>
</tbody>
</table>

### Existence Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists()</td>
<td>Tests for the existence of a fact, object. If any fact exists then a value of true is returned. Exists is applicable for checking the existence of facts. Exists will return a value of true when a</td>
<td>exists({@Assets}) Tests if any fact value is reported for assets and returns true or false</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Examples</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td><code>exists(list())</code></td>
<td>Returns a value of true.</td>
<td></td>
</tr>
<tr>
<td><code>missing(list())</code></td>
<td>Returns a value of false.</td>
<td></td>
</tr>
<tr>
<td><code>missing({covered @Assets})</code></td>
<td>Tests if any fact value is not reported for assets. Covered needs to be used with missing if determining if the fact exists in the document at all.</td>
<td></td>
</tr>
<tr>
<td><code>first-value</code></td>
<td>Takes a list of expressions and returns the first expression that has a value. If no value is returned, then no value is returned. This is the same as a factset that does not return any facts.</td>
<td><code>first-value({@Assets},{@CurrentAssets})</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Unit Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unit()</code></td>
<td>Pass this function the unit URI and local name to define a comparable unit with the instance.</td>
<td><code>{@unit = unit(iso4217:USD)}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns all fact values reported in USD</td>
</tr>
<tr>
<td><code>convert-unit(fact, resulting_unit)</code></td>
<td>Converts the value of the fact to the resulting unit and returns the result. Only works in those cases where the conversion rate is constant between the unit of the fact and the resulting unit. The function requires that the datatype of the concept is the same as the datatype defining the unit in the unit registry. Conversions cannot be done where the unit of the fact does not have the same datatype as the resulting member. I.e. a value of 3 feet in length with a datatype of <code>lengthItemType</code> cannot be converted to a unit with a datatype of <code>lengthItemType</code>.</td>
<td><code>convert-unit($fact, unit(uri:m))</code></td>
</tr>
</tbody>
</table>
### DTS Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>qname(namesp ace, local- name)</td>
<td>Creates a valid qname by providing the namespace and localname as parameters.</td>
<td>qname(‘$ext_namespace’,’FairValueInputsLevel2AndLevel3Member’)</td>
</tr>
<tr>
<td></td>
<td>Defines a qname based on the extension namespace</td>
<td></td>
</tr>
<tr>
<td>taxonomy()</td>
<td>Creates a taxonomy object based on the taxonomy entry point. If no parameter is passed the DTS of the current instance is used.</td>
<td>$us-gaap = taxonomy(‘<a href="http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-2016-01-31.xsd%E2%80%99">http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-2016-01-31.xsd’</a>)</td>
</tr>
<tr>
<td></td>
<td>Returns the US-GAAP taxonomy</td>
<td></td>
</tr>
<tr>
<td>entry-point-namespace(tax onomy)</td>
<td>Returns the namespace of the entry point for the taxonomy (DTS) used. Takes the taxonomy object as a parameter.</td>
<td>entry-point-namespace($us-gaap)</td>
</tr>
<tr>
<td></td>
<td>Returns the namespace of the us-gaap taxonomy “<a href="http://xbrl.fasb.org/us-gaap/2019%E2%80%9D">http://xbrl.fasb.org/us-gaap/2019”</a></td>
<td></td>
</tr>
<tr>
<td>entry-point()</td>
<td>Returns the entry point uri of the taxonomy object passed as a parameter</td>
<td>entry-point($us-gaap)</td>
</tr>
</tbody>
</table>

### Range Function

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
</table>
| range(start, stop, step) | The range function generates a list of integer numbers between the given start integer to the stop integer, which is generally used to iterate through a Loop. The range function accepts an integer and returns a list of integers. A single argument represents the stop | range(5)  
Returns: list(1,2,3,4,5)  
range(4,10)  
Returns: list(4, 5, 6, 7, 8, 9, 10) |
integer, two arguments are the start and stop integers. The default start integer is 1. The default step integer is 1 unless specified otherwise.

\[
\text{range}(4, 10, 2) \\
\text{Returns:} \\
\text{list}(4, 6, 8, 10)
\]

Use in a loop with list $path$:

\[
\text{for } i \text{ in range}($path.length) \\
\text{$path[i]$}
\]

## Data Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>csv-data()</td>
<td>csv-data takes 4 arguments. The first 2 are required.</td>
<td><code>csv-data('https://xbrl.us/dqc_06_data_bounds.csv', true, list('string', 'string', 'string'))</code></td>
</tr>
<tr>
<td></td>
<td>1. file url</td>
<td>Defines a csv file as a source of data.</td>
</tr>
<tr>
<td></td>
<td>2. has_headers - either true or false. If it is true, the first line is ignored.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. list of types - optional. If supplied, it must have a type for each column</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. as dictionary - optional. If supplied and true, the result for the row will be a dictionary using the Column names from the header as the key.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The function returns a list of rows. The row is either a list or a dictionary (if the 4th argument is supplied and true).</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Limitations:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● If the list of types is supplied, no row can have more columns than the length of the list of types.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● If headers are in the file and returning as dictionary, no row can have more columns than the header row.</td>
<td></td>
</tr>
<tr>
<td>json-data()</td>
<td>json-data takes 1 parameter which is the url to the json file.</td>
<td><code>json-data('https://xbrl.us/json-file.json')</code></td>
</tr>
<tr>
<td></td>
<td>The function returns a dictionary which matches the structure of the json file.</td>
<td>Defines a json file as a source of data.</td>
</tr>
</tbody>
</table>
Information Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
</table>
| rule-name()     | rule-name returns the name of the current executing rule. The rule name is the generated rule name composed of the rule-name-prefix and rule-name-separator and the specific rule name if there is a rule-name prefix, otherwise it is just the specific rule name. It does not include a rule-suffix, as this cannot be determined during rule processing. When evaluating constants, the rule-name() returns none. | output abc rule-name()  
Returns: 'abc' |

Custom Functions

Often a user may write a number of rules that repeat the same logic. Rather than duplicating the rule logic XULE supports defining custom functions that can be defined once and used by many rules. Functions allow the user to pass values to the function and return the results of the function. Xule functions return the value of the body of the function. In addition, any variables or tags defined within the function are available to use in a message. Functions are defined with the keyword ‘function’.

```
function add_two_numbers($a , $b)  
    $a + $b  
This function will add two variables passed to it and return the result.
```

```
assert sum_less_zero satisfied  
$sum_assets_liabilities = add_two_numbers({@Assets} , {@Liabilities});  
$sum_assets_liabilities < 0
```

```
message  
"The value of {$sum_assets_liabilities.concept} with a value of {$sum_assets_liabilities} is less than zero. Please enter a positive amount"
```

This rule uses the function to add Assets and Liabilities and check the sum is less than 0.

Alignment of facts is maintained when using a function.

Recursive functions are not supported by the XULE processor implementation.

Functions do not have to have arguments. In the example below the function defines a set.
function non_neg_concepts()
    set(Assets, Liabilities)

Formatting Strings
TO BE DEFINED

Defining Namespaces
In order to determine which namespace an element is in prefixes can be associated with concepts. The prefixes are defined in the rule file in the following manner.

/* DECLARE NAMESPACES USED IN THE RULES */
namespace iso4217 = http://www.xbrl.org/2003/iso4217
namespace us-types = http://fasb.org/us-types/2017-01-31
namespace exch = http://xbrl.sec.gov/exch/2017-01-31
namespace http://fasb.org/us-gaap/2017-01-31
namespace currency = http://xbrl.sec.gov/currency/2017-01-31
namespace invest = http://xbrl.sec.gov/invest/2013-01-31
namespace nonum = http://www.xbrl.org/dtr/type/non-numeric
namespace num = http://www.xbrl.org/dtr/type/numeric

The default namespace is declared by not including a prefix. The namespaces only need to be defined once and not in every file.

Assertion Types
Xule requires different assertion types to be defined. The assertion types supported by XULE are as follows:

<table>
<thead>
<tr>
<th>Assertion Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>This will perform a value assertion. This requires a boolean result and will produce output dependent on the assertion</td>
</tr>
<tr>
<td>output</td>
<td>This will return the results without a severity</td>
</tr>
</tbody>
</table>

Satisfied Types
A rule assertion type can be satisfied or unsatisfied. If not defined the default is satisfied. This indicates if a boolean result of true will return results. The satisfied keyword returns an assertion if the result is true. The key word is used after the rule name. In this case abc.0001.

    assert abc.0001 satisfied
    $severity = “error”;
    {@)#nonnegitem < 0
message
"The value of {$nonnegitem.concept} with a value of {$nonnegitem} is less than zero. Please enter a positive amount"
severity

This will return a message if the fact is less than zero.

The unsatisfied keyword is the opposite of satisfied. The following rule will return a result if the fact is greater than zero.

assert abc.0001 unsatisfied
$severity = “error”;
{@}#negativeitem <= 0

message
"The value of {$negativeitem.concept} with a value of {$negativeitem} is greater than zero. Please enter a negative amount.”
severity

This will return a message if the fact is greater than zero.

In the second case a value of 100 will be returned as false as it is greater than zero. Because the rule is defined as unsatisfied then only facts which fail the expression will be returned.

These keywords cannot be used on a output assertion. If no severity is provided then this defaults to “error” for assertions. Output defaults to a severity of info.

Consistency Assertion (Not Implemented)

A consistency assertion specifies how to determine whether an output fact, produced by the associated formula, is consistent with all aspect matched facts in the input XBRL instance. For example, if a formula $c = $a + $b produced an output fact item eg:c, its consistency assertion could state that the output eg:c must be within a tolerance margin, of the value of a corresponding input fact.

The assertion can state that the output eg:c must be within a tolerance margin, called acceptance radius, of the value of a corresponding input fact. The acceptance radius can be defined as an absolute or proportional amount of the input fact.

To distinguish if the assertion is a consistency assertion the keyword fact is used after the rule name. To evaluate a fact comparison or a consistency assertion in xbrl formula the following syntax is used.

assert abc.0001 fact [@CommonStockSharesOutstanding] absolute 100

{@}@dei:EntityCommonStockSharesOutstanding)#fact1

assert abc.0001 fact [@CommonStockSharesOutstanding] proportional 0.20

{@}@dei:EntityCommonStockSharesOutstanding)#fact1

assert abc.0001 fact [@CommonStockSharesOutstanding] tolerance-for-decimals
In XULE the evaluation of a consistency assertion is not strict. This means that if the input value is not in the instance then no result is returned.

**Rule Output**

**Output Attributes**

The output of XULE allows for additional attributes to be associated with the output of a rule. This is useful for classifying rule types for applications using XULE. In addition this can allow for refinement on severity levels or additional details that you would otherwise have to extract from a message string.

Rule output attributes are assigned as part of a set of rules, different attributes cannot be defined on a rule by rule basis. Output attributes are defined using the output-attribute qualifier in the file. See example below.

```
output-attribute concept
This defines an attribute called concept. This can then be defined at the bottom of the rule.
```

Xule includes predefined output attributes called *message*, *rule-suffix*, *rule-focus* and *severity* that do not have to be defined as an output attributes. The severity attribute will default to error if not defined. Severity has the following enumerated values: error, warning, ok.\(^7\)

The format for a rule that checks for any negative items in an instance with a concept would be as follows:

```
assert abc.0001 satisfied

If {@ where $fact < -1000000}#nonnegitem
$severity = "error";
true
else If {@ where $fact < 0}#nonnegitem
$severity = "warning";
true
else
false

message
"The value of {$nonnegitem.concept} with a value of {$nonnegitem} is less than zero. Please enter a positive amount"

concept
$nonnegitem.concept.name

severity
$severity
```

\(^7\) For consistency with the XBRL formula specification.

---

7 For consistency with the XBRL formula specification.
Rule-focus $nonnegitem

Passing Variables to Rule Output
Variables can be used in an output string. Variables are indicated by using a $ sign enclosed in curly brackets to indicate that it is a variable. For example:

“The value of {\$nonnegitem.concept} with a value of {\$nonnegitem} is less than zero. Please enter a positive amount”

Any expression defined in the rule can be expressed in the output returned from the rule. In addition properties of a variable can also be returned using the dot notation. In the above message to return the local name of a concept would be represented as follows:

“The value of {\$nonnegitem.concept.name.local-name} with a value of {\$nonnegitem} is less than zero. Please enter a positive amount”

Tagging Values for use in Output
In many cases the result of a factset needs to be passed to the rule output. Rather than forcing every factset or property to be defined as a variable these items can be tagged and then used in the output. This means the output can access a variable defined in the rule or a tag defined in the rule. An item can be tagged using the # symbol. In the example the tag #nonnegitem is used to tag those items that are less than zero. The # is used immediately after the factset. The tag also has properties that can be referenced in the message. Below the element name and balance type of the item can be displayed in the message by representing the properties of the tag.

assert abc.0001 satisfied
{@ where $fact < 0)#nonnegitem

message
“The value of {\$nonnegitem.concept} with a balance type of {\$nonnegitem.concept.balance} is less than zero. Please enter a positive amount”

Note that tag names cannot contain a period.

Fact Properties and Rule Focus
The output from xule will also send the properties of a fact object or concept in the message. This will normally default to the first item in a list if it is more than one. The xule syntax allows you to control how these properties are returned by providing a hint to the processor. Within the grammar you can define the object for which the properties are returned. This is done using the keyword ‘rule-focus’ as a result name. The rule focus must evaluate to a fact or a concept. For Example:

assert a satisfied
$liab = @Liabilities;
$e = @Equity;
($liab + $e) < 0

message
"Liabilities with value {$liab} plus Equity with value {$e} is less than zero"

rule-focus $e

If rule-focus is not present, than the first evaluated fact will be used for the properties returned. So if the example did not have rule-focus, the properties returned would have been Liabilities instead of Other Assets.

The rule focus can also return a set of properties if the rule-focus is passed a list.

Rule Value
The value of the rule or output expression is recorded in a variable called $rule-value. If a rule evaluates to a boolean of true, then the value of $rule-value will be recorded as “true”. $rule-value can be used in the message to indicate the result of an output or the result of running the rule.

For example the result of an output value can be reported using rule-value.

output add-two-numbers
   @assets#a + @liabilities#b

message
"{$a} + {$b} = {$rule-value}"

Labels in Messages
In many cases labels of elements need to be returned in output from the rule to make them easier to read. To access the label of a concept the label object which is associated with a concept is used. So if the rule is checking if a value is negative the concept can be returned and the associated label in the dts can also be returned.

$Assets = {@concept = Assets where $fact < 0}

This rule will return all Assets where the value is less than zero.

The output can then be expressed using the label as follows:

message
"The value of the concept {$Assets.concept.label.text} is less than zero"

Special Characters in Messages
When using quotes in a message the rule has to have the quotes escaped. This is because a message is represented as a string encapsulated with quotes. To escape a quote or other control character such as a tab or a return the backslash is used.
<table>
<thead>
<tr>
<th>Character</th>
<th>Defined as</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>tab</td>
<td>\t</td>
</tr>
<tr>
<td>return</td>
<td>\n</td>
</tr>
<tr>
<td>Curly Bracket</td>
<td>{</td>
</tr>
</tbody>
</table>

### Rule Name Prefix

In many cases rule names have a common prefix naming convention. Xule allows a common prefix to be defined for all rules below the rule name prefix declaration. In the example below the keyword `rule-name-prefix` is used to set a standard prefix for all rules below the declaration.

```plaintext
rule-name-prefix my_rules
assert a_rule satisfied
  $liab = @Liabilities;
  $e = @Equity;
  ($liab + $e) < 0
message
  "Liabilities with value {$liab} plus Equity with value {$e} is less than zero"
rule-focus $e
```

*When the results of the rule are returned the rule will be referenced as my_rules.a_rule.*

Between the rule prefix and the rule name a period is added to distinguish the prefix from the rule. The period is the default separator. This however can be changed by using the keyword `rule-name-separator`

A rule name prefix cannot be defined as a variable.

### Rule Name Separator

The `rule-name-separator` declaration is used to change the separator added by rule-name-prefix and rule-suffix. To change the separator from the default period to a colon the following declaration is made:

```plaintext
rule-name-separator : 
```

*Changes the rule name separator for a period to a colon. From the example above the rule will be referenced as my_rules:a_rule.*

### Rule Suffix

The `rule-suffix` keyword can be used to add a suffix to a rule. Unlike the `rule-name-prefix` keyword, the rule-suffix applies to a specific rule. The rule-suffix can be passed as a variable which allows rule names to be defined at run-time. This means a single rule can be defined that can generate rule results with different rule numbers depending on the input or processing within the rule. The example below shows this.
rule-name-prefix my_rules
assert a_rule satisfied
$e = @Equity;
if $e < 0
    $suffix = "equity_less_zero"
else
    $suffix = "equity_greater_equal_zero"
    $liab = @Liabilities;
($liab + $e) < 0
message
"Liabilities with value {$liab} plus Equity with value {$e} is less than zero"
rule-focus $e
rule-suffix $suffix

When the results of the rule are returned the rule will be referenced as my_rules.a_rule.equity_less_zero or my_rules.a_rule.equity_greater_equal_zero depending on the data in the XBRL instance.

Between the rule suffix and the rule name the default period is added to distinguish the suffix from the rule. This can be changed using the rule name separator declaration.

Iterations and Alignments

The XULE processing model can run multiple times for a given rule, in some cases returning a result as a message or returning no message. The rule starts with a single iteration, when a factset is encountered iterations are added for each value of the factset. Iterations are also created when a for loop is encountered. An iteration is created for each loop of the for expression. For example if the value of assets is tested to determine that it is less than zero. If assets is reported for 3 periods the rule will test that assets is less than zero for all three periods. If assets is also reported in multiple currencies each of these currency disclosures will be tested by the same rule. To do this the processor looks at the number of times a value appears in the instance and executes the rule for each occurrence of the fact. If the fact does not exist in the instance then no iterations are created for each fact and the rule will not produce a message.

For example the following says print a message if BelowMarketLeaseAcquired doesn't exist in the instance.

output exits_iteration
If (exists (@BelowMarketLeaseAcquired))
"This item exists and this string is output for every occurrence of the fact."
else
"If BelowMarketLeaseAcquired does not exist no iteration occurred and this string will never be reported."

The problem with this rule is that the else condition is never reported because there is no iteration to create the message. Because the message is created for each occurrence of the fact and the fact does not exist then no message is output. The factset is aligned and matches based on alignment and if no alignment matches no iteration of the rule occurs. If the alignment
is removed from the factset then a single iteration will occur. This is because when a rule executes a single iteration will occur if no factsets or for loops are encountered. The alignment is removed from a factset using the `covered` keyword. The following output will produce a message:

```
output exists_iteration
If (exists ({covered @BelowMarketLeaseAcquired}))
  "This item exists and this string is output for every occurrence of the fact."
else
  "If the BelowMarketLeaseAcquired does not exist a single iteration occurs because the covered gets a true result and this string will reported once."
```

Multiple Factsets
In the above examples the iteration impacted a single factset. In some cases multiple factsets are compared. For example to expand the above example:

```
output exists_and_missing
if missing({@BelowMarketLeaseAcquired}) AND exists({@Assets})
  "Below Market leases are not in the filing but assets are. This message is reported for the number of times an asset value appears in the filing."
else
  "Below market leases are in the filing. This will appear for the number of occurrences of below market leases."
```

In the case above the number of iterations can vary. If below market leases is missing the number of iterations will be the same as the number of occurrences as assets. This is because it answers the question, how many cases of assets are present where below market leases is not. If below market leases are present for a given alignment, such as 2017, then a false value is returned and the else condition occurs for the number of occurrences of below market leases. The existence of the assets is not even checked because the first condition is false and the entire statement will be false. The XULE processor implements the Lazy AND to reduce processing time. The number of iterations can vary if the rule was written as follows:

```
output exists_and_missing
if exists({@Assets}) AND missing({@BelowMarketLeaseAcquired})
  "Below Market leases are not in the filing but assets are. This message is reported for the number of times an asset value appears in the filing."
else
  "Assets values are not in the filing. This message will appear for the number of occurcences of assets which in this case is 0 which means there will be zero iterations."
```

In the case above if no assets are present there will be zero iterations, whereas in the first case there would be iterations that reflect the number of times that below market leases appeared. Depending on how many error message the user wants returned they need to consider the sequence of their logic statements.
Impact of Syntax on Performance

The way rules are written can have a performance impact and in many cases a rule can execute faster or slower depending on the amount of work that needs to be done by the processor. When writing rules in XULE the following should be considered:

- If a value is a constant always define it as a constant. This allows software to load these first and use them for the execution of every rule without having to recompute them and load into memory each time a rule is run. On server versions of a XULE processor the performance should be vastly improved.

- Secondly, avoid for loops if possible. For loops require heavy processing because of the need to track alignments for all variables for each loop. In addition, it can be hard to predict how many loops may be specific to a given filing. Instead of using a loop try to use the filter operator instead as this is much more efficient. In addition use built in properties and functions that are more efficient than a for loop in XULE, for example the join property.

- Reuse constants across many rules where possible.

- Define your own functions so that variables that have already been calculated can be cached and reused by the processor.

- Try and avoid a where clause in a factset when a filter can be used instead. I.e 
  \{@Assets\} is more efficient than \{@ where $fact.concept == Assets\}.

- Consider the use of lazy AND. The sooner you can find a false for an AND the more efficient the processor can be.
Appendix 1 - XULE Operators

Operators

Operators are used between variables and values in XULE. For example the where filter supports standard comparison operators. The operators defined below are consistent across the entire xule syntax.

<table>
<thead>
<tr>
<th>Boolean Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>==</td>
<td>Equivalent to. Can be used on a set or a list</td>
</tr>
<tr>
<td>!=</td>
<td>Not equivalent to. Can be used on a set or list</td>
</tr>
<tr>
<td>and</td>
<td>Logical AND. This is lazy</td>
</tr>
<tr>
<td>or</td>
<td>Logical OR This is lazy</td>
</tr>
<tr>
<td>in</td>
<td>Evaluates if an item is in a set.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Numeric Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition of numbers, strings, lists and sets.</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction, unary</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>&lt;+&gt;</td>
<td>Addition that will occur if both left and right side present</td>
</tr>
<tr>
<td>&lt;+&gt;</td>
<td>Addition will only occur if left side is present.</td>
</tr>
<tr>
<td>+&gt;</td>
<td>Addition will only occur if right side is present.</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Subtraction that will only occur if both values are present on either side of the subtraction.</td>
</tr>
<tr>
<td>&lt;-</td>
<td>Subtraction will only occur if left side is present.</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Subtraction will only occur if right side is present.</td>
</tr>
<tr>
<td>power()</td>
<td>Power. This is a property and not the operator ^ as this is used for symmetric difference.</td>
</tr>
</tbody>
</table>

Set Operators
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Union of 2 sets</td>
</tr>
<tr>
<td>&amp;, intersect</td>
<td>Intersection of 2 sets</td>
</tr>
<tr>
<td>-</td>
<td>Difference of 2 sets</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Test if a set is a subset</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Test if a set is a superset</td>
</tr>
<tr>
<td>[]</td>
<td>Return value of an index</td>
</tr>
</tbody>
</table>

### Order of Precedence

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parenthesis</td>
</tr>
<tr>
<td>#</td>
<td>Tagging</td>
</tr>
<tr>
<td>[ ]</td>
<td>Index</td>
</tr>
<tr>
<td>.</td>
<td>Property expression</td>
</tr>
<tr>
<td>+, -</td>
<td>Unary, union</td>
</tr>
<tr>
<td>*, /</td>
<td>Multiplication and Division</td>
</tr>
<tr>
<td>+, -</td>
<td>Addition and subtraction</td>
</tr>
<tr>
<td>&amp;, intersect</td>
<td>Set intersections</td>
</tr>
<tr>
<td>^</td>
<td>Symmetric Difference</td>
</tr>
<tr>
<td>&gt;, &lt;, &gt;=, &lt;=, in, not in, ==, !=</td>
<td>Comparison Operators</td>
</tr>
<tr>
<td>not</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix 2

## Version 1.1

Where possible we have identified specific requirements (highlighted in grey within the document) that have not been implemented in the reference implementation. These will be implemented in version 1.1.

In addition there is additional functionality we know needs to be addressed in Version 1.1 that is not yet documented in this document (As it conflicts with the current implementation) or is not implemented in the reference implementation. These requirements are summarized below:

<table>
<thead>
<tr>
<th>No</th>
<th>Feature Name</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Footnotes of a Fact</td>
<td>Add support for returning the footnotes of a fact. A footnote property will be added that will allow specific footnotes to be returned. I.e. $fact.footnote(role)</td>
</tr>
<tr>
<td>2</td>
<td>Date Casting</td>
<td>For business users it would be more straightforward if they can directly write @period.start='2016-01-01' without the date constructor. @period.start and @period.end always returns an instant, so an automatic type cast is possible as it is the case for units, e.g. @unit=iso417:USD</td>
</tr>
<tr>
<td>3</td>
<td>Include/Import</td>
<td>Currently all xule files are compiled from a given directory. However, the user should be able to include specific XULE files from another file location using an include statement. Should also have the ability to import compiled XULE files such as a compiled group of XULE functions.</td>
</tr>
<tr>
<td>4</td>
<td>Namespace declaration as an expression</td>
<td>Namespaces are currently declared as a string and cannot be an expression. However in many cases these need to be evaluated at run time as the namespaces in an instance may not be known in advance. Namespaces should be able to be defined as an expression. This is a major change and will mean old rule would not be able to be run on a new processor implementing this functionality.</td>
</tr>
<tr>
<td>5</td>
<td>Update return types from a navigate</td>
<td>The return types from the navigate expression should be expressions. This means that the properties cannot be accessed directly without some form of lookup. For example in the return options to get the source name, a return option of source-name is used rather than relationship.source.name.</td>
</tr>
<tr>
<td>6</td>
<td>Rule Link</td>
<td>Add a standard output attribute called rule link that links to the documentation of the actual rule. Such as <a href="https://xbrl.us/data-">https://xbrl.us/data-</a></td>
</tr>
<tr>
<td>Parameters</td>
<td>Should be able to pass parameters to the XULE processor that match the parameters in the XULE syntax. For example: person running the rule passes the parameter called user with a value such as user=jblow. In message can then output who ran the rule as the parameter is defined {user}. Should also have default value option.</td>
<td></td>
</tr>
</tbody>
</table>
Appendix 3

Examples

DQC.US.0001.51
/** Define the base taxonomy as a constant**/
/** Determine the members on the axis in the taxonomy using the definition tree **/
constant $member_USGAAP_fair_value = navigate dimensions dimension-member descendants from FairValueByFairValueHierarchyLevelAxis taxonomy $us-gaap returns set (target-name))
/** Define a function to identify a concept as an extension**/
constant $extension_ns = first(filter taxonomy().concepts where is_extension($item)).name.namespace-uri
function is_extension($test_extension)
  $test_extension not in $us-gaap.concepts
assert dqc.us.0001.51 satisfied
/** Define allowable extensions with the qname function**/
$allowable_extensions = set(qname($extension_ns,'FairValueInputsLevel1AndLevel2Member'), qname($extension_ns,'FairValueInputsLevel2AndLevel3Member'), qname($extension_ns,'InvestmentsNetAssetValueMember'))
/** Union the sets**/
$allowed_members = $allowable_extensions + $member_USGAAP_fair_value;
/** Evaluate if facts exists with the unallowable members **/
exists (@FairValueByFairValueHierarchyLevelAxis = * as $FV where ($allowed_members not in $FV#member))#fact
message "The concept {$fact.concept.name.local-name} with a value of {$fact} is dimensionally qualified with the FairValueByFairValueHierarchyLevelAxis and the unallowable member {$member.local-name}. The filer should use members from the US GAAP taxonomy that are children of the FairValueByFairValueHierarchyLevelAxis axis or the allowable extensions of: {$allowable_extensions.join(', ')}.
The properties of the fact for {$fact.concept.name.local-name} are:
  Period: {$fact.period}
  Dimensions: {$fact.dimensions.join(', '=')}
  Unit: {$fact.unit}"
severity error
DQC.US.0004.16

This rule tests that the value reported for the element Assets equals the value reported for the element Liabilities and Equity. The rule allows a tolerance for rounding between the values based on the scale of the values. For example, if the values are reported in millions, the rounding tolerance would be $2 million.

/** Define a function that works out the tolerance between 2 values with different decimals. The decimal tolerance factor for this rule takes the value of 2**/

function Tolerance_For_Decimals($left, $right, $decimal_tolerance_factor)
    $tolerance1 = if ($left.decimals < $right.decimals)
        $left.decimals
    else $right.decimals;
    $tolerance2 = if ($tolerance1 == inf)
        0
    else (10^(-1 * $tolerance1)) * $decimal_tolerance_factor;
    if (abs(round($left,$tolerance1) - round($right,$tolerance1)) > $tolerance2)
        true
    else false

assert DQC.US.0004.16 satisfied
$Assets = {@Assets};
$LiabilitiesAndStockholdersEquity = {@LiabilitiesAndStockholdersEquity};
Tolerance_For_Decimals($Assets, $LiabilitiesAndStockholdersEquity, 2)

message
"{$Assets.concept.label.text} with a value of {$Assets} is not equal to the total of
{$LiabilitiesAndStockholdersEquity.concept.label.text} with a value of
{$LiabilitiesAndStockholdersEquity}. These values should be equal.
The properties of this {$Assets.concept} fact are:
  Period :{$Assets.period}
  Dimensions :{$Assets.dimensions.join(’=’,’
  Unit :{$Assets.unit}
  Rule Version :{$ruleVersion}";

severity
error

DQC.US.0044.6834

assert dqc.us.0044.6834 satisfied

$Accrual_items_in_cashflows  = navigate summation-item descendants from
    list(NetCashProvidedByUsedInFinancingActivities,
NetCashProvidedByUsedInFinancingActivitiesContinuingOperations) where $relationship.target.name in $Accrual_Items;

$Accrual_fact = {concept in $Accrual_items_in_cashflows};

$Accrual_fact != 0

message
"The concept {$accrual_fact.concept} with a value of {$accrual_fact} is an accrual-based item in the US GAAP taxonomy that is included in the sum of cash provided by (used in) financing activities in the cash flows of the extension taxonomy.

The properties of this {$accrual_fact.concept} fact are:
Period :{$accrual_fact.period}
Dimensions : {$accrual_fact.dimensions.join(‘,’‘)}
Unit : {$accrual_fact.unit}
Rule Version : {$ruleVersion}"

severity
error

/** This rule uses a constant to represent the accrual items in the taxonomy. These are determined as follows **/

constant $Accrual_Items =
    (navigate summation-item descendants from
    list(ComprehensiveIncomeNetOfTax, NetIncomeLoss, NetIncomeLossAvailableToCommonStockholdersBasic, NetIncomeLossAvailableToCommonStockholdersDiluted, IncomeLossIncludingPortionAttributableToNoncontrollingInterest, IncomeLossAttributableToParent, NetIncomeLossAllocatedToGeneralPartners, NetIncomeLossAllocatedToLimitedPartners, StockholdersEquityPeriodIncreaseDecrease, PartnersCapitalAccountPeriodIncreaseDecrease) taxonomy $us-gaap where $relationship.target.is-monetary == true returns set (target-name)) +
    (navigate parent-child descendants from list((IncomeStatementAbstract, StatementOfIncomeAndComprehensiveIncomeAbstract, StatementOfStockholdersEquityAbstract, StatementOfPartnersCapitalAbstract) taxonomy $us-gaap where $relationship.target.is-monetary == true returns set (target-name))

assert dqc.us.0045.6835 satisfied

/** This rule identifies elements in the investing section of the cash flow calculation that are operating items. It uses a function to build a list of misplaced items by navigating the calculation tree of the filing dts and the calculation of the US-GAAP taxonomy. **/

$misplaced_concept  =
compare_baseCalc_to_extensionCalc(NetCashProvidedByUsedInOperatingActivitiesContinuingOperations,
NetCashProvidedByUsedInInvestingActivitiesContinuingOperations);

for $x in $misplaced_concept true

message
"The concept {$x} appears in the investing cash flows of the company's cash flow statement. {$x} is an operating item and it is expected that this item would only appear in the cash flow generated from operating activities. Please review the calculations defined for the cash flow statement to determine that the correct element has been used for this item.
Rule Element Id:6835
Rule Version: 5.0.0"

severity
error

function compare_baseCalc_to_extensionCalc($baseConcept, $extensionConcept)

$extensionNames = navigate summation-item descendants from ($extensionConcept);

navigate summation-item descendants from ($baseConcept) taxonomy $us-gaap where $relationship.target in $extensionNames and not ($relationship.target.name in $cash_flow_exceptions)

constant $cash_flow_exceptions =
set(ProceedsFromDepositsWithOtherInstitutions, InterestPaidCapitalized, ProceedsFromFederalHomeLoanBankAdvances, PaymentsForFederalHomeLoanBankAdvances, ProceedsFromPaymentsForTradingSecurities, PaymentsForDepositsWithOtherInstitutions, ProceedsFromPaymentsForInSecuritiesSoldUnderAgreementsToRepurchase, IncreaseDecreaseInFederalFundsPurchasedAndSecuritiesSoldUnderAgreementsToRepurchaseNet, IncreaseDecreaseInRestrictedCash, IncreaseDecreaseOfRestrictedInvestments)

DQC.US.0046.6839

assert DQC.US.0046.6839 satisfied

EffectOfExchangeRateOnCashAndCashEquivalents in navigate summation-item descendants from (NetCashProvidedByUsedInContinuingOperations) returns set (target-name)

message
"The element NetCashProvidedByUsedInContinuingOperations (Net Cash Provided by (Used in) Continuing Operations) does not include EffectOfExchangeRateOnCashAndCashEquivalents (Effect of Exchange Rate on Cash and Cash Equivalents) as defined in the US GAAP Taxonomy.

However, in the companies extension taxonomy NetCashProvidedByUsedInContinuingOperations includes EffectOfExchangeRateOnCashAndCashEquivalents as a summation-child. Consider using either CashAndCashEquivalentsPeriodIncreaseDecrease or CashPeriodIncreaseDecrease, instead of NetCashProvidedByUsedInContinuingOperations. \nEffectOfExchangeRateOnCashAndCashEquivalents {NetCashProvidedByUsedInContinuingOperations}\n
Rule Element Id:6839
Rule Version: 5.0.0"

severity error
assert dqc.us.0047.7481 satisfied

for ( $cashOperating in 
set(NetCashProvidedByUsedInOperatingActivitiesContinuingOperations,NetCashProvidedByUsedInOperatingActivities))

    $misplaced_concept = navigate summation-item descendants from ($cashOperating)
    /** tests if the element has no balance type **/
    where $relationship.target.balance == none
    /** Excludes as this is a known exception **/
    and $relationship.target.name  != NetCashProvidedByUsedInOperatingActivitiesContinuingOperations
    /** Does not flag an error if the element is an extension element **/
    and is_base($relationship.target);

    for $x in $misplaced_concept true

message
"In the company's extension taxonomy the concept {taxonomy().concept($cashOperating).label.text} includes {$misplaced_concept} as a summation-child. The concept {$misplaced_concept} should not appear as a child of {$cashOperating} because it does not have a balance type. Increase (Decrease) items without balance attributes are used in a roll forward and should not be used in the cash flow statement as they represent the impact on the balance sheet item which is the opposite of the impact on cash."

Rule Element Id:7481
Rule Version: 5.0"

Element_id  7481

severity  error
assert dqc.us.0049.7483 satisfied

$must_be_present_concepts = set('CashAndCashEquivalentsPeriodIncreaseDecrease', 'CashPeriodIncreaseDecrease', 'CashAndCashEquivalentsPeriodIncreaseDecreaseExcludingExchangeRateEffect', 'CashCashEquivalentsRestrictedCashandRestrictedCashEquivalentsPeriodIncreaseDecreaseIncludingExchangeRateEffect', 'CashCashEquivalentsRestrictedCashandRestrictedCashEquivalentsPeriodIncreaseDecreaseExcludingExchangeRateEffect')

$nonallowed_root_elements = filter (navigate summation-item descendants from CashAndCashEquivalentsPeriodIncreaseDecreaseExcludingExchangeRateEffect taxonomy $us-gaap returns set (target-name)) returns $item.local-name

$networkPresRole = filter taxonomy().networks(parent-child) where ($item.concept-names.contains(StatementOfCashFlowsAbstract) or $item.role.uri.lower-case.contains('cashflow')) and $item.role.description.contains('- Statement ') and not $item.role.uri.lower-case.contains('parenthetical') returns $item.role;

/* This uses navigation. It finds the root relationships and returns the networks. Since it returns a set, the dups will be eliminated. */

for ($calcNetwork in filter taxonomy().networks(summation-item) where $item.role in $networkPresRole)
    $roots = set(
        for $root in $calcNetwork.roots
            if ($root.name.namespace-uri != $extension_ns and $root.name != NoncashOrPartNoncashAcquisitionNetNonmonetaryAssetsAcquiredLiabilitiesAssumed 1)
                $root.name.local-name
            else
                none)

$root_string = $roots.join(', ');

($roots intersect $must_be_present_concepts).length > 0 and ($roots intersect $nonallowed_root_elements).length > 0

message "The following elements {$root_string} are parent (root) elements defined in the calculation relationship for the cash flow statement using the group {$calcNetwork.role.uri}. The cash flow statement should only have one calculation parent for durational concepts representing the increase or decrease in cash during the period. If the company has adopted ASU-2016-18 then the root element used to represent the
aggregate change in cash should be the element
CashCashEquivalentsRestrictedCashAndRestrictedCashEquivalentsPeriodIncreaseDecreaseIncludingExchangeRateEffect. If the company specifically excludes the exchange rate effect from the total then the element
CashCashEquivalentsRestrictedCashAndRestrictedCashEquivalentsPeriodIncreaseDecreaseExcludingExchangeRateEffect should be used.

Rule Element Id:7483
Rule Version: 5.0"

Element_id 7483

severity error
Using External Data

**assert values_compare satisfied**
/** Shows how durational data in an instance can be compared to a previous filing using an api that returns the previous filing as json from an XBRL API. An API could also be used to get data such as stock quotes or exchange rates that could be used in a rule **/

/** Returns an quarterly Apple Inc. filing in an XBRL OIM format **/

/** Get durational data from the current filing **/
$instance_element = [@concept.period-type = duration @concept.is-numeric = true];
$eop = $instance_element.period.end + time-span("P0D");
$sop = $instance_element.period.start + time-span("P0D");
$element_name = $instance_element.name.local-name;

/** Get data from prior filing **/
$old_value = filter $prior_filing['facts'] where
    $concept_name = $item['aspects'][xbrl:concept];
    $num_dims = length($item['aspects']);
    $concept_local-name = $concept_name.substring($concept_name.index-of(':')+1);
    $concept_local-name == $element_name and date($item['aspects'][xbrl:periodEnd]) == $eop and date($item['aspects'][xbrl:periodStart]) == $sop and $num_dims == 5 and $item['value'].number != $instance_element returns $item['value'];

length($old_value) > 0

message
"The fact with a value of {$instance_element} for the element {$instance_element.name.local-name} does not match the prior filing value of {$old_value[1].number}.

The properties of this {$instance_element.concept} fact are:
Period :{$instance_element.period}
Dimensions :{$instance_element.dimensions.join(',','=')}
Unit :{$instance_element.unit}"

severity error