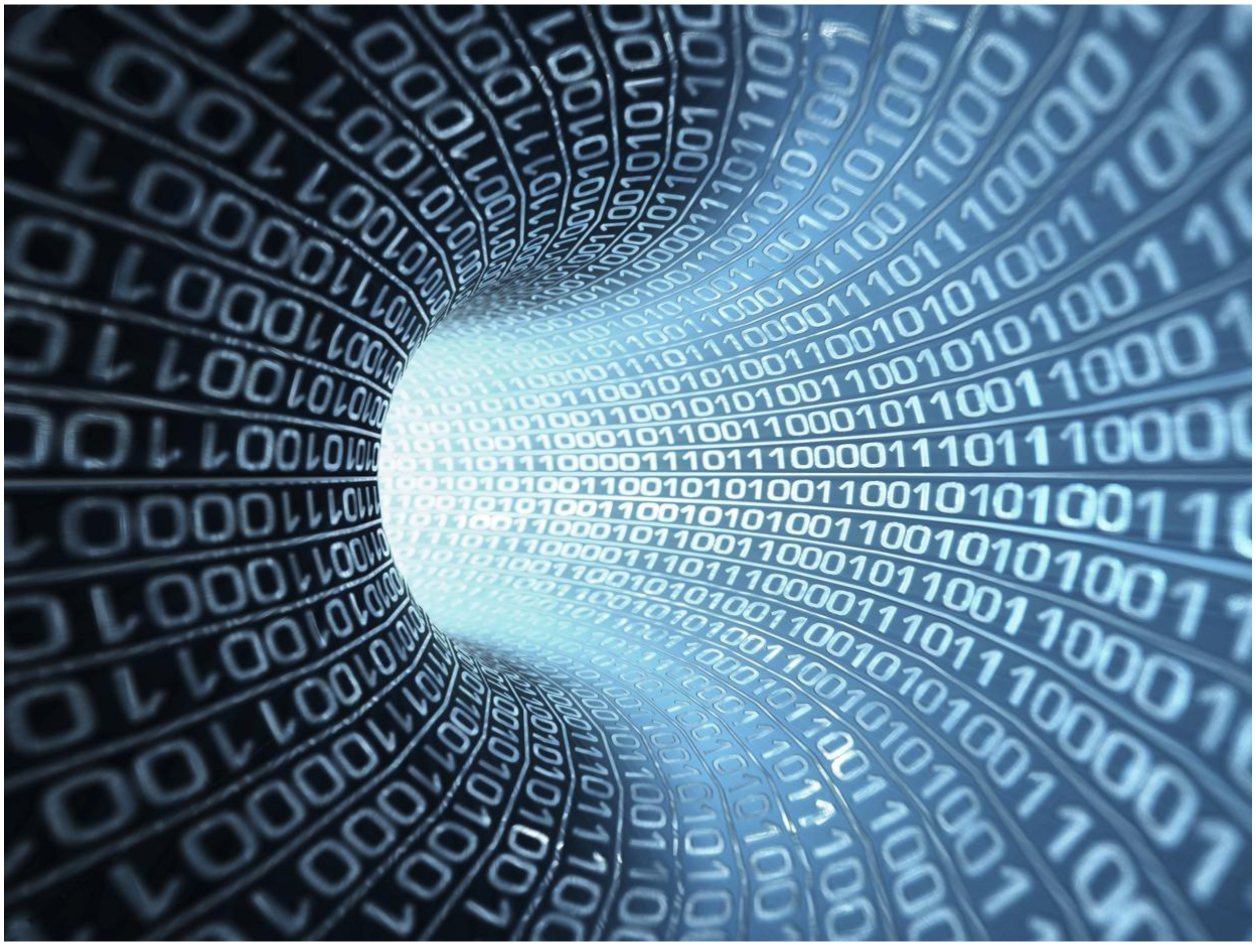
XULE Language Syntax for XBRL - V1.2.2

Version 1.2.2



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 fact filtering and the second is taxonomy navigation.

Instance Document Filtering

Every fact has dimensions associated with it that define what the fact is, how it is disaggregated, the period of time at which the fact was measured,and the duration over which it occurred. Fact query filtering is the action of extracting data from an XBRL instance based on these features 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 a collection, basic manipulation can be performed on the data such as a union or intersection or complement. A rule can filter the facts in an instance not only using the dimensions of a fact, but also the properties of the core or taxonomy defined dimensions. 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 fact queries. 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 facts returned from fact queries to be aligned. XULE also allows any dimension 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 fact query 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 is a short form of us-gaap:Assets. The Asset concept has the default namespace (in this case us-gaap) applied to it. If you get the qname of Assets you can access the namespace and local-name but not its label, balance, period type etc. To obtain 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(). For example, to obtain the concepts assets used by the instance being processed use the function:

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

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)
```

Here 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

Fact queries 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 fact query is encountered, iterations are added for each value of the fact query. Iterations are also created when a for loop is encountered. An iteration is created for each loop of the for expression. For example, the value of assets is tested to determine if it is less than zero. If assets are reported for 3 periods, the rule will test that assets are less than zero for all three periods or 3 iterations. If assets are 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

Fact Query

A fact query in XULE returns a group of XBRL facts. The information about a fact not only includes the values of a fact but all the “dimensions” (defining characteristics such as units, time period, etc.) of that fact. This allows the properties of each specific fact to be queried.

The curly bracket notation is used to explicitly state in the XULE language that something is a dimension filter of a fact query. The **@** is used to indicate a feature of a fact, and all facts that have this feature 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

Fact Query 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 fact window.

[**@**] 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.

Fact Query Filters

The fact query body consists of filters that allow filtering of facts in an instance that returns facts. Without filters, all XBRL facts available in the instance would be returned by the Xule processor. The filters work by explicitly stating the dimensions that are required. The following are dimension filters that can be used:

- concept
- period
- unit
- dimension (actual dimension name, for example LegalEntityAxis)
- dimensions
- entity
- cube
- instance

In addition, the value of a fact can be filtered using a **where** filter, which allows more refined filtering of the facts that has already been filtered by dimensions, 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 dimension for the facts numerical value. The where filter is described in greater detail below.

To indicate that a dimension filter is being used in a fact query, XULE needs to know that it is a dimension. The **@** sign is used to indicate dimensions in a fact query. For example:

```
{@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:

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

This can also be defined in the short form:

```
{@Assets}
```

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

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

Filter Name	Xule Syntax	Filter by
Name	Default or name	QName of fact element. May have multiple QName choices or an expression. Concepts can also be filtered by a concept object. `{@concept = Assets}`
Local Name	local-name	Local name of fact element. `{@concept.local-name = 'Assets'}`

Period type	period-type	Concept schema declared period type, for example, instant or duration.
		`{@concept.period-type = instant}`
		`{@concept.period-type = duration}`

Balance	balance	Concept schema declared balance, for example, credit, debit, * , none. (* will return all concepts with either a balance of credit or debit)
		`{@concept.balance = debit}`
		`{@concept.balance = credit}`
		`{@concept.balance = none}`
		`{@concept.balance = *}`

Custom attribute	attribute(name)	Concept schema element declaration custom attribute value, for example, Value, * , none (* will return all concepts with any attribute value associated with the concept) `{@concept.attribute(someattr) = *}`
Data Type	data-type	Concept schema declared data type. `{@concept.data-type = xbrli:monetaryItemType}`
	base-type	Base XBRL type that the data type is derived from. `{@concept.base-type = xbrli:stringItemType}`
	data-type-ancestry	Returns a list of types in order of the type ancestry.
	has-enumerations	Values can be true or false {@concept.has-enumerations = true}
	is-monetary	Values can be true or false (Defaults to true)

		`{@concept.is-monetary = true}`
		Can express as:
		`{@concept.is-monetary}`

	is-numeric	Values can be true or false (Defaults to true)
		`{@concept.is-numeric = true}`
		Can express as:
		`{@concept.is-numeric}`

Substitution group	substitution	Concept schema declared substitution group. `{@concept.substitution = xbrli:item}`
	substitution-ancestry	Returns a list of qnames of substitution groups in order of the ancestry.
Namespace	namespace-uri	URI of concept namespace. `{@concept.namespace-uri = 'http://xbrl.sec.gov/dei/2014-01-31'}`

```
        {@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 dimension with the following properties:

Filter Name	Xule Syntax	Filter by
Period	default	A period expression i.e. forever, date('2016-12-31'), duration('2016-01-01','2016-12-31'), *
Period start	start	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.
Period end	end	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.
Period duration	days	Specify duration of time in days to return values that match that period.

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 returns all duration facts that are 90 days in length.

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

The period 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 = 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:

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

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 including those that do not have a unit and will take the unit out of dimension alignment - see Nested Alignment.

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

Entity Filter

Filter Name	Xule Syntax	Filter by
Entity	default	entity(scheme, identifier), * `@entity = entity('http://www.xxx.com','0000320193')` `@entity = *`
Scheme	scheme	Pass the uri of the scheme. I.e. `@entity.scheme = 'xxx'`
Identifier	id	Pass the id of the entity I.e. `@entity.id = 'xxx'`

Examples

```
{@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 with entities in the instance. In XBRL JSON where an entity is not required, these facts will not be returned.

Taxonomy Defined Dimension Filter

Filter Name	Xule Syntax	Filter by

Dimension member	(dimension member name) Qname	Value of the dimension member which is either a qname, a variable, *, none or empty.
		`@dei:LegalEntityAxis = ParentCompanyMember`
		`@dei:LegalEntityAxis = *`
		`@dei:LegalEntityAxis = none`

No additional dimensions	[ ], dimensions() != *	Limits the facts to those that do not have the dimension specified in the fact query.
		`{@dei:LegalEntityAxis != *}`
		
		`[]`

Dictionary of dimension member combinations	dict	Allows the filtering by a combination of dictionary member pairs. Uses the keyword **dimensions** `{@dimensions = $dictA}`

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

```
    {@dei:LegalEntityAxis = *}
```
This will return all the facts with the legal entity axis, except for facts with no dimensions (default values). It also returns facts 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 dimensions.

```
{@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 not 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(RevenueRemainingPerformanceObligationExpectedTimingOfSatisfa
ctionStartDateAxis) in $fact.dimensions-typed and ($fact.dimensions-
typed)[taxonomy().concept(RevenueRemainingPerformanceObligationExpectedTimingOfS
atisfactionStartDateAxis)] == none}
```

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

```
{@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.

In some cases the dimensions to be filtered on are not known until runtime. The value of a dimension can be passed as a variable if the value of the dimension or member is not known in advance.

```
{@concept = Assets @$dimension = $member}
```

This will return all facts for Assets that have the dimension and member represented by the variables $dimension and $member.

/** Added in version 2.2.1 **/

In those cases where the number of and nature of the taxonomy defined dimensions is not known in advance a dictionary of dimension member pairs can be passed to the filter using the dimensions filter.

```
$Assets = {@Assets}.dimensions
{@concept = Liabilities @dimensions = $Assets}
```

This will return all facts for Liabilities that have the same taxonomy defined dimensions as the fact value for $Assets. These dimensions will be taken out of alignment. It will also select facts that have the same dimensions as $Assets and any additional dimensions.

To keep values in alignment and use the dimensions filter as a method to filter facts and maintain alignment a `@@` can be used in front of the taxonomy defined dimensions filter.

```
$Assets = {@Assets}.dimensions
{@concept = Liabilities @@dimensions = $Assets}
```

The dimensions filter only applies to taxonomy defined dimensions and does not include the period, concept, entity or unit dimensions. The dimensions filter is overridden by any explicit taxonomy defined dimension filters defined in the fact query expression that ***conflict*** with the values defined in the dimensions filter. Only 1 taxonomy defined dimension filter can be defined in a fact query.

To cover all dimensions in a manner similar to the covered-dims keyword the taxonomy defined dimension filter with no dictionary value can be used.

```
{@concept = Liabilities @dimensions}
```

This is equivalent to the following:

```
{covered-dims @concept = Liabilities}
```

The taxonomy defined filter can also be defined with a wildcard to select those facts that have taxonomy defined dimensions. This will exclude any facts in the default.

```
{@concept = Liabilities @dimensions=*}
```

This will return all facts for Liabilities that have a dimension and member defined.

The taxonomy defined filter can also be defined with a none value to select those facts that have no taxonomy defined dimensions. This will include only those facts in the default.

```
{@concept = Liabilities @dimensions=none}
```

This will return all facts for Liabilities that have no taxonomy defined dimensions.

This is equivalent to the following:

```
[@concept = Liabilities]
```

Specific taxonomy defined dimensions can be used with the dimensions filter. If they conflict, the named dimension takes precedence. If the dimension filter equals none, an additional filter is not considered a conflict.

```
{@concept = Liabilities @dimensions=none @dei:LegalEntityAxis = *}
```

This will return all facts for Liabilities that only have the LegalEntityAxis as a dimension on the fact. This is the equivalent to:

```
[@concept = Liabilities @dei:LegalEntityAxis = *]
```

Instance Filter

Facts can also be filtered by the XBRL instance reported. XULE filters on the instance of the current filing being evaluated. XULE also allows multiple instances to be processed simultaneously. If no instance filter is defined then all instance documents are filtered on.

To add instances an instance object is defined that references the instance.

```
$wsfs = instance('https://www.sec.gov/edgar/wsfs-20211231.htm')
```

This loads the wsfs instance into the XULE processor.

Fact query selection defaults to the default instance document. To select facts from an alternative instance document the fact query selection criteria must reference the instance object.

```
{@instance = $wsfs}
```
This returns all the facts from the wsfs instance.

Sets of facts can be returned from multiple instances using a list of instance objects.

```
{@instance in list($wsfs, $appl)}
```
This returns all the facts from the wsfs and appl instances.

The default instance can also be specified using instance()

```
$default_instance = instance()
```

To select facts from the default instance the following syntax can be used.

```
      {@instance = instance()}
```
This returns all the facts from the default instance.

The fact query above is the same as the following:

.
```
      {@}
```
This returns all the facts from the default instance if no additional instance objects are defined.

Facts from multiple instances can be returned by including instances in a list.

```
      {@instance in list(instance(),$wsfs,$appl)}
```
This returns all the facts from the default instance and the instance defined by the named variables.

Because there is no defined set of instances the wildcard and not in selectors are invalid. The following examples are **invalid**.

```
      {@instance not in list(instance(),$wsfs,$appl)}

      {@instance = *}
```

Dimension Filter Operators

The following operators may be used with any dimension 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:

```
      @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.

```
      @concept in list(Assets, Liabilities)
```
This will return the facts defined with the concept assets or liabilities.

Combining Dimension Filters

When used together, dimension filters are separated with a space.

```
{@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 data selected 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:

```
{ @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.[1] This can be done with any of the filters.

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

```
{@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 a dimension. Although it is more effective to filter by the dimension; the **where** filter allows further refinement of results *after* all dimension filters have been applied.

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 fact query. 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.

[1] = is used in the same way that "in" would be used in other languages.

To evaluate fact query 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:

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

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

```
{@} < 0
```

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

```
{@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 query 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 a dimension filter.

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

```
{@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 dimensions and properties of the $fact variable are accessed with dot notation. The properties match the dimension filter names:

Dimension and Fact Property	Definition
$fact.concept	Returns the concept of the fact.
$fact.period	Returns the period of the fact.
$fact.unit	Returns the unit of the fact.
$fact.entity	Returns the entity of the fact.
$fact.decimals	Returns the decimals of a fact.
$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.

$fact.dimensions() $fact.dimensions-explicit() $fact.dimensions-typed()	Returns a dictionary of key value pairs of dimension keys and member values.
$fact.id	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.
$fact.inline-scale	Returns the inline scale of the fact if defined in an inline XBRL document. Returns none if no scale is defined.
$fact.sid()	Returns a semantic id of the fact. The semantic id is based on the concept, period, unit, entity, dimensions, decimals and the fact value.

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

```
{@concept = Assets  @dei:LegalEntityAxis =* where $fact > 0 and
$fact.dimension(dei:LegalEntityAxis).name == ParentCompanyMember }
```

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

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

This will return all facts with the ParentMember without having to know that the fact is on the legal entity dimension.[2]

Dimension Alias

XULE allows the setting of aliases for any dimension. 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 “$lea” to represent a set of members on the legal entity axis.

```
{@concept = Assets @dei:LegalEntityAxis=* as $lea }
```

The alias can be used in the where expression:

[2] See Dictionary Operators and Properties dictionary operators later in the document.

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

This could also be written as follows:

```
{@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 Dimensions

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

```
{@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.

```
{@ 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 fact query of values from another fact query of values. Using the dimension filters to extract the values we need to do the calculation.

```
{@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 a dimension 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 return the following results :

1. Assets(WidgetsC0) - Liabilities(WidgetsCo) = She => $20
2. Assets(WidgetsC0) - Liabilities(SnapsCo) = She => $30
3. Assets(SnapsC0) - Liabilities(WidgetsC0) = She => $-10
4. Assets(SnapsC0) - Liabilities(SnapsCo) = She =>$10

When the user actually only wanted this:

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

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

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

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

If a dimension is left out of the fact query expression, that dimension defaults to implicit matching and is matched automatically. For example, these two expressions are identical:

```
{@concept = Assets }
{@concept = Assets @@period @@unit @@entity @@dei:LegalEntityAxis}
```

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

Covering

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

In some cases a user may want to un-align or cover all dimensions of a fact, but is unaware of what the actual dimensions to cover on the fact are. In these cases, the keyword **covered** is used to un-align or cover all dimensions 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 taxonomy defined dimensions and be reported for many periods, it would not align with the EntityFilerCategory. The EntityFilerCategory only exists in a single period and has no taxonomy defined dimensions. By covering the entityFilerCategory in the fact query, it can be lined up and evaluated against each value for assets irrespective of its dimensions.

```
count(list({covered @concept = Assets}))
```

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

```
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 '**covered-dims**' can be used to cover all dimensions in the returned facts.

```
exists({covered-dims @concept = Assets}) and {@concept = Liabilities
where $fact < 0}
```

A variable which represents the result of a fact query cannot be covered. The underlying fact query defining the variable must be covered.

Nested Alignment (Alignment Windows)

In some cases, multiple fact queries need to be aligned using different dimensions when adding, subtracting, multiplying and dividing. Generally, adjusting alignments is done with **@@**. However, when addition and subtraction operations are required between fact queries with *different* *dimensions*, then the facts with the *same dimension 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:

```
{@actualMonthlyPayment} - {@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 dimension 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:

```
{@contractedMonthlyPayment} - {@actualMonthlyPayment} -
{@actualMonthlyReimbursement}
```

This calculation would produce incorrect results because the contracted monthly payment has no period dimension 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.

```
{@contractedMonthlyPayment @period} -  {@actualMonthlyPayment} -
{@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 fact query result:

```
{{@actualMonthlyPayment} - {@actualMonthlyReimbursement}}
```

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

```
{@period {@actualMonthlyPayment} - {@actualMonthlyReimbursement}}
```

The full calculation is expressed as follows:

```
{@contractedMonthlyPayment @period}  - {@period {@actualMonthlyPayment}
- {@actualMonthlyReimbursement}}
```

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

Month	actualMonthlyPayment	actualMonthlyReimbursement	Difference
Jan	***210***	***12***	***198***
Feb	***205***	***11***	***194***
Mar	***212***	***10***	***202***
Apr	***210***	***6***	***204***

Applying the full calculation from above produces the following results:

Month	Actual Monthly Payment	Actual Monthly Reimbursement	Netted Payment	Contracted Amount $200	Difference
Jan	***210***	***12***	***198***	***200***	***2***
Feb	***205***	***11***	***194***	***200***	***6***
Mar	***212***	***10***	***202***	***200***	***-2***
Apr	***210***	***6***	***204***	***200***	***-4***

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 fact query remains in alignment with the period and the outer fact query is not in alignment with the ***@period***.

```
{@period {@actualMonthlyPayment} - {@actualMonthlyReimbursement}}
```

The first calculation is nested within the outer fact query. Nesting fact queries allows dimensions to be removed depending on how many dimensions 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:

DefinedBenefitPlanFairValueOfPlanAssets / DefinedBenefitPlanBenefitObligation = 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, the nested structure can be used:

```
{@unit{@DefinedBenefitPlanFairValueOfPlanAssets} /
{@DefinedBenefitPlanBenefitObligation}}
```

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

```
{@DefinedBenefitPlanFundedPercentage @unit} !=
{@unit
      {@DefinedBenefitPlanFairValueOfPlanAssets}
      /
       {@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 fact query results. A XULE processor can be instructed to exclude nil values from the fact query when processing is initiated. In addition, a fact query 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 @DefinedBenefitPlanFundedPercentage}
```

This will exclude nil facts from the fact query result. The "**nonils**" keyword overrides the processor settings. In addition, the keyword `nils` can be set to include nils. This is used if the processor is set to exclude 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 values 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 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 dimensions. 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.

```
      none + none = skip
```

If the value is none plus a string value the empty none is treated as an empty string.

```
      none + “hello” = “hello”.
```

When adding none to a collection such as a dictionary, set or list the result is the same set, list or dictionary.

```
      set(1,2,3) + none = set(1,2,3)
```

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.

```
      7 / none = skip
```

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

```
      none > 0 = none
```

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.

```
      none >= none = true
```

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

1. `none and TRUE` : SKIP the iteration
2. `none and FALSE` : return FALSE
3. `none and none` : SKIP the iteration
4. `none or TRUE` : return TRUE
5. `none or FALSE` : SKIP the iteration
6. `none or none` : 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.

```
      exists(none) = true
```

[3] See operator section of this document to control bindings if a value is none.

Skipping an Iteration

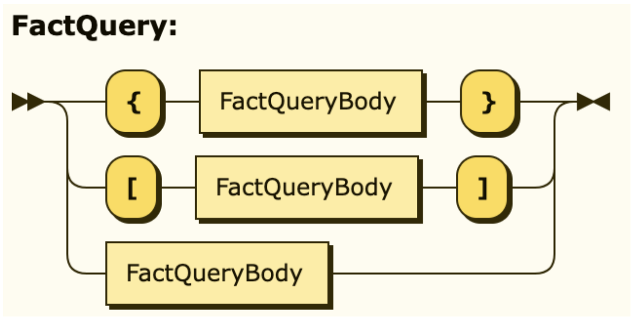
An iteration can be skipped if certain conditions are defined in the rule. This could be done in an if statement for example:

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

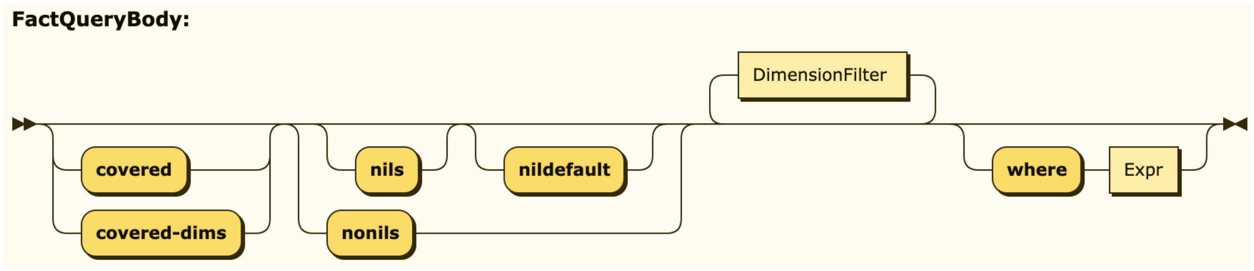
This will check if the document type is reported. If it is then the processor will skip the iteration without returning any value.

Fact Query Grammar Syntax

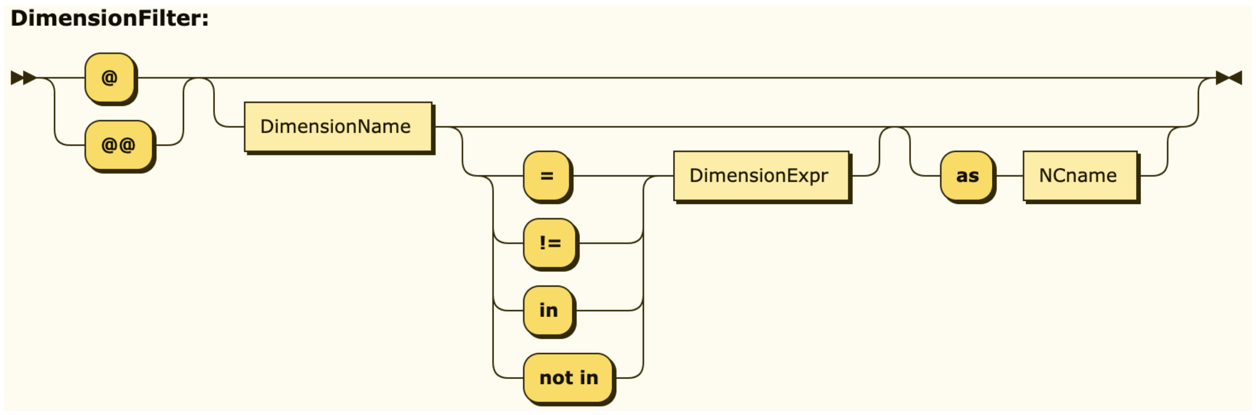
The following diagram shows the syntax of the fact query and the available options.



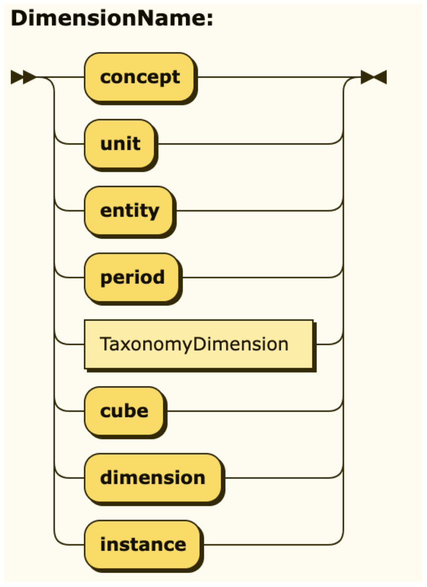
The fact query body is comprised of the following:



The dimension filter has the following options available:



A dimension name can be comprised of any or all of the following:



Defining Facts as Variables

When a fact is assigned to a variable, it maintains all the properties and attributes of the fact from which it was derived. In an example above, a user calculated shareholders equity by deducting Liabilities from Assets. The shareholders equity variable would maintain the alignments that resulted from the calculation. The variable is defined as follows:

```
$she = {@concept = Assets @@dei:LegalEntityAxis = (WidgetsCO, SnapsCO)  } -
      {@concept = Liabilities @@dei:LegalEntityAxis = (WidgetsCO, SnapsCO)};
```

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

Cube as a Filter

A fact query defines the facts to query based on a set of dimensions, and filtering results with a **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 a dimension filter, a cube filter can be defined to do this.

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

```
{@cube.name = StatementTable}
```

This will return all the facts associated with any cube named StatementTable.

Facts in a cube can also be specified by defining the drs role[4].

```
{@cube.drs-role = BalanceSheet}
```

This will return all the facts associated with the any cube in drs role BalanceSheet.

Using @cube has no effect on the alignment of the returned facts. @cube is only used to filter facts for a fact query.

~~{@cube.name =StatementTable @cube.drs-role = BalanceSheet }~~

```
{@cube=taxonomy().cube(StatementTable, BalanceSheet)}
```

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

```
{@cube != none}
```

This will return all the facts associated with any cube.

```
{@cube = none}
```

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

The filters of a fact query (anything with `@`...) all have to be true to select a fact. In the case where you have a fact that is both in `ParentheticalTable/IncomeStatement` and `StatementTable/BalanceSheet` and a fact query selector of {`@`cube.name=`StatementTable` `@`cube.drs-role=`IncomeStatement`}, both filters `@`cube=`StatementTable` and `@`cube.drs-

[4] See DRS Role.

role=`IncomeStatement` are true for the fact and therefore the fact should be selected. The correct way to filter for facts in `StatementTable` AND `IncomeStatement` is to use the `@`cube filter. This can be used with the cube property of a taxonomy as shown below:

```
{@cube=taxonomy().cube(StatementTable, IncomeStatement)}
```

Since there is not a cube for `StatementTable` AND `IncomeStatement`, the `taxonomy().cube(StatementTable, IncomeStatement)` will return none and the fact query will not select any facts.

When using `@`....=.... syntax, there are two things going on. First, a filter identifies facts to select and second if this is a dimension, it specifies a dimension to to cover. `@`cube is NOT a dimension. It is only used for filtering. With dimensions, each fact can only have one value for a dimension (i.e. a fact cannot have 2 concepts or 2 periods or 2 units or 2 values for a dimension). So when specifying multiple dimension properties, they all have to be true for the fact to be selected. For example:

```
{@concept.local-name='Assets' @concept.namespace-uri =
'http://fasb.org/us-gaap/2018'}
```

Only facts that both have a local name of 'Assets' AND the namespace is us-gaap/2018 will be selected. This is because both these filters have to be true for the fact and since a fact can only have one value for the concept dimension.

This is not true for `@`cube. A fact can be in multiple cubes. Hence, to satisfy all the filters for:

```
{@cube.name=StatementTable @cube.drs-role=IncomeStatement}
```

The filter will get the facts in a `StatementTable` cube and in the `IncomeStatement drs-role`.

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.

```
navigate parent-child descendants
```

This will return all the descendent concepts in the presentation parent-child relationships in the 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`
- `navigate ‘http://www.xbrl.org/2003/arcrole/parent-child’ descendants`

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

If the last path component of the uri is not unique within the taxonomy then all uri's with the same last path component are returned.

The Arcrole navigation component can be a uri, a string, an arc-role object, a set or a list.

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.

When using ancestors and parents without a starting element the default start point is the root of the tree. This means the following expression will return no results:

```
navigate ancestors
```

This will return no results.

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.

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.

```
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**'.

```
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.

If there are multiple instances documents, the taxonomy of the referenced instance can be identified by using the taxonomy property of the instance.

```
navigate parent-child descendants from Assets taxonomy
$myInstance.taxonomy
```

This will return all the descendants of Assets in the referenced instance document.

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 relationship 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.

```
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".

Dictionary 1

Key	Value
source	*Assets*
target	*AssetsCurrent*
role	*balanceSheet*

Dictionary 2

Key	Value
source	*Assets*
target	*AssetsNoncurrent*
role	*balanceSheet*

Dictionary 3

Key	Value
source	*Assets*
target	*AssetsOther*
role	*balanceSheet*

Dictionary 4

Key	Value
source	*AssetCurrent*
target	*Cash*
role	*balanceSheet*

Dictionary 5

Key	Value
source	*AssetsCurrent*
target	*CashEquivalents*
role	*balanceSheet*

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.

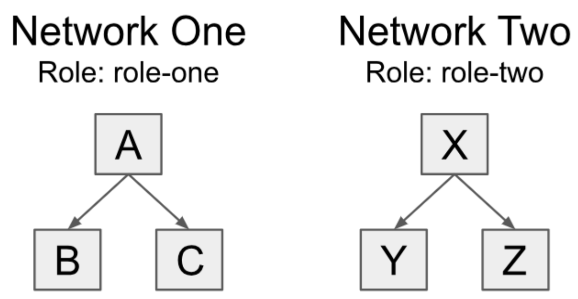
```
navigate parent-child descendants from Assets returns list
```

This will return a list of 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 network by using the '**by network**' keyword.



```
        navigate parent-child descendants returns by network
```

This will return the following dictionary of lists:

Key	***Value***
Network One	*(B, C)*
Network Two	*(Y, Z)*

```
        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.

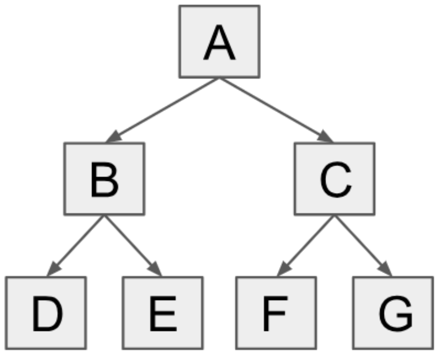
Key	Value			
Network One	*List of dictionaries*			
	Dictionary 1		*Dictionary 2*	
	Key	Value	Key	Value
	target	*B*	*target*	*C*
	role	*role-one*	*role*	*role-one*
Network Two	*List of dictionaries*			
	Dictionary 1		*Dictionary 2*	
	Key	Value	Key	Value
	target	*Y*	*target*	*Z*
	role	*role-two*	*role*	*role-two*

```
$a = navigate parent-child descendants include start returns by network
(target, role) as dictionary
$Network_uri = set(for $network in $a.keys()
      if ($a[$network][1][‘target’] ) == “B” )
```

```
        $a[$network][1][‘role’].uri  OR  $a[$network].role.uri
    else
        none)
```
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.



```
    navigate parent-child descendants include start from A returns paths
```
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))*

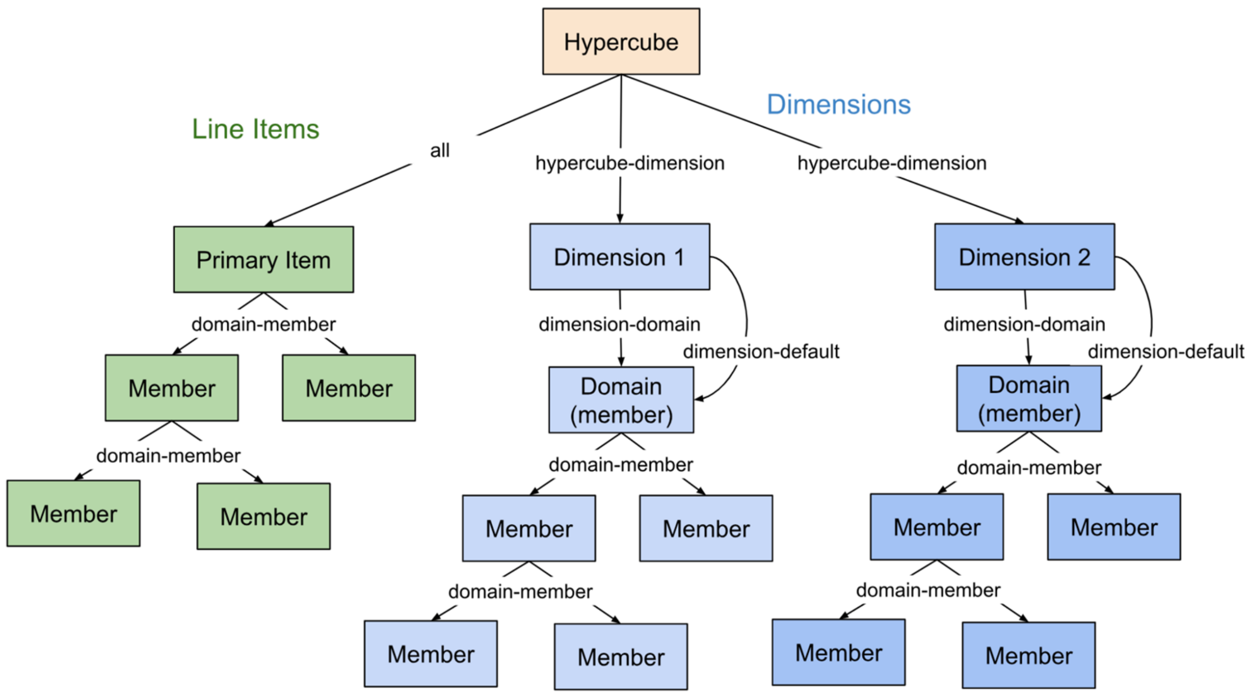
The following example demonstrates how paths can be used to calculate the effective weight between two elements.

```
    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 relation set for a table



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 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:

pseudo arc role	Description
hypercube-primary	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 the opposite direction.
dimension-member	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).
primary-member	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).

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 and the BalanceSheet extended link role.

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:

Return component	Description
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 for different relationships. The drs-role is constant for 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.

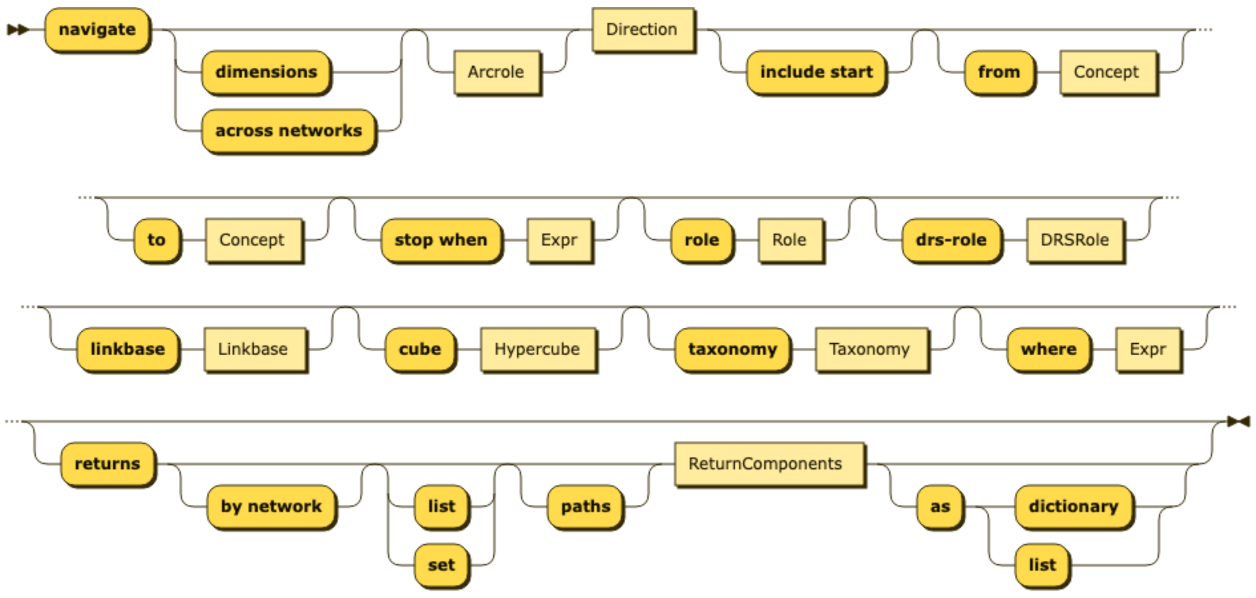
Alternative to dimensional navigation

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

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

Navigation Expression

```
navigate dimensions|across networks {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.

```
        navigate descendants include start
```
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.

```
        filter $a
```
This returns all the items in the set.

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

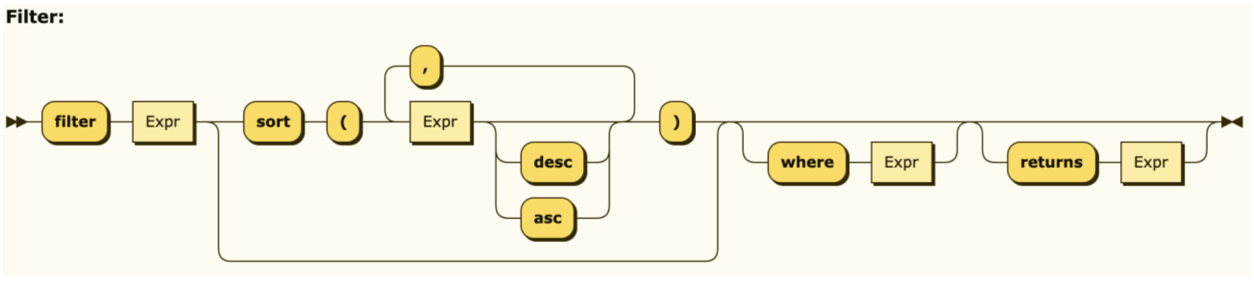
```
        filter set(1,2,3) where $item > 1
```
This returns a set of (2,3)

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:

```
        filter $networkQname returns $item.local-name
```

This returns a set of local names from the set of qnames defined in the variable $networkQname.

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



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

```
	(filter $sub-periods returns "\t" + $item.period.string + "\t" +
	$item.string).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.

The filter supports sorting of the returned set or list, by using the sort expression. The sort in a filter will always return a list. These can be sorted as desc or asc as optional keywords.

```
	filter set(1,2,3,4) sort $item desc where $item > 1 returns $item
```

The sort expression defines the keys that are sorted on and applies that to the filter values. There can be more than 1 sort expression.

```
	filter set(list(1,”e”,5),list(2,”b”,2)) sort ($item[2], $item[1]) desc
returns $item
```

This will return list(list(2,”b”,2),list(1,”e”,5))

Conditional and Iterative Statements

Iterative Statements (Loops)

In XULE, a for loop can be used to iterate through a set or list 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|list))
		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 fact query the where clause is used to perform the same function.

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

```
      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:

```
$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:

```
$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.

1. A set is a group of unique items.
2. A list is a group of ordered items.
3. A dictionary is a group of items with key names that must be unique.

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 facts are 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 facts should generally be assigned to a list and not a set.

Set and List Operators and Properties

Operation	Operator	Property	Sets	Syntax	Result
Union of 2 sets	+	union()	$A = set(a,b,c); $B = set(c,d,e);	$A + $B $A.union($B)	set(a,b,c,d,e)
Intersection of 2 sets	&, intersect	intersect()	$A = set(a,b,c); $B = set(c,d,e);	$A & $B A.intersect($B)	set(c)
Difference of 2 sets	-	difference()	$A = set(a,b,c); $B = set(c,d,e);	$A -$ B	set(a,b) New set with elements in A but not B
Symmetric difference of 2 sets	^	symmetric-difference()	$A = set(a,b,c); $B = set(c,d,e);	$A^$B	set(a,b,d,e)
Test if item is in set	in		$A = set(a,b,c);	c in $A	true
Test if set contains item		contains()	$A = set(a,b,c);	$A.contains(c)	true
Test if item not in set	not in		$A = set(a,b,c);	b not in$ A	false
Get length of a set		length()	$A = set(a,b,c);	$A.length	3
Convert list to a set		to-set()	$A = list(a,b,c,b);	$A.to-set	set(a,b,c)
Convert set to a list		to-list()	$A = set(a,b,c,d);	$A.to-list	list(a,b,c,d)
Convert a set to a string		join()	$A = set(a,b,c);	$A.join(',')	“a,b,c,”
Test if a set is a subset	<=	is-subset()	$A = set(a,b,c); $B = set(a,b,c,d,e);	$A.is-subset($B)	true
Test if a set is a superset	>=	is-superset()	$A = set(a,b,c); $B = set(a,b,c,d,e);	$B.is-superset$(A)	true
Return value of an index	[]	index()[5]	$B = list(a,b,c,d,e);	$B.index(1)	“a”
Converts a set or a list to a dictionary		to-dict()	$A= set(list(‘AAxis’,’AMember’), list(‘BAxis’,’BMember’))	$A.to-dict	dict(list(‘AAxis’,’AMember’))

[5] The index starts at 1 rather than 0.

Converts a list, set or dictionary to a json string format.		to-json()		$A.to-json	
Converts a list,or list of lists to a csv format. The separator is optional. A comma is the default if not supplied. The csv dialect is excel.		to-csv(separator)		$A.to-csv(“,”)	

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. All sort values must be the same type. If string and numbers are mixed the sort is done on string values.		sort()	$A = list(a,c,b);	$A.sort('desc')
				
				$A.sort('asc')
					list(c,b,a)
					
					list(a,b,c)

Aggregate a 2 dimensional collection (set/list of lists) to a dictionary. Arguments represent the index of the list that is used as the dictionary key. If there is no index a value of none is used for the key.**(V1.2.1)**		agg-to-dict(key positions)	$A = list(list('a', 'b', 'c'), list('e', 'f', 'g'), list('h', 'i', 'j'), list('a', 'x', 'y'))	$A.agg-to-dict(1) $A.agg-to-dict(1,2)	dictionary a=list(list('a', 'b', 'c'), list('a', 'x', 'y')), e=list(list('e', 'f', 'g')), h=list(list('h', 'i', 'j')), dictionary list('a', 'b')=list(list('a', 'b', 'c')) list('a', 'x')=list(list('a', 'x', 'y')), list('e', 'f')=list(list('e', 'f', 'g')), list('h', 'i')=list(list('h', 'i', 'j')),

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 fact query definition.

Dictionary Operators and Properties

Operation	Operator	Property	Dict	Syntax	Result
Convert a dictionary to a string		join()	$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));	$A.join(', ', '=')	"AAxis=AMember , BAxis=BMember"
Return the number of key-value pairs in a dictionary.		length()	$A = dict(list('AAxis','AMember', 'BAxis','BMember'));	$A.length	2
Return value of a key	[ ]		$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));	$A['AAxis']	'AMember'
Return a set of the key values in dictionary, can add value to get matching keys		keys()	$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));	$A.keys	set('AAxis', 'BAxis)
		keys(value)	$A = dict(list('AAxis','AMember', list('BAxis','BMember'));	$A.keys('BMember')	set('BAxis')
Test if key is in dictionary	in	has-key()	$A =dict(list('AAxis','AMember'),list('BAxis':'BMember'));	$A.has-key('AAxis')	true
Return a list of the values from key-value pairs		values()	$A =dict(list('AAxis','AMember'), list('BAxis','BMember'));	$A.values	set('AMember','BMember')
Add dictionaries. If the same key is added, then the add is not performed for the matching key. **(V1.2.1)**	+	union	$A = dict(list('AAxis','AMember'), list('BAxis','BMember')); $B = dict(list('YAxis','YMember'));	$A + $B $A.union($B)	dict(list('AAxis','AMember'), list('BAxis','BMember'), list('YAxis','YMember'));
Subtract dictionaries. The keys and values	-	difference	$A = dict(list('AAxis','AMember'), list('BAxis','BMember'));	$A - $B $A.difference($B)	dict(list('AAxis','AMember'));

must match. If not then the left dictionary is maintained. **(V1.2.1)**			$B = dict(list('BAxis','BMember'));		
Subtract list of keys. If keys don't match the left dictionary is maintained. **(V1.2.1)**	-		$A = dict(list('AAxis','AMember'), list('BAxis','BMember')); $B = list('BAxis');	$A - $B	dict(list('AAxis','AMember'));
Converts a dictionary with a list of lists to a spreadsheet format.		to-spreadsheet()		$A.to-spreadsheet()	

Instance Objects (V1.2)

The following objects are available for an XBRL Instance.

Instance Object

XULE provides access to the default instance object. Additional instance documents can be defined using the instance object by providing the url of the instance document.

$wsfs = *instance('https://www.sec.gov/edgar/wsfs-20211231.htm')*

This loads the wsfs instance into the XULE processor.

Fact query selection defaults to the default instance document. To select facts from an alternative instance document the fact query selection criteria must reference the instance object.

```
{@instance = $wsfs}
```

This returns all the facts from the wsfs instance.

Instance Properties

Name	Definition	Examples
document-location **(V1.2)**	Returns the document location of the instance	`$myInstance.document-location` **Returns the URI of the instance**

facts **(V1.2)**	Returns the list of facts in an instance. These will all be unaligned. To get aligned facts, use a fact query such as {@instance=$myInstance}. Note that the returned value is a list and not a set. Sets eliminate duplicates by value, which would eliminate different facts that happen to have the same value. The facts property returns all facts including duplicates. This differs from the fact filter which removes duplicates.	`$myInstance.facts`
taxonomy **(V1.2)**	Returns the taxonomy used by the instance.	`$myInstance.taxonomy` **Returns the taxonomy of the instance document.**

Fact Object

The fact object is returned when defining a fact query. Each fact returned 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.

Name	Definition	Examples

aspects()	The taxonomy defined and built in dimensions associated with a fact are returned as a dictionary where the aspect is the dictionary key.	`{@Revenues}.aspects`
		
		Returns the aspects of the fact.
		dictionary(period=2021-06-30,unit=USD,entity=http://www.sec.gov/CIK=0000831259,concept=us-gaap:Revenues)

[6] Listed later in the document.

concept()	The concept of the fact value	`{@ where $fact.concept.name == Revenues}`
cubes **(V1.2)**	Returns a set of cube objects that the fact is valid in.	`{@Revenues}.cubes` **Returns the cube objects that the fact is valid in.**
decimals()	The decimal value of the fact value	`{@Revenues where $fact.decimals == -6}`
dimension(qname of dimension)	Returns the member of the fact for the specified dimension	`{@ where $fact.dimension(dei:LegalEntityAxis).name == ABC}`

dimensions()	Returns a dictionary of key values pairs of dimension keys and member values. These are returned as concepts.	`{@Revenues}.dimensions()`
		
		`{@Revenues}.dimensions.keys.name }`
		Returns the qnames of the dimensions on the fact.

dimensions-explicit()	Returns a dictionary of key values pairs of explicit dimension keys and member values. These are returned as concepts.	`{@Revenues}.dimensions-explicit()`
		
		`{@Revenues}.dimensions-explicit.values.name`
		
		Returns the qnames of the members on the fact.

dimensions-typed()	Returns a dictionary of key values pairs of typed dimension keys and member values. These are returned as concepts.	`{@Revenues}.dimensions-typed()`
entity()	Returns the entity of the fact. The properties schema and id can be added as additional properties.	`{@ where $fact.entity.id = ‘00000000001’}` **Returns all facts for entity 00000000001**
footnotes **(V1.2)**	Returns the footnote objects associated with a fact.	`{@Revenues}.footnotes`

id	Returns the id of the fact. If the fact does not have an id, none is returned.	`{@Revenues}.id`
sid **(V1.2)**	Returns the semantic id of the fact. The semantic id is based on the concept, unit, period, entity, dimensional qualifications, decimals value and the value of the fact.	`{@Revenues}.sid`
instance **(V1.2)**	Returns the instance object that the fact is in.	`{@Revenues}.instance`
period()	The period of the fact. The period object supports properties of start, end, days	`{@Revenues where $fact.period.days > 100 and $fact.period.start > date('2014-12-31')}`
unit()	The unit of measure of the fact.	`{@ where $fact.unit == unit(xbrli:pure)}` **Returns all facts that are pure.**
Inline XBRL Properties		
inline-is-hidden()	Returns a boolean if the fact is hidden in an inline document. Returns none on non-inline documents.	`{ where $fact.inline-is-hidden == true}` **Returns all hidden facts**
inline-scale()	Returns the scale of a fact in an inline xbrl document. Returns none on non-inline documents.	`{ where $fact.inline-scale == 6}` **Returns all facts that have a scale of 6.**
inline-format()	Returns the format qname associated with the fact in an inline xbrl document. Returns none on non-inline documents.	`{ where $fact.inline-format == ixt:datemonthdayyearen}` **Returns all facts formatted with the `datemonthdayyearen` format.**
inline-display-value()	Returns the display value associated with the fact in an inline xbrl document. Returns none on non-inline	`{ where $fact.inline-display-value == 'June 30, 2018'}`

	documents.	**Returns all facts displayed as 'June 30, 2018'.**
inline-negated()	Returns a boolean if the fact has a sign in an inline document. Returns none on non-inline documents.	`{ where $fact.inline-negated == true}` **Returns all facts that have the sign attribute**
inline-parents() **(V1.2)**	Returns the parent facts of a fact in an inline document as a list. This function navigates the html structure and links in continuations if they are present in the filing.	`{ where $fact.inline-parents.length == 0}` **Returns all facts that are not contained within another fact such as a text block as a list of facts.**
inline-children() **(V1.2)**	Returns a list of child facts of a fact in an inline document. This function navigates the html structure and links in continuations if they are present in the filing.	`{ where $fact.inline-children.length == 0}` **Returns all facts that have no children.**
inline-ancestors() **(V1.2)**	Returns the ancestor facts of a fact in an inline document as a list. This function navigates the html structure and links in continuations if they are present in the filing.	`{ where $fact.inline-ancestors.length > 0}` **Returns a list of facts that are ancestors of a given fact. The list is ordered from closest to furthest fact.**
inline-descendants() **(V1.2)**	Returns the descendant facts of a fact in an inline document as a list.	`{ where $fact.inline-descendants.length > 0}` **Returns a list of facts that are descendants of a given fact. The list is ordered from closest to furthest fact.**

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

Name	Definition	Examples

days()	Returns the number of days in a given duration period.	`{@Revenues where $fact.period.days > 100}`
end()	Returns the end date of a durational period or the date of an instant	`{@Assets where $fact.period.end > date('2014-12-31')}`
start()	Returns the start date of a durational period	`{@Revenues where $fact.period.start > "2014-12-31"}`

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

Name	Definition	Examples
numerator()	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	`where $fact.unit.numerator == unit(iso4217:USD)` **Tests if the numerator of the fact is USD**
denominator()	Returns the xbrl measure of the denominator as a qname. If there is no denominator an empty value is returned.	`where $fact.unit.denominator == "iso4217:USD"` **Tests if the denominator of the fact is USD**
id	Returns the id of the unit used in the instance.	`$fact.unit.id`
utr()	Returns the symbol of the unit from the units registry if a value exists.	*$fact.unit.utr(**symbol)*** ***Returns the symbol of the fact. Such as $***

Footnote Object

Footnote Properties

Xule includes the following properties for a footnote object.

Name	Definition	Examples
arcrole	Returns the arc-role of the footnote.	`footnote.arcrole` **Could return http://www.xbrl.org/2009/arcrole/fact-explanatoryFact**
content	Returns the content of the footnote. If it is a fact it returns a fact object	`footnote.content`
fact	Returns a set of the facts that the footnote came from.	`footnote.fact`
lang	Returns the language of the footnote.	`footnote.lang`
role	Returns the footnote-role of the footnote.	`footnote.role`

Taxonomy Objects

Concept Object

Concept Equality

Concepts are compared based on qname. A concept used in two different dts 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.

Name	Definition	Examples

attribute(name)	Returns the value of a custom attribute based on the name provided to the function.	`$fact.concept.attribute(abc)`
balance	Returns the balance attribute of a fact. This can be either debit or credit or none.	`$fact.concept.balance,` `{@concept.balance = debit}` **For Assets will return debit**
base-type	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 derived from.	`$fact.concept.base-type.name` **For BasisOfAccount (which has a data type of nonnum:textBlockItemType) will return the type object for xbrli:stringItemType.**
data-type	Returns the type of a concept.	`$fact.concept.data-type.name` **For Assets will return the type object for xbrli:monetaryItemType.**
enumerations	Returns a set of enumerated values allowed for the concept. These are the enumerations defined in the type of the concept.	`concept.data-type.enumerations`
has-enumerations	Returns a true or false if the concept has enumerations in the datatype. These are the enumerations defined in the type of the concept.	`concept.data-type.has-enumerations`
is-abstract	Returns true if the concept has an abstract value of true. This attribute can only be on the concept object.	`concept.is-abstract,` `{@concept.is-abstract = false}` **For Assets will return false**
is-monetary	Returns a boolean result if the concept has that type.	`$fact.concept.is-monetary` **For Assets will return true**
is-numeric	Returns a boolean result if the concept has that type.	`$fact.concept.is-numeric` **For Assets will return true**

is-type(type)	Returns a boolean result if the concept has that type. The type is provided as a qname.	`$fact.concept.is-type(xbrli:monetaryItemType)` **For Assets will return true**
label(label role, language)	For a concept the label property label(label role, language) can be used to return one 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. The roles are searched in an ordered list.	`taxonomy().concept(Assets).label.text` **For Assets will return the string of Assets**

all-labels(label role, language)	Returns a set of all labels for the concept. Label role and language will filter the result to only return labels that match the label role and/or language specified. To return all labels for a given language, use none for the label role.	`taxonomy().concept(Assets).all-labels`
		
		Will return all the labels associated with the Assets concept
		
		`taxonomy().concept(Assets).all-labels(none,'en')`
		
		Will return all the labels associated with the Assets concept in english.

local-name	Returns the local name of the concept name.	`$fact.concept.name.local-name` **For Assets will return the string Assets**
name	Returns the qname of the concept. This includes the local name and URI.	`$fact.concept.name` **For Assets will return us-gaap:Assets**
namespace-uri	Returns the uri of the concept name.	`$fact.concept.name.namespace-uri` **For Assets will return us-gaap**

clark	Returns a string in clark notation when provided a qname or concept.	`$US-GAAP-2020.concept(Assets).clark` **Returns a string of {http://fasb.org/us-gaap/2020-01-31}Assets**
period-type	Returns the period type, instant or duration.	`$fact.concept.period-type` **For Assets will return instant**
references(reference-role)	Returns a set of references associated with a concept for a given dts. If no role is provided it defaults to the first role that has a reference. The roles checked are based on an ordered list.	`taxonomy().concept(Assets).references("http://www.xbrl.org/2003/role/presentationRef")` **Returns a set of reference objects for Assets based on the role provided or the default role.**
all-references()	Returns a set of all references associated with a concept. (includes all roles)	`taxonomy().concept(Assets).all-references` **Returns a set of all reference objects for Assets**
relationships	Returns all relationships associated with the concept.	
source-relationships	Returns the relationships where the concept is the source.	
substitution() **(V1.1)**	Returns the qname substitution group of the concept.	$fact.concept.substitution **For Assets will return xbrli:item**
target-relationships()	Returns the relationships where the concept is the target.	
traits() **(V1.3)**	Returns a set of traits associated with a concept. Uses the trait relationship and class-subclass relationships defined in the taxonomy.	$fact.concept.traits **For AssetsCurrent will return a set of traits associated with the concept i.e. set(CurrentMember)**

Part Element Object

A part element represents the element definition of the reference part.

Part Element Equality

Part Elements are compared based on qname. A part element used in two different dts are considered to be the same concept for equality as the qnames are compared.

Part Element Properties (V1.2)

Xule includes the following properties for a concept object.

Name	Definition	Examples
attribute(name)	Returns the value of a custom attribute based on the name provided to the function.	`$fact.concept.attribute(abc)`
data-type	Returns the type of a part element.	`$fact.concept.data-type.name` **For Assets will return the type object for xbrli:monetaryItemType.**
enumerations	Returns a set of enumerated values allowed for the concept. These are the enumerations defined in the type of the part element.	`concept.data-type.enumerations`
has-enumerations	Returns a true or false if the concept has enumerations in the datatype. These are the enumerations defined in the type of the part element.	`concept.data-type.has-enumerations`
is-abstract	Returns true if the part element has an abstract value of true. This attribute can only be on the part element object.	`concept.is-abstract, {@concept.is-abstract = false}` **For Assets will return false**
is-numeric	Returns a boolean result if the part element has that type.	`$fact.concept.is-numeric` **For Assets will return true**
is-type(type)	Returns a boolean result if the part element has that type. The type is provided as a qname.	`$fact.concept.is-type(xbrli:monetaryItemType)` **For Assets will return true**
local-name	Returns the local name of the part	`$fact.concept.name.local-name`

	element name.	For Assets will return the string Assets
name	Returns the qname of the part element. This includes the local name and URI.	`$fact.concept.name` For Assets will return us-gaap:Assets
namespace-uri	Returns the uri of the part element name.	`$fact.concept.name.namespace-uri` For Assets will return us-gaap
clark	Returns the clark notation of the part element name.	
substitution() **(V1.1)**	Returns the substitution group of the concept.	$fact.concept.substitution **For Assets will return xbrli:item**

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.

Name	Definition	Examples
part-by-name(part qname)	Returns the reference part for a reference based on part name.	`reference.part-by-name(cod:Topic).part-value` **Returns the reference part value with a qname of cod:Topic**
parts()	Returns a set of parts for a reference	`reference.parts` **Returns all the reference parts associated to a reference**
role()	Returns the reference role of a reference.	`reference.role.uri` **Returns the reference role associated with a reference such as 'http://fasb.org/srt/role/changeNote/changeNote'**

concepts()	Returns a set of concepts that are linked to this reference in the taxonomy	

Part Object

The following table details the properties associated with reference parts object:

Name	Definition	Examples
part-value()	Returns the value associated with a reference part	`reference().parts.part-value` **Returns the reference part value**
name()	Returns the qname of the reference part.	`reference().parts.name` **Returns the reference part name**
namespace-uri()	Returns the namespace uri of the reference part	`reference().parts.namespace-uri` **Returns the reference part namespace**
local-name()	Returns the local name of the reference part	`reference().parts.local-name` **Returns the reference part local name**
order()	Returns the order of the part	`reference().parts.order`
part-element() **(v1.2)**	Returns the part element for the reference part	`reference().part-by-name(ref:Name).part-element`

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))
codification-part:URI -
http://asc.fasb.org/extlink&amp;oid=6876686&amp;loc=d3e534808-122878
```

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

Name	Definition	Examples
text()	Returns the text of a label as a string	`label.text`
lang()	The language of a label	`label.lang`
role()	The role of the label returned as a role object.	`label.role`
concepts()	Returns a set of concepts that are linked to the label in the taxonomy	

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.

Data Type Object

A type object represents a data type of a concept.

Type Object Properties

Name	Definition	Examples
name()	QName of the type, for simple types	`concept.data-type.name`
enumerations()	Returns a set of allowed values for the type	`concept.data-type.enumerations`
has_enumerations()	Returns a boolean if the type is restricted to a list of enumerated values.	`concept.data-type.has_enumerations`
type-ancestry() **(V1.2.1)**	Returns a list of the data type ancestry.	`Concept.data-type.type-ancestry` **Returns list(xbrli:stringItemType)**
parent-type() **(V1.2.1)**	Returns the type of the parent type.	`Concept.data-type.parent-type` **Returns xbrli:stringItemType**
base-type() **(V1.2.1)**	Returns the base type of the type.	`Concept.data-type.base-type` **Returns xbrli:stringItemType**
min-inclusive **(V1.2.1)**	Returns an integer value of the min inclusive cardinal value for a type.	`concept.data-type.min-inclusive` **Returns 1**
max-inclusive **(V1.2.1)**	Returns an integer value of a max inclusive cardinal value for a type.	`concept.data-type.max-inclusive` **Returns 10**
min-exclusive **(V1.2.1)**	Returns an integer value of a min exclusive cardinal value for a type.	`concept.data-type.min-exclusive` **Returns 1**
max-exclusive **(V1.2.1)**	Returns an integer value of a max exclusive cardinal value for a type.	`concept.data-type.max-exclusive` **Returns 10**
total-digits **(V1.2.1)**	Returns an integer value of the total digits of a value.	`concept.data-type.total-digits` **Returns 6**
fraction-digits **(V1.2.1)**	Returns an integer value of the number of digits to the right of the decimal place.	`concept.data-type.fraction-digits` **Returns 3**

value-length **(V1.2.1)**	Returns an integer value of the length of a string value.	`concept.data-type.value-length` **Returns 4**
min-length **(V1.2.1)**	Returns an integer value of the minimum length of a string value.	`concept.data-type.min-length` **Returns 2**
max-length **(V1.2.1)**	Returns an integer value of the maximum length of a string value.	`concept.data-type.max-length` **Returns 4**
white-space **(V1.2.1)**	Returns a string of one of the following: 'preserve', 'replace' or 'collapse'	`concept.data-type.white-space` **Returns 'preserve'**
pattern **(V1.2.1)**	A single regex expression or a list/set of regex expressions	`concept.data-type.pattern`

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.

Name	Definition	Examples
cube-concept	Returns the hypercube concept of the cube	`cube.cube-concept()` **Returns the cube concept.**
drs-role()	Returns an extended link role object of the role that the specified cube is included in.	`cube.drs-role().uri == “BalanceSheet”` **Returns the cube in the balance sheet extended link role.**
dimensions()	Returns the dimensions of a cube as a dimension object.	`cube.dimensions` **Returns all the dimensions associated with a given cube.**
primary-concepts()	Returns the primary concepts of a cube as a set of primary concepts.	`cube.primary-concepts` **Returns the primary concepts associated with the cube.**

members()	Returns the member concepts of a cube.	`cube.members` **Returns all the members on a specific cube.**
closed()	Returns a boolean result of true if the cube is closed.	`cube.closed` **Returns true if the cube is closed.**
facts()	Returns all the facts associated with a cube.	`cube.facts` **Returns all the facts in a given cube.**

Dimension Object

The dimension object is different from 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.

Name	Definition	Examples
dimension-type()	Returns typed dimensions.	dimension.dimension-type **Returns a string value of typed or explicit.**
members() **(V1.2)**	Returns the members on a dimension	dimension.members
default()	Returns the default of a given dimension.	`dimension.default` **Returns the default domain of the dimension object as a concept.**
concept()	Returns the concept for the dimension.	`dimension.concept` **Returns the dimension concept.**
domains()	Returns the domains of a given dimension	dimension.domains
useable-members()	Returns the useable members on a dimension.	dimension.useable-members
nonuseable-members()	Returns the non-useable members on a dimension.	dimension.non-useable-members

cube **(V1.2)**	Return the cube object that the dimension is on.	`dimension.cube`
data-type	Returns the datatype of a typed dimension.	dimension.data-type.name **Returns the datatype of the typed dimension**

Members Object (Not Yet Implemented)

Name	Definition	Examples
concept()	Returns the concept associated with a member.	members.concept
dimensions()	Returns the dimensions associated with a member	members.dimensions

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.

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.

taxonomy('http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-2016-01-31.xsd')

This loads the US GAAP elements taxonomy.

Taxonomy Properties

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

Name	Definition	Examples

concepts()	Returns a set of concepts representing every concept in a taxonomy or a network. Can be used on a taxonomy type or a network type.	`taxonomy().concepts` **Returns all the concepts in the instance taxonomy. (Whether they have values or not, or are included in a tree or not.)**
concept(QName)	Returns the concept in a taxonomy based on the concept qname.	`taxonomy().concept(Assets)` **Return the Assets concept from the taxonomy.**
concepts_by_trait((Concept or QName or list or set)	This will return the concepts that have all the traits listed in the argument. The argument can either be a single value or a collection (set/list) of values. The values can be qnames or concepts (concepts that are traits).	`taxonomy().concepts_by_trait(`
cube(Concept or QName,Role)	Returns a cube from the taxonomy. The first argument is the concept or qname of the cube concept. The role is the drs-role.	`taxonomy().cube(StatementTable, ShareholdersEquity)` **Returns the cube object**
cubes()	Returns all the cubes in the taxonomy as a set.	`taxonomy().cubes` **Returns all the cubes in a taxonomy**
effective-weight(QName, Qname)	Returns the effective weight between two concepts aggregated across all calculation networks. Two qname parameters must be passed.	`taxonomy().effective-weight(NetCashProvidedByUsedInOperatingActivities,IncomeLossFromEquityMethodInvestments)` **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.**
effective-weight-network(QName ,QName,*Role*)	Returns the effective weight between two concepts in a given calculation network. Returns a set	`taxonomy().effective-weight-network(NetCashProvidedByUsedInOp`

	of lists containing the network and effective weight between the 2 concepts. The third parameter (role) is optional.	`eratingActivities,IncomeLossFromEquityMethodInvestments)`
		
		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 the statement of cashflows.

dimensions() **(V1.1)**	Return all the dimensions in a taxonomy that are dimension objects. Dimensions are determined based on the defined hypercubes in the taxonomy, not the datatype of the concept.	taxonomy().dimensions **Returns all the dimensions in a taxonomy**
dimension(QName) **(V1.1)**	Returns the dimension object with a provided qname.	`taxonomy().dimension(us-gaap:CountryAxis).default()` **Returns the default member of the us-gaap:CountryAxis**
dimensions-explicit() **(V1.1)**	Return all the dimensions in a taxonomy that are explicit dimension objects.	taxonomy().dimensions-explicit **Returns all the explicit dimensions in a taxonomy**
dimensions-typed() **(V1.1)**	Return all the dimensions in a taxonomy that are typed dimension objects.	taxonomy().dimensions-typed **Returns all the typed dimensions in a taxonomy**

dts-document-locations	Returns the location of all documents comprising the dts.	`$US-GAAP.dts-document-locations` **Returns the uri of all dts documents as a set.**
entry-point-namespace	Returns the namespace of the entry point for the taxonomy (DTS) used.	`$US-GAAP.entry-point-namespace` **Returns the namespace of the us-gaap taxonomy** "http://xbrl.fasb.org/us-gaap/2019"
entry-point()	Returns the entry point url of the taxonomy object	`$US-GAAP.entry-point` **Returns the url** http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-2016-01-31.xsd

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.
		
		`taxonomy().networks(parent-child,'http://www.abc.com/role/ConsolidatedBalanceSheets')`
		
		Returns the parent-child network for the consolidated Balance Sheet in the taxonomy.
		
		`taxonomy().networks(parent-child,ConsolidatedBalanceSheets)`
		
		Returns the parent-child network for the consolidated Balance Sheet in the taxonomy. This uses the short name (No quotes)

namespaces() **(V1.1)**	Return a set of namespace uris that are defined in the taxonomy.	taxonomy().namespaces

		Returns the namespaces that are defined in the taxonomy.
roles() **(V1.2)**	Return a set of roles associated with the dts.	taxonomy().roles
arcroles() **(V1.2)**	Return a set of arc roles associated with the dts.	taxonomy().arcroles
part-elements() **(V1.2)**	Return a set of reference part elements associated with the dts	taxonomy().part-elements
elements() **(V1.2)**	Return a set of elements that are not concepts or part elements	taxonomy().elements()
element(QName) **(V1.2)**	Returns the element in a taxonomy based on the element qname.	`taxonomy().element(CustomElement)` **Return the CustomElement from the taxonomy of the instance.**
data-types() **(V1.2)**	Return a set of data types associated with the dts.	`taxonomy().data-types`

concepts-by-trait() **(V1.3)**	Takes a single argument of a trait concept or trait concept qname or a set/list of trait concepts or trait concept qnames and returns a set of concepts that have all the supplied traits.
	
	Uses the trait relationship and class-subclass relationships defined in the taxonomy.
		taxonomy().concepts-by-trait(OperatingActivity)
		
		Returns all the concepts that have the Operating Activity trait
		
		taxonomy().concepts-by-trait(list(OperatingActivity, Current)
		
		Returns all the concepts that have the Operating Activity trait and Current trait.

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.

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.

Name	Definition	Examples
arcrole()	Returns the arc-role of the network.	`Network().arcrole` **Could return summation-item, parent-child etc.**
concept-names()	Returns a set of qnames representing every concept in a network.	`network().concept-names` **Returns all the qnames in a network.**
concepts()	Returns a set of every concept in a network including target and source	`network().concepts` **Returns all the concepts in a network.**
source-concepts()	Returns the source concepts from a network.	`network().source-concepts`
target-concepts()	Returns the target concepts in a network	`network().target-concepts`
relationships()	Returns a set of relationships that can be looped to get the relationship object	`network().relationships` **Returns all the relationships in a given network.**

role()	Returns the role of the network. The role as 3 properties of uri, description and tree (linkbase) applicable to. (Is the extended link role)	`network.role.uri`
		
		Returns the uri of the role
		
		`network().role.description`

		Returns the description of the role `network().role.used-on` **Returns a set of the trees the role is used on.**
roots()	Returns a set of concepts representing the root concepts of a network	`network().roots` **Returns the root concepts of the defined network**

Role Object

Role Properties

Name	Definition	Examples
uri()	Returns the uri of a role.	`network.role.uri` **Returns the uri of the role** network.arcrole.uri **Returns the uri of the arcrole**
description()	Returns the description of the role.	`network.role.description` **Returns the description of the role** network.arcrole.description **Returns the description of the arcrole**
used-on()	Returns a set of qnames of **the extended link element names** the role is used on.	`network.role.used-on` **Returns a set of the extended link element names the role is used on.** network.arcrole.used-on

		Returns a set the qnames of the arc element the role is used on.
cycles()	Indicates if the arcrole allows cycles.	

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.

Name	Definition	Examples
source()	The source concept of the relationship	`relationship().source`
source-name()	The QName of the source concept of the relationship	`relationship().source-name`
target()	The target concept of the relationship	`relationship().target`
target-name()	The QName of the target concept of the relationship	`relationship().target-name`
order()	The value of the order attribute on the relationship	`relationship().order`
weight()	The value of the weight attribute on the relationship	`relationship().weight`
preferred-label()	The value of the preferred Label attribute on the relationship	`relationship().preferred-label`
role()	The extended link role of the network	`relationship().role`

arcrole()	The arcrole of the network	`relationship().arcrole`
arcrole-uri()	The arcrole uri of the network	`relationship().arcrole-uri`
arcrole-description()	The description of the arcrole of the network	`relationship().arcrole-description`
link-name()	The QName of the extended link element	`relationship().link-name`
arc-name()	The QName of the arc element	`relationship().arc-name`
network()	The network object the relationship is in.	`relationship().network`

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 an object.

Numerical Properties and Functions

Name	Definition	Examples
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.**
int()	Returns the value as an integer. The int function will always round down. It effectively cuts off any decimal places.	*Property:* `10.98.int` **Returns a value of 10.**
log10()	Returns the log of a number. Can be used on an integer, float, decimal and fact type. A negative number returns none.	*Property:* `{@Assets}.log10` *Function:* `log10({@Assets})` **Returns the log10 value of assets.**
power()	Returns the power of a number. Can	*Property:* `4.power(2).`

	be used on an integer, float, and decimal type. If you want to determine the square a number it is used as 4.power(2).	**Returns a value of 16.**
signum()	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.	*Property:* `{@Assets}.signum` *Function:* `signum({@Assets})`
trunc(number, places)	Truncates the decimal places on a number. Places defaults to zero if not provided.	*Property:* `{@Assets}.trunc(2)` *Function:* `trunc({@Assets},2)` **truncates the value of assets to two decimal places**
round(number,places)	Rounds a number to the decimal places indicated. Rounds to the nearest even.	*Property:* `{@Assets}.round(2)` *Function:* `round({@Assets},2)` **Rounds the value of assets to two decimal places**

mod(numerator, divisor)	Returns the mod of a numerator and a divisor.	*Property:*
		`{@Assets}.mod({@Liabilities})`
		Function:
		`mod({@Assets},{@Liabilities})`

random(places) **(V1.2)**	Returns a random number between 0 and 1. Places defaults to 4 if not provided.	*Function:* `random()` **Returns random number between 0 and 1.**

String Functions and Properties

Name	Definition	Examples
clark(*qname*)	Returns a string in clark notation when provided a qname.	`$US-GAAP-2020.concept(Assets).clark`

		Returns a string of {http://fasb.org/us-gaap/2020-01-31}Assets
contains(*string,string*)	Returns a boolean result if the provided value is contained in the provided string.	`'http://some/role/for/cashflow/'.lower-case.contains("cashflow")` **Returns a boolean of true**
index-of(*string,string*)	The index-of returns an integer of the position of the first instance of a given string in another string. If the string is not found a value of 0 is returned.	`“Hello Mr. Stains”.index-of(’llo’)` **Returns a value of 3**

inline-transform(*transform,type*) **(V1.2)**	The inline-transform is a property of a string. The function will transform a string value to a valid defined data type using the inline transformation registry. The first property is the qname of the transform and the second optional parameter is the datatype. If not provided it will default to a string.	`'1,234,567.89'.inline-transform(ix4:num-dot-decimal)`
		Returns a string value of 1234567.89
		
		`'1,234,567.89'.inline-transform(ix4:num-dot-decimal, 'decimal')`
		Returns a decimal value of 1234567.89
		
		`inline-transform('1,234,567.89',ix4:num-dot-decimal, 'decimal')`
		Returns a decimal value of 1234567.89

last-index-of(*string,string*)	The last-index-of returns an integer of the position of the last instance of a given string in another string.	`'Hello Mr Stains'.last-index-of(’o’)  = 5` **Returns a value of 5**
length(*string*)	Returns the length as an integer of a string. Can also be used on a set, list or a dictionary to return the number of items in the collection.	`'Hello Mr. Stains'.length()  = 16` **Returns a value of 16**
lower-case(*string*)	Returns a string as lowercase characters.	`'http://some/role/for/cashflow/'.lower-case()` **Returns the string in lowercase**

number(*string*)	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	`'3.4'.number = 3.4` **Returns a decimal number value of 3.4**

regex-match(*pattern*) **(V1.1)**	Returns a dictionary of information about the regular expression match. The dictionary contains:
	end: The character position in the string where the match ended.
	groups: Returns matching portions within parenthesis in the regular expression. Groups are based on occurrence and order of left parenthesis. Returns a list of matching expressions.
	match: The text of the string that matched the regular expression pattern.
	Match-count: The number of matches
	start: The character position in the string where the match started.
	
	If there is no match, the 'match' value is none and the 'match-count' is zero.
		`'abcdefg'.regex-match('de+')`
		
		Returns a dictionary of:
		dict('end': 6, 'groups': list(), 'match': 'de', 'match-count': 1, 'start': 4)

regex-match-all(*pattern*) **(V1.1)**	Same as regex-match except that the "-all" will return a list where a match is made and then the pattern is applied to the rest of the string and the next match is made.	`'abcdefg'.regex-match-all('de+')`
		
		Returns a list of:
		list(dict('end': 6, 'groups': list(), 'match': 'de', 'match-count': 1, 'start': 4))

regex-match-string(*pattern, optional group number*) **(V1.1)**	Returns the string that matches the regular expression pattern. If there is no match, the value is none. The second parameter allows	`'abcdefg'.regex-match-string('de+')`

	the number of the group to be returned. Groups are defined in the pattern using parenthesis.	**Returns the string 'de'** `'abcdefgabcdefg'.regex-match-string('(c)(d)', 2)` **Returns 'd' because the '(d)' is the second group.**

regex-match-string-all(*pattern, optional group number*) **(V1.1)**	Same as regex-match-string except that the "-all" will return a list where a match is made and then the pattern is applied to the rest of the string and the next match is made.	`'abcdefggabcdefg'.regex-match-string-all('[cf]([dg])')`
		
		Returns list(cd, fg, cd, fg).
		
		`'abcdefggabcdefg'.regex-match-string-all('[cf]([dg])', 1)`
		
		Returns list(d, g, d, g)

replace(search string, replacement string, number of times) **(V1.2)**	Returns a new string based on an existing string. Replace the search string with the replacement string. The 3rd argument indicates how many times to do the replace from left to right. If left off all search items will be replaced.	`'Hello Mr Stains'.replace('Stains','Smith', 1)  = 'Hello Mr Smith'`
repeat()	Repeats a string value the number of times defined by the argument. The argument value must be an integer.	`'-'.repeat(4) = '----'`
split(*delimiter*)	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.	`'Hello Mr Stains'.split(' ')  = list('Hello','Mr','Stains')`
starts-with()	Returns a value of true or false if the string starts with a given string.	`"Value of Derivatives".starts-with('Value') = true` **Returns a value of true**

ends-with()	Returns a value of true or false if the string ends with a given string.	`“Value of Derivatives”.ends-with(‘ves’)` **Returns a value of true**

string()	Converts a value to string using builtin formatting. For integers, decimals and floats, the built-in formatting includes comma thousand separators and a limit of 4 decimal places.	`3.string`
		
		Returns a string value of “3”
		
		`1234.string`
		
		Returns a string value of “1,234”

plain-string()	Converts a value to string. Converts a value to string without formatting.	`3.plain-string`
		
		Returns a string value of “3”
		
		`1234.plain-string`
		
		Returns a string value of “1234”

substring(*begin index, end index*)	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. Thus the length of the substring is endIndex-beginIndex + 1. If the last parameter is left off it continues to the end of the string.	`'Hello Mr. Stains'.substring(1,5)` **Returns the string of “Hello”**

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’ .

trim(*side*) **(V1.1)**	Trim removes white space from either the left side or right side or both sides of the string. The property has optional string values of “left”, “right”, or “both”. If no value is provided the property defaults to both. *(Added V1.1)*	`'Abc '.trim = 'Abc'`
		
		 Returns a string without spaces at end.
		
		 `'Abc '.trim('right') = ‘Abc'`
		
		 Returns a string without spaces at end.

upper-case()	Returns a string as uppercase characters.	`'cashflow'.upper-case()` **Returns the string of “CASHFLOW”**

Generic Properties

Name	Definition	Examples
is-fact	Returns true or false if a value represents a fact object.	`{@Assets}.is-fact` **Returns true.** 4.is-fact **Returns false.**
is-nil	Returns true or false if a fact value is nil.	`{@Assets}.is-nil` **Returns true if the value is nil**

document-location	Returns the document uri of the document that contains the object. If there is no document uri, then it returns none	`taxonomy().concepts(Assets).document-location`
		 `Returns the uri where the concept Assets is defined.`

Date Properties and Functions

Name	Definition	Examples

contains(duration)	Is a property of a duration used to check if one duration period is contained within another duration period. If the values overlap a value of false is returned. Two matching durations will return true using contains.	`duration('2024-01-01','2025-01-01').contains(duration('2024-03-01','2024-06-01'))` **Returns boolean value of true**

date(string)	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.	`date('2017-12-31')`
		OR
		`'2017-12-31'.date`
		
		Converts the string to a date.

day()	Returns the day (integer) from a given date.	`day(date('2017-12-31'))` Returns a value of 31.
days()	Returns the number of days in a duration object.	`duration('2022-01-01', '2022-03-31').days` **Returns an integer of 89**
duration(start-date, end-date))	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.	*Function:* `{@period = duration('2016-01-01', '2016-12-31')}` **Converts two string dates to a duration for comparing or filtering a fact.**
forever()	Generates a period equal to the forever period	*Function*: `{@period = forever }` **Returns all periods that have a period of forever.**
month()	Returns the month (number) from a given date.	*Function:* `month(date('2017-12-31'))` **Returns a value of 12.**

is-month-day	Checks that the value of the element with a datatype of gMonthDayItemType is valid.	*Property:* `{@CurrentFiscalYearEndDate where $fact.is-month-day != true}` **Identifies if the value of CurrentFiscalYearEndDate is valid.**
is-leap-year	Identifies if a year(number) is a leap year	*Property:* `year(date(“2017-12-31”)).is-leap-year`
time-span()	Allows a span of time to be defined for example time-span(“P90D”). This uses the XML duration format to return the number of days in the period. This can also be used as a property of any duration.	*function:*`$document_period_end_date + (time-span("P4D"))` **Adds 4 days to the value of document period end date.**
year()	Returns the year (number) from a fact or the year from a given date. given date.	*Property:* `{where $fact.end.year == '2017'}` *Function:* `year(date('2017-12-31'))` **The function and property returns a value of 2017.**

Aggregation Functions

Aggregation functions only work on sets and lists. The values returned from a fact query need to be expressed as a list or set before the aggregation functions below can be used. These functions can also be represented as a property of a list or set. I.e. list(1,2,3).avg

Name	Definition	Examples
avg(set|list)	Will return the average of values in a set or a list.	*Function*: `avg(list({@PlanAssets @DefinedBenefitPlansDisclosuresDefinedBenefitPlansAxis = *}))` **This will return the average value of plan assets across all plans.**

count(set|list)	Will return the count of values in a set or a list.	*Function* : `count(set(a,b,c,d,e))` **This will return a value of 4**
max(set|list)	Returns the maximum value of a set or a list	*Function:* `max(list({@concept = PlanAssets @@DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))` **This will return the largest value of the Plan Assets from all plans.**
min(set|list)	Returns the minimum value in a set or list. This can be numerical or string items. None cannot appear in the list.	*Function:* `min(list( {@concept = PlanAssets @@DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))` **This will return the lowest value of the Plan Assets from all plans.**
sum(set|list)	Returns the sum of a set or a list, this operates on any numerical and string type. Strings are concatenated.	*Function:* `sum(list({@concept = PlanAssets @@DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))` **This will sum the value of the Plan Assets for every Plan member.**
stdev(set|list)	Returns the standard deviation of a set or a list.	*Function:* `stdev(list({@concept = PlanAssets @@DefinedBenefitPlansDisclosures DefinedBenefitPlansAxis = *}))` **This will return the stdev of the Plan Assets for every Plan member.**
prod(set|list)	Returns the product of a set or a list. The prod of an empty list or set returns 1.	*Function:* `prod(list({@concept = PlanAssets @DefinedBenefitPlansDisclosuresDefinedBenefitPlansAxis = *}))` **This will return the product of Plan Assets for every Plan member.**

all(set|list)	Returns true if all values in a list or a set are true	*Function:* `all(list(true,false,false,false))` *Property:* `list(true,false,false,false).all` **This will return the value of false.**
any(set|list)	Returns true if any value is true in a list or a set.	*Function:* `any(list(true,false,false,false))` **This will return the value of true.**
first()	Returns the first value found in a list	*Function:* `first(list(1,2,3,4))` **This will return the value of 1.**
last()	Will return the last value in a list.	*Function:* `last(list(1,2,3,4))` **This will return the value of 4.**
denone()	Removes none values from a list or set	*Function:* `denone(list(1,2,3,none,4)` *Property:* `list(1,2,3,none,4).denone` **This will return the list(1,2,3,4).**

Statistical Functions

Name	Definition	Examples
corr(*Y*, *X*)	correlation coefficient	
regr_r2(*Y*, *X*)	square of the correlation coefficient	

Existence Functions

Name	Definition	Examples
exists()	Tests for the existence of a fact, object. If any fact exists then a	**exists({@Assets})**

	value of true is returned. Exists is applicable for checking the existence of facts. Exists will return a value of true when a none value is returned in a set or list. If a fact has a value of nil in the instance then exists returns true.	**Tests if any fact value is reported for assets and returns true or false** **exists(**list()**)** **Returns a value of true.**

missing()	Tests for the existence of a fact. If a fact does not exist then a value of true is returned.	**missing({*covered* @**Assets**})**
		
		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.
		
		missing(list()**)**
		Returns a value of false.

first-value	Takes a list of expressions and returns the first expression that has a value. If no value is returned then no value is returned and the iteration is skipped. This is the same as a fact query that does not return any facts. I.e it is unbound. The function calculates values until it gets one that is not none.	**first-value({@**Assets**},{@**CurrentAssets**})** **Returns the value of CurrentAssets if Assets is not present. If neither value is present then the iteration is skipped.**
first-value-or-none **(V1.2)**	This does the same thing as first-value(), except if there are no values in the arguments, then it returns a 'none' value instead of skipping the iteration.	**first-value-or-none({@**Assets**},{@**CurrentAssets**})** **Returns the value of CurrentAssets if Assets is not present. If neither value is present then a none value is returned.**

Unit Functions

Name	Definition	Examples

unit()	Pass this function the unit URI and local name to define a comparable unit with the instance	**{@unit = unit(iso4217:USD)}** **Returns all fact values reported in USD**
convert-unit(fact, resulting_unit)	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 lengthItemType cannot be converted to a unit with a datatype of areaItemType.	**convert-unit**($fact, unit(uri:m))

DTS Functions

Name	Definition	Examples
qname(namespace, local-name)	Creates a valid qname by providing the namespace-uri and localname as parameters.	`qname($ext_namespace,'FairValueInputsLevel2AndLevel3Member')` **Defines a qname based on the extension namespace**
taxonomy()	Creates a taxonomy object based on the taxonomy entry point. If no parameter is passed the DTS of the current instance is used.	`$US-GAAP = taxonomy('http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-2016-01-31.xsd')` Returns the US-GAAP taxonomy **taxonomy**()

		Returns the DTS of the current instance
entry-point-namespace(taxonomy)	Returns the namespace of the entry point for the taxonomy (DTS) used. Takes the taxonomy object as a parameter.	**entry-point-namespace**($US-GAAP) **Returns the namespace of the us-gaap taxonomy** "http://xbrl.fasb.org/us-gaap/2019"
entry-point()	Returns the entry point uri of the taxonomy object passed as a parameter	**entry-point**($US-GAAP) **Returns the uri** http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-2016-01-31.xsd
dts-document-locations	Returns the location of all documents comprising the dts.	**dts-document-locations**($us-gaap) **Returns the uri of all dts documents as a set.**

Range Function

Name	Definition	Examples

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 integer, two arguments are the start and stop integers. The default start integer is 1. The default step integer is 1 unless specified otherwise.	`range(5)`
		Returns:
		`list(1,2,3,4,5)`
		
		`range(4,10)`
		Returns:
		`list(4, 5, 6, 7, 8, 9, 10)`
		
		`range(4,10,2)`
		Returns:
		`list(4, 6, 8, 10)`
		
		Use in a loop with list $path:
		`for $i in range($path.length)`
		`      $path[$i]`

Data Import and Transformation Functions

Name	Definition	Examples

csv-data()	csv-data takes 4 arguments. The first 2 are required.
	
	1. file-url: (required) Location and name of the file
	2. has headers: (required) either true or false. If it is true, the first line is ignored. Defaults to false.
	3. list of types: (optional). If supplied it must have a type for each column. If omitted the columns will be strings. Possible values are string, date, decimal, float, int, boolean, qname, list().
	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.
	
	The function returns a list of rows. The row is either a list or a dictionary (if the 4th argument is supplied and true).
	
	Limitations:
	
	● If the list of types is supplied, no row can have more columns than the length of the list of types.
	● If headers are in the file and returning as a dictionary, no row can have more columns than the header row.
	Defining types
	Types are defined as a list. I.e. list('string', 'date', 'date'). In some cases the format of the csv is formatted. In these cases XBRL transforms can be used to transform the data to the appropriate type. The transform and type are provided as a
		`csv-data('https://xbrl.us/dqc_06_date_bounds.csv',true, list( 'string', 'string', 'string'))`
		
		Defines a csv file as a source of data.
		
		`csv-data('https://xbrl.us/trans.csv', true, list('string',list(ix4:num-dot-decimal, 'decimal'), list(ix4:date-day-month-year, 'date'), list(ix4:date-day-monthname-year-fr, 'date')) )`
		
		Defines a csv format for the following:
		name,amount,date,date2
		Mark,"1,234,567.89",06/05/2020,05-juin-1900
		Phillip,"9,876.54",09/01/2019,01-septembre-1901

	list i.e. list(ix4:date-day-month-year, 'date'),	
excel-data() **(V1.2)**	Excel data takes 5 arguments. The first arg is required 1. file-url: (required) Location and name of the file 2. Range: (optional) This can be one of the following: a. name of sheet, b. named range, c. cell range i.e sheet1!A1:D3 If not defined it defaults to the active sheet. To add subsequent parameters a value of none must be provided. 3. Has headers: (optional) Boolean value where the default is false. 4. list of types: (optional). If supplied it must have a type for each column. If omitted the columns will be strings. Possible values are string, date, decimal, float, int, boolean, qname, list(). 5. 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.	`excel-data('https://xbrl.us/excel-file.xls', 'sheet1!A1:D3',true,list('string',list(ix4:num-dot-decimal, 'decimal'), list(ix4:date-day-month-year, 'date'), list(ix4:date-day-monthname-year-fr, 'date')) )` **Defines a an excel file with the following in sheet1:** name,amount,date,date2 Mark,"1,234,567.89",06/05/2020,05-juin-1900 Phillip,"9,876.54",09/01/2019,01-septembre-1901
json-data()	json-data takes 1 parameter which is the url to the json file. The function returns a dictionary which matches the structure of the json file.	`json-data('https://xbrl.us/json-file.json')` **Defines a json file as a source of data.**

xml-data-flat() **(V1.2)**	xml-data-flat takes.takes 3 to 5 arguments. The first 3 are required:
	1. file url
	2. xpath expression to find the nodes you want
	3. list of xpath expressions to return the field you want. The node will be the starting point for the xpath
	4. list of return types. There should be 1 for each field in the 3rd argument. This is optional. if not supplied, the result will be a string
	5. a dictionary of namespace mappings. The key is the prefix and the value is the namespace. This is optional. The namespaces declared in the rule file are always available. Default namespaces are not supported. Any namespace must have a prefix.
		`xml-data-flat('http://www.xbrl.org/lrr/lrr.xml','lrr:arcroles/lrr:arcrole', list('lrr:roleURI', 'lrr:status'))`
		
		Returns a list of role URI and status.
		
		`xml-data-flat('http://www.xbrl.org/lrr/lrr.xml', '/lrr:lrr', list('@version', 'lrr:status'), list('decimal', 'string'))`
		
		The version number is converted to a decimal
		
		`xml-data-flat('https://www.sec.gov/Archives/edgar/data/1099219/000162828023018062/information_table.xml', 'j:infoTable', list('j:nameOfIssuer', 'j:titleOfClass', 'j:cusip','j:value'), list('string','string','string', 'decimal'), dict(list('j','http://www.sec.gov/edgar/document/thirteenf/informationtable')))`
		
		Example showing namespace map

Information Functions

Name	Definition	Examples
rule-name()	rule-name returns the name of the current executing rule. The rule	**`output abc`**

	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.
		`rule-name()`
		
		Returns:
		`'abc'`

alignment() **(V1.2)**	Returns a dictionary of the alignment for a fact. Can only be used as a function.	`$alignment = alignment()` **Returns a dictionary with keys of entity, unit, period, concept, dimension and the associated values for the keys**
_type()	Returns the object type of a xule object.	

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)
```

Recognising Qnames

The grammar parser automatically identifies qnames. However because a period "." is used to identify a property the parser cannot always accurately differentiate a qname from a property. If the qname contains a period then a backslash should be used to indicate that a period comprises the qname. For example **`aapl:FixedRateRangefrom0.875to4.300`** contains two periods. To identify this as a qname if written in XULE a backslash "\" is used to identify the period as a qname. I.e. **`aapl:FixedRateRangefrom0\.875to4\.300`**

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 dei = http://xbrl.sec.gov/dei/2014-01-31
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.

Namespace Group (V1.2)

In some cases it is unknown what the namespace of an element will be exactly. Most sets of XULE expressions expect a given taxonomy, but this is not always assured. In many cases one filer will file with the 2022 US-GAAP taxonomy and the 2022 SEC DEI taxonomy. In some cases filers will use the 2022 US-GAAP taxonomy and the 2021 SEC DEI taxonomy. Historically this issue has been addressed by using element local-names to avoid the mismatch of concepts using a prior version or later version of the taxonomy. Namespace groups averts this issue by using qname prefixes that can represent a group of namespaces.

The following shows two dei namespaces defined as follows:

namespace dei = http://xbrl.sec.gov/dei/2021

namespace dei-2022 = http://xbrl.sec.gov/dei/2022

A namespace group can be defined such as dei-all (Or any name) . This is then associated with any namespace containing the text dei.

```
namespace-group dei-all = list('dei')
```

To define a rule that selects all the fact values for the dei element DocumentPeriodEndDate in the filing the following filter can be used:

```
{@concept = dei-all:DocumentPeriodEndDate}
```

As opposed to the following:

```
{@concept.local-name = 'DocumentPeriodEndDate'}
```

Using namespace group eliminates the risk of duplicate element name clashes across different taxonomies. It also means that single rulesets can be defined that can be used across updated releases of a taxonomy which has an incremented namespace.

A namespace group prefix can only be used in a fact query expression.

Assertion Types

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

Assertion Type	Description
assert	This will perform a value assertion. This requires a boolean result and will

	produce output dependent on the assertion
output	This will return the results without a severity

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”;
{@concept.is-numeric = true}#nonnegitem < 0

message
“The value of {$nonnegitem.concept} with a value of {$nonnegitem} is less than zero. Please
enter a positive amount”
severity
$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”;
{@concept.is-numeric = true}#negativeitem <= 0

message
“The value of {$negativeitem.concept} with a value of {$negativeitem} is greater than zero.
Please enter a negative amount.”
severity
$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 an output assertion. If no severity is provided then this defaults to “error” for assertions. Output defaults to a severity of info.

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 attribute. 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
      rule-focus $nonnegitem
```

[7] For consistency with the XBRL formula specification.

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 fact query needs to be passed to the rule output. Rather than forcing every fact query 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 fact query. 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 key word '**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, then 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.

Character	Defined as
Quote	\”
tab	\t
return	\n
Curly Bracket	\{

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.

```
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:

```
    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.

Predefined Output Attributes

Xule includes output attributes that allow XULE to output rule information to a file. The attribute `file-content` defines the content that will be output. This has to be a string and can be derived from the to-json, to-csv, to-spreadsheet or string properties. The content cannot contain XULE objects. The output attribute `file-location` defines where the file content is written to.

Multiple rules can be written to the same file. To prevent the previous content from being overwritten the attribute `file-append` is used. The value of `file-append` can be true or false.

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 fact query is encountered iterations are added for each fact. 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 are reported for 3 periods the rule will test that assets are less than zero for all three periods. If assets are 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 a fact and the fact does not exist then no message is output. The fact 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 fact query then a single iteration will occur. This is because when a rule executes a single iteration will occur if no facts or for loops are encountered. The alignment is removed from a fact query 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 Fact Queries

In the above examples the iteration impacted a single fact query. In some cases multiple fact queries 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 occurences 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.

Lists, Sets and Iterations

The use of a list or a set creates an unbound iteration. The iteration allows processing to happen in the set. For example a list can be used to check the existence of a fact as follows:

```
output exits_iteration
If (list({@BelowMarketLeaseAcquired}).length > 0)
“This item exists and this string is output for every occurrence of the fact.”
```

```
else
"If  BelowMarketLeaseAcquired does not exist the unbound list iteration will be used and this string will be reported."
```

This has the advantage over the exists that the covered does not have to be used. The covered has the disadvantage that all the facts are taken out of alignment and the existence function will return true when you may be processing a specific alignment where that fact does not actually exist.

FACT Iterations versus FOR iterations

Facts defined in a list with multiple facts in different alignments will create a list for each fact alignment. For example the following list:

```
output Assets_in_list
list(@concept = Assets)
```

If there are three values for Assets this will create three iterations of the list with the value assets. If iterations are created using a for statement in a list these iterations are contained within the list to produce separate values within the list.

```
output For_Values_in_List
list(for $x in list(1,2,3)
            $x)
```

This will create a list with a value of `list(1,2,3)`. It will not create 3 lists with values of `list(1)`, `list(2)`, `list(3)`

SKIP in a list or Set

A skip appearing in a list or set will not cause the iteration to be skipped. A SKIP in a list or set is treated as an empty list. The following output will result in an empty list.

```
output skip_in_list
list(skip)
```

Any items in the list will be retained and the skip is ignored.

```
output skip_in_list
list(1,2,3,skip)
```

The above output will produce `list(1,2,3)`

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, then it should be defined 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 fact query when a filter can be used instead. I.e
  `{@Assets}` is more efficient than `{@ where $fact.concept == Assets}`

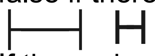
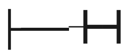
- Consider the use of lazy AND. The sooner you can find a false for an AND the more efficient the processor can be.

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.

Boolean Operators	Description
`<`	Less than
`>`	Greater than
`>=`	Greater than or equal
`<=`	Less than or equal
`==`	Equivalent to. Can be used on a set or a list
`!=`	Not equivalent to. Can be used on a set or list
`and`	Logical AND. This is lazy
`or`	Logical OR This is lazy
`in`	Evaluates if an item is in a set or a list.
Numeric Operators	
`+`	Addition of numbers, strings, lists and sets.
`-`	Subtraction, unary
`*`	Multiplication
`/`	Division
`<+>`	Addition that will occur if both left and right side are present
`<+`	Addition will only occur if left side is present.
`+>`	Addition will only occur if right side is present.
`<->`	Subtraction that will only occur if both values are present on either side of the subtraction.
`<-`	Subtraction will only occur if left side is present.

`->`	Subtraction will only occur if right side is present.
`power()`	Power. This is a property and not the operator ^ as this is used for symmetric difference.
Set Operators	
`+`	Union of 2 sets
`&, intersect`	Intersection of 2 sets
`-`	Difference of 2 sets
`<=`	Test if a set is a subset
`>=`	Test if a set is a superset
List Operators	
`+`	Union of 2 lists
`[]`	Return value of an index
Duration Operators	
`>`	Test if a time duration is **after** another time duration. Returns false if there is an overlap.  If the end and start date meet this is considered after. 
`<`	Test if a time duration is **before** another time duration. Returns false if there is an overlap.
`==`	Test if two durations are the same duration. The start and end times must match.
`!=`	Test if two durations are not the same duration.
`>=`	Test if a time duration is the same or after another time duraiton.
`<=`	Test if a time duration is the same
`in`	Test if a duration is wholly within another duration.
Date (left) and Duration (right) Operators	
`>`	Test if a date is after the end date of the duration
`<`	Test if a date is before the start date of a duration

`in`	Test if a date is greater than or equal to the start date of the duration and the date is less than or equal to the end date of the duration.

Order of Precedence	
`()`	Parenthesis
`#`	Tagging
`[ ]`	Index
`.`	Property expression
`+, -`	Unary, union
`* , /`	Multiplication and Division
`+, -`	Addition and subtraction
`&, intersect`	Set intersections
`^`	Symmetric Difference
`>,<,>=,<=,in,not in, == , !=`	Comparison Operators
`not`	
`and`	
`or`	

Appendix 2

Versions

The following features are available in either version 1.1 or 1.2 or planned in version 1.3.

No	Version	Feature Name	Detail
1.	1.2	Footnotes of a Fact	Added support for returning the footnotes of a fact. A footnote property was added that allows specific footnotes to be returned. I.e. $fact.footnote(role)
2.	1.3	Date Casting	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
3.	1.1	Parameters	Allows passing parameters to the XULE processor that match the constants in the XULE syntax. For example: a person running the rule passes the parameter called user with a value such as user=jblow. The message can then output who ran the rule as the constant is defined constant $user
4.	1.1	Regular Expressions	Added a string matching function to match regular expressions. Returns a match object and a simple version that returns the match. match(”regex pattern”, “string”) Returns true or false.
5.	1.1	Properties of none	The string value of a none value returns an empty string.
6.	1.1	Taxonomy Namespaces	Added a property to taxonomy() that returns the URI of the namespaces in the filing as a set.
7.	1.1	Default Handling	When filtering a fact query if a member is defined as the default member then all facts with the default will be returned. Previously if a member was identified that was the default then no values would be returned.
8.	1.2	Namespace Groups	Added namespace groups to allow the definition of a prefix that references multiple namespaces. This allows comparison when the namespace version of the XBRL file being processed is not known in advance. I.e. us-gaap 2022 vs us-gaap 2021

9	1.2	Ability to define and create instance documents	Added support to define a fact and output as an instance document. This support is provided as a separate plugin called XINCE. Xule has added additional functions to support this. Specifically the alignment function.
10	1.2.1	Ability to create a csv or json file from an output rule.	Added support to define a file and its content as part of the rule. This means you can query the taxonomy and output it in csv or json format of the users choice.
11	1.2	Ability to open multiple instance documents and include them in alignment.	Added support to open multiple instance documents. These are included in the fact query.
12	1.2	Added properties to support XINCE and XODEL and general requirements.	Added the following properties: 1. excel-data 2. xml-data-flat 3. first-value-or-none 4. inline-transform 5. random 6. roles 7. arc-roles 8. inline-parents 9. inline-children 10. inline-ancestors 11. inline-descendants 12. Instance : Instance of a fact 13. Footnotes: on a fact 14. cubes : Cubes a fact is applicable to 15. taxonomy : Taxonomy used by an instance 16. facts : facts in an instance 17. document location 18. to-csv

Upcoming Versions

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 upcoming versions.

In addition there is additional functionality we know needs to be addressed in future 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:

No	Version	Feature Name	Detail
1.	1.3	Include/Import	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.
2	1.3	Namespace declaration as an expression	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 xule would not be able to be run on a new processor implementing this functionality.
3	1.3	Update return types from a navigate	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.
4	1.3	Rule Link	Add a standard output attribute called rule link that links to the documentation of the actual rule. Such as https://xbrl.us/data-rule/dqc_0001/. This would allow users to easily navigate to the rule definition page in XULE enabled editors.
5	1.3	Run rules in streaming mode	Ability to define rules to run in streaming mode.
6	1.3	Namespace Groups	Currently namespace groups are a convenience for matching a concept name in a fact query, but it doesn't affect how the results of the fact query are aligned. For example, you are working with us gaap data that comes from 2 different versions: us gaap 2022 and us gaap 2021. If you have a fact query {@}, Revenues for 2022 would not be in the same alignment as Revenues for 2021. This is because the alignment uses the full namespace uri. This proposal is to use the namespace group name

			instead of the actual namespace uri for defining alignment. So if a namespace group is declared as: namespace-group = 'us-gaap' then a fact query: {@} would align Revenues regardless of the version of the us gaap taxonomy because they would have the same namespace group prefix. If a concept can be matched to more than one namespace group, an error would be raised. This would be
7	1.3	Concept Traits	Add a property of a concept called traits that are defined using the traits relationship defined in the link role registry. The traits property returns all traits associated with a concept using the traits-concept arc-role and the class-subclass arcrole. Add a property of a taxonomy called concepts-by-trait which takes a trait or a list of traits and returns the concepts that match all supplied traits.
8	1.3	Extensible Enumerations	Return the extensible enumeration values of a concept.
9	1.3	Filter on a dictionary of dimensions.	In some cases you do not know what dimensions to filter on until you inspect the filing. This feature allows you to define the filtered dimensions at run time by passing a dictionary of taxonomy defined dimensions and members to the fact filter. The key is the dimension and the value is the member. The values are passed as qnames or concepts. If the dimension is already defined in a fact query filter then the value in the dictionary of dimensions is ignored. The dimensions can be kept in alignment by using the double @@. The syntax is as follows {@concept=Assets @dimensions=$dictOfDimensions}

Appendix 3

Examples

DQC.US.0001.51

```
/** Define the base taxonomy as a constant**/
constant $US-GAAP = taxonomy('http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-
2016-01-31.xsd')

/** 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 2 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))


constant $US-GAAP = taxonomy('http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-
2016-01-31.xsd')
```

DQC.US.0045.6835

```
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(NetCashProvidedByUsedInOperatingActivitiesC
ontinuingOperations,
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,
IncreaseDecreaseInFederalFundsPurchasedAndSecuritiesSoldUnderAgreementsToRepu
rchaseNet, IncreaseDecreaseInRestrictedCash,
IncreaseDecreaseOfRestrictedInvestments)

constant $US-GAAP = taxonomy('http://xbrl.fasb.org/us-gaap/2016/elts/us-gaap-
2016-01-31.xsd')
```

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.\n EffectOfExchangeRateOnCashAndCashEquivalents
{NetCashProvidedByUsedInContinuingOperations}}

Rule Element Id:6839
Rule Version: 5.0.0"

severity
error
```

DQC.US.0047.7481

```
assert dqc.us.0047.7481 satisfied

for ( $cashOperating in
set(NetCashProvidedByUsedInOperatingActivitiesContinuingOperations,NetCashPro
videdByUsedInOperatingActivities))

      $misplaced_concept = navigate summation-item descendants from
      ($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
```

DQC.US.0049.7483

```
assert dqc.us.0049.7483 satisfied

$must_be_present_concepts =
set('CashAndCashEquivalentsPeriodIncreaseDecrease','CashPeriodIncreaseDecreas
e','CashAndCashEquivalentsPeriodIncreaseDecreaseExcludingExchangeRateEffect',
'CashCashEquivalentsRestrictedCashandRestrictedCashEquivalentsPeriodIncreaseD
ecreaseIncludingExchangeRateEffect','CashCashEquivalentsRestrictedCashandRest
rictedCashEquivalentsPeriodIncreaseDecreaseExcludingExchangeRateEffect')

$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
CashCashEquivalentsRestrictedCashAndRestrictedCashEquivalentsPeriodIncreaseDecreaseIncludingExcha
ngeRateEffect.  If the company specifically excludes the exchange rate effect from the total then the
element
CashCashEquivalentsRestrictedCashAndRestrictedCashEquivalentsPeriodIncreaseDecreaseExcludingExch
angeRateEffect 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 **/
constant $prior_filing = json-
data('https://csuite.xbrl.us/php/dispatch.php?Task=xbrlValues&API_Key=4cd4576a-aada-
4d35-9e40-dc7f0734f8d6&Accession=0001628280-17-
000717&Ultimus=false&DimReqd=false&Format=json')

/** 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
```

Appendix 4

XULE Properties

Property Name	Parameters (-ve is optional)	Object Applies To	Parameters
abs	0	int, float, decimal, fact	
agg-to-dict	1 or more	Set of lists, list of lists	
all	0	list, set	
all-labels	0	concept, fact	
all-references	0	concept, fact	
any	0	list, set	
arc-name	0	relationship	
arcrole	0	network, relationship	
arcrole-description	0	network, relationship	
arcrole-uri	0	network, relationship	
aspects	0	fact	
attribute	1	concept, relationship	
avg	0	list, set	
balance	0	concept	
base-type	0	concept, fact	
clark	0	Qname, concept, fact, reference-part	
concept	-1	fact, taxonomy, dimension	
concept-names	0	taxonomy, network	
concepts	0	taxonomy, network	
contains	1	list, set, string, uri	
content	0	footnote	
count	0	list, set	
cube	2	taxonomy	
cube-concept	0	cube	

cubes	0	taxonomy	
data-type	0	concept, fact	
date	0	string	
day	0	instant	
days	0	instant, duration	
decimal	0	Int, float, decimal, string, fact	
decimals	0	fact	
default	0	dimension	
denominator	0	unit	
description	0	role	
difference	1	set	set
dimension	1	fact, taxonomy	qname
dimension-type	0	dimension	
dimensions	0	fact, cube, taxonomy	
dimensions-explicit	0	fact, cube, taxonomy	
dimensions-typed	0	fact, cube, taxonomy	
document-location		all	
drs-role	0	cube	
dts-document-locations	0	taxonomy	
effective-weight	2	taxonomy	qname, qname
effective-weight-network	-3	taxonomy	qname, qname, network
end	0	instant, duration	
entity	0	fact	
entry-point	0	taxonomy	
entry-point-namespace	0	taxonomy	
enumerations	0	type, concept, fact	
fact	0	footnote	
facts	0	Cube, instance	

first	0	list, set	
footnotes	0	fact	
has-enumerations	0	type, concept, fact	
has-key	1	dictionary	
id	0	entity, unit, fact	
index	1	list, dictionary	
index-of	1	string, uri	
inline-ancestors	0	fact	
inline-children	0	fact	
inline-descendants	0	fact	
inline-display-value	0	fact	
inline-format	0	fact	
inline-hidden	0	fact	
inline-negated	0	fact	
inline-parents	0	fact	
inline-scale	0	fact	
inline-transform	-2	string	
instance	0	fact	
int	0	Int, float, decimal, string, fact	
intersect	1	set	
is-abstract	0	concept, fact	
is-fact	0	fact	
is-monetary	0	concept, fact	
is-nil	0	fact	
is-numeric	0	concept, fact	
is-subset	1	set	
is-superset	1	set	
is-type	1	concept, fact	
join	-2	list, set, dictionary	string, string
keys	-1	dictionary	

label	-2	concept, fact	
lang	0	label, footnote	
last	0	list, set	
last-index-of	1	string, uri	Integer
length	0	list, set, string, uri, dictionary	
link-name	0	relationship	
local-name	0	qname, concept, fact, reference-part	
log10	0	int, float, decimal, fact	
lower-case	0	string, uri	
max	0	list, set	
min	0	list, set	
mod	1	int, float, decimal, fact	int, float, decimal, fact
month	0	instant	
name	0	fact, concept, reference-part, type	
namespaces	0	taxonomy	
namespace-map	0	fact	
namespace-uri	0	qname, concept, fact, reference-part	
network	0	relationship	
networks	-2	taxonomy	
number	0	int, float, decimal, fact	
numerator	0	unit	
order	0	relationship, reference-part	
part-by-name	1	reference	
part-value	0	reference-part	
parts	0	reference	
period	0	fact	
period-type	0	concept	
plain-string	0	all	
power	1	int, float, decimal	
preferred-label	0	relationship	

primary-concepts	0	cube	
prod	0	list, set	
references	-1	concept, fact	uri
regex-match	1	string, uri	string
regex-match-all	1	string, uri	string
regex-match-string	-2	string, uri	string, number
regex-match-string-all	-2	string, uri	string, number
relationships	0	network	
role	0	network, label, reference, relationship	
role-description	0	network, label, reference, relationship	
role-uri	0	network, label, reference, relationship	
roots	0	network	
round	1	int, float, decimal, fact	int, float, decimal, fact
scheme	0	entity	
sid	0	fact	
signum	0	int, float, decimal, fact	
sort	-1	list, set	string enum = “ASC”, “DESC”
source	0	relationship	
source-concepts	0	network	
source-name	0	relationship	
split	1	string, uri	string
start	0	instant, duration	
stdev	0	list, set	
string	0	all	
substitution	0	concept, fact	
substring	-2	string, uri	
sum	0	list, set	

symmetric-difference	1	set	
target	0	relationship	
target-concepts	0	network	
target-name	0	relationship	
taxonomy	0	instance	
text	0	label	
time-span	0	String, duration	
to-csv	1	list	
to-dict	0	list, set	
to-json	0	list, set, dictionary	
to-list	0	list, set	
to-qname	0	string	
to-set	0	list, set, dictionary	
to-spreadsheet	0	dictionary	
trim	-1	string, uri	string enum = “left”, “right”,”both”
trunc	-1	int, float, decimal, fact	
union	1	set	
unit	0	fact	
upper-case	0	string, uri	
uri	0	role, taxonomy	
used-on	0	role	
values	0	dictionary	
weight	0	relationship	
year	0	instant	

Appendix 5

XULE Functions that are not Properties

Function Name	Parameters	Optional Parameters	Parameters
alignment	0	0	Used to get the alignment of a fact.
csv-data	4	2	File url, has headers, list of types, output as dictionary
dict			Define a dictionary
duration	2	0	Start date, end date
excel-data	5	4	File url, range, has headers,list of types, output as dictionary
exists	1	0	Object to test exists.
first-value	unlimited		Must have at least one parameter
first-value-or-none	unlimited		Must have at least one parameter
json-data	1	0	File url
missing	1	0	Object to test is missing.
qname	2		
random	1	1	Number of decimal places, defaults to 4 if none provided
rule-name	0	0	Returns the rule name.
range	3	2	Accepts integers the first is the stop, the second is the start and the third is the step.
xml-data-flat	5	3	File url, xpath expression, xpath expressions return,list of return types, output as dictionary

Appendix 6

Error Codes

Run Time Errors

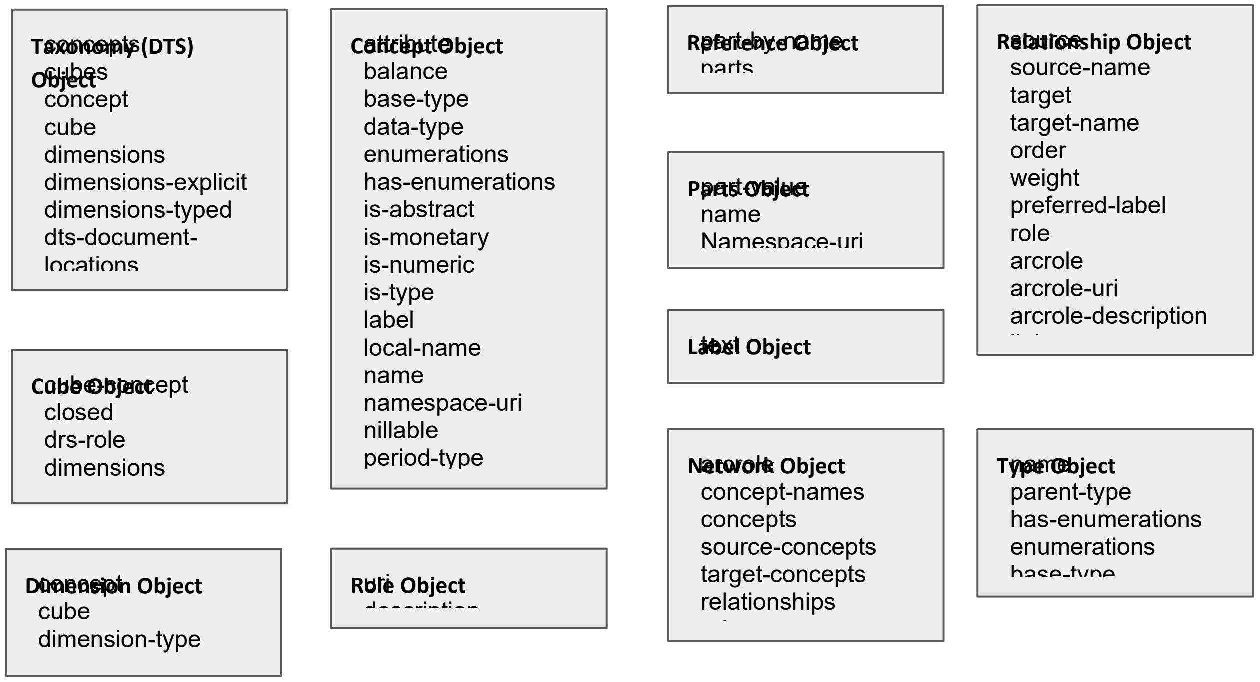
Error Code	Description
DivByZero	Represents a division by zero error.
ForLoopNotSetList	The for loop must run over either a set or a list.
IndexOutOfRange	Index value does not exist in the list
InvalidArgument	The argument to the function or property is not valid
InvalidProperty	The property used with the object is not valid.
NotIndexable	The index expression '[]' can only operate on a list or dictionary. An index expression on a different object such as a set will result in an error.
ShortNameDuplicate	The role short name resolves to more than one arcrole in the taxonomy.
TooFewArguments	The function or property contains too few arguments
TooManyArguments	The function or property contains too many arguments
UnsupportedOperandType	An operator is used on a type that does not support the operand.

Compile Errors

Error Code	Description
DuplicateName	A duplicate name for a rule, an output, a constant, a namespace or a namespace group has been defined.
DuplicatePrefix	Duplicate namespace prefix with different namespace.
MissingVariable	A variable is not defined or cannot be found.

MissingNamespacePrefix	Namespace prefix does not have a namespace or namespace-group declaration.
NoOutputAttributeDefined	In rule the result name is not defined as an output-attribute.

Taxonomy Object Model



Taxonomy (DTS) Object
concepts
cubes
concept
cube
dimensions
dimensions-explicit
dimensions-typed
dts-document-locations

Cube Object
cube-concept
closed
drs-role
dimensions

Dimension Object
concept
cube
dimension-type

Concept Object
attribute
balance
base-type
data-type
enumerations
has-enumerations
is-abstract
is-monetary
is-numeric
is-type
label
local-name
name
namespace-uri
nillable
period-type

Rule Object
uri
description

Reference Object
part-by-name
parts

Parts Object
part-value
name
Namespace-uri

Label Object
text

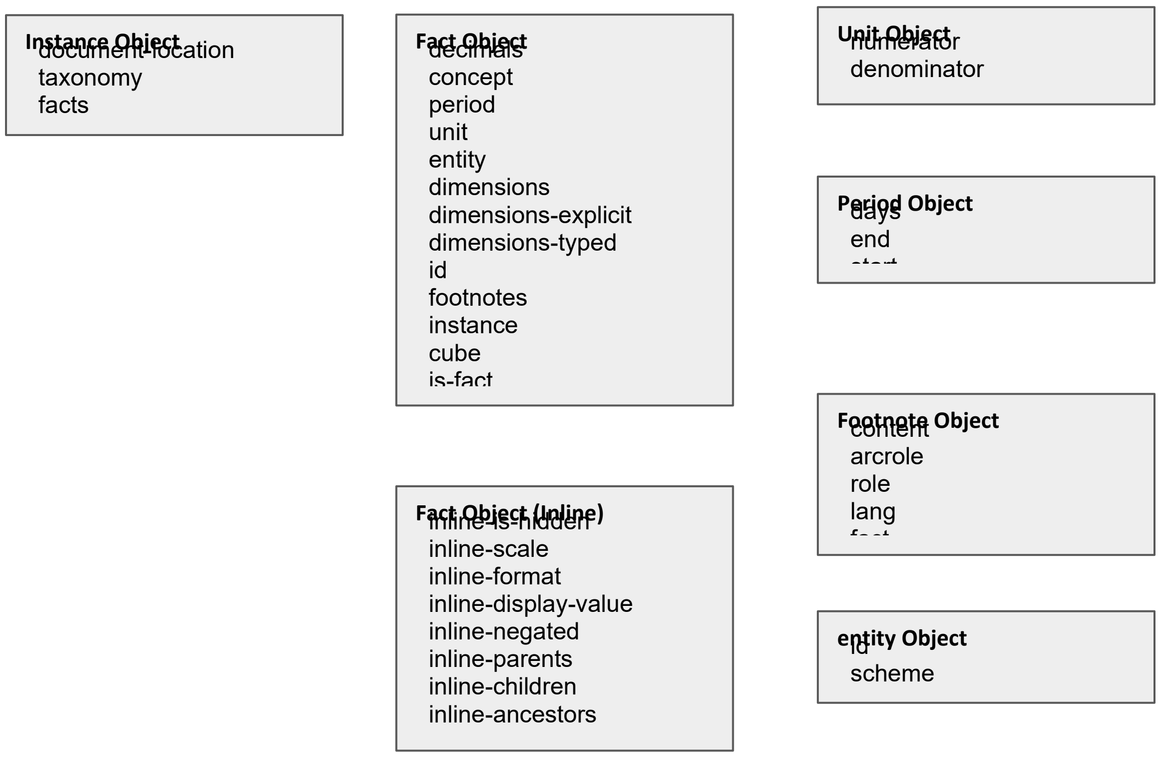
Network Object
arcrole
concept-names
concepts
source-concepts
target-concepts
relationships

Relationship Object
source
source-name
target
target-name
order
weight
preferred-label
role
arcrole
arcrole-uri
arcrole-description

Type Object
name
parent-type
has-enumerations
enumerations
base-type

Appendix 8

Instance Object Model



Appendix 9

EBNF Grammar

```
QueryFile ::= ( QueryNameSeparator | QueryNamePrefix | NamespaceGroupDeclaration |
NamespaceDeclaration | OutputAttributeDeclaration | FunctionDeclaration |
ConstantDeclaration  | AssertDeclaration | OutputDeclaration )

FunctionDeclaration
         ::= 'function' FunctionName '(' ( '$' ArgName ( ',' '$' ArgName  )* )? ')'
FunctionBody

FunctionName ::= SimpleName

ArgName ::= SimpleName

FunctionBody ::= Expr


NamespaceGroupDeclaration ::= "namespace-group" NamespaceGroupName "=" Expr

NamespaceDeclaration ::= 'namespace' (Prefix '=')? NamespaceURI

OutputAttributeDeclaration ::= "output-attribute" OutputAttributeName

NamespaceGroupName ::= SimpleName

OutputAttributeName ::= SimpleName

ConstantDeclaration ::= 'constant' '$' ConstantName "=" Expr

ConstantName ::= SimpleName

AssertDeclaration ::= "assert" NCName ( "satisfied" | "unsatisfied" )? Expr
QueryResult*

OutputDeclaration ::= "output" NCName Expr QueryResult*

QueryResult ::= ( ( ("message" LanguageExpr?) | "severity" | "query-suffix" | "query-
focus" | OutputAttributeName  ) Expr )

LanguageExpr ::= Expr

Expr ::= FuncRef | VarRef | BlockExpr | PropertyExpr | IndexExpr | TaggedExpr | IfExpr
| ForExpr | Navigate | Filter | FactQuery | UnaryExpr | BinaryExpr  | FloatLiteral |
IntegerLiteral | StringLiteral | BooleanLiteral | SeverityLiteral | BalanceLiteral |
PeriodTypeLiteral | NoneLiteral | SkipLiteral | ForeverLiteral | QName

FuncRef ::= FunctionName '(' (Expr (',' Expr )* )* ')'

VarRef ::= '$' VarName
```

```
VarName ::= SimpleName

BlockExpr ::= VarDeclaration ( VarDeclaration)* Expr

VarDeclaration ::= '$' VarName '=' Expr ";"?

IndexExpr ::= Expr '[' Expr ']'

TaggedExpr ::= Expr '#' SimpleName

IfExpr ::= 'if' ConditionExpr Expr 'else' Expr

ForExpr ::= 'for' '$' VarName 'in' Expr Expr

BinaryExpr ::= Expr ("*" | "/" | "+>" | "->" | "+" | "-" | "<+>" | "<+" | "<->" | "<-"
| "^" |"&" | "intersect"| "==" | "!=" | "<=" | "<" | ">=" | ">" | 'in' | ( 'not' 'in')
| 'and' | 'or' ) Expr

UnaryExpr ::= ('+'| '-'| 'not') Expr


Navigate ::= 'navigate' ('dimensions' | 'across networks')?
Arcrole? Direction
'include start'?
('from' Concept)?
('to' Concept)?
('stop when' Expr)?
('role' Role)?
('drs-role' DRSRole)?
('linkbase' Linkbase)?
('cube' Hypercube)?
('taxonomy' Taxonomy)?
('where' Expr)?
('returns' ('by network')? ( 'list' | 'set' )?
          'paths'? (ReturnComponents ('as' ('dictionary' | 'list'))?) )?


ReturnComponents ::=  '('? (ReturnComponent (',' ReturnComponent )* )* ')'?

ReturnComponent ::= ('source' | 'source-name' | 'target' | 'target-name' | 'order' |
                    'weight' | 'preferred-label' | 'preferred-label-role' |
'relationship'|
                    'role' |'role-description' | 'arcrole' | 'arcrole-uri'|'arcrole-
description'|
                    'arcrole-cycles-allowed'| 'link-name' |'arc-name' |'network'|
                    'cycle' | 'navigation-order' | 'navigation-depth' | 'result-order'
|
                    'dimension-type' | 'dimension-sub-type' | 'drs-role' | QName)

Arcrole ::= Expr

Role ::= Expr

Taxonomy ::= Expr
```

```
Concept ::= Expr

Linkbase ::= Expr

Hypercube ::= Expr

DRSRole ::= Expr


Direction ::= 'descendants' | 'children' | 'ancestors' | 'parents' | 'siblings' |
'previous-siblings' | 'next-siblings'

ConditionExpr ::= Expr

PropertyExpr ::= Expr '.' PropertyName

PropertyName ::= SimpleName

Filter ::= 'filter' Expr ('sort' '(' (Expr ('desc'|'asc')?) (',' (Expr
('desc'|'asc')?))* ')') ?  ('where' Expr)? ('returns' Expr)?

FactQuery ::=
('{' FactQueryBody '}') |
('[' FactQueryBody ']') |
(FactQueryBody)

FactQueryBody ::=  ('covered'|'covered-dims')? (('nils'
'nildefault'?)|('nildefault'?)|'nonils')? (DimensionFilter)* ('where' Expr)?

DimensionFilter ::= ('@' | '@@') (DimensionName (('=' DimensionExpr)?| ('!='
DimensionExpr)?| ('in'  DimensionExpr)? |('not in' DimensionExpr)?)  ('as' NCName)? )?

DimensionName ::= ('concept' | 'unit' | 'entity' | 'period' | TaxonomyDimension |
'cube' | 'dimension'| 'instance')

StringLiteral ::= '"' [^"]* '"' | "'" [^']* "'"

NameStartChar
         ::= ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6] | [#xF8-#x2FF] |
[#x370-#x37D] | [#x37F-#x1FFF] | [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] |
[#x3001-#xD7FF] | [#xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]

NameChar ::= NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-#x036F] | [#x203F-
#x2040]

Name     ::= NameStartChar (NameChar)*

NCName   ::= Name - (Char* ':' Char*)
          /* An XML Name, minus the ":" */

QName    ::= PrefixedName
           | UnprefixedName

PrefixedName
```

```
         ::= Prefix ':' LocalPart

UnprefixedName
         ::= LocalPart

Prefix   ::= NCName

LocalPart
         ::= NCName

SimpleName ::= NameStartChar (SimpleNameChar)*

SimpleNameChar ::= NameStartChar | "-" | [0-9] | #xB7 | [#x0300-#x036F] | [#x203F-
#x2040]

Sign ::= "+" | "-"

SciNot ::= "e"

DecimalPoint ::= "."

Digits ::= [0-9]+

IntegerPart ::= sign digits

IntegerLiteral ::= integerPart

InfLiteral ::= sign "INF"

FloatLiteral ::= ( ( DecimalPoint Digits SciNot IntegerPart ) |
                    ( IntegerPart DecimalPoint Digits [0-9]* SciNot IntegerPart ) |
                    InfLiteral ) "float"

StringEscape ::= "\\"

StringExpr ::= "{" Expr "}"


BooleanLiteral ::= ( "true" | "false" )

NoneLiteral ::= "none"

SkipLiteral ::= "skip"

ErrorLiteral ::= "error"

WarningLiteral ::= "warning"

OkLiteral ::= "ok"

PassLiteral ::= "pass"

SeverityLiteral ::= ( ErrorLiteral | WarningLiteral | OkLiteral | PassLiteral )

BalanceLiteral ::= ( "debit" | "credit" )
```

```
PeriodTypeLiteral ::= ( "instant" | "duration" )

ForeverLiteral ::= "forever"
```

Appendix 10 - Arelle Reference Implementation of XULE

Command Line Instructions for XULE Plugin

Command	Description
`--xule-compile XULE_COMPILE`	Xule files to be compiled. This may be a file or directory. When a directory is provided, all files in the directory will be processed. Multiple file and directory names are separated by a '|' character.
`--xule-compile-type=XULE_COMPILE_TYPE`	Determines how the compiled rules are stored. Options are 'pickle', 'json'. Json files are significantly larger than native pickle files, but are readable in an editor.
`--xule-compile-workers`	Used to specify the number of processes for compiling rules in parallel. The default value is 1 and a value of 0 will use the number of CPUs available on the system.
`--xule-rule-set XULE_RULE_SET`	RULESET to use. This is the zip file created from compiling the rules. This file is then passed as a parameter when executing the rules
`--xule-run`	Indicates that the rules should be processed.
`--xule-arg XULE_ARG`	Redefines a constant. In the form of 'name=value'. This allows constants to be parsed to the processor such as a username or any other parameter that is used in processing the ruleset. The constant must exist in the ruleset I.e myDate=2022-12-31. Multiple arguments can be passed as follows: **`--xule-arg abc=value --xule-arg xyz=value2`**
`--xule-add-packages=XULE_ADD_PACKAGES`	Add packages to a xule rule set. Multiple package files are separated with a |.
`--xule-remove-packages=XULE_REMOVE_PACKAGES`	Remove packages from a xule rule set. Multiple package files are separated with a |.
`--xule-show-packages`	Show list of packages in the rule set.

`--xule-bypass-packages`	Indicates that the packages in the rule set will not be activated.
`--xule-time XULE_TIME`	Output timing information. Supply the minimum threshold in seconds for displaying timing information for a rule. I.e. `--xule-time .005`
`--xule-trace`	Output trace information.
`--xule-trace-count=XULE_TRACE_COUNT`	Name of the file to write a trace count.
`--xule-debug`	Output trace information.
`--xule-debug-table`	Output trace information.
`--xule-debug-table-style=XULE_DEBUG_TABLE_STYLE`	The table format. The valid values are tabulate table formats: plain, simple, grid, fancy_gri, pipe, orgtbl, jira, psql, rst, mediawiki, moinmoin, html, latex, latex_booktabs, textile.
`--xule-test-debug`	Output testcase information.
`--xule-crash`	Output trace information if get a xule:error
`--xule-pre-calc`	Pre-calc expressions
`--xule-filing-list=XULE_FILING_LIST`	File name of file that contains a list of filings to process. The filing list can be a text file or a JSON file. If it is a text file, the file names are on separate lines. If the file is a JSON file, the JSON must be an array. Each item in the array is a JSON object. The file name is specified with 'file' key. Additional keys can be used to specific --xule options to use. These options will override options specified on the command line. Example: [{'file' : 'example_1.xml}, {'file' : 'example_2.xml', 'xule_rule_set'}]
`--xule-max-recurse-depth=XULE_MAX_RECURSE_DEPTH`	The recurse depth for python. The default is 2500. If there is a 'RecursionError: maximum recursion depth exceeded' error this argument can be used to increase the max recursion depth.
`--xule-stack-size=XULE_STACK_SIZE`	Stack size to use when parsing rules. The default stack size is 8Mb. Use 0 to indicate that the operating system default stack size should be used. Otherwise

	indicate the stack size in megabytes (i.e. 10 for 10 Mb).
`--xule-server=XULE_SERVER`	Launch the webserver.
`--xule-multi`	Turns on multithreading
`--xule-cpu=XULE_CPU`	Overrides number of cpus per processing to use.
`--xule-async`	Outputs on screen output as the filing is being processed.
`--xule-numthreads=XULE_NUMTHREADS`	Indicates number of concurrent threads will run while the Xule Server is active
`--xule-skip XULE_SKIP`	List of rules to skip. Rules are comma separated with no space.
`--xule-run-only XULE_RUN_ONLY`	List of rules to run. Rules are comma separated with no space.
`--xule-no-cache`	Turns off local caching for a rule.
`--xule-precalc-constants`	Pre-calculate constants that do not depend on the instance.
`--xule-exclude-nils`	Indicates that the processor should exclude nil facts. By default, nils are included.
`--xule-include-dups`	Indicates that the processor should include duplicate facts. By default, duplicate facts are ignored.
`--xule-version`	Display version number of the xule module.
`--xule-display-rule-set-map`	Display the rule set map currently used.
`--xule-update-rule-set-map=XULE_UPDATE_RULE_SET_MAP`	Update the rule set map currently used. The supplied file will be merged with the current rule set map.
`--xule-replace-rule-set-map=XULE_REPLACE_RULE_SET_MAP`	Replace the rule set map currently used.
`--xule-reset-rule-set-map`	Reset the rule set map to the default.
`--xule-validate`	Validate ruleset
`--logNoRefObjectProperties`	Excludes fact properties from the log file.

`--xule-output-constants CONSTANT_NAMES`	Will return the value of constants, provided as a comma separated list.
`--xule-rule-stats-file FILENAME.json`	Creates a json file showing statistics of xule run must specify file name and path. Leave a space between the option and the filename
`--xule-rule-stats-log`	Outputs the statistics of xule run to the log file. Defaults to false. If option is there it adds to the log.
`--xule-args-file`	Loads constants from a json file prepared by --xule-output-constants-file
`--xule-output-constants-file`	File path to contain reloadable output constants, or write to log if absent.
xule/xulecat Plugin	
Explanation	Provides information about a ruleset zip file. Prints out the rule set file with version information and rule structure.
Example	`--plugin 'xule|xule/xulecat' --xule-rule-set dqc-us-2021-V20-ruleset.zip --xule-cat cat`
`--xule-cat cat`	Prints out the rule set file with version information and rule structure.
xule/saveXuleQNames Plugin	
Explanation	Use to generate a list of qnames from a taxonomy that can be used by an editor to check that qnames are valid.
Example	`--plugin 'xule/savexuleqnames' --xule-qnames-dir '/GitHub/xule.dqc/dqc_us_rules/taxonomy/us/2023/' --xule-qnames-format json -f 'https://xbrl.sec.gov/ecd/2023/ecd-2023.xsd' --noCertificateCheck`
`--xule-qnames-dir`	Directory where concept qnames are output as a json or xml file for use by a xule editor.
`--xule-qnames-format`	Format of the qnames file. This can be json or xml.

Glossary

Term	Definition
Alignment	The process of aligning facts with corresponding dimensions. For example the facts for the 2017 year end are aligned so that Revenue for 2017 is aligned with Expenses for 2017 so that Net Income for 2017 can be calculated correctly. These facts have been aligned on the period dimension, the unit dimension and the entity dimension. They are not aligned on the concept dimension.
Arc	
Arcrole	
Assertion	Term used to define a rule assertion. The rule will return a boolean result if the assertion passed or failed.
Attribute	A property of an element including its name, balance, data type, and whether the element is abstract.
Axis	(pl. axes) – An instance document contains facts; an axis differentiates facts and each axis represents a way that the facts may be classified. For example, Revenue for a period might be reported along a business unit axis, a country axis, a product axis, and so forth.
Balance	An attribute of a monetary item type designated as debit, credit, or neither; a designation, if any, should be the natural or most expected balance of the element “credit” or “debit” and thus indicates how calculation relationships involving the element may be assigned a weight attribute (-1 or +1).
Boolean	A data type expression used to indicate if a value an be true or false.
Child	Term used to define a relationship between two nodes when represented in a tree. A child node is further from the root element in a tree.
Concept	A taxonomy element that provides the meaning for a fact. For example, "Profit", "Turnover", and "Assets" would be typical concepts. [approximate technical term: concept (XBRL v2.1) or primary item (XBRL Dimensions). Concept, as defined here, excludes abstract concepts, and elements that are used to define hypercubes, dimensions and members]
Concepts	
Default	
Definition	
DEI	SEC Approved Taxonomy used to capture Document and Entity Information.
Descendants	All descendants of a node.i.e. The children of the children recurring.
Dictionary	A data store used to store data. A dictionary is a multi dimensional array.
Dimension	XBRL technical term for axis
DQC	Data Quality Committee of XBRL US.

DRS	
Element	XBRL components (items, domain members, dimensions, and so forth). The representation of a financial reporting concept, including: line items in the face of the financial statements, important narrative disclosures, and rows and columns in tables.
Entity	
Exists	A function used to determine if a value exists in an XBRL instance.
Expression	
Extended	
Extension	
Fact	Represents a fact value in an instance and the associated aspects and attributes of the fact.
Fact query	Represents the query that selects facts from an instance. Uses dimension filters and a where clause to select facts. The fact query is also used to control the alignment of facts.
Filter	
Fiscal	
Function	
GAAP	Generally Accepted Accounting Principles.
Instance	XML file that contains business reporting information and represents a collection of financial facts and report-specific information using tags from one or more XBRL taxonomies
Item	
Iteration	
Key	
Label	
Legal entity	
Length	
Linkbase	Technical construct that defines relationships, for example, those used to create a presentation tree or calculation tree.
List	
Local	
Loop	
Member	
Message	
Namespace	A namespace is a globally unique identifier that differentiate names created by

	different sources. In XBRL usage, namespaces are used to disambiguate the taxonomy element names defined in taxonomies. For example, different regional accounting standards might define a concept called "Profit". Namespaces are used to differentiate the UK GAAP definition of "Profit" from the US GAAP definition of "Profit". Namespaces are URIs, which are identifiers that follow the same format as URLs, which are used to locate resources on the internet.
Navigate	
Navigation	
Networks	
Networktype	
Number	
Object	
Order	
Output	
Parent	
Part	
Period	
Primary	
Property	
Qname	
Quarter	
Reference	
Relationship	
Reported	
Return	
Role	
Root	
Set	
Severity	
Source	
String	
Syntax	
Cube	A view of a taxonomy or report that is designed to replicate tables for presentation or data entry purposes. Cube structures are typically used

	to cope with the complex, dimensional reports often seen in prudential reporting.
Target	
Taxonomy	An XBRL taxonomy defines taxonomy components that provide meaning for the facts in an XBRL report. For example, a taxonomy for an accounting standard would include definitions of concepts such as "Profit", "Turnover", and "Assets". Taxonomies may contain a very rich set of information, including multi-language labels, links to authoritative definitions (for example, accounting standards or relevant local laws) , validation rules and other relationships.
Unit	
Values	
Variable	
Xule	
Uri	