ICARUS2 Query Language Specification

Markus Gärtner

2020

Contents

List of Figures

List of Tables

Todo list

1 Introduction

Queries in the IQL are designed to be self-contained with logical sections for specifying all the information required to determine the target of a query and its granularity, resolve additional dependencies such as extensions or scripts, link and validate constraints to parts of the target corpus or corpora and finally optional pre- and post-processing steps. To achieve this complex task IQL embeds a keyword-based syntax for the query payload within a JSON-LD structure to drive declaration of all the aforementioned information. As a side effect queries can become quite verbose and potentially cumbersome to define manually. As a countermeasure the overall structure of a query is composed of blocks that can be glued together incrementally and that make it very easy for an application built on top of it to provision boilerplate query code based on settings or a GUI so that the user only needs to type the actual constraints used in the query (the so called *query payload*, cf. Section 3). This document lists the basic building blocks of queries and their compositions.

Section 2 gives an overview on the JSON-LD elements available in the “outer” section of the IQL protocol. Section 3 provides the specification for the actual query language used in a query’s “inner” payload and Section 4 finally contains various collections of utility functions available to make queries both simpler to declare and more efficient to evaluate.

2 JSON-LD Elements

The “outer layer” of every serialized IQL query is a JSON-LD object with various (optional) properties to hold all the required information for proper query evaluation.

The following snippet shows the mockup of a simple query that searches the TIGER corpus for word pairs starting with an adjective and ending in a word with the surface form “test”, ignoring case and only returning up to 100 hits in a keyword-in-context (KWIC) style. The query also features an import statement for the STTS part-of-speech tagset, allowing more controlled expressions inside query constraints.

```
{ "@context" : "http://www.ims.uni-stuttgart.de/icarus/v2/jsonld/iql/query"
  "@type" : "iql:Query",
  "@id" : "query/000001",
  "iql:imports" : [ {
    "@type" : "iql:Import",
    "@id" : "import/000001",
    "iql:name" : "common.tagsets.stts"
  } ],
  "iql:setup" : [ {
    "@type" : "iql:Property",
    "iql:key" : "iql.string.case.off"
  } ],
  "iql:streams" : [ {
    "@type" : "iql:Stream",
    "@id" : "stream/000001",
    "iql:corpus" : {
      "@type" : "iql:Corpus",
      "@id" : "corpus/000001",
      "iql:name" : "TIGER v2"
    },
    "iql:rawPayload" : "FIND ADJACENT [pos==stts.ADJ][form==\"test\"]",
    "iql:result" : {
      "@type" : "iql:Result",
      "iql:resultTypes" : [ "kwic" ],
      "iql:limit" : 100
    }
  } ]
}
```

say something about the namespace and general iql: prefixing

2.1 Binding

A binding associates a collection of member variables (3.5) with the content of a specific item layer or derived layer type.

Attributes of `iql:Binding`:

Attribute	Type	Required	Default
distinct	Boolean	no	false
edges	Boolean	no	false
target	string	yes	

`iql:distinct` Enforces that the bound member references in this binding do **not** match the same target items during evaluation. Depending on the structural constraints used in the query, this setting might be redundant (e.g. when using the member references as identifiers for tree nodes who already are structurally distinct), but can still be used to make that fact explicit.

`iql:edges` Signals that the member labels are to be used for edges within a structure.

`iql:target` The name or alias of the layer to whose content the member variables should be bound.

Nested Elements of `iql:Binding`:

Element	Type	Required
members	array of `iql:Reference` (2.15)	yes

`iql:members` Non-empty collection of member references that are bound to the target layer's content. Every such instance of `iql:Reference` (2.15) must be unique within the surrounding `iql:Payload` (2.11).

2.2 Constraint

Constraints represent the actual content filtering of every query.

Attributes of `<Constraint>`:

Attribute	Type	Required	Default
id	string	yes	
solved	Boolean	no	false
solvedAs	Boolean	no	false

iql:id Identifier to uniquely identify the constraint within the entire query.

iql:solved Hint for the evaluation engine that this constraint has already been solved, either by a back-end implementation or as a result of (partial) query evaluation by the engine itself.

iql:solvedAs Specifies to what Boolean value (`true` or `false`) the constraint has been evaluated.

2.2.1 Predicate

Wraps a Boolean `iql:Expression` into an atomic constraint element that represents the smallest unit of evaluation for the top-level evaluation engine.
Extends <Constraint>(2.2).

Nested Elements of iql:Predicate:

Element	Type	Required
expression	`iql:Expression` (2.6)	yes

iql:expression The actual expression to be evaluated to a Boolean result. Note that typically this expression **cannot** be composed of directly nested Boolean conjunctions or disjunctions, as the engine will have parsed those into `iql:Term` (2.2.2) objects already during the first processing phase.

2.2.2 Term

A collection of constraints with a logical connective.
Extends <Constraint>(2.2).

Attributes of iql:Term:

Attribute	Type	Required	Default
operation	enum	yes	

iql:operation The Boolean connective to be applied to all the constraint items. Legal values are the strings “conjunction” or “disjunction”.

Nested Elements of iql:Term:

Element	Type	Required
items	array of `iql:Constraint` (2.2)	yes

iql:items The constraints which are to be combined by the specified `iql:operation`.

2.3 Corpus

Top-level entry point for querying a single stream.
Attributes of iql:Corpus:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	

iql:id Identifier to uniquely identify the corpus within the entire query.

iql:name The identifier used by the query engine’s manifest registry for the corpus.

2.4 Data

Allows to embed binary data in the query and make it usable from within constraint expressions via a designated reference.

Attributes of `iql:Data`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
content	string	yes	
codec	string	no	hex
checksum	string	no	
checksumType	enum	no	

`iql:id` Identifier to uniquely identify the corpus within the entire query.

`iql:name` The identifier used for the expression (3.6) which can be used to reference the binary payload from within query constraints.

`iql:content` The actual content of the payload in textual form. How to properly convert the textual form to a binary stream is defined by the `iql:codec` attribute.

`iql:codec` Specifies the mechanism of converting the `iql:content` data into an actual binary stream. If left empty, defaults to `hex`.

`iql:checksum` Optional hex-string of the checksum to check the `iql:content` against.

`iql:checksumType` Defines the algorithm for computing the checksum. Currently only `MD5` is supported as legal value.

2.5 Element

Abstract base type for all logical and/or structural units that can be matched against content of a target corpus.

Attributes of `<Element>`:

Attribute	Type	Required	Default
id	string	yes	
consumed	Boolean	no	false

`iql:id` Identifier to uniquely identify the element within the entire query.

`iql:consumed` Signals that the element has already been *used up* in the context of a partial query evaluation. An element that has been consumed can safely be ignored in the further evaluation of the query. Note that this state can be propagated according to the following rules:

- A `iql:Node`(2.5.2) can be marked as `consumed` if its `iql:constraint` is marked as `solved` and its match count satisfies the `iql:quantifiers` requirement. Note that cross-referencing constraints can only be considered solved when all other aspects of the involved elements support the `consumed` state.

- A iql:TreeNode(2.5.3) can be marked as consumed if above conditions are met and all nested iql:children are marked consumed.
- An iql:Edge(2.5.4) is considered consumed when both its terminals are consumed and the same conditions regarding its iql:constraint are fulfilled as mentioned above.
- A iql:ElementDisjunction(2.5.5) is considered consumed if at least one of its iql:alternatives has been marked consumed.

2.5.1 Node Set

Wrapper around a list of iql:Element (2.5) instances to group them for either nesting or disjunction.

Extends iql:Node(2.5.2).

Attributes of iql:NodeSet:

Attribute	Type	Required	Default
nodeArrangement	enum	no	unspecified

iql:nodeArrangement Defines what kind of order or arrangement should be assumed between the elements in this set. Legal values are unspecified, ordered (matched elements must occur in exactly the order specified in this set but need not form a continuous span) or adjacent (matched elements must form a continuous span).

Nested Elements of iql:NodeSet:

Element	Type	Required
nodes	array of <Element> (2.5)	no

iql:nodes List of nested <Element> instances. Legal types depend on the context in which this set is being used.

2.5.2 Node

Logical unit for sequence or graph matching in a target corpus. May contain local constraints and can also be quantified.

Extends <Element>(2.5).

Attributes of iql:Node:

Attribute	Type	Required	Default
label	string	no	

iql:label Identifier to bind the node through a previously defined iql:Binding (2.1) declaration.

Nested Elements of iql:Node:

Element	Type	Required
constraint	<Constraint> (2.2)	no
quantifiers	array of iql:Quantifier (2.13)	no

iql:constraint Optional local constraint to be matched against the content of potential target candidates during query evaluation.

iql:quantifiers Optional quantifiers to define the multiplicity of matches of this node required for a positive evaluation. Multiple quantifiers behave disjunctively. Note that IQL defines some restrictions on the legal combinations of quantifiers: The all-quantifier (* or all) and not-quantifier (! or not) can only be used in isolation, all other quantifiers can be combined in disjunctive fashion.

2.5.3 Tree Node

Extension of the simple iql:Node type (2.5.2) to add implicit hierarchical constraints related to dominance within tree structures.
Extends iql:Node(2.5.2).

Nested Elements of iql:TreeNode:

Element	Type	Required
children	instance of <Element> (2.5)	no

iql:children Optional nested nodes or node alternatives.

2.5.4 Edge

Specialized element extension to query structural information in graphs.
Extends <Element>(2.5).

Attributes of iql:Edge:

Attribute	Type	Required	Default
label	string	no	
edgeType	enum	yes	

iql:label Identifier to bind the edge through a previously defined iql:Binding (2.1) declaration.

iql:edgeType The type specification for this edge, primarily a directionality information. Legal values are simple, one-way or two-way.

Nested Elements of iql:Edge:

Element	Type	Required
constraint	<Constraint> (2.2)	no
source	iql:Node (2.5.2)	yes
target	iql:Node (2.5.2)	yes

iql:constraint Optional local constraint to be matched against the content of potential target candidates during query evaluation.

iql:source Source node declaration.

iql:target Target node declaration.

For complex graph declarations multiple nodes can be defined having the same iql:label. The evaluation engine will treat them as being the same node. Note however, that at most **one** node per label is allowed to declare a local iql:constraint attribute!

2.5.5 Element Disjunction

Allows declaration of multiple alternative element definitions. When evaluating the query, each such alternative that is matched successfully will cause this element declaration to evaluate positively.

Extends `<Element>`(2.5).

Nested Elements of `iql:ElementDisjunction`:

Element	Type	Required
alternatives	array of `iql:Element` (2.5)	yes

`iql:alternatives` The alternative element declarations, each of which constitutes a legal match for this element declaration. Must not contain less than 2 elements!

2.6 Expression

Wraps the textual form of an arbitrarily complex IQL expression, which can be a formula, literal, method invocation, a combination of those or a great many other types of expressions. For more details see Section 3.6.

Attributes of `iql:Expression`:

Attribute	Type	Required	Default
content	string	yes	
resultType	string	no	

`iql:content` The textual form of the expression. Must be valid according to the specifications in Section 3.6.

`iql:resultType` An optional specification regarding the return type of the expression. Redundant when the expression is used as a constraint, as those are required to always evaluate to a Boolean result value anyway.

2.7 Group

Provides a mechanism to collect successful matches into dedicated groups, either for result visualization or use in further result processing.

Attributes of `iql:Group`:

Attribute	Type	Required	Default
id	string	yes	
label	string	yes	

`iql:id` Identifier to uniquely identify the group declaration within the entire query.

`iql:label` Label (ideally human readable) to be used for referencing this group in subsequent result processing or for generating textual result reports.

Nested Elements of `iql:Group`:

Element	Type	Required
groupBy	`iql:Expression` (2.6)	yes
filterOn	`iql:Expression` (2.6)	no
defaultValue	`iql:Expression` (2.6)	no

iql:groupBy The mandatory expression used to extract the value from matches based on which the actual grouping occurs.

iql:filterOn Optional mechanism to exclude certain matches from being used for grouping.

iql:defaultValue If matches cannot produce a valid value for grouping but should still be included in the process, this optional field provides the means of declaring a kind of “fall back” group. Be aware of potential overlap in groups when using default values that are not distinct from the regular grouping results.

2.8 Import

To allow for flexible integration of macro definitions or bigger language extensions, IQL provides an optional section in the query that lets users specify exactly what additional modules besides the bare IQL core are required for evaluating the query. Each import target is specified by providing it’s unique name and telling the engine whether or not the import is to be considered optional.

Attributes of iql:Import:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
optional	Boolean	no	false

iql:id Identifier to uniquely identify the import within the entire query.

iql:name The original name of the extension to be added.

iql:optional Defines whether or not the referenced extension is optional. Non-optional imports that cannot be resolved to an actual extension during the query evaluation phase will cause the entire process to fail.

2.9 Lane

Lanes serve as a means of splitting queries for a single corpus stream into multiple logical subqueries that target different structural and/or logical layers, e.g. multiple syntactic analyses for the same source text.

Attributes of iql:Lane:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
alias	string	no	
laneType	enum	yes	

iql:id Identifier to uniquely identify the lane within the entire query.

iql:name The unique identifier of the item layer or structure layer that serves as target for this lane.

iql:alias If items of this lane in their entirety are meant to be used as part of query expressions inside this field holds the label used for the respective member variable. It is recommended to keep the chosen `alias` close to the original `name` to avoid confusion.

iql:laneType The type of structure this lane is meant to match, effectively defining the basic complexity class for evaluation. legal values are `sequence`, `tree` and `graph`. Note that the initial evaluation engine for IQL does not support the `graph` type!

Nested Elements of iql:Lane:

Element	Type	Required
elements	`<Element>` (2.5)	yes

iql:elements The structural constraints to be used for evaluation of this lane.

2.10 Layer

Every layer selector either references an entire subgraph of the corpus' member-graph directly or constructs a partial selection as part of a `iql:Scope` (2.18). When using the first approach, an item layer is referenced and all its dependencies and associated annotation layers will be made available implicitly. This is an easy way of accessing simple corpora, but can lead to costly I/O overhead when loading vast parts of a complex corpus that aren't actually needed to evaluate the query. For a more fine-grained alternative, scopes allow to create a scope that spans an exactly specified collection of layers. If multiple layer selectors are defined, up to one can be declared as "primary" to represent the granularity of returned items for the search or scope. In case no layer is explicitly marked as "primary", the one specified by the corpus or context will be used for that role by default.

Attributes of iql:Layer:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
alias	string	no	
primary	Boolean	no	false
allMembers	Boolean	no	false

iql:id Identifier to uniquely identify the layer within the entire query.

iql:name Identifier used to reference the layer within its host corpus.

iql:alias Optional identifier to rename the layer for referencing within the query.

iql:primary Signals that the layer is intended to act as the primary layer in the query or scope and as such defines the level of granularity for obtaining chunks in the corpus.

iql:allMembers When this layer definition is used inside a `iql:Scope` (2.18), effectively adds the entire member-subgraph of this layer to the scope. This property is redundant when the layer is part of the regular `iql:layers` declaration in a `iql:Stream` (2.20), as in that case all member subgraphs for each layer are already being added to the global scope!.

2.11 Payload

Every payload encapsulates all the (processed) query constraints to be evaluated against a single stream of corpus data.

Attributes of `iql:Payload`:

Attribute	Type	Required	Default
id	string	yes	
name	string	no	
queryType	enum	yes	
queryModifier	enum	no	

`iql:id` Automatically generated identifier to uniquely identify the payload within the entire query.

`iql:name` Custom identifier to uniquely identify the payload within the entire query. This attribute is deprecated but currently being kept to shift its use case.

`iql:queryType` The overall type of query strategy to be applied for this query payload. Legal values are `all` (returns the entire corpus and disallows any kind of constraint, leaving only the `iql:Result` (2.16) declaration as option to modify the result volume), `plain` (disabling any kind of structural constraints/lanes), `singleLane` and `multiLane`. The last two values dictate the minimal/maximal number of `iql:Lane` definitions in this payload.

`iql:queryModifier` Allows to limit the number of times an individual units-of-interest (UoIs) will be returned in the result. Supported values are `first`, `last` and `any`. The specific semantics of this modifier are described in more details in Section 3.7.

Nested Elements of `iql:Payload`:

Element	Type	Required
bindings	array of `iql:Binding` (2.1)	no
lanes	array of `iql:Lane` (2.9)	no
filter	`iql:Constraint` (2.2)	no
constraint	`iql:Constraint` (2.2)	no

`iql:bindings` Optional collection of bindings used within this payload. Note that member variables inside constraints or structural query elements will not resolve unless previously bound to corpus members.

`iql:lanes` If `iql:queryType` is set to `singleLane` or `multiLane`, this array is expected to hold either exactly 1 or at least 2 `iql:Lane` declarations that define structural constraint for the evaluation.

`iql:filter` If `iql:queryType` is set to anything other than `plain`, this constraint expression allows to filter contextual UoIs prior to the actual structural matching.

`iql:constraint` If `iql:queryType` is set to `plain`, this is expected to contain the basic constraints for matching candidates. In any version involving `iql:Lane` declarations, global constraints can be defined here as a means of implementing complex query features that are tested once the lanes have produced preliminary result candidates.

2.12 Property

Allows customization of the evaluation process by changing parameters or switching certain features on/off.

Attributes of `iql:Property`:

Attribute	Type	Required	Default
key	string	yes	
value	string	no	

`iql:key` The identifier of the targeted parameter or switch. The evaluation engine might report unknown keys as errors.

`iql:value` The actual value to apply to the specified property in case it is not a switch.

2.12.1 Switches

For increased flexibility, IQL supports a collection of switches to turn certain optional features on or off when needed. Switches are static and cannot be changed for the active query evaluation once set. All the native IQL switches use the prefix `iql:` for their name. Any extensions that offer additional switches should declare and use their own namespace for those switches! Currently supported switches are shown in Table 1.

2.13 Quantifier

Specifies the multiplicity of an associated `<Element>` (2.5).

Attributes of `iql:Quantifier`:

Attribute	Type	Required	Default
quantifierType	enum	yes	
value	integer	no	
lowerBound	integer	no	
upperBound	integer	no	

`iql:quantifierType` Defines how to interpret the other attributes. Legal values are `all` (universal quantification), `exact`, `atMost` (0..n), `atLeast` (n+), `range` (n..m).

`iql:value` Target or limit value when `iql:quantifierType` is set to `exact`, `atMost` or `atLeast`.

`iql:lowerBound` Used for `range` quantification to define the minimum multiplicity.

`iql:upperBound` Used for `range` quantification to define the maximum multiplicity.

2.14 Query

Encapsulates all the global configuration and extension of the query engine, as well as shared embedded data. Each query contains at least one `iql:Stream` declaration that in turn holds the actual query payload with constraints for the matching process.

Attributes of `iql:Query`:

Name	Description
iql.string.case.off	Turns of case sensitivity when performing string operations such as equality checks.
iql.string.case.lower	Another approach to case insensitivity, this switch turns all strings into lower case.
iql.expansion.off	Effectively shuts down value expansion Section 3.6.13.
iql.string2bool.off	Deactivates the interpretation of strings as Boolean values as described in Section 3.7.
iql.int2bool.off	Deactivates the interpretation of integers as Boolean values as described in Section 3.7.
iql.float2bool.off	Deactivates the interpretation of floating point numbers as Boolean values as described in Section 3.7.
iql.obj2bool.off	Deactivates the interpretation of arbitrary objects as Boolean values as described in Section 3.7.
iql.any2bool.off	Deactivates the interpretation of anything non-Boolean as Boolean value. This is a combination of "iql.string2bool.off", "iql.int2bool.off", "iql.float2bool.off" and "iql.obj2bool.off".
iql.direction.reverse	Reverses the direction used to traverse corpus data for a search.
iql.array.zero	Change array access (3.6.4) to be 0-based.
iql.markers.position.relative	Allow position markers (4.1) to use relative (percentage) arguments.
iql.warnings.off	Deactivates all warnings, potentially resulting in confusing results if there are mistakes in the query.
iql.parall.off	Forces the query evaluation engine to run single-threaded. This does however only affect the actual matcher, not additional. modules such as monitoring or item caches

Table 1: Currently supported switches in IQL and their explanations.

Attribute	Type	Required	Default
id	string	yes	
dialect	string	no	1.0

iql:id Identifier for the query, chosen by the client. In more complex (asynchronous) query workflows this id is used to map answers and results to the correct query.

iql:dialect Specifies which basic version of IQL to use. The initial version of IQL is "1.0" and by leaving the dialect part of a query blank the engine will default to this initial version.

Nested Elements of iql:Query:

Element	Type	Required
imports	array of `iql:Import` (2.8)	no
setup	array of `iql:Property` (2.12)	no
embeddeData	array of `iql:Data` (2.4)	no
streams	array of `iql:Stream` (2.20)	yes

`iql:imports` Defines extensions to be applied to the evaluation engine prior to actual query evaluation.

`iql:setup` Allows to configure the core evaluation engine or already defined extensions in a simple manner.

`iql:embeddeData` Binary data to be used in the evaluation process, such as audio or video fragments.

`iql:streams` Corpus data streams to be queried. In the initial version, the engine only supports single-stream querying!

2.15 Reference

Models references usable from within query expressions for accessing corpus members or variables.

Attributes of `iql:Reference`:

Attribute	Type	Required	Default
id	string	yes	
name	string	yes	
referenceType	enum	yes	

`iql:id` Identifier to uniquely identify the reference within the entire query.

`iql:name` The local identifier to be used for addressing this reference. Note that this is the bare name without any type-specific prefixes (such as '$' for members, cf. Section 3.5).

`iql:referenceType` Specifies the nature of this reference. Legal values are `reference`, `member` or `variable`.

2.16 Result

Encapsulates all the information on result processing and preparation.

Attributes of `iql:Result`:

Attribute	Type	Required	Default
limit	integer	no	
percent	Boolean	no	false

`iql:limit` Optional limitation on the total size of the result to be returned. If the `iql:percent` flag is not set to `true`, this number is in reference to the units provided by the query's primary item layer.

iql:percent If set to true the value defined in iql:limit is treated as a integer percentage value in the interval 1 to 99, with boundaries included.

Nested Elements of iql:Result:

Element	Type	Required
resultTypes	array of enum	yes
resultInstructions	array of iql:ResultInstruction (2.17)	no
sortings	array of iql:Sorting (2.19)	no

iql:resultTypes Defines the result format or type the engine should return data in. At least one result type must be declared and the engine can also be instructed to return the results in multiple formats simultaneously. In the first iteration only kwic (keyword-in-context) and custom (as a placeholder for the raw corpus members) are supported.

iql:resultInstructions Optional collection of additional processing instructions to generate (textual) result reports.

iql:sortings Allows to sort matches before generating result reports.

2.17 Result Instruction

Currently unused dummy for declaring post-processing instructions on the query result to perform conversions and/or tabular calculations.

2.18 Scope

Very detailed vertical filtering of the layers available in a query.

Attributes of iql:Scope:

Attribute	Type	Required	Default
id	string	yes	

iql:id Identifier to uniquely identify the scope within the entire query.

Nested Elements of iql:Scope:

Element	Type	Required
layers	array of iql:Layer (2.10)	yes

iql:layers The layer members of this scope.

2.19 Sorting

Defines a single rule for sorting query results based on an arbitrarily complex expression.

Attributes of iql:Sorting:

Attribute	Type	Required	Default
order	enum	yes	

iql:order Hint on sorting direction, legal values are `asc` or `desc` for ascending or descending order, respectively.

Nested Elements of iql:Sorting:

Element	Type	Required
expression	`iql:Expression` (2.6)	yes

iql:expression The actual sorting expression. It can use any (member) reference or variable available in the query to compute its result and must return a type that is comparable to allow stable sorting. Per default any of the primitive numerical types (`int` or `float`), the text type `string` and any member of the ICARUS2 framework implementing the `java.lang.Comparable` interface can be used as return type.

2.20 Stream

A stream encapsulates all the information and query constraints to extract, evaluate and prepare data from a single corpus. Note that many of the attributes and/or elements below are marked as optional, but the following restrictions are in effect:

- Either `iql:rawPayload` or `iql:payload` must be provided by the client.
- Either `iql:layers` or `iql:scope` must be provided to define the granularity of data being loaded for evaluation.

Attributes of iql:Stream:

Attribute	Type	Required	Default
id	string	yes	
primary	Boolean	no	false
rawPayload	string	no	
rawGrouping	string	no	
rawResult	string	no	

iql:id Identifier to uniquely identify the stream within the entire query.

iql:primary Flag to indicate that the primary layer of this stream is meant to be used as primary layer of the entire search result. Only one stream can declare this property and it primarily dictates the order of result elements in a multi-stream query or which stream is allowed to dictate sorting.

iql:rawPayload The textual (raw) form of the payload for this stream, i.e. all the constraints and structural query content.

iql:rawGrouping The textual (raw) grouping definitions to be applied for results of this stream.

iql:rawResult The textual (raw) result configuration and post-processing instructions for this stream.

Nested Elements of `iql:Stream`:

Element	Type	Required
corpus	`iql:Corpus` (2.3)	yes
layers	array of `iql:Layer` (2.10)	no
scope	`iql:Scope` (2.18)	no
payload	`iql:Payload` (2.11)	no
grouping	array of `iql:Grouping` (2.7)	no
result	`iql:Result` (2.16)	yes

`iql:corpus` The corpus to extract data from.

`iql:layers` Vertical filtering to be applied to the corpus prior to actual query evaluation.

`iql:scope` Another and more fine-grained form of vertical filtering that allows for more precise selection of layers to be part of this stream's data.

`iql:payload` The processed form of `iql:rawPayload`.

`iql:grouping` The processed form of `iql:rawGrouping`.

`iql:result` The processed form of `iql:rawResult`.

3 Inner IQL ELements

Certain parts of an IQL query can be defined in *raw* form, that is, in a keyword-driven formal language. During the first phase of query evaluation they get (partly) translated into their respective JSON-LD counterparts described in Section 2 (unless of course the query or query fragments are provided fully processed). This section defines the syntax and additional rules for those raw statements. Note that the textual form of all following IQL elements is expected to be encoded in UTF-8, so no special escape mechanisms are needed for unicode content.

3.1 Reserved Words

The following list of keywords is reserved and any of the words may not be used as direct identifier strings in a query. They are reserved in both all lowercase and all uppercase variants, and while camel-cased versions are technically permitted, it is highly discouraged to use them:

`ADJACENT`
`ALL`
`AND`
`ANY`
`AS`
`ASC`
`BY`
`COUNT`
`DEFAULT`
`DESC`
`DISTINCT`
`DO`
`EDGES`
`END`
`EVEN`
`FALSE`
`FILTER`
`FIND`
`FIRST`
`FOREACH`
`FROM`
`GROUP`
`HAVING`
`IN`
`LABEL`
`LANE`
`LAST`
`LIMIT`
`NOT`
`NULL`
`ODD`
`OMIT`
`ON`
`OR`
`ORDER`
`ORDERED`
`RANGE`
`STEP`
`TRUE`
`WITH`

In addition the following strictly lowercase words are reserved as type identifiers and may not be used otherwise:

`boolean` `int` `float` `string`

3.2 Comments

IQL supports single-line comments, indicated by “//”. All remaining content in a line after the comment indicator will be ignored when parsing and evaluating a query.

3.3 Literals

Literals are statically-typed fixed-value expressions in IQL. They are parsed only once during the initial processing part of a query.

3.3.1 String Literals

IQL uses simple double quotes (‘`"`’ or U+0022) to define string literals. String literals may not contain any of the following symbols directly:

```
\n line break
\r carriage return
\t tab
\ backslash
" nested quotation mark
```

Any of those symbols listed above can be embedded into a string literal as part of an escape sequence with a preceding backslash. At the current time there is no planned mechanism to provide additional escape support for unicode symbols, since the default encoding scheme for IQL is UTF-8.

Examples for valid string literals:

```
"string"
"123"
"some fancy number (123.456e-789) and special symbol ♣"
"a more complex string!"
"a\n multiline\n string..."
```

3.3.2 Boolean Literals

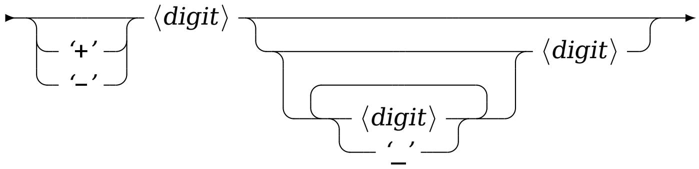
Boolean literals are limited to either all lowercase or all uppercase versions of the literals `true` and `false`.

3.3.3 Integer Literals

Signed Integer Literals Literals representing regular `int` (32bit) or `long` (64bit) integers consist of an optional initial sign (‘`+`’ or ‘`-`’) and the body consisting of digits (‘`0`’ to ‘`9`’) or underscore (‘`_`’) characters. Underscore characters may only appear inside the integer literal, never at the beginning or end (not counting the sign symbol).

Grammar Snippet 1

⟨*integerLiteral*⟩*:*



Examples for valid (signed) integer literals:

```
1
+123
-123
1_000_000
-99_000000_0
```

Pure Integer Literals Some parts of the IQL syntax only allow unsigned "pure" integers and will explicitly state this fact. In those special cases integer literals may neither contain the initial sign symbol nor intermediate underscores.

3.3.4 Floating Point Literals

Floating point literals are constructed by using a (signed) integer literal for the pre-decimal part, a dot '`.`' as delimiter and a decimal part made up by a unsigned integer literal. They represent either single-precision `float` (32bit) or double-precision `double` (64bit) values.

Grammar Snippet 2

⟨*floatingPointLiteral*⟩*:*

►— ⟨*signedInteger*⟩ – '`.`' – ⟨*unsignedInteger*⟩ —►

Examples for valid (signed) floating point literals:

```
1.0
+123.456
-123.456
1_000_000.999
-99_000000_0.000_000_001
```

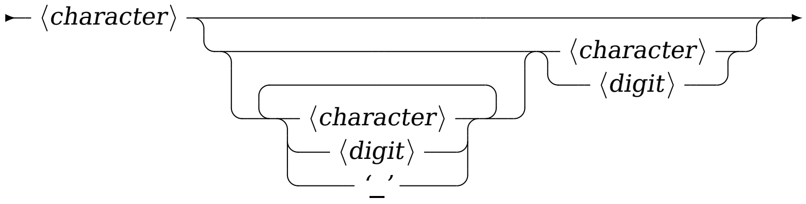
While many languages offer to express floating point literals in the scientific notation with explicit exponent declaration, we do not include this in the initial draft of IQL.

3.4 Identifiers

Identifiers in IQL are combinations of lowercase or uppercase alphabetic `[a-zA-Z]` characters that may contain underscore symbols _ between the first and last position and may also contain digits `[0-9]` on any position except as initial symbol (cf. snippet 3).

Grammar Snippet 3

⟨identifier⟩:



Examples for valid identifiers:

```
x
myIdentifier
x1
x_1
x__1
x321
some_random_id
someRandomId002
random_2_4
notTheBest______________example
```

Identifiers are limited in length by the engine to a total of 255 characters. This is a purely arbitrary choice to keep queries readable and not subject to any technical limitations.

3.5 Variables and References

In IQL all top-level (i.e. not part of the tail expression in a hierarchical path) identifiers are expected to reference 'something' from the global namespace available to the query. This namespace is populated with all the globally available constants, methods and helper objects from the IQL core and any imported extensions, as well as all the corpus members defined in the scoping part of the query. Outside this global namespace any dynamically created identifiers from within a query reside in the variable namespace and are marked with a preceding '`@`' (e.g. `@myVariable`). They can be used the same way as any regular identifier, with the exception of additionally allowing assignment expressions when inside script blocks. In addition any corpus members bound within a constraint section are prefixed with a '`$`' sign, such as `$token1`. Table 2 provides a compact overview of the available identifiers and their capabilities/features.

Type	Prefix	Example	Scope	Fixed[1]	Final	Re-Assign
Reference		`max()`	global	X	X	
Variable	`@`	`@myVar`	limited	(X)		X
Member	`$`	`$token`	limited	X	(X)	

Table 2: Identifier types available in IQL and their properties.

Special remarks: Variables are more or less general-purpose storage objects for arbitrary values and without a fixed type. Their first assignment however hints at the implied type to be used and as such they can cause cast errors when used for situations where an incompatible type would be needed.

Member identifiers are final in the sense that they cannot be re-assigned explicitly but will be implicitly for every iteration of the query on a new part of the corpus. For example, above `$token` member will point to a new token object every time the inner constraint parts of the query are evaluated. Therefore member identifiers could be viewed as a sort of loop variable.

3.6 Expressions

Expressions are the foundation of every query. Each expression has a (usually fixed) result type and evaluates to a value of that type. They can take any of the following forms and a overview diagram is available in the appendix as snippet 21.

3.6.1 Primary Expressions

Any literal of types `boolean`, `string`, `int` or `float` can serve as a primary expression of that type. See Section 3.3 for examples and a more detailed specification of the various types of literals in IQL.

References Any reference or variable as described in Section 3.5 is also a valid primary expression. Note that the process for resolving references strongly relies on the context the reference is being used in, for instance to allow simple references to be used as aliases for method calls.

Additionally a special construct is available as primary expression to more easily access annotation values from within a node definitions (3.8.7). Normally the evaluation engine tries to resolve any otherwise unknown identifier inside a node as the key (aliased or original) of an annotation that should be fetched for the item matched to this node. But since this is not always possible in an unambiguous way or there exists a name clash between an annotation key and another identifier available in the current environment, a ⟨*qualifiedIdentifier*⟩ (snippet 4) can be used instead.

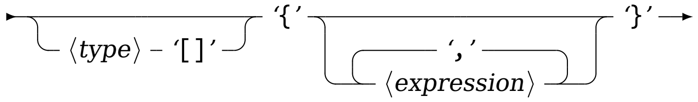
Grammar Snippet 4 (⟨*qualifiedIdentifier*⟩)
⟨*qualifiedIdentifier*⟩ ::= ⟨*identifier*⟩ '`::`' ⟨*identifier*⟩

A qualified identifier consists of two identifiers that are joined by a double colon '`::`'. The first (or left) identifier is expected to unambiguously resolve to a annotation layer available for the current context. The second (or right) identifier in turn acts as specification of an annotation key (or its alias) available from the previously defined annotation layer.

List Expression Arrays (or more generally "lists") in IQL can be defined by enclosing a sequence of expressions within curly brackets ('`{`' and '`}`'), using the comma symbol ('`,`') as separator. Optionally the desired type for elements in the list can be made explicit with a special type marker in front of the opening bracket as illustrated by snippet 5. If no type is provided, the overall element type will be determined by checking the types of all elements and choosing the least restrictive one. Note that for empty lists (no elements are given inside the curly brackets) it is mandatory to specify the element type!

Grammar Snippet 5 (⟨*listExpression*⟩)
⟨listExpression⟩ ::= (⟨type⟩ ‘`[]`*’)? ‘*`{`*’ (⟨expression⟩ (‘*`,`*’ ⟨expression⟩)*)? ‘*`}`*’*

⟨listExpression⟩:



3.6.2 Path Expressions

For navigating hierarchically structured object graphs or namespaces, expressions can take the form of paths, consisting of a original expression, a dot as separator and finally an identifier that denotes the path element or “field” within the context of whatever the original expression returned.

Grammar Snippet 6 (⟨*path*⟩)
⟨path⟩ ::= ⟨expression⟩ ‘`.`*’ ⟨identifier⟩*

Examples:

```
someObejct.someProperty
some.really.long.winded.path
```

Note that for a lot of native classes of the ICARUS2 framework, IQL provides convenient path-based alternatives to method invocations. For example in the context of navigating a structure, “someStructure.getParent(someItem)“ can be replaced by “someItem.parent“ as long as “someStructure“ is unambiguous in the current context and already bound.

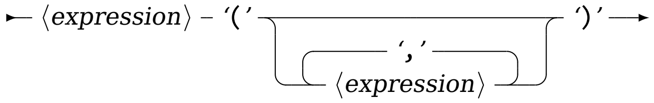
3.6.3 Method Invocation

Method invocations consist of an expression that points to the actual method (such as an identifier in the global namespace or a path expression) and round brackets for the invocation with an optional argument list:

Grammar Snippet 7 (⟨*method*⟩)
⟨method⟩ ::= ⟨expression⟩ ‘`(`*’ ⟨arguments⟩? ‘*`)`*’*

⟨arguments⟩ ::= ⟨expression⟩ (‘`,`*’ ⟨expression⟩)**

⟨method⟩:



Examples:

```
myFunction()
myNamespace.someFunction(someArgument, anotherArgument)
min(123, 456, dynamicContent())
some().chained().methods()
```

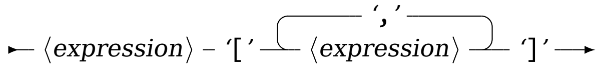
3.6.4 List Access

Lists or arrays are accessed by an expression pointing to the list or array object itself and an index expression in square brackets indicating the position(s) of the desired element(s) within the array. Note that the index or indices expression must evaluate to values within 32bit signed integer space. Positive values indicate the position beginning from the start of the 0-based array, whereas negative values allow backwards referencing of elements with ‘`-1`’ pointing to the last array element and ‘`-2`’ to the second to last one. For multi-dimensional arrays several index statements can be chained or even combined in a single comma-separated list.

Grammar Snippet 8 (*⟨array⟩*)
⟨array⟩ ::= ⟨expression⟩ ‘`[`*’ ⟨indices⟩ ‘*`]`*’*

⟨indices⟩ ::= ⟨expression⟩ (‘`,`*’ ⟨expression⟩)**

⟨method⟩:



Examples:

```
myArray[1]
myArray[-1]
myArray[-myArray.length] // same as myArray[0]
complexArray[1][2][3]
complexArray[-1][2][-3]
complexArray[1, 2][3]
complexArray[1, 2, 3]
```

Note that IQL provides convenient ways of using array access patterns to access list-like data structures and/or classes of the framework: Every ItemLookup implementation, such as Container or Structure that would traditionally access its content via “myContainer.getItemAt(someIndex)“ can be used the same as any regular array with the expression “myContainer[someIndex]“.

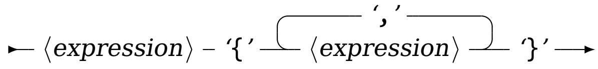
3.6.5 Annotation Access

The ICARUS2 Corpus Modeling Framework (ICMF) models segmentation, structure and content of a corpus resource as different aspects. As such the information about any annotation attached to a given Item is stored apart from it and therefore is not easily accessible from the item alone. To simplify the usage of annotations within a query, IQL provides the following expression as syntactic sugar for accessing (multiple) annotations directly from an item:

Grammar Snippet 9 (*⟨annotation⟩*)
⟨annotation⟩ ::= ⟨expression⟩ ‘`{`*’ ⟨keys⟩ ‘*`}`*’*

⟨keys⟩ ::= ⟨expression⟩ (‘`,`*’ ⟨expression⟩)**

⟨method⟩:



The first expression must evaluate to an item reference and the annotation pointers inside curly brackets must evaluate to strings (if only a single expression is given, it can evaluate to a list or array and be expanded, cf. Section 3.6.13) that uniquely denote annotation layers in the current context of the query. Typically users will use string literals in double quotes to explicitly state the annotations to be accessed, but the IQL syntax allows for very flexible extraction statement. If the evaluation of those annotation pointers yields more than one string, the result will be an array-like object containing the resolved values for each of the annotation keys in the same order as those were specified.

Examples:

```
myItem{"pos"}
myItem{"form", "pos", "lemma"}

// extract values from multiple concurrent annotation layers
// and pick the first one present
firstSetValue(myItem{"parser1.head", "parser2.head"})
```

3.6.6 Type Cast

Expressions in IQL are automatically cast to matching types according to the actual consumer's needs (unless this feature gets deactivated via the corresponding switch, cf. Section 2.12.1). Explicit casts can be performed by preceding an expression with one of the type keywords listed above (3.1) in round brackets.

Examples:

```
(int) myValue
(int) 12345.678
(float) average(myVector)
(string) 123.456
```

3.6.7 Wrapping

Expression hierarchy and evaluation order follows the order the different types of expressions are listed here. To dictate another order, expressions can be wrapped into round brackets. This will cause the inner expression to be evaluated independent of potential hierarchical rules from the outside context.

Examples:

```
6 + 4 * 2   // multiplication evaluated first -> result 14
(6 + 4) * 2 // addition is evaluated first -> result 20
```

3.6.8 Unary Operation

IQL only allows four unary operators to be used directly in front of an expression, the exclamation mark '!' and the NOT keyword for Boolean negation, the minus sign '–' for negating numerical expressions and the '~' symbol of bitwise negation of integer numbers.

Examples:

```
!someBooleanFunction()
NOT someBooleanValue
-123
-myNumericalFunction()
~123
~myIntegerFunction()
```

3.6.9 Binary Operation

Binary operations between two expressions take the following simple form:

Grammar Snippet 10 (⟨*binary_op*⟩)
⟨*binary_op*⟩ *::=* ⟨*expression*⟩ ⟨*operator*⟩ ⟨*expression*⟩

Binary operators follow an explicit hierarchy, listed in Table 3 in the order of priority, from highest to lowest.

Operators	**Explanation**
`* / %`	multiplication, division and modulo
`+ -`	addition and subtraction
`<< >> & | ^`	shift left, shift right, bitwise and, bitwise or, bitwise xor
`< <= > >=`	less, less or equal, greater, greater or equal
`=~ !~ =# !#`	string operators: matches (regex), matches not (regex), contains, contains not
`== !=`	equals, equals not
`&& AND`	logical and
`|| OR`	logical or

Table 3: Binary operators available in IQL and their hierarchical order.

Basic Numerical Operations Basic numerical operations follow the standard mathematical rules for priorities. While the basic numerical types (int, float) can be arbitrarily mixed inside those expressions, the type used during the expression and as result will be determined by the least restrictive type of any operand involved.

Bit Operations Bitwise operations ('&', '|' and '^') take integer expressions (or any other form of *bitset*) as inputs and generate a result of the corresponding type. If different types are used (e.g. int and long), one must be cast 3.6.6 to match the other. If value expansion 3.6.13 is active, any array-like data can also be used and will be subject to element-wise bit operations.

The two shift operations (‘<<’ and ‘>>’) take arbitrary integer types as left operand and an `int` value as right operand.

Ordered Comparisons Comparisons are special binary operators that take two expressions of equal or compatible result type and produce a Boolean value. Note that their exact semantics are type specific, e.g. when comparing strings, the operation is performed lexicographically and may be subject to case conversions (2.12.1).

String Operations To account for the ubiquity of textual annotations in corpora, IQL provides a set of dedicated string operators to perform substring matching (with the *contains* operator ‘=#’ or its negated form ‘!#’) and regular expression matching (via ‘=~’ and ‘!~’). Per default IQL uses the Java regex syntax, but for the future, additional switches (2.12.1) are planned to allow finer control over regex details.

Examples:

```
// find verbal forms
somePosAnnotation # "V"
// alternative to the set predicate with more flexibility
somePosAnnotation !~ "NN|NS"
```

Equality Equality checks follow the same basic conditions as ordered comparisons (3.6.9), but with the following rules for comparable values “a“ and “b“:

```
a == b iff !(a<b) && !(a>b)
a != b iff a<b || a>b
```

More generally, equality between expressions in IQL is based on content equality and therefore type specific. Note that trying to check two expressions of incompatible types (such as `int` and `string`) for equality will always evaluate to `false` and also emit a warning.

Logical Composition All Boolean expressions can be combined via disjunction (either double pipes ‘||’ or the `OR` keyword) or conjunction (double ampersand ‘&&’ or the `AND` keyword), with conjunction having higher priority. While not strictly mandatory, evaluation of IQL expressions is recommended to employ optimized interpretation such that only the first operand is evaluated if possible. When the first operand of a disjunction evaluates to `true`, the entire expression is already determined, same for a conjunction’s first operand yielding `false`.

Examples:

```
a>1 && b<2
x==1 or x==3
```

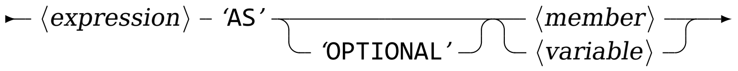
Note that aforementioned optimization strategy also applies to chained boolean expressions, such as `a && b && c` and that the evaluation engine is free to reorder operands here as it sees fit. As such individual operands may never be evaluated at all and the overall correctness of a query expression must not rely on side-effects stemming from specific expressions being evaluated or not.

3.6.10 Assignment

IQL offers a special construct to assign (member) variables and at the same time verify the successful assignment with a Boolean result value. Snippet 11 illustrates the basic syntax for assignment operations inside a constraint, allowing the result of an arbitrary expression to be assigned to any previously defined member variable (or dynamically allocated variable).

Grammar Snippet 11 (⟨*assignment*⟩)
⟨*assignment*⟩ ::= ⟨*expression*⟩ *'*`AS`*' '*`OPTIONAL`*'? (*⟨*member*⟩ *|* ⟨*variable*⟩*)*

⟨*assignment*⟩*:*



If a member variable is used for the assignment or the variable has already been used previously and thereby has been type-bound, the result type of the ⟨*expression*⟩ has to be compatible! Assignment operations provide a Boolean return value that is `true` iff the assignment was successful and the (member) variable holds a non-null value. If the assignment is declared to be `OPTIONAL`, it will always evaluate to `true` regardless of the final value the (member) variable is holding. As the name of that keyword implies, this allows to have parts of the query effectively becoming optional when it comes to capturing information. Keep in mind that optional (member) variables might not be assigned a valid non-null value when trying to read them in subsequent constraints or final result processing. As such care must be taken not to create errors when trying to access their content.

Examples:

```
// global constraint over two tree nodes to fetch their first
// ancestor, failing if $x and $y do not have a common ancestor
... HAVING ancestor($x, $y) AS $a

// optionally assign the last child of $x as a member variable,
// not failing if $x has no children
... HAVING lastChild($x) AS OPTIONAL $c
```

3.6.11 Set Predicate

Also called ’containment predicate’, this expression allows to check if a given value is a member of a specified set (or generally speaking ’collection’) as shown in snippet 12. The entire expression evaluates to a Boolean value and will be `true` iff the input expression (left-most one) evaluates to the same value as any of the elements inside the set definition to the right (typically a list expression, cf. Section 3.6.1 or snippet 5). See about equality operators in Section 3.6.9 on how elements are compared. Note that methods or collections used inside the set definition are subject to the expansion rules described in Section 3.6.13. The primary use case for set expressions is to greatly simplify the declaration of constraints for multiple alternative target values.

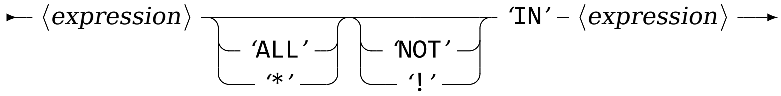
Set predicates can be directly negated (apart from wrapping 3.6.7 them and negating 3.6.8 the entire expression) with an exclamation mark '!' or the keyword NOT in front of the IN keyword. If the input expression evaluates to an array-like object, the set predicate will expand its content and evaluate to true if at least *one* of its elements is found to be contained in the set. The set predicate can be universally quantified with a star '*' or the ALL keyword in front to change the overall behavior such that the result will be true iff *all* of the elements are contained in the set (or none of them are, if the set predicate is directly negated).

Grammar Snippet 12 (⟨*setPredicate*⟩)
⟨*setPredicate*⟩ ::= ⟨*expression*⟩ ⟨*all*⟩? ⟨*not*⟩? 'IN' ⟨*expression*⟩

⟨*all*⟩ ::= 'ALL' | '*'

⟨*not*⟩ ::= 'NOT' | '!'

⟨*setPredicate*⟩:



Examples:

```
someAnnotationValue IN {"NP","VP","-"}
someAnnotationValue NOT IN {"NN","DET"}
someAnnotationValue NOT IN {"NN","DET", STTS.getVerbTags()}
"John" IN getLegalNames()
fetchCharacterNamesInChapterOne() IN getOrcishNames()
```

3.6.12 Ternary Operation

A single ternary operation is supported in IQL, which is the popular if-then-else replacement with the following syntax:

Grammar Snippet 13 (⟨*ternary*⟩)
⟨*ternary*⟩ ::= ⟨*expression*⟩ '?' ⟨*expression*⟩ ':' ⟨*expression*⟩

The first expression must evaluate to a boolean value and determines which of the following two alternatives will be evaluated for the final value of the expression. Note that the second and third expressions must have compatible result types.

Examples:

```
x<2 ? "text for smaller value" : "some other text"
```

3.6.13 Value Expansion

IQL supports expansion of arrays, lists and array-like method return values for situations where an immediate consumer supports lists of values as input. Assuming the method "randomPoint()" returns an array of 3 integer values or a *array-like* data type (such as

Type	**Condition**	**Value**
`string`	empty or null	`false`
`int`	`0`	`false`
`float`	`0.0`	`false`
any object	`null`	`false`

Table 4: Rules for converting arbitrary values or objects in a query to Boolean values.

a 3D point) and another method “invertPoint(int, int, int)” takes 3 integer arguments, then the invocation of “invertPoint(randomPoint())” is legal and the array or object from the inner method call will be automatically expanded into the separate 3 values. This is especially handy when dealing with multidimensional arrays, as regular indexing would require manual extraction of method return values into variables to then be used in accessing the different array dimensions. With automatic expansion, a three-dimensional array could directly be accessed with aforementioned method via “array[randomPoint()]”.

3.7 Constraints

Simply put, constraints are expressions that evaluate to a Boolean result. Apart from native Boolean expressions (such as comparisons, Boolean literals or Boolean functions), IQL allows certain evaluations as syntactic sugar, listed in Table 4. Note that those conversions are only active if the respective switches to disable them (2.12.1) have not been set.

Result Expansion Besides their obvious role as filters, constraints can also be used to expand the set of captured members in a match, subsequently usable for instance in advanced result processing.[2] The assignment operation (3.6.10) allows arbitrary expressions to be evaluated and having their results stored in either a general variable or, in case of a result type compatible to items, actual member variable (3.5).

3.8 Payload Structure

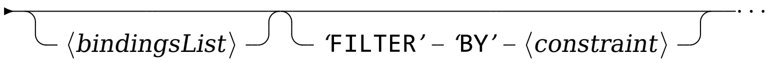
The Payload section in IQL consists of either the sole `ALL` keyword or a selection statement (3.8.4) with optional binding (3.8.3) definition and filter constraints (3.8.1) preceding it. If the `ALL` keyword is used, no constraints whatsoever can be defined and the engine is instructed to return the entire target corpus. In this case the only way of restricting results is by using the `iql:Result` section (2.16) of a query.

Grammar Snippet 14 (⟨*payload*⟩)

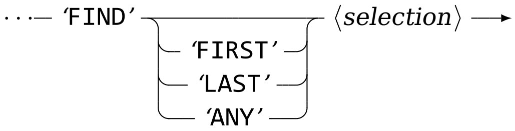
⟨*payload*⟩ ::= ‘`ALL`’
| ⟨*bindingsList*⟩? (‘`FILTER`’ ‘`BY`’ ⟨*constraint*⟩)? ‘`FIND`’ ⟨*modifier*⟩? ⟨*selection*⟩

⟨*modifier*⟩ ::= ‘`FIRST`’ | ‘`LAST`’ | ‘`ANY`’

⟨*payload with content*⟩:



[2] Per default members defined as bindings (3.8.3) are captured when used as labels for nodes (3.8.7) in the query.



3.8.1 Filter Constraints

For complex (i.e. structural) queries, IQL offers a way of filtering the UoIs before they are processed by the matchers for sequence, tree or graph structures (cf. Sections 3.8.8 to 3.8.10). A dedicated `FILTER BY` section in the query payload preceding the actual structural constraints is available to define filtering rules that have to evaluate to `true` for a UoI to be considered for actual matching. Constraints within a filtering rules have only access to general properties of the UoIs, such as sentence length, tree height or similar information. The do **not** have access to bound member variables, apart from those defined for the top-level members of lanes (3.8.5)! Note that filter constraints are **not** compatible with flat constraints (3.8.6) as they both essentially fill the same function and flat constraints take precedence.

3.8.2 Match Modifiers

Per default, the search in IQL is expected to be exhaustive, i.e. the evaluation engine will attempt to find all of the instances in a target corpus that match the query constraints, potentially reporting individual UoIs (such as sentences) multiple times if they contain several occurrences. For instance, the sentence "The dog chased the rabbit down the hill." will be treated as tree entries in the result if the query was only meant to find instances of the lemma "the". This default behavior can be adjusted to only return each UoI no more than once by using one of the modifiers (`FIRST`, `LAST`, `ANY`) listed in snippet 14. The semantics of the `ANY` modifier are such that the engine may freely pick any one match within a UoI. Note however, that to support reproducible search results, repeated evaluations of the same query on a corpus are still required to yield the same instances here. The exact semantics of `FIRST` and `LAST` are depending on the type of structural constraints used in the payload, but generally are based on the natural order of items within the corpus (typically this is the flow of words in a text). The evaluation behavior for them is subsequently covered in Sections 3.8.8 to 3.8.10.

3.8.3 Bindings

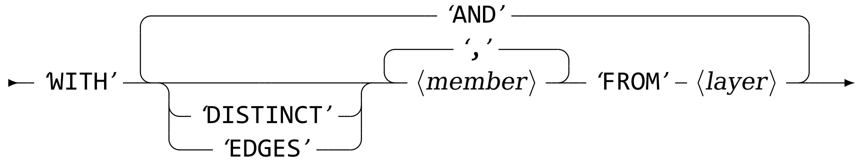
A binding is a collection of member references (3.5) that get declared to belong to a certain member type or part of the corpus. The `DISTINCT` keyword enforces that multiple bound member references in this binding do **not** match the same target. Depending on the local constraints used in the query, this might be redundant (e.g. when using the member references as identifiers for tree nodes who already are structurally distinct), but can still be used to make that fact explicit. Additionally the `EDGES` keyword signals that the bound members of a structure are actually edges. In this case using `DISTINCT` is redundant, as bound edges are implicitly assumed to be distinct when matching.

Grammar Snippet 15 (Bindings)
⟨*bindingsList*⟩ ::= 'WITH' ⟨*binding*⟩ *(*'AND' ⟨*binding*⟩*)**

⟨binding⟩ := *⟨option⟩? ⟨member⟩ (*‘,’ *⟨member⟩)** ‘FROM’ *⟨layer⟩*

⟨option⟩ := ‘DISTINCT’ | ‘EDGES’

⟨bindingsList⟩:



Raw binding definitions in the payload are parsed and stored in their JSON counterpart (`iql:Binding`, 2.1) during query processing.

3.8.4 Selection Statement

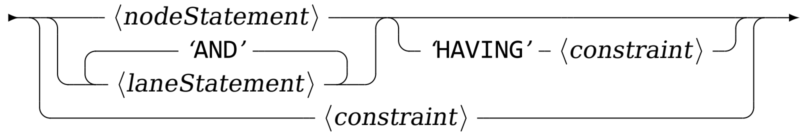
Constraints are further divided into local constraints as part of node or edge definitions and global ones (with the HAVING keyword). Local constraints are obligatory and define the basic complexity of the query (flat, tree or graph). They also introduce certain limitations on what can be expressed or searched (e.g. a “flat” local constraints declaration will not provide implicit access to tree information). However, global constraints can introduce arbitrary constraints or relations and thereby increase the evaluation complexity, potentially without limits. Since there is no way for an evaluation engine to assess the complexity of user macros or extensions, extensive use of global constraints could in fact lead to extremely slow searches or even create situations where an evaluation will never terminate at all.

Grammar Snippet 16 (*⟨selectionStatement⟩*)
⟨selectionStatement⟩ ::= *⟨constraint⟩*
| (*⟨nodeStatement⟩* | *⟨laneStatementsList⟩*) (‘HAVING’ *⟨constraint⟩*)?

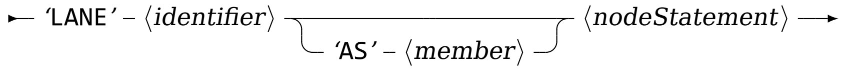
⟨laneStatementsList⟩ ::= *⟨laneStatement⟩* (‘AND’ *⟨laneStatement⟩*)*

⟨laneStatement⟩ ::= ‘LANE’ *⟨identifier⟩* (‘AS’ *⟨member⟩*)? *⟨nodeStatement⟩*

⟨selectionStatement⟩:



⟨laneStatement⟩:



3.8.5 Lanes

Lane statements can be used to extract information from concurrent structures that exist for the UoI of the payload. Each lane statement is introduced by the LANE keyword and an identifier that matches the name or alias of a layer in the outer query definition (cf.

snippet 16). Optionally the source layer of a lane can also be assigned a member variable (3.5) so that it can be explicitly referenced in the payload.[3] During query processing raw lane statements will be parsed into `iql:Lane` objects (2.9).

Constraints are further divided into local constraints as part of node or edge definitions and global ones (with the `HAVING` keyword). Local constraints are obligatory and define the basic complexity of the query (flat, tree or graph). They also introduce certain limitations on what can be expressed or searched (e.g. a "flat" local constraints declaration will not provide implicit access to tree information). However, global constraints can introduce arbitrary constraints or relations and thereby increase the evaluation complexity, potentially without limits. Since there is no way for an evaluation engine to assess the complexity of user macros or extensions, extensive use of global constraints could in fact lead to extremely slow searches or even create situations where an evaluation will never terminate at all.

3.8.6 Flat Constraints

Flat constraints provide no extra helpers to declare structural properties of the query. They consist of arbitrary basic constraints Section 3.7 and disallow both global constraints (3.8.11) and filter constraints (3.8.1). Note that flat constraints rely on the availability of member references from the binding or lane sections in the query payload to have access to any content at all. In this regard they behave very similar to global constraints.

3.8.7 Structural Constraints

IQL provides several classes of structural constraints that each feature distinctive syntax features to express structures of increasing complexity. Those structures are sequences (3.8.8), trees (3.8.9) and graphs (3.8.10). They all get explained in more detail in their respective sections, but the syntactic basics for all of them will be defined here. To simplify the overall IQL grammar, a general syntax exists for the declaration of nodes (and edges). This general form honors the aspects specific to each of those structure types, but generally over-generates and only some of its features are actually applicable in concrete use cases. Snippet 17 shows the basic for defining structural constraints in IQL. More detailed illustrations of the various components (⟨*node*⟩, ⟨*quantifier*⟩ and ⟨*edge*⟩) follow below.

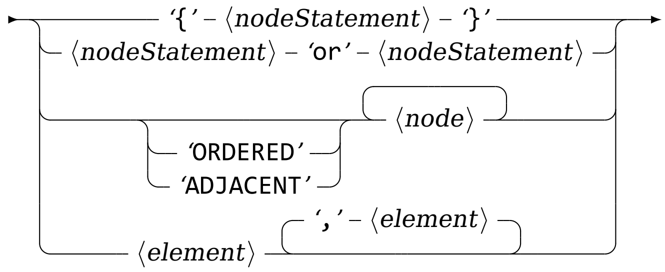
Grammar Snippet 17 (⟨*nodeStatement*⟩)

```
⟨nodeStatement⟩ ::= '{' ⟨nodeStatement⟩ '}'
  |  ⟨nodeArrangement⟩? ⟨node⟩+
  |  ⟨element⟩ (',' ⟨element⟩)*
  |  ⟨nodeStatement⟩ 'or' ⟨nodeStatement⟩

⟨nodeArrangement⟩ ::= 'ORDERED' | 'ADJACENT'

⟨memberLabel⟩ ::= ⟨member⟩ ':'
```

[3] This is particularly useful when using the global constraints to compare content of different lanes. Imagine for instance a query that searches for a certain syntactic construct `C` to be present in two concurrent parse trees `A` and `B`, but will only consider sentences where `C` is embedded deeper inside `A` compared to its embedding depth in `B`.

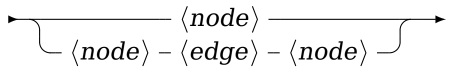
⟨element⟩ ::= ⟨node⟩ | *⟨node⟩ ⟨edge⟩ ⟨node⟩*

⟨nodeStatement⟩:



'{' – ⟨nodeStatement⟩ – '}'
⟨nodeStatement⟩ – 'or' – ⟨nodeStatement⟩
⟨node⟩
'ORDERED'
'ADJACENT'
',' – ⟨element⟩
⟨element⟩

⟨element⟩:



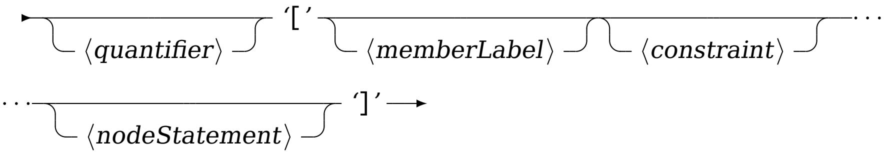
There are four (partly recursive) approaches to express node statements, i.e. grouping, node sequence, element sequence and disjunction. The distinction between node and element sequences exists to easily distinguish sequence or tree queries from graph definitions. Sequence queries do not include hierarchical structural information and as such have no use for edges. In the syntax used for tree nodes in IQL information about the incoming edge is implicitly available from every nested node and constraints related to outgoing edges are to be attached to the respective child terminals of those edges. For graphs where no simple association between nodes and edges exists, there is a necessity to have explicit edge declarations available for querying. As such the ⟨*element*⟩ rule in snippet 17 is a placeholder that can be filled with either node or edge declarations.

Nodes IQL uses square brackets ('[' and ']') to mark individual nodes. snippet 18 gives an overview of the syntax used for defining nodes with all the optional content elements.

Grammar Snippet 18 (⟨*node*⟩)
⟨node⟩ ::= ⟨quantifier⟩? '[*' ⟨memberLabel⟩? ⟨constraint⟩? ⟨nodeStatement⟩? '*]*'*

⟨memberLabel⟩ ::= ⟨member⟩ ':*'*

⟨node⟩:



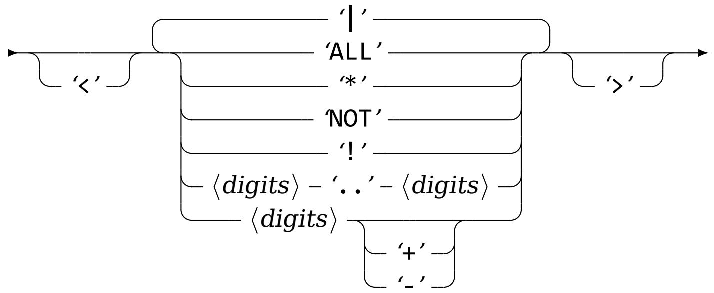
Declaring a node in a structural constraint implicitly marks it as existentially quantified. Additionally, nodes can be **explicitly quantified** with an arbitrary combination of universal quantification, negation, explicit quantification, *at-most* (0 to n), *at-least* (n to inf) or bounded range quantification. The snippet 19 simplifies the overall rules for ⟨*quantifier*⟩ to keep it compact. Albeit being shown here as unrelated to each other, the appearance of the angle brackets ('<' and '>') before or after the actual quantifier content is restricted to either *both* of them being used (for a proper wrapping, such as '<3..10 >') or *none* of them (for plain quantifiers, such as '1|4|ALL'). IQL allows multiple quantifiers to be separated by the pipe symbol '`|`' to express disjunction between quantifiers. This way complex constraints can be defined very neatly, such as the "all or nothing" quantification '`all|not`'. This quantifier combination ensures that either all targets in a certain context match the node in question, or none does.[4]

Grammar Snippet 19 (⟨*quantifier*⟩)

⟨*quantifier*⟩ ::= ⟨*simpleQuantifier*⟩ ('`|`' ⟨*simpleQuantifier*⟩)*
 | '`<`' ⟨*simpleQuantifier*⟩ ('`|`' ⟨*simpleQuantifier*⟩)* '`>`'

⟨*simpleQuantifier*⟩ ::= ('`ALL`' | '`*`')
 | ('`NOT`' | '`!`')
 | ⟨*digits*⟩ ('`+`' | '`-`')?
 | ⟨*digits*⟩ '`..`' ⟨*digits*⟩

⟨*quantifier*⟩:



A node's inner content can optionally have an initial **member label** (identifier with a colon '`:`' afterwards) to link this node to a previously defined member binding (3.8.3). Such as binding restricts the type of corpus member that the node can be matched against. It also provides a point of reference that subsequent constraints (e.g. in the global constraints section, cf. Section 3.8.11) can use to access information of the target the node has been matched against. Note that cross-referencing between nodes from within local constraints (such as in `[$x:][$y: pos!=$x.pos]`) is discouraged[5] and global constraints should be used for this. This approach guarantees that by the time such cross-reference constraints (or "joins", to use database terminology) are evaluated all involved member variables will be assigned preliminary candidates.

Nodes can also optionally define **local constraints** that must evaluate to `true` for target item to be considered as result candidate. Local constraints have full access to properties and annotations of the target item currently being inspected and can take any form described in Section 3.7. Note that it is up to the evaluation engine how to optimize and potentially prune the evaluation process of constraint expressions. Correct evaluation of a conjunctive local constraints with external function calls `[$x: func1($x) && func2($x) && func3($x)]` must not rely on the premise of any particular function (`func1`, `func2` or

[4] Example: find all sentences that either have no word with more than five characters or all of their words have five or more characters.

[5] The reason behind this is that the ICARUS2 Corpus Query Processor (ICQP) per default is not required to honor the order of nodes defined in a query or the linking relations between them when planning the automaton for evaluation. As such there is no guarantee that node `x` will have already been matched against a valid target when the cross-reference constraint inside `y` is evaluated. This would cause an error during evaluation time to occur, which in turn will abort the entire search.

func3) actually being called at all. The evaluation semantics of conjunctive Boolean concatenation allow an early determination of the final result as soon as the stable predicate of one of the inner terms evaluating to `false` is met. Therefore it is perfectly legal (and in parts expected from an efficient evaluation engine) to not evaluate the calls to `func2` and `func3` after `func1` has already caused the result to remain `false`.

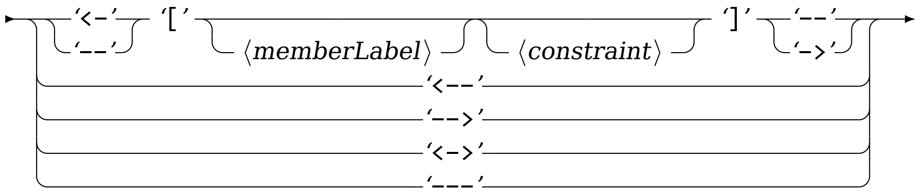
If a ⟨*node*⟩ is used within a tree environment, it can also contain a **nested** ⟨*nodeStatement*⟩ declaration to define structural constraints on child nodes. The target item a node is matched against during query evaluation defines the structural context that is then in turn being used for matching the nested ⟨*node*⟩ instances. Note that this opens up the entire spectrum of node grouping, disjunctions and arrangement modifiers to define constraints on (sub-)trees.

Edges Edges are a structural element that is available exclusively to **graph** constraints (3.8.10). Each edge minimally consists of a source and target node and a type indicator[6] to signal whether the edge is meant to be undirected, uni- or bidirectional and in the case of unidirectional edges which direction between source and target node it takes.

Grammar Snippet 20 (⟨*edge*⟩)

⟨edge⟩ ::= '<--' | '-->' | '<->' | '---'
| ('<-' | '--') '[' ⟨memberLabel⟩? ⟨constraint⟩? ']' ('--' | '->')

⟨edge⟩:



Optionally, an edge can also be assigned its own member label and/or local constraints. In that case the type indicator gets split into two separate parts[7] with a node-like part in the middle to host the edge's label and/or constraints. Note that the same guidelines for cross-referencing from within local constraints that were mentioned above in the section about nodes, also apply to local edge constraints: Correct evaluation of an edge's local constraints must not rely on the availability of cross-referenced external nodes (including the edge's own source and target nodes) or edges. As usual global constraints should be used to define constraints that link individual elements such as edges to other (external) parts of the payload. The section about global constraints (3.8.11) contains further hints on the optimization the ICQP implements in order to prevent unnecessary exploration of the search space.

The **quantification** of edges follows slightly different rules as compared to regular nodes. Every edge can have **up to one** explicit quantifier declaration attached to it, either on the source or target node. If no quantifier is present, the entire edge is by default existentially quantified, otherwise the following rules apply: The node not bearing

[6] Type indicators for empty edges (i.e. edges without an inner constraint or label definition) always consist of three symbols, cf. snippet 20

[7] When split, each part of a type indicator uses two symbols, which are the respective 2/3 of the original three-symbol indicator.

any quantifier is existentially quantified (or "fixed") and the quantifier on the other node acts as an indicator for the multiplicity of the edge itself. The following examples illustrate some use cases for quantification on edges:

```
// a node x linked to 3 nodes that match y
[$x:]--> <3>[$y:]
// 3 nodes matching x that are linked to the same node y
<3>[$x:]-->[$y:]
// node x with no direct link to a node that would match y
[$x:]-->![$y:]
```

Note that edge definitions cannot be chained, so the query "find x linked to y, linked to z" must be expressed in two separate edge definitions, one linking `x` to `y` and one linking `y` to `z`. This means that graph constraints (3.8.10) require some redundancy, but the linked section also explains some easy strategies to minimize overhead.

Node Grouping Nodes (or elements) can be grouped together within curly brackets ('`{`' and '`}`') as defined by the first ⟨*nodeStatement*⟩ rule in snippet 17. This is useful for either restricting the scope of modifiers or directives such as the `ADJACENT` keyword to only a selected few nodes or when expressing a disjunction. Note that a group counts as an individual node statement inside the outer scope and as such is subject to order directives defined there. However, those directives are **not** automatically **inherited** to the inner collection of nodes in the group, allowing for expressions such as the following node sequence:

```
ADJACENT [$x:] {[$a:][$b:]} {[$c:][$d:]} [$y:]
```

This would read as "Find x immediately followed by a, later followed by b+c, later followed by d+y". Note that the adjacency modifier does not apply to the inner sequences a+b and c+d, which are only subject to the implicit order of the sequence declaration.[8] The concept of node grouping is especially important for the tree (3.8.9) and graph (3.8.10) constraints introduced below, as by default those do not impose an a priori order of nodes.

Node Sequence Nodes usable for sequences (3.8.8) and trees (3.8.9) are defined in a simple sequence style (second ⟨*nodeStatement*⟩ rule in snippet 17). Instances of ⟨*node*⟩ in a sequence are defined one after another without special separator symbols. They may optionally be preceded by a ⟨*nodeArrangement*⟩ directive to guide the matching progress. Currently there are only two directives available to specify the node arrangement (`ORDERED` and `ADJACENT`), but this might increase in the future, making node grouping a very important tool for defining complex structural compositions.

Element Sequence Similar to node sequences, ⟨*element*⟩ instances can also be used in a list-style collection (third rule of ⟨*nodeStatement*⟩ in snippet 17), but with noticeable differences: Element sequences do use a separator symbol (a simple comma '`,`') between ⟨*element*⟩ definitions. Since IQL does not use keywords to signal the structural type to

[8] An additional `ORDERED` in each of the groups would make that explicit, but is redundant.

be expected in a query payload[9] this approach was necessary to easily detect the type of structure. It also hints at the second difference, that is, element sequences do not support arrangement modifiers (as ⟨*element*⟩ instances can be either nodes or edges, with the latter not being suitable for this kind of ordering) and as such can be more intuitively be understood as sets of ⟨*element*⟩ instances.

Structural Disjunction As the forth option of ⟨*nodeStatement*⟩ in snippet 17, the disjunction of entire node statements provides a very powerful tool to express complex queries. Two node statements are linked with the `OR` keyword to mark them as "either or" cases. Since this opens up recursion, a complex disjunction may contain more than two node statements in total. Note that the order of node statements in a disjunction does not imply a specific priority the evaluation engine has to follow. For illustration, the second example below might match an instance of `[$y:]` first:

```
[$x:] or [$y:]          // x or y
[$x:] or [$y:] or [$z]  // x or y or z
[$x:] or {[$y:][$z]}    // either x alone or a sequence y+z
[$x: {[$y:] or [$z]}]   // tree node x with either y or z as child
// complex nested disjunctions
[$x: [$y: [$z: [$a:] or [$b:]]] or [$z]]
```

3.8.8 Sequence Constraints

As the most basic form of structural constraints this type is used to match sequences of nodes to items in the target corpus. Multiple nodes in a sequence declaration are required to match to items in exactly the order they are defined in (but not necessarily adjacent to each other, use the `ADJACENT` directive in front of a node sequence for that).

Examples:

```
[]                    // empty node
[pos=="NN"]           // node with local constraint
<2>[$x:]              // node x exactly 2 times
[$x:]<2-5>[][$y:]     // nodes x & y with 2-5 nodes in between
[$x:] ![$y:]          // node x without any node y following
ADJACENT [$x:][$y:] // node y directly following node x
[$x:] or [$y:]        // disjunction: either x OR y
{[$x:][$y:]} or [$z:] // disjunction: either group x+y OR z
```

Sequence Matching Sequence constraints only provide a single dimension for *moving* the query sequence through the search space of the target corpus. Matching is performed greedily in order of node appearance in the query, following the direction specified by the corpus itself. If the switch to reverse direction (cf. 2.12.1) is set, then matching attempts will start from the very end of each matching context (such as a container or structure), but the order of nodes to match will remain the same. That is, in a node sequence

[9] An earlier draft made use of `TREE` and `GRAPH` keywords to distinguish those types from the basic node sequence, but in an effort to reduce the overall number of keywords (that users had to learn) this approach was dropped.

`[$x:][$y:]` the node matched by `x` will **always** be before the node matched by `y` according to the original direction of items in the corpus. Only the direction from which the overall matching attempt will start changes with the respective switch. This also leads to a very simple and intuitive semantic for the `FIRST` and `LAST` modifier keywords: They stay true to their names and limit the returned match to either the first or last, with respect to the current direction.

Special Note: Evaluation of quantified nodes in IQL is done greedily, i.e. the engine will try to match as many instances as possible. However, since an empty node does only provide the constraint of existential quantification, the above example will find the first occurrence of pair c+d after the initial hit on pair a-b. An extension of the quantifier syntax or a change in the "at most" semantics is on the drawing board, but currently not considered to be implemented (a new keyword `inf` could potentially be used with the "at most" quantifier to create a hint to greedily expand empty nodes to the total limit instead of a "first hit" eager strategy, such as '`<inf->`').

Empty nodes with quantifiers can be used as proxies to model distance constraints, as seen in above examples. Since the `ADJACENT` directive changes the behavior of an entire node sequence, some creativity can be necessary to achieve mixed cases, such as *"find an adjacent pair a+b that is later followed by another adjacent pair c+d"*. Possible (and simple) solutions for this query could be the following:

```
ADJACENT [$a:][$b:] <0+>[] [$c:][$d:]
{ADJACENT [$a:][$b:]} {ADJACENT [$c:][$d:]}
```

3.8.9 Tree Constraints

Located between mere sequences (3.8.8) and graphs (3.8.10) this type of structural constraints is meant to target tree structures, such as (but not limited to) syntax trees, coreference structures, discourse, etc. To simplify query syntax, IQL uses a similar approach as the original ICARUS project, which in turn took inspiration from PML-TQ: To signal parent-child relations, child nodes are nested within their respective parent, effectively making each node yet another scope for a sequence of child nodes. Contrary to bare sequence constraints (3.8.8) the order of (child) nodes to be matched in the corpus is **not** implicitly defined by the order of constraint nodes! Instead, the `ORDERED` or `ADJACENT` keywords need to be used explicitly to signal that a specific kind of order should be honored.[10] Apart from this little addendum tree constraints behave basically the same as nested sequence constraints: They can be individually quantified or existentially negated, as well as grouped and linked via the `OR` keyword to expression disjunctions.

Examples:

[10] Also note that `ORDERED` and `ADJACENT` when used inside tree nodes are referring to the order of nodes as defined by the parent, **not** their overall position when projected onto the underlying foundation layer!

```
[[]]               // anonymous nesting of nodes
[$x: [$y:]]        // nesting of node y inside x
[[$x:] [$y:]]      // nesting of siblings x and y
[$x: [$y: [$z:]]] // deep nesting chain
[$x: <2->[$y:]]    // at most 2 y nested inside x
[$x: ![$y:]]       // node x without any child matching y

// internal disjunction
[[$x:] or [$y:] or {[$z: <4+>[]]}]
```

Tree Matching As opposed to sequences (3.8.8), trees (and subsequently also graphs, cf. 3.8.10) offer an additional dimension of matching freedom over the bare iteration of items in a container or structure to be matched. This requires further specification of the matching process to guarantee consistent results and define basic expectations. Below explanations are primarily intended to specify the behavior in the presence of limiting modifiers (`FIRST`, `LAST`, `ANY`) in the payload, but are also of interest for the expected order of returned matches if no limiting modifiers are defined.

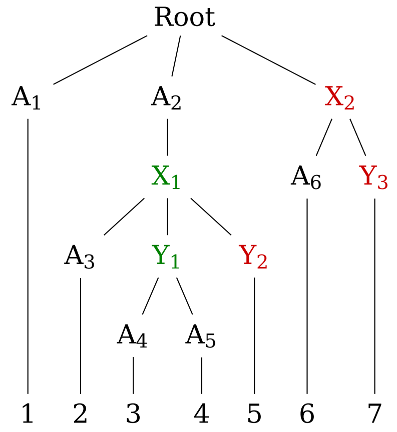


Figure 1: Example tree with highlighted hits for the simple tree query `[$x: [$y:]]`.

Consider a simple tree query `[$x: [$y:]]` where `x` and `y` are bound nodes with individual constraints. The nodes X_n or Y_n in the example tree (Fig. 1) then denote the n-th node that match the constraints of query nodes `x` and `y`, respectively. The nodes A_1 to A_6 are placeholders for nodes that match neither `x` nor `y`.

The tree contains 3 possible matches for the query, specifically the pairs {X_1,Y_1}, {X_1,Y_2} and {X_2,Y_2}.
[Root [A_1] [A_2 [X_1 [A_3] [Y_1 [A_4] [A_5]] [Y_2]]] [X_2 [A_6] [Y_3]]]

expand on the order of matches

3.8.10 Graph Constraints

content, explain node and edge composition, mention referencing as a strategy to minimize edge declaration overhead, etc...

3.8.11 Global Constraints

Global constraints can be any basic constraint Section 3.7 and follow after the main section of structural constraints, indicated by the `HAVING` keyword (cf. snippet 16).

Evaluation Priorities If global constraints are present, the evaluation process changes to a two-stage strategy: Matchers for the associated structural constraints produce preliminary result candidates and the global constraints are then evaluated for each such candidate. This makes global constraints both very powerful as they have access to more information compared to regular (internal or local) constraints (e.g. they already *know* that all local constraints evaluated to `true` and the exact candidates produced for structural constraints) and also very critical when it comes to performance. It can be very tempting to construct queries such as the following one (bindings section omitted) that only matches when y is the last child of x:

```
FIND [$x: [$y:]] HAVING $x.indexOf($y) == $x.size-1
```

This will cause the structural matcher to potentially propose **all** children of x as candidates to be processed by the global constraints section. Subsequently, for a node of size N this will produce $N-1$ candidates that are bound to fail the global constraint check. Section 4 lists several families of utility markers and functions that can be used to signal the evaluation engine that certain local constraints are to be treated as special filters. With the use of those utility markers, above query looks like the following and will be vastly more efficient to evaluate:

```
FIND [$x: [$y: isLastChild]]
```

Similarly global constraints are not the place to perform filtering on general properties of the current UoI, such as sentence length (use the FILTER BY expression for that, cf. Section 3.8.1).

Constraint Hoisting Per default, global constraints are second-class constraints, that are only consulted once the structural and local constraints in the other sections of a query payload have been evaluated. This provides them with a lot more decision power and available information, but also can lead to rather inefficient evaluation scenarios. Consider a query FIND [$x: [$y: [$z]]] HAVING $x.pos!=$y.pos that looks for nested nodes x, y and z where in addition to local constraints the nodes x and y are required to have different part-pf-speech tags. With the default two-stage evaluation process described above this would result in a serious overhead when evaluating a target tree: For every successful match of [$x: [$y:]] the engine would continue to look for children of y matching z, regardless of x and y satisfying the global constraint.

Through a optimization technique dubbed *constraint hoisting* the ICQP will try to work around this performance issue and attach global constraints to the nodes in the (tree) automaton that matches a certain part of the query. In the example above the matcher would evaluate the global constraint $x.pos!=$y.pos as soon as it successfully identified candidates for x and y, thereby reducing the overhead of searching for potential candidates for z that would ultimately fail due to x and y not satisfying the global constraint.

Note that only very specific global constraints can successfully be hoisted!

list properties required for hoisting and mention hoisting of constraint fragments, since IQL splits boolean expressions into

3.9 Result Processing

There be dragons...

(Content of the result section will be added as IQL evolves)

4 Utility Markers & Functions

The following utility features are provided by the ICQP but are **not** part of the core specification. As such it is possible for engine extensions to override them, change their behavior or completely remove them if desired. They are listed here as per default they all are available and provide valuable improvements for performance and usability.

4.1 Position Markers

Every node (3.8.7) in a query has an implicit Container or Structure context that it is hosted or contained in.[11] IQL offers a variety of predefined helper functions to exploit this fact and to optimize queries. Table 5 lists the functions currently available, their arguments and matching conditions. Note that legal arguments for a node that occurs in a container of size N reside in the closed integer interval $[1..N]$ for regular index values and $[-N..-1]$ for reverse indices (see below).[12] The syntax for providing arguments follows the normal rules for method invocations in IQL 3.6.3, e.g. `isOutside(4, 12)`. Arguments can be provided as literals, which is is the preferred way, or as any other kind of expression that evaluates to the necessary type (or can be converted, depending on the query configuration). It is not possible to access properties from the node itself for which the marker is defined, as position markers are evaluated **before** the corresponding nodes are matched and subsequently any such attempts will result in an error during query evaluation. However, globally accessible information or any data provided by surrounding containers or structures can be used to specify marker arguments.[13]

Label	**Arguments**	**Matching Condition**
isFirst	-	Node is the first in its context ($index = 1$).
isLast	-	Node is the last in its context ($index = N$).
isAfter	int	Node occurs after designated position ($index > arg$). Legal values are: $1 \leq arg < N$.
isBefore	int	Node occurs before designated position ($index < arg$). Legal values are: $1 < arg \leq N$.
isInside	int, int	Node occurs inside designated interval ($arg_1 \leq index \leq arg_2$). Legal values are: $1 \leq arg_1 < arg_2 \leq N$.
isOutside	int, int	Node occurs outside designated interval ($index < arg_1$ and $index > arg_2$). Legal values are: $1 \leq arg_1 < arg_2 \leq N$.
isAt	int	Node occurs at specific position ($index = arg$). Legal values are: $1 \leq arg \leq N$.
isNotAt	int	Node occurs at any position except the designated one ($index \neq arg$). Legal values are: $1 \leq arg \leq N$.

Table 5: Position markers provided by the default evaluation engine for IQL. The conditions in the last column assume a container of size N as context. If a marker function takes arguments, they appear in the conditions as arg for the sole argument or arg_i for the i-th argument list, with arg_1 being the first argument.

Direction Position markers follow the direction specified for the whole query, same as all structural constraints (3.8.7). This means they are also subject to reversing via switch (2.12.1) the order in which UoIs are traversed. Note that this effectively reverses the semantics of all the markers listed in Table 5 (see below section about negation for examples

[11] The ICMF specifies that each item can only be hosted (or more accurately *owned*) by a single container or derived object, but be contained within an arbitrary number of additional containers or structures.

[12] This is in contrast to the 0-based access of array or list elements in IQL.

[13] For example, a node could be defined to be in first position for short sentences and in last position for long ones.

on how construct markers that cover the inverse index interval). For instance `isFirst` will target the last item when traversing a container in the original direction. Users need to keep this in mind when mixing position markers with switches that influence the order of traversal during node matching.

Negation Position markers cannot be negated when used in a constraint expression.[14] For every function/marker in Table 5 there exists a complement that can be used to easily express the negated condition. For instance `isBefore(n)` can be negated into `isAfter(n-1)`, but the engine will not perform this conversion and it up to the user to properly provide the intended negated form. Trying to negate a position marker inside structural constraints will result in an error during the preprocessing of the query payload.

Reverse Indices Any position marker that takes at least one argument also supports negative values that are treated as reverse indices, starting from the end of the surrounding container's size. This follows the same semantics as the index scheme for arrays or lists (3.6.4). Note that certain position markers with interval arguments (such as `inBetween(`arg_1`, `arg_2`)`) still require actual interval boundaries to be in the proper order (generally $arg_1 \leq arg_2$).

Relative Indices Per default all the position markers that take argument expect them to be integer values for explicit designation of the desired index or index range. This means that any other type of argument will be automatically converted to an integer value if the query configuration permits it. However, if the associated switch (2.12.1) is active[15], position markers can use relative arguments in the form of floating-point numbers. Relative indices are expected to be in the open interval $(0..1)$ (or $(-1..0)$ for negative reverse indices) and are treated as percentages of the total size of the surrounding container. The following examples illustrate this approach (assume the markers are used inside a token node that is directly hosted within a sentence):

```
// the token can only occur in the first half of the sentence
isBefore(0.5)

// the token must occur after the first 20% and before the last 80%
isBetween(0.2, 0.8)
```

The actual index values to be used for determining the legal position of the node are calculated by multiplying the context's size attribute (e.g. the length of the surrounding sentence) with the marker's relative argument and then rounding down the result to the nearest integer. It is easily possible to define the same position marker with relative indices in both normal and reverse form. For instance, `isBefore(0.8)` and `isBefore(-0.2)` both describe the same 80% interval. As a general rule, relative indices should only be used with regular (positive) arguments.

[14] Depending on the query configuration doing so would either fail the evaluation process with an error or silently ignore the issue, potentially producing invalid results.

[15] This switch is off by default as it interferes with the automatic type conversion of arguments. If it is active, arguments must explicitly be cast if not already an integer or floating-point number. If the switch is not active, any floating-point argument will be cast to an integer (typically 0 if any of the arguments are intended to be reverse indices).

Marker Stacking Position markers generally may **only** be used as top-level terms in a constraint expression. IQL allows multiple position markers to occur in the same constraint expression, with the following condition: When the expression is viewed in conjunctive normal form, positional markers may not be mixed with non-marker terms inside a single clause. Multiple marker-only clauses are allowed however.

```
// legal use cases
[isLast]
[pos!="NN" && isBefore(10)]
[pos!="NN" && (isBefore(10) || isAfter(24))]
[pos!="NN" && isNotAt(5) && (isBefore(10) || isAfter(24))]

// illegal use of position markers (see rules above)
[pos!="NN" || isLast]
// above constraint must be expressed via node disjunction
{[pos!="NN"] or [isLast]}
```

The reasons for this limitation in stacking position markers stems from the underlying interval arithmetic. Each positional marker effectively describes a non-empty interval of possible index values for the node. Conjunction of markers results in the intersection of their intervals, possibly creating an empty set, which will render the node constraint impossible to satisfy. Disjunction of markers on the other hand creates a union of their intervals, either joining them if they overlap or creating a set of disjoint intervals that denotes a discontinuous collection of legal values. The final result is a fixed[16] collection of index values that can be iterated to check for potential result candidates. In the absence of any kind of positional marker this default index set to be traversed is the complete interval $[1..N]$ for a host container of size N. In case of disjunction between a positional marker and a non-marker term, this would render the marker's filter function useless.[17]

Markers in Global Constraints Global constraints (3.8.11) have access to an extended version of the "local" position markers that acts as a general predicate for items (and additional index arguments, depending on the marker). So for example `isLast($x)` is the method-equivalent to the local marker `[$x: isLast]`. Note that there is no performance benefit by using those methods in global constraints, as actual position markers are only recognized within structural constraints. The version for global constraints merely serves as syntactic sugar. However, since those methods are not subject to the same limitations as mentioned int the section about stacking above, they can be used freely inside local constraints (both in nested and disjunctive expressions).

4.2 Tree Markers

While the basic positional markers described in Section 4.1 limit a node's overall position in the surrounding container, tree markers work on the hierarchical properties of tree structures. As such they define legal positions of a node within its parent's list of children. All tree markers are subject to the same rules as position markers regarding negation,

[16] Fixed in the sense that it only depends on the overall size of the surrounding container.

[17] Joining the complete interval $[1..N]$ with any sub-interval does not reduce the number of indices to be traversed

direction, relative and reverse indices, as well as stacking. Note however, that the default order for traversing child nodes in a tree does not necessarily[18] reflect the actual order of the child nodes position when projected onto the underlying foundation layer. It can therefore make sense to define additional precedence constraints via global constraints if needed.

Label	**Arguments**	**Matching Condition**
isFirstChild	-	Node is the first child of its parent ($childIndex = 1$).
isLastChild	-	Node is the last child of its parent ($childIndex = N - 1$).
isChildAfter	int	Node occurs as child after designated position ($childIndex > arg$). Legal values are: $1 \leq arg < N$.
isChildBefore	int	Node occurs as child before designated position ($childIndex < arg$). Legal values are: $1 < arg \leq N$.
isChildInside	int, int	Node occurs as child inside designated interval ($arg_1 \leq childIndex \leq arg_2$). Legal values are: $1 \leq arg_1 < arg_2 \leq N$.
isChildOutside	int, int	Node occurs as child outside designated interval ($childIndex < arg_1$ and $childIndex > arg_2$). Legal values are: $1 \leq arg_1 < arg_2 \leq N$.
isChildAt	int	Node occurs as child at specific position ($childIndex = arg$). Legal values are: $1 \leq arg \leq N$.
isChildNotAt	int	Node occurs as child at any position except the designated one ($childIndex \neq arg$). Legal values are: $1 \leq arg \leq N$.

Table 6: Positional markers for trees provided by the default evaluation engine for IQL. The conditions in the last column assume a parent node with N children as context and $childIndex$ being the desired position of the node within its parent's list of children. If a marker function takes arguments, they appear in the conditions as arg for the sole argument or arg_i for the i-th argument list, with arg_1 being the first argument.

4.2.1 Hierarchy Markers

Apart from the horizontal domain described above, tree markers also cover the hierarchical aspect of tree structures. A series of additional markers is available to specify a node's vertical location in the tree, as shown in Table 7. These markers inherit all the rules of general tree markers, except that they cannot be used with relative indices.

All the additional hierarchical markers in Table 7 operate purely on the vertical axis within a tree structure (with the exception of the two "descendants" markers that use the fringes of the subtree spanned by a node's parent to place candidate nodes in the target tree). The initial lot of them (`isRoot`, `isNoRoot`, `isLeaf`, `isNoLeaf` and `isIntermediate`) are based on the presence or absence of child nodes and/or a parent. The remaining markers use the concept of *generations*[19] in the tree. Given a tree structure T and the

[18] The ICMF does however strongly encourage tree structures to order children according to their begin-offsets if it does not interfere with their intended semantics.

[19] Also often called "levels" in tree structures.

Label	**Arguments**	**Matching Condition**
isRoot	-	Node is a designated root in the structure.
isNoRoot	-	Node has a parent.
isLeaf	-	Node is a leaf, i.e. it has no children.
isNoLeaf	-	Node is not a leaf, i.e. it has at least one child.
isIntermediate	-	Node is neither a designated root nor a leaf.
isGeneration	int	Node is a member of the arg-th generation of the parent node. Note that only values of $arg \geq 2$ make any sense, as per default parent-child nesting in tree constraints implies membership in the 1st generation.
isNotGeneration	int	Node is nested arbitrarily deep in its parent, but not in the designated generation.
isGenerationAfter	int	Node is nested at least $arg + 1$ generations deep in its parent.
isGenerationBefore	int	Node is nested no more than $arg - 1$ generations deep in its parent.
isAnyGeneration	-	Node is nested arbitrarily deep in its parent. This is the equivalent of a full transitive closure over the tree dominance relation between parent and child nodes.
isLeftmostt	-	Node is located along the path of leftmost descendants (according to the actual position on the underlying foundation layer, **not** necessarily the order of child nodes along each step). This marker is generally combined with one of the generation-based markers to provide an indicator on the vertical location of the desired descendant.
isRightmost	-	Represents the symmetric opposite of above `isLeftmost`, marker, following the path of rightmost descendants.

Table 7: Additional markers for hierarchical properties in tree structures. Note that "parent" in the matching conditions refers to the node in a target tree that has been matched by the original node's parent.

root node R, the set of nodes in T are partitioned into disjoint sets of generations G_0 to G_n where n is the height of T and G_i is the set of nodes belonging to the i-th generation, that is all the nodes with a depth of i (i.e. path distance i to R). Note that R is the only member in G_0 and G_n can only contain leaf nodes. Generations in structural IQL queries are always calculated based on the immediate parent of the node declaration that contains the marker(s). If a constraint related to the nesting distance between transitively nested nodes or other non-immediate relations is desired, global constraints (3.8.11) should be used, possibly employing dedicated tree functions (4.5).

If no generation markers are used within a node constraint and the node is nested, a direct parent-child relation is assumed, which is equivalent to `isGeneration(1)`. Nodes that represent the root of a tree query can also use generation-based markers to indicate where in a prospective target tree the associated item should be located. Regular tree hierarchy markers can also be combined with generation-based markers to further specify the location within a tree. For instance, `[$x: [isGenerationAfter(2) && isNoLeaf]]` will match any node that is nested at least 3 steps deep within `x` but ignores leaves. The `isRoot` marker however cannot be combined with any other hierarchy-related markers, as only designated roots in the target structure can match it and the only sensible combinations would be with normal (horizontal) position markers (4.1). If the `isRoot` marker is being used within a nested node, an error will be issued as the query will be impossible to satisfy.[20]

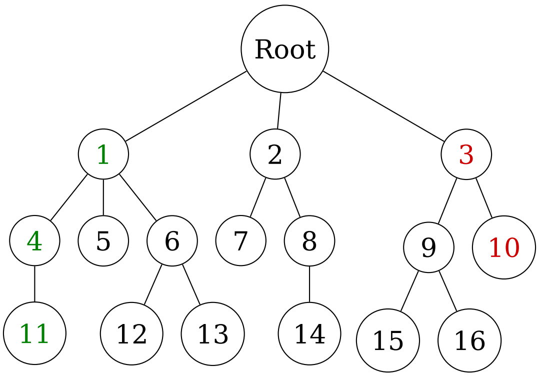


Figure 2: Example tree with highlighted nodes on the fringes for the `isLeftmost` (green, nodes 1, 4 and 11) and `isRightmost` (red, nodes 3 and 10) markers.

The two **fringe** markers `isLeftmost` and `isRightmost` play a special role as they effectively combine horizontal and vertical navigation into a single marker. They signal that node candidates must be located at the left (or right) fringe of a target tree as illustrated by Fig. 2. As such they describe a path starting from a particular node downwards rather than a specific position within that node's list of immediate children. Note that the *leftmost* and *rightmost* properties do **not** refer to the position of children within their parent's list of child nodes. Instead the actual position on the underlying foundation layer is taken as basis for determining which nodes belong to the respective fringes. More specifically, an item's begin index is responsible for its position in the lift of *leftmost* candidates, and the end index is taken as the decisive property when it comes to the *rightmost* fringe. Since the ICMF does not require structures to order outgoing edges for a parent node according to the positional indices of its children, using those fringe markers can incur a certain computational overhead.

Fringe markers are typically combined with one or more generation-based markers to further specify a node's position in the tree. If no additional hierarchical marker is provided only the immediate children of a node are taken into account. Otherwise the fringe marker will guide the selection of child nodes along the path to the desired vertical position.

Examples:

```
[isLeftmost && isLeaf]            // node 11
[isRightmost && isGeneration(-2)] // node 3
```

[20] The ICQP does this for any kinds of obviously erroneous queries when the planning phase results in unsolvable issues.

4.2.2 Evaluation Performance

Tree-related methods for fetching hierarchical information (such as height and depth of a node) are optional in ICMF. As a result the ICQP might have to compute the data required for evaluating a query at runtime, generating some additional overhead from tree traversals. This is especially critical for hierarchical markers that use reverse indices, as knowledge about the overall tree height is required to effectively translate relative indices into absolute intervals of legal (generation) values.

4.3 Sequence Functions

finish

correct description/caption in table

Label	**Arguments**	**Result**	Description
isAdjacent	node[]	Boolean	Check whether the given nodes form a continuous interval on the underlying foundation layer.
isOrdered	node[]	Boolean	Check whether the given nodes form an ordered sequence, as indicated by their begin indices, allowing for nodes to overlap.
isOrderedDisjoint	node[]	Boolean	Check whether the given nodes form a disjoint ordered sequence, as indicated by their begin indices.

Table 8: Utility functions for advanced sequence matching. Primarily these functions are intended to provide equivalents for the position markers in Section 4.1 usable in global constraints and batch methods for checking the horizontal arrangement of nodes.

4.4 Spatial Functions

To provide fine control over horizontal positioning of nodes, as set of utility functions is provided that models specific spatial relations regarding (relative) positioning, overlap or inclusion of a pair of nodes.

4.5 Tree Functions

The nested style of defining tree queries in IQL in top-down manner and the associated matcher in the ICQP also producing top-down automata for evaluation cause some obvious shortcomings: For problems in bottom-up style, for instance “Find the first common ancestor of adjacent phrases x and y” it can be more intuitive (and usually more efficient wrt evaluation) to define the query bottom-up. Using the tree functions listed in Table 10 a simple query for this question could look like the following:

```
FIND ADJACENT [$x:][$y:] HAVING ancestor($x,$y) AS $a}
```

This will first try to find two adjacent instances for `x` and `y` from the set of nodes in the target tree and then use the `ancestor($x,$y)` method to fetch their first common

Label	Matching Condition
isLeftOf	Node n_1 located completely outside and to the left of n_2 ($n_1.right < n_2.left$).
isRightOf	Node n_1 located completely outside and to the right of n_2 ($n_1.left > n_2.right$).
overlaps	Nodes n_1 and n_2 overlap in some way ($n_1.left \leq n_2.right \land n_2.left \leq n_1.right$).
overlapsNot	Nodes n_1 and n_2 do not overlap ($n_1.left > n_2.right \lor n_2.left > n_1.right$). This function is a negation of `overlaps` that mainly exists as as syntactic sugar.
overlapsLeft	Node n_1 overlaps n_2 on the left side ($n_1.left \leq n_2.left \land n_1.right \geq n_2.left$).
overlapsRight	Node n_1 overlaps n_2 on the right side ($n_1.right \geq n_2.right \land n_1.left \leq n_2.right$).
surrounds	Node n_1 fully surrounds n_2, i.e. it overlaps n_2 on both sides ($n_1.left \leq n_2.left \land n_1.right \geq n_2.right$).
fits	Node n_1 exactly fits n_2, i.e. its boundaries match on both sides ($n_1.left = n_2.left \land n_1.right = n_2.right$).
alignsLeft	Node n_1 and n_2 share the same left boundary ($n_1.left = n_2.left$).
alignsRight	Node n_1 and n_2 share the same right boundary ($n_1.right = n_2.right$).

Table 9: Spatial functions to model relative positioning between two nodes. All functions listed take exactly two node arguments, n_1 and n_2. For the matching condition column, $n.left$ stands for the begin index (the left boundary of the span covered by n when projected onto the common foundation layer) and $n.right$ for the end index with $n.left \leq n.right$.

Label	Arguments	Result
ancestor	node[]	Returns the first common ancestor of the given nodes.
parentAt	node, int	Returns the arg_2-th parent of arg_1. The immediate parent of arg_1 is reached with $arg_2 = 1$, its grandparent with $arg_2 = 2$ and so on . . . Note that a node's immediate parent can (in the presence of a single unambiguous structure) be reached with the `parent` shortcut field.

Table 10: Utility functions for advanced tree matching. Primarily these functions are intended to complement the top-down tree matching strategy employed by the ICQP with bottom-up evaluation and navigation options.

ancestor node and assign it to `a` for further use. The same can be achieved in a top-down fashion with a slightly more verbose query:

```
FIND [$a: childCount>=2
    [$x: isAnyGeneration]
    [$y: isAnyGeneration]]
HAVING isAdjacent($x, $y)
```

Besides the differences in appearance those two approaches also differ greatly in their respective evaluation complexity. While the bottom-up query can be efficiently solved in $\mathcal{O}(nh)$ time for trees with n nodes and height h [21], the top-down query results in a vastly more expensive search. Since node a does not provide any strong filter mechanism besides needing at least 2 child nodes, the matcher will have to move the search tree through the target tree and repeatedly check any possible combination of descendant nodes matching x and y to be adjacent.

4.6 Graph Functions

There be dragons. . .

(Content of the this section will be added as IQL evolves)

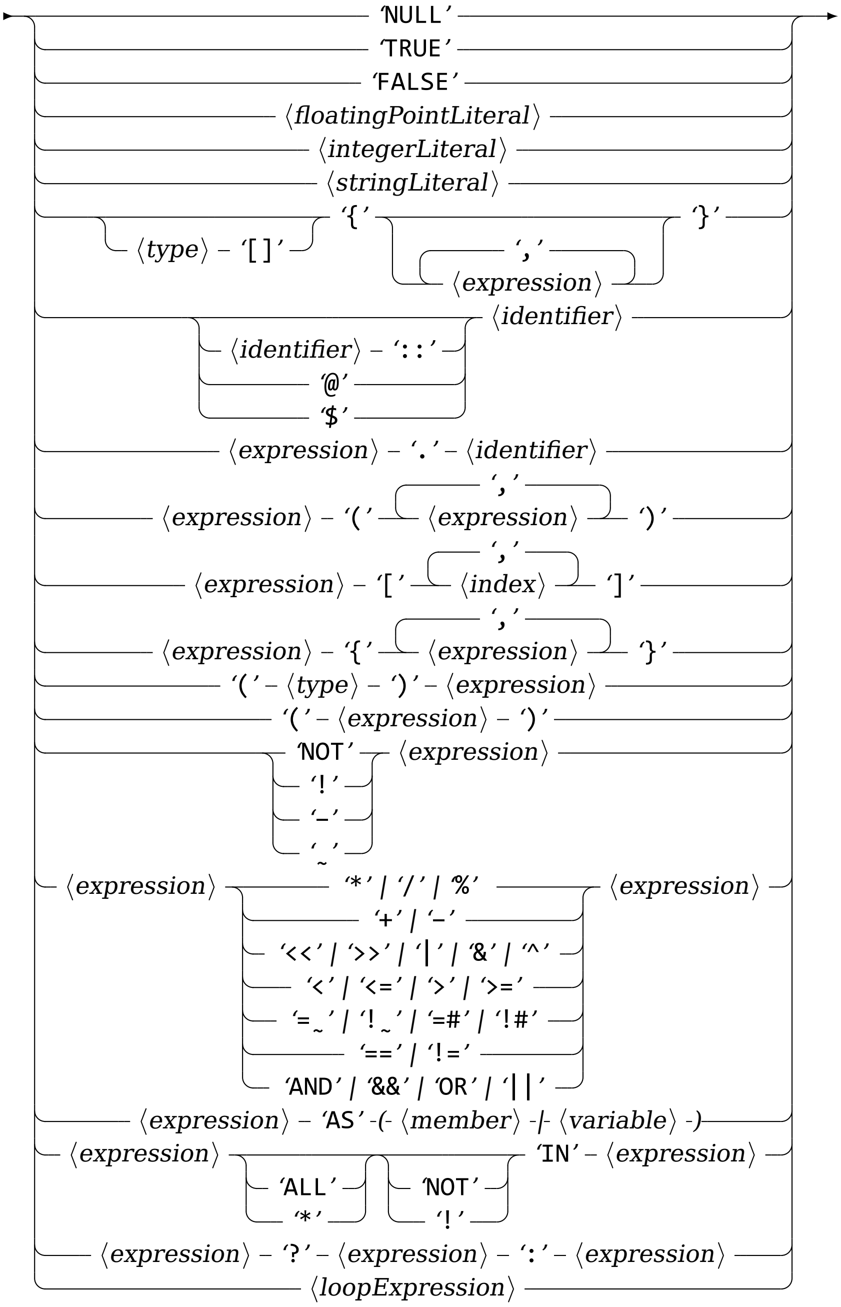
Appendices

A Extended Grammar Diagrams

Grammar Snippet 21

⟨*expression*⟩*:*

[21] Construction of a reverse lookup between indices and the nodes beginning/ending there and subsequent production of adjacent pairs for x and y requires $\mathcal{O}(n)$ time. Without a dedicated utility data structure, determining the first common ancestor of x and y is linear in the maximum depth of the two nodes.


'NULL'
'TRUE'
'FALSE'
⟨floatingPointLiteral⟩
⟨integerLiteral⟩
⟨stringLiteral⟩
⟨type⟩ – '[]' '{' ',' ⟨expression⟩ '}'
⟨identifier⟩ – '::' '@' '$' ⟨identifier⟩
⟨expression⟩ – '.' – ⟨identifier⟩
⟨expression⟩ – '(' ',' ⟨expression⟩ ')'
⟨expression⟩ – '[' ',' ⟨index⟩ ']'
⟨expression⟩ – '{' ',' ⟨expression⟩ '}'
'(' – ⟨type⟩ – ')' – ⟨expression⟩
'(' – ⟨expression⟩ – ')'
'NOT' '!' '–' '~' ⟨expression⟩
⟨expression⟩ ⟨expression⟩
'*' | '/' | '%'
'+' | '–'
'<<' | '>>' | '|' | '&' | '^'
'<' | '<=' | '>' | '>='
'=~' | '!~' | '=#' | '!#'
'==' | '!='
'AND' | '&&' | 'OR' | '||'
⟨expression⟩ – 'AS' -(- ⟨member⟩ -|- ⟨variable⟩ -)
⟨expression⟩ 'ALL' '*' 'NOT' '!' 'IN' – ⟨expression⟩
⟨expression⟩ – '?' – ⟨expression⟩ – ':' – ⟨expression⟩
⟨loopExpression⟩