ICARUS2 Query Language Specification

Markus Gärtner

2020

Contents

1 Query Structure **3**

2 JSON-LD Elements **3**

2.1 Binding . . . 3

2.2 Constraint . . . 4

2.2.1 Predicate . . . 5

2.2.2 Term . . . 5

2.3 Corpus . . . 6

2.4 Data . . . 6

2.5 Element . . . 7

2.5.1 Node . . . 7

2.5.2 Tree Node . . . 8

2.5.3 Edge . . . 8

2.5.4 Element Disjunction . . . 9

2.6 Expression . . . 10

2.7 Group . . . 10

2.8 Import . . . 11

2.9 Lane . . . 11

2.10 Layer . . . 12

2.11 Payload . . . 13

2.12 Property . . . 14

2.12.1 Switches . . . 14

2.13 Quantifier . . . 14

2.14 Query . . . 16

2.15 Reference . . . 17

2.16 Result . . . 17

2.17 Result Instruction 18
2.18 Scope 18
2.19 Sorting 18
2.20 Stream 19

3 Inner IQL ELements 20

3.1 Reserved Words 20
3.2 Comments 20
3.3 Literals 21
3.3.1 String Literals 21
3.3.2 Boolean Literals 21
3.3.3 Integer Literals 21
3.3.4 Floating Point Literals 22
3.4 Identifiers 22
3.5 Variables and References 23
3.6 Expressions 24
3.6.1 Primary Expressions 24
3.6.2 Path Expressions 24
3.6.3 Method Invocation 25
3.7 List Access 25
3.7.1 Annotation Access 26
3.7.2 Type Cast 27
3.7.3 Wrapping 27
3.7.4 Set Predicate 27
3.7.5 Unary Operation 28
3.7.6 Binary Operation 29
3.7.7 Ternary Operation 31
3.7.8 Value Expansion 31
3.8 Constraints 32
3.9 Structural Constraints 32
3.9.1 Basic Constraints 33
3.9.2 Flat Constraints 33
3.9.3 Tree Constraints 33
3.9.4 Graph Constraints 33
3.9.5 Global Constraints 33
3.10 Result Processing 34
3.11 Query Payload 34

1 Query Structure

Queries in 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*). This document lists the basic building blocks of queries and their compositions.

```
{
  "@context" : "http://www.ims.uni-stuttgart.de/icarus/
      jsonld/iql/query"
}
```

2 JSON-LD Elements

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`(2.5.1) 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`(2.5.2) can be marked as `consumed` if above conditions are met and all nested `iql:children` are marked `consumed`.
- An `iql:Edge`(2.5.3) 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`(2.5.4) is considered `consumed` if at least one of its `iql:alternatives` has been marked `consumed`.

2.5.1 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.

2.5.2 Tree Node

Extension of the simple `iql:Node` type (2.5.1) to add implicit hierarchical constraints related to dominance within tree structures.
Extends `iql:Node`(2.5.1).

Nested Elements of `iql:TreeNode`:

Element	Type	Required
children	array of `<Element>` (2.5)	no

`iql:children` Optional list of nested nodes or node alternatives.

2.5.3 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` (2.5.1)	yes
target	`iql:Node` (2.5.1)	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!

2.5.4 Element Disjunction

- alternatives

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 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

- id
- groupBy
- filterOn
- label
- defaultValue

XXX

Attributes of `iql:XXX`:

Attribute	Type	Required	Default
xxx	xxx	x	

`iql:xxx` XXX.

Nested Elements of `iql:XXX`:

Element	Type	Required
xxx	array of `iql:XXX` (**??**)	no

`iql:xxx` XXX.

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

- id
- name
- laneType
- elements
- nodeArrangement

XXX

Attributes of `iql:XXX`:

Attribute	Type	Required	Default
xxx	xxx	x	

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

`iql:xxx` XXX.

Nested Elements of `iql:XXX`:

Element	**Type**	**Required**
xxx	array of `iql:XXX` (**??**)	no

`iql:xxx` XXX.

2.10 Layer

- id
- name
- primary
- allMembers

Every layer selector either references an entire subgraph of the corpus' member-graph directly or constructs a partial selection with the "SCOPE" keyword. When using the first approach, an [item layer](../../icarus2-model-api/src/main/java/de/ims/icarus2/model/api/layer/ItemLayer.java) 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, the "SCOPE" keyword allows to create a scope that spans an exactly specified collection of layers. If a layer in the list of scope elements is appended the "*" symbol, the entire member-subgraph for this layer will be added to the scope.

If multiple layer selectors are defined, up to one can be declared as "primary" to represent the granularity of returned items for the search. In case no layer is explicitly marked as "primary", the one specified by the corpus or context will be used for that role. XXX

Attributes of `iql:XXX`:

Attribute	**Type**	**Required**	**Default**
xxx	xxx	x	

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

`iql:xxx` XXX.

2.11 Payload

- id
- name
- queryType
- bindings
- lanes
- constraint

XXX

Attributes of `iql:XXX`:

Attribute	Type	Required	Default
xxx	xxx	x	

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

`iql:name` Identifier to uniquely identify the payload within the entire query.

`iql:xxx` XXX.

Nested Elements of `iql:XXX`:

Element	Type	Required
xxx	array of `iql:XXX` (**??**)	no

`iql:xxx` XXX.

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 Section 2.12.1.

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).

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 3.7.8.
iql.string2bool.off	Deactivates the interpretation of strings as boolean values as described in Section 3.8.
iql.int2bool.off	Deactivates the interpretation of integers as boolean values as described in Section 3.8.
iql.float2bool.off	Deactivates the interpretation of floating point numbers as boolean values as described in Section 3.8.
iql.obj2bool.off	Deactivates the interpretation of arbitrary objects as boolean values as described in Section 3.8.
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 Section 3.7 to be 0-based.
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.

`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

- id
- dialect
- imports
- setup
- embeddeData
- streams

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. XXX

Attributes of `iql:XXX`:

Attribute	**Type**	**Required**	**Default**
xxx	xxx	x	

`iql:xxx` XXX.

Nested Elements of `iql:XXX`:

Element	**Type**	**Required**
xxx	array of `iql:XXX` (**??**)	no

`iql:xxx` XXX.

2.15 Reference

XXX

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

- resultTypes
- resultInstructions
- limit
- percent
- sortings

XXX

Attributes of `iql:XXX`:

Attribute	Type	Required	Default
xxx	xxx	x	

`iql:xxx` XXX.

Nested Elements of `iql:XXX`:

Element	Type	Required
xxx	array of `iql:XXX` (**??**)	no

`iql:xxx` XXX.

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 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`.

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

- id
- primary
- corpus
- layers
- scope
- rawPayload
- payload
- rawGrouping
- grouping
- rawResult
- result

XXX

Attributes of `iql:XXX`:

Attribute	Type	Required	Default
xxx	xxx	x	

`iql:xxx` XXX.

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

Nested Elements of `iql:XXX`:

Element	Type	Required
xxx	array of `iql:XXX` (**??**)	no

`iql:xxx` XXX.

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.

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, any while camel-cased versions are technically permitted, it is highly discouraged to use them:

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

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

3.3.1 String Literals

IQL uses double quotes 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.

Examples for valid string literals:

```
"string"
"123"
"some fancy number (123.456e-789) and emoji "
"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 or underscore (_) characters. Underscore characters may only appear inside the integer literal, never at the beginning or end (not counting the sign symbol).

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.

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.

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.

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`.
The following table provides a compact overview of the available identifiers and their capabilities/features:

Type	Prefix	Example	Scope	Fixed[1]	Final	Re-Assign
Reference	none	`max()`	global	X	X	
Variable	`@`	`@myVar`	limited	(X)		X
Member	`$`	`$token`	limited	X	(X)	

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 reassigned 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 and can take any of the following forms:

3.6.1 Primary Expressions

Any literal (boolean, string, integer or floating point) can serve as a primary expression.

3.6.2 Path Expressions

For navigating hierarchically structured object graphs or namespaces, expressions can take the form of paths:

$\langle path \rangle$ ::= $\langle expression \rangle$ '`.`' $\langle identifier \rangle$

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) and round brackets for the invocation with an optional argument list:

$\langle$*method*$\rangle$ ::= $\langle$*expression*$\rangle$ '`(`' $\langle$*arguments*$\rangle$? '`)`'

$\langle$*arguments*$\rangle$ ::= $\langle$*expression*$\rangle$ ('`,`' $\langle$*expression*$\rangle$)*

Examples:

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

3.7 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 integer values within `int` space. Positive values indicate the position beginning from the start of the array (with `1` being the first position for better human readability per default, see Section 2.12.1 for an option to switch to 0-based indices), 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 multidimensional arrays several index statements can be chained or even combined in a single comma-separated list.

$\langle$*array*$\rangle$ ::= $\langle$*expression*$\rangle$ '`[`' $\langle$*indices*$\rangle$ '`]`'

$\langle$*indices*$\rangle$ ::= $\langle$*expression*$\rangle$ ('`,`' $\langle$*expression*$\rangle$)*

Examples:

```
myArray[1]
myArray[-1]
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]“.

3.7.1 Annotation Access

The ICARUS2 framework 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:

$\langle annotation \rangle$ ::= $\langle expression \rangle$ ‘{’ $\langle keys \rangle$ ‘}’

$\langle keys \rangle$ ::= $\langle expression \rangle$ (‘,’ $\langle expression \rangle$)*

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 3.7.8) 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"}
firstSetValue(myItem{"parser1.head", "parser2.head"})
```

```
// extract values from multiple concurrent annotation layers
// and pick the first one present
```

3.7.2 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 in round brackets.

Examples:

```
(int) myValue
(long) 12345.678
(float) average(myVector)
```

3.7.3 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 is evaluated first -> result 14
(6 + 4) * 2 // addition is forced to be evaluated first -> result 20
```

3.7.4 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'). The basic form of a set predicate looks as follows:

$\langle$*set_predicate*$\rangle$ ::= $\langle$*expression*$\rangle$ 'IN' '{' $\langle$*items*$\rangle$ '}'

$\langle$*items*$\rangle$ ::= $\langle$*expression*$\rangle$ (',' $\langle$*expression*$\rangle$)*

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 expressions inside the curly brackets (the set definition). See the section about equality operators in Section 3.7.6. Note that methods or collections used inside the set definition are subject to the expansion rules described in Section 3.7.8. 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 wrpping Section 3.7.3 them and negating Section 3.7.5 the entire expression) with 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).

The complete syntax with all options looks as follows:

$\langle$*set_predicate*$\rangle$ ::= $\langle$*expression*$\rangle$ $\langle$*all*$\rangle$? $\langle$*not*$\rangle$? 'IN' '{' $\langle$*items*$\rangle$ '}'

$\langle$*all*$\rangle$ ::= 'ALL' | '$*$'

$\langle$*not*$\rangle$ ::= 'NOT' | '!'

$\langle$*items*$\rangle$ ::= $\langle$*expression*$\rangle$ (',' $\langle$*expression*$\rangle$)*

Examples:

```
someAnnotationValue IN {"NP","VP","-"}
someAnnotationValue NOT IN {"NN","DET"}
myValue IN {getLegalNames()}
fetchCharacterNamesInChapterOne() IN {getOrcishNames()}
```

3.7.5 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()
```

3.7.6 Binary Operation

Binary operations between two expressions take the following simple form:

⟨*binary_op*⟩ ::= ⟨*expression*⟩ ⟨*operator*⟩ ⟨*expression*⟩

Binary operators follow an explicit hierarchy, listed below 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

Basic Numerical Operations Basic numerical operations follow the standard mathematical rules for priorities. While the basic numerical types (`int`, `long`, `float` and `double`) 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 Section 3.7.2 to match the other. If value expansion Section 3.7.8 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.

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 Section 2.12.1 are planned to allow finer control over regex details.

Examples:

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

Equality Equality checks follow the same basic conditions as ordered comparisons Section 3.7.6, 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
```

3.7.7 Ternary Operation

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

$\langle \textit{ternary} \rangle$ ::= $\langle \textit{expression} \rangle$ '?' $\langle \textit{expression} \rangle$ ':' $\langle \textit{expression} \rangle$

The first expression must evaluate to a `boolean` value and determines which of the following two alternatives will be evaluated for the total 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"
```

3.7.8 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 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()]”.

3.8 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 the following evaluations as syntactic sugar:

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

3.9 Structural Constraints

The constraints section in IQL consists either of the sole “ALL“ keyword or of an optional bindings definition and the actual constraints themselves. A binding is a collection of member references Section 3.5 that get declared to belong to a certain type and/or part of the corpus. The ’DISTINCT’ keyword enforces that the bound member references in this binding do **not** match the same target. Depending on the localConstraint 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.

“‘ bindingsList := ’WITH’ binding (’AND’ binding)* binding := member (’,’ member)* ’AS’ ’DISTINCT’? qualifiedIdentifier “‘

Constraints are further divided into local constraints (signaled by the “WHERE“ keyword) 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 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.

3.9.1 Basic Constraints

Constraints can either be predicates, loop predicates, bracketed (with “(“ or “)“) constraints or boolean disjunctions (via “OR“) or conjunctions (with “AND“) of constraints.

Predicate Predicates are essentially expressions Section 3.6 that evaluate to boolean value. See the previous sections on constraints Section 3.8 for information on how non-boolean types are interpreted as boolean values and the switches **??** section for ways to influence this behavior.

Loop Predicate TODO

3.9.2 Flat Constraints

Flat constraints provide no extra helpers to declare structural properties of the query. They consist of arbitrary basic constraints Section 3.9.1 and typically make global constraints redundant.

3.9.3 Tree Constraints

TODO

3.9.4 Graph Constraints

content

3.9.5 Global Constraints

Global constraints con be any basic constraint Section 3.9.1.

3.10 Result Processing

There be dragons.

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

3.11 Query Payload

The innermost part of every IQL query is the collection of constraints used to actually extract and match data from the corpus. It is composed in a keyword-style syntax to differentiate between the different *dialects* or constraint modes.