Basic-Grammar

Namespaces

One important feature of IQL is the integration of Java types. The DSL can refer to Java types from Java standard libraries, Odysseus Bundles or other external libraries. Namespaces are useful in this context to refer Java types by a simple name and not by a fully qualified name. The static-keyword allows the static methods of a java class to be referenced without qualifying the class name.

Grammar

```
ID                      ::= ("a.."z"|"A.."Z"|"_")("a.."z"|"A.."Z"|"_"|"0.."9")*.
QName                   ::= ID ("::" ID)*
Namespace               ::= "use" ("static")? QualifiedNameWithWildcard ";".
QualifiedNameWildcard   ::= QualifiedName ("::*")?
```

Example

```
use java::util::*;
use de::uniol::inf::is::odysseus::core::ISubscription;
use com::google::common::collect::ImmutableMap;
use static java::lang::System;
```

Class, Interface

Besides the integration of Java types it is also possible to create your own classes and interfaces.

Grammar

```
Class                   ::= "class" ID ("extends" QName)? ("implements" QName ("," QName)*)? "{" (Attribute |
Method)* "}".
Interface               ::= "interface" ID ("extends" QName ("," QName)*)? "{" (MethodDeclaration)* "}".
Attribute               ::= VariableDeclaration (VariableInit)? ";".
VariableDeclaration     ::= Type ID.
Type                    ::= (PrimitiveType | QName) ("[" "]")*.
PrimitiveType           ::= "boolean" | "byte" | "char" | "short" | "int" | "float" | "long" | "double".
Method                  ::= ("override")? MethodDeclaration StatementBlock.
MethodDeclaration       ::= ID (MethodParameters)? (":" Type)?.
MethodParameters        ::= "(" (VariableDeclaration ("," VariableDeclaration)*)? ")".
```

Example Class

```
class Point implements IPoint{
    int x;
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    override getX() : int{
        return this.x;
    }

    override getY() : int{
        return this.y;
    }
}
```

Example Interface

```
interface IPoint {
        getX() : int;
    getY() : int;
}
```

Statements

Methods, constructors and other blocks are sequences of statements. They can be divided into atomic and compound statements. Atomic statements are not made up of other statements and end with a semicolon (e.g. VariableStatement). Compound statements (also called control structures) contain other statements (e.g. WhileStatement).

Grammar

```
Statement             ::= ( StatementBlock | IfStatement | SwitchStatement | WhileStatement | DoWhileStatement
| ForStatement | ForEachStatement
                          | VariableStatement | ConstructorStatement | ExpressionStatement | BreakStatement |
ContinueStatement | ReturnStatement ).
StatementBlock        ::= "{" (Statement)* "}".
IfStatement           ::= "if" "(" Expression ")" Statement ("else" Statement)?.
SwitchStatement       ::= "switch" "(" Expression ")" "{" ("case" Expression ":" (Statement)*)* ("default" ":"
(Statement)*)? "}"
WhileStatement        ::= "while" "(" Expression ")" Statement.
DoWhileStatement      ::= "do" Statement "while" "(" Expression ")" ";".
ForStatement          ::= "for" "(" VariableDeclaration ";" Expression ";" Expression ")" Statement.
ForEachStatement      ::= "for" "(" VariableDeclaration ":" Expression ")" Statement.
VariableStatement     ::= VariableDeclaration VariableInit ";".
ConstructorStatement  ::= ("super" | "this") "(" (ArgsList)? ")" ";".
ExpressionStatement   ::= Expression ";".
BreakStatement        ::= "break" ";".
ContinueStatement     ::= "continue" ";".
ReturnStatement       ::= "return" (Expression)? ";".
```

Example if-Statement

```
int i = 2;
if (i>2) {

}
```

Example switch-Statement

```
int i = 2;
switch(i) {
        case 2 :
                return true;
        case 4 :
                return true;
        default :
                return false;
}
```

Example while-Statement

```
int i = 2;
while (i>2) {

}

do {

} while (i>2);
```

Example for-Statement

```
for (int j = 0; j<10; j++) {

}

List list = [1,2,3];
for (Object element : list) {

}
```

A new object can be created by calling a constructor. It is also possible to assign values to attributes in this context. This makes it easier to configure operators when building a query with QDL because it is not necessary to create a new statement for each parameter.

Example

```
// Constructor Call
Point p1 = new Point(2, 5);
Point p2(2,5);

// Assigning values to attributes
Point p3 = new Point{x = 2, y = 5};
Point p4{x = 2, y = 5};

Select sel{predicate="bid.price <=200"};
Project proj(sel){attributes=[bid.PRICE]};
```

Expressions

An expression is used as part of other expressions or statements and produces a value as result.

Grammar

```
Expression           ::= ( AssignmentExpression | LogicalExpression | EqualityExpression | ArithmeticExpression
| UnaryExpression | CastExpression
                                                    | PrefixExpression | PostfixExpression |
InstanceOfExpression | CreateExpression | AttributeExpression | MethodExpression
                                                    | ArrayExpression | "this" | "super" | ID |
LiteralExpression | ("(" Expression ")").
AssignmentExpression ::= Expression ("=" | "+=" | "-=" | "*=" | "/=" | "%=") Expression.
LogicalExpression    ::= Expression ("&&" | "||") Expression.
EqualityExpression   ::= Expression ("==" | "!=" | ">" | ">=" | "<" | "<=") Expression.
ArithmeticExpression ::= Expression ("+" | "-" | "*" | "/"| "%") Expression.
UnaryExpression      ::= ("+" | "-" | "!") Expression.
CastExpression       ::= "(" Type ")" Expression.
PrefixExpression     ::= ("++" | "--") Expression.
PostfixExpression    ::= Expression ("++" | "--").
InstanceOfExpression ::= Expression "instanceof" Type.
NewExpression        ::= "new" QName (("[" "]") | (("(" (ArgsList)? ")")? ("{" ArgsMap "}")?)).
MemberCallExpression ::= (Expression ".")? ID ("(" (ArgsList)? ")")?.
ArrayExpression      ::= Expression ("[" ArgsList "]")+.
LiteralExpression    ::= Integer | Double | Boolean | Char | String | Range | List | Map | Null
VariableInit         ::= ((("(" (ArgsList)? ")")? ("{" ArgsMap "}")?) | ("=" Expression)).
ArgsList             ::= Expression ("," Expression)*.
ArgsMap              ::= ArgsMapKeyValue ("," ArgsMapKeyValue)*.
ArgsMapKeyValue      ::= ID "=" Expression
```

If a data type has a getter- or setter-method there will always be a corresponding attribute to access. So it is often possible to use assignment expressions instead of method calls.

Example

```
Tuple tuple = createTuple();

ITimeInterval time = tuple.metadata;
tuple.metadata = time;

//is equivalent to
ITimeInterval time1 = tuple.getMetadata();
tuple.setMetadata(time1);
```

Metadata

Grammar

```
Metadata              ::= ID "=" MetadataValue.
MetadataValue         ::= MetadataValueSingle | MetadataValueList | MetadataValueMap.
MetadataValueSingle   ::= INTEGER | Double | CHAR | STRING | BOOLEAN | QName.
MetadataValueList     ::= "[" (MetadataValue ("," MetadataValue)*)? "]".
MetadataValueMap      ::= "[" (MetadataValueMapEntry ("," MetadataValueMapEntry)*)? "]".
MetadataValueMapEntry ::= MetadataValue ":" MetadataValue.
```

See operators in ODL or queries in QDL for examples.

Literals

Literals are sequence of characters that represent constant values to be stored in variables.

Grammar

```
Integer                ::= (0..9)+.
Double                 ::= (0..9)* "." (0..9)+.
Boolean                ::= ("true" | "false").
Char                   ::= "'" Unicode-Character "'".
String                 ::= '"' (Unicode-Character)* '"'.
Range                  ::= (0..9)+ ".." (0..9)+.
List                   ::= "[" (Expression ("," Expression)*)? "]".
Map                    ::= "[" (MapKeyValue ("," MapKeyValue)*)? "]".
MapKeyValue            ::= Expression ":" Expression.
Null                   ::= "null".
```

Example

```
Range r = 1..10;
List l  = [1, 2, 3];
Map m   = ["key1":1, "key2":2];
Tuple t = [5, true, 3.4];
IPunctuation p = 100;
```

Java-Code

IQL-code is always translated into Java-code. Sometimes it might be useful to write Java-code directly instead of IQL-code.

Grammar

```
GPLCode               ::= "$*" GPL Code "*$"
```

Example

```
$*
public static void main(String[] args) {

}
*$
```

Comments

IQL supports single- and multi-line comments.

Grammar

```
Comment                ::= SingeLineComment | MultiLineComment.
SingeLineComment       ::= "//" Text (("\r")? "\n")?.
MultiLineComment       ::= "/*" Text "*/".
```

Example

```
// This is a single line comment

/*
 * This is a multi line comment
 */
```