Processing IQL Queries and Migrating Data in the AutoMed toolkit

Edgar Jasper, Alex Poulovassilis, Lucas Zamboulis
School of Computer Science and Information Systems, Birkbeck

July 25, 2003

1 Introduction

This technical report describes how IQL queries are processed in the AutoMed heterogeneous data integration system, and also how data migration can be supported.

We start with an outline of the IQL language in Section 2. We then consider in Section 3 an abstract representation of this textual IQL and describe the `ASG` class that implements this abstract representation.

Given a query expressed using the constructs of one schema, it may be necessary to reformulate this query so that it is expressed over the constructs of other target schemas that are linked to it by AutoMed transformation pathways. For example, a query expressed on a global virtual schema needs to be reformulated into a query over the local data sources in order to be evaluated. Software that does this is described in Section 4.

Next, we consider how IQL queries are evaluated. The implementation of software to evaluate IQL queries that contain various built-in functions is described in Section 5. Retrieving data from data sources is described in Section 6. In Section 7 some example code is given that shows how an IQL query would be processed from 'top to bottom'. Some examples of IQL queries are also given. Finally, Section 8 discusses how data migration from one or more data sources to a target data source can be supported[1].

2 Outline of IQL

The AutoMed Intermediate Query Language (IQL) [3] is a functional language. Its purpose is to provide a common query language that queries written in various high level query languages (such as SQL) can be translated into and out of. For the complete syntax of IQL see Appendix A.

IQL includes constants which may be strings, booleans, and real and integer numbers. It is envisaged that dates and times will be incorporated soon. Variables and identifiers are also included. These may represent built-in functions. Appendix B lists the current set of built-in functions, but this set is easily extensible.

[1] This report is largely based on an earlier report by Edgar Jasper: "Processing IQL Queries", 4th June 2003.

IQL also supports tuples (eg. {1,2,3}) and lists (eg. [1,2,3]). Anonymous functions may be defined using lambda abstractions. For example:

```
lambda {x,y,z} ((+) ((+) x y) z)
```

This function adds the three components of its argument triple together. More on lambda abstractions and functional languages can be found in [2]. We note that most of IQL's built-in functions are only supported in prefix form, exceptions being the infix operators `++` and `--`, denoting list append and monus (bag difference), respectively.

IQL also supports list comprehensions. For example:

```
[{x,y} | x<-[1,2,3]; y<-['a','b']; (>) x 1]
```

This undertakes a Cartesian product of the two lists followed by a selection, and the result is

```
[{2,'a'},{2,'b'},{3,'a'},{3,'b'}]
```

Comprehensions do not actually add expressiveness to IQL but are 'syntactic sugar' allowing queries that are easy to write and read. Also, there already exists work on optimising comprehensions so it will be possible to draw on this for the task of query optimisation (this is ongoing work at present). Furthermore, it is expected that it will prove easier to translate into and out of comprehensions when it comes to translating between IQL and various high level query languages (this is also ongoing work).

Comprehensions comprise a head expression followed by a list of qualifiers which may be either filters or generators. Generators iterate a pattern over a list-valued expression, where a pattern is either a variable or a tuple of patterns. Filters are boolean-valued expressions that act as filters on the variable instantiations generated by the generators of the comprehension.

Comprehensions may be translated into simple function applications and lambda abstractions as follows:

$$
\begin{array}{ll}
[e|p \leftarrow s; Q] & \Longrightarrow flatmap\ (lambda\ p\ [e|Q])\ s \\
[e|e'; Q] & \Longrightarrow if\ e'\ [e|Q]\ [] \\
[e|] & \Longrightarrow [e]
\end{array}
$$

The if function takes three arguments and returns the second if the first is true and the third if the first is false. The $flatmap$ function behaves as follows:

$$
\begin{array}{lll}
flatmap\ f\ [] & = & [] \\
flatmap\ f\ (Cons\ x\ xs) & = & (f\ x)\ +\!+\ (flatmap\ f\ xs)
\end{array}
$$

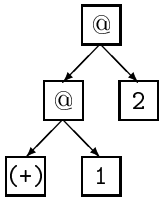
where $Cons\ x\ xs$ represents a list of which x is the head and xs is the rest of the list.

Also supported by IQL are let expressions (ie. `let v = e1 in e2`) and some special constants including `Void` and `Any`. Like comprehensions, let expressions add no expressivity to the language but help to make it more readable.

3 Abstract representation of IQL

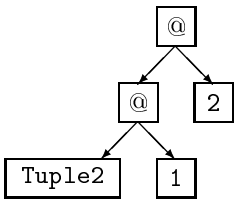
The string representations of IQL queries must be parsed to create an abstract syntax tree representation. A binary tree representation is used for the latter. These trees represent repeated function applications. All non-leaf cells are either

apply cells (@) or lambda cells (λ). An apply cell represents the left child being applied to the right child. For example:

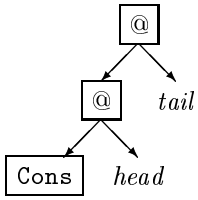


This tree represents the query `(+) 1 2` which evaluates to 3. All functions are considered to be of arity 1 at the fundamental level — though this is not necessarily how they are implemented. Thus, `(+) 1` returns a function which adds 1 to its argument. The whole query could be written as `((+) 1) 2`. So we see how the result of an apply cell is considered to be the result of its left child applied to its right child.

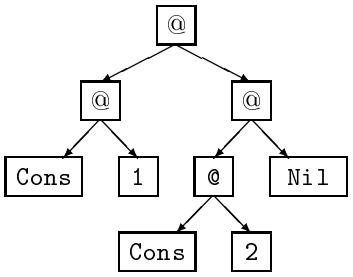
Leaf cells may be constants such as `1`, `2.3`, `'Edgar'` or `True`. They may also be variables or function names like `(+)`, `flatmap` or `x`. They may also be *constructors*. These are special functions used to represent structured values such as tuples and lists. Tuples are simple to represent. For example, this represents the pair `{1,2}`:



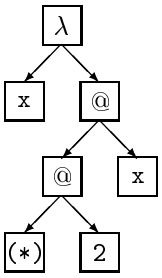
Lists are slightly more complicated. Two constructors are used. The `Nil` constructor represents an empty list. Non-empty lists are represented using the `Cons` constructor:



In the above diagram which illustrates how the `Cons` constructor works, *head* is the place where the first element of the list would be put. In place of *tail* would be another list representing the original list without the first element. The tree representation of a list is thus a recursive structure built using the `Cons` and `Nil` constructors. The following diagram represents the list `[1,2]`.



The following diagram shows the abstract syntax tree for the lambda abstraction `lambda x ((*) 2 x)`:



The above abstract representation for IQL queries is implemented as the `ASG` class. ASG stands for abstract syntax graph. The purpose of the `ASG` class is to hide from the user the implementation details of the cells in the abstract syntax graph and to provide useful methods to work with abstract syntax graphs[2]. It is intended that `ASG` objects be the standard way to store IQL queries and that the various elements of the AutoMed query processing software should act upon `ASG` representations of queries.

Details of the constructors and methods of the `ASG` class can be found in Appendix C. The following code sample shows the creation of an `ASG` object:

```
String query = "<<person,pname>>";
ASG q = new ASG(query);
```

Note the use of an AutoMed scheme in the above query, delimited by double chevrons `<<` ... `>>`. Evaluating any scheme in IQL results in a list of values of the appropriate type. For example, if `person` is a relational table with a single key attribute `pid` and a non-key attribute `pname`, then the above query would return a list of pairs of the form {`i`,`n`} where `i` is a value of `pid` and `n` the corresponding value of `pname`.

4 Query Reformulation

Query reformulation is performed by `QueryReformulator` objects. In outline the process is as follows. A `QueryReformulator` object is created by passing

[2]Note that we are dealing with an abstract syntax graph rather than an abstract syntax tree. Although parsing an IQL query results in a tree, it is possible that when the tree is processed some substructures are referenced more that once. Rather than duplicate these substructures to retain a tree structure, the original abstract syntax tree may become an acyclic graph. Hence the `ASG` class is for abstract syntax graphs.

a source schema and one or more target schemas to the constructor. The constructor generates and stores views of the source schema constructs in terms of the constructs of the target schemas. Pathways from source to targets are obtained from the Schemas and Transformations Repository (STR) [1].

For some part of the start of their length these transformation pathways may be the same as each other. Thus the set of transformation pathways may be viewed as a tree with the source schema at the root and the target schemas at the leaves.

View definitions for each source schema construct are derived by traversing through the tree in the following manner. Initially, each source schema construct has as its view definition merely a query consisting of that construct. Each node in the tree is visited in a top-down fashion. Each node corresponds to a transformation in one or more transformation pathways.

The only transformations that are significant are those that delete, contract or rename an extensional construct. These transformations are significant because the current view definitions may query constructs that no longer exist after such a transformation. Each of these types of transformation is handled as follows if it is encountered during the tree traversal:

- delete: This has an associated query which shows how to reconstruct the extent of the construct being deleted. Any occurrence of the deleted construct within the current view definitions is replaced by this query.
- contract: If it has an associated reconstructing query, it is treated in the same way as a delete. Otherwise, any occurrence of the contracted construct within the current view definitions is replaced by the `Void` constructor.
- rename: All references to the old construct in the current view definitions are replaced by references to the new construct.

As this may not be a simple transformation pathway but a tree, at some points it may branch. When this happens, constructs of the parent schema are semantically identical to constructs that have the same scheme within the child schemas. The possibility of using all paths is retained within the view definitions by replacing each construct of the parent schema by a disjunction (`OR`) of the corresponding constructs in the child schemas. A child construct may later be made `Void` by a contract transformation further down the tree.

The tree is traversed top to bottom until all nodes are visited. The transformations are processed as above and the resulting set of view definitions are the appropriate view definitions for the source schema constructs over the target schemas. The `QueryReformulator` object stores these view definitions and can be used for reformulated queries accordingly. The following code sample shows the construction and use of a `QueryReformulator` object:

```
ASG q = new ASG(...);
Schema src = Schema.getSchema("GLOBAL");
Schema[] tgts = new Schema[2];
tgts[0] = Schema.getSchema("LOCAL1");
tgts[1] = Schema.getSchema("LOCAL2");
QueryReformulator qr = new QueryReformulator(src,tgts);
qr.reformulate(q);
```

The `QueryReformulator` class has a couple of other methods that are covered in Appendix D.

5 Query Evaluation

5.1 Background

Evaluating a query expressed in a functional language consists of performing reductions on reducible expressions until no more reductions can be performed. It is then said to be in *normal form*. There may be more than one reducible expression within the query at any one time and thus a choice of reductions. Irrespective of the order in which reductions are made, the same normal form will result provided the evaluation terminates. However, choices made about which reduction is performed may impact on the efficiency, or indeed the termination, of the evaluation.

IQL's query evaluation software always reduces the leftmost reducible expression that it can — this is known as normal-order reduction and has the best possible termination behaviour. It also has the effect that, in many cases, function arguments are not evaluated unless they are actually needed for the evaluation of the function. (In practice it is not quite as straightforward as this. An IQL query is represented internally as an acyclic graph. Thus, several identical sub-expressions of it may, in effect, be evaluated at the same time.)

5.2 The `Evaluator` class

Evaluation is performed by `Evaluator` objects. There are methods for full evaluation — reduction to normal form — and for reduction to so-called *weak head normal form*. (Information on weak head normal form can be found in [2].) These methods are the public methods of `Evaluator` objects.

The reductions performed by `Evaluator` objects are of two types — reduction of lambda expressions and reduction of built-in functions. Also, various higher level constructs — let expressions and comprehensions for example — are based upon lambda expressions. These higher level constructs are evaluated by being translated into their equivalent lambda expression which the evaluator must be able to evaluate.

The built-in functions of IQL fall into two classes. There are ordinary built-in functions, such as `sort`, `(+)` and the like, which are part of the IQL language the user is intended to use. There are also special built-in functions, such as `Comp` and `Let`, which are not intended for direct use but exist to represent internally constructs of IQL such as comprehensions and let expressions. These special functions provide a view of these constructs that turns them into simple function applications like the rest of IQL.

When an `Evaluator` object is created, the constructor is passed a table of built-in functions that is used when evaluating queries.

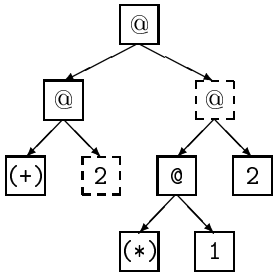
5.3 Implementation of built-in functions

The `FunctionTable` object that is passed to the `Evaluator` constructor is used by the evaluator to obtain the built-in functions it needs to evaluate `ASG`s. It has a `getFunction()` method that given a string representing the name of the

function — the string that occurs in the query — returns a `BuiltInFunction` object.

`BuiltInFunction` objects are instances of classes that implement the `BuiltInFunction` interface. This interface has 2 methods. A `getArity()` method returns the arity of the function. The evaluator needs to know the arity of a built-in function — how many arguments it takes. Though we may consider all functions to be of arity one and to be capable of returning other functions as results, in practice built-in functions are implemented as taking their full complement of arguments.

The other method of `BuiltInFunction` is `perform(Cell[] args, Evaluator e)`. To understand this method consider the query '`(+) 2 ((*) 1 2 )`'. The following diagram shows the tree representation of this query that the evaluator will work on.



In evaluating this query the `Evaluator` will obtain a `BuiltInFunction` for the `(+)` function from its `FunctionTable`. It will then call the `perform()` method of this `BuiltInFunction` passing it references to the two cells marked with dashed boxes as the `args` argument. However, the `perform()` method cannot simply work out the result because one of the cells it is passed represents not a number but a query — `(*) 1 2` — in need of evaluation. Because the `perform()` method will frequently need to evaluate some of the `args` it is passed before it can return a result it must also be provided with a reference to the `Evaluator` it should use to accomplish this task. This is the purpose of the argument `e`. So the `Evaluator` calls the `perform()` method of a `BuiltInFunction` passing it the function arguments and a reference to itself. The `BuiltInFunction` may then call back out to the `Evaluator` to evaluate sub-queries it needs evaluating in order to provide a result. The decision to evaluate these sub-queries is left to the `BuiltInFunction` rather than being done by the `Evaluator` before the `perform()` method is called because the `Evaluator` doesn't know whether it will be necessary to evaluate them or not. For example, consider a query of the form '`and False X`' where `X` represents a, possibly enormous, sub-query. The `Evaluator` will not know it but the `BuiltInFunction` need not evaluate `X` — it can return `False` without ever looking at `X`.

Built-in functions are implemented by classes that implement the `BuiltInFunction` interface. The `Evaluator` obtains the appropriate `BuiltInFunction` from a `FunctionTable` passed to its constructor when it is created. `FunctionTable` has 2 methods: `addFunction()` adds a string and corresponding `BuiltInFunction` to the table, and `getFunction()` returns a `BuiltInFunction` object given a string. See Appendix F for more details.

A number of standard functions have been implemented. The class

`StandardFunctionTable` extends `FunctionTable` to add various standard functions and corresponding strings automatically.

The following code gives an example of how to use the evaluator software to evaluate a query.

```
ASG q = new ASG(...);
FunctionTable ft = new StandardFunctionTable();
Evaluator e = new Evaluator(ft);
e.evaluate(q);
```

6 Retrieving data

To retrieve data from data sources, the references to schemes in an ASG have to be replaced by actual data. AutoMed's data source wrappers can be used to retrieve this data. The `executeIQL` method provided by an AutoMed wrapper takes an IQL query `ASG`, translates it into the data source's query language, submits this query for evaluation at the data source, and returns the result in the form of an `ASG`. It is possible to generate an appropriate wrapper object for a given schema by using information stored about that schema in the STR.

A simple architecture for using these wrappers from the IQL evaluator has been developed. It will not be the final solution but has been implemented to provide a basic demonstration of query processing from top to bottom.

A `FragmentProcessor` object has 1 method — `process(ASG)` — that traverses the `ASG` representing a reformulated IQL query, replacing references to schemes with a call to a 2-ary function `$wrapper`. Its first argument is the appropriate wrapper object, as obtained from the STR using the schema associated with the scheme[3]. The second argument is the `ASG` — in this case merely representing the scheme — to be evaluated at the data source. As the `Evaluator` object evaluates, it reduces the `$wrapper` function (an appropriate `BuiltInFunction` is included in `StandardFunctionTable`) when it reaches that part of the overall query graph, and the `ASG` is submitted to the wrapper's `executeIQL` method at that time.

The `FragmentProcessor` does not submit queries to wrappers itself as this will probably be an expensive operation which should only be done if needed. Only as evaluation is performed can it be established whether or not sending a query to a wrapper is necessary.

The following code shows the use of the `FragmentProcessor` class.

```
ASG q = new ASG(...);
FragmentProcessor fp = new FragmentProcessor();
fp.process(q);
```

7 Example Code for Processing a Query

The following code shows the 'top to bottom' processing of an IQL query:

```
String query = "<<person,pname>>";
ASG q = new ASG(query);
```

[3] This information will have been added to the ASG by the reformulator constructor.

```
Schema src = Schema.getSchema("GLOBAL");
Schema[] tgts = new Schema[2];
tgts[0] = Schema.getSchema("LOCAL1");
tgts[1] = Schema.getSchema("LOCAL2");
QueryReformulator qr = new QueryReformulator(src,tgts);
qr.reformulate(q);

FragmentProcessor fp = new FragmentProcessor();
fp.process(q);

FunctionTable ft = new StandardFunctionTable();
Evaluator e = new Evaluator(ft);
e.evaluate(q);
System.out.println(q);
```

7.1 Some Example IQL Queries

We now illustrate IQL by means of some example IQL queries. These are posed on the following global schema, expressed in the AutoMed Relational data model:

```
schema:uni_zb

table:<<dept>>
field:<<dept,dname>>
primary_key:<<dept_pk,dept,<<dept,dname>>>>

table:<<person>>
field:<<person,id>>
field:<<person,name>>
field:<<person,dname>>
primary_key:<<person_pk,person,<<person,id>>>>

table:<<male>>
field:<<male,id>>
primary_key:<<male_pk,male,<<male,id>>>>

table:<<female>>
field:<<female,id>>
primary_key:<<female_pk,female,<<female,id>>>>
```

The above global schema is a virtual integration of the following two relational database schemas, expressed in the Data-Source-Oriented Relational model:

```
schema:uni_s1_source

tuple_sql_table:<<dept,dname>>
tuple_sql_primary_key:<<dept_pk,<<dept,dname>>,dname>>
```

```
tuple_sql_table:<<staff,id,name,sex,dname>>
tuple_sql_primary_key:<<staff_pk,<<staff,id,name,sex,dname>>,id>>

schema:uni_s2_source

tuple_sql_table:<<female,id>>
tuple_sql_primary_key:<<female_pk,<<female,id>>,id>>

tuple_sql_table:<<male,id>>
tuple_sql_primary_key:<<male_pk,<<male,id>>,id>>

tuple_sql_table:<<person,id,name,dname>>
tuple_sql_primary_key:<<person_pk,<<person,id,name,dname>>,id>>
```

The equivalent AutoMed Relational model representation of these two schemas is:

```
schema:uni_s1

table:<<dept>>
field:<<dept,dname>>
primary_key:<<dept_pk,dept,<<dept,dname>>>>

table:<<staff>>
field:<<staff,id>>
field:<<staff,name>>
field:<<staff,sex>>
field:<<staff,dname>>
primary_key:<<staff_pk,staff,<<staff,id>>>>

schema:uni_s2

table:<<person>>
field:<<person,id>>
field:<<person,name>>
field:<<person,dname>>
primary_key:<<person_pk,person,<<person,id>>>>

table:<<female>>
field:<<female,id>>
primary_key:<<female_pk,female,<<female,id>>>>

table:<<male>>
field:<<male,id>>
primary_key:<<male_pk,male,<<male,id>>>>
```

The transformation pathways from `uni_s1_source` and `uni_s2_source` to `uni_s1` and `uni_s2` are generated automatically by AutoMed's SQL wrapper. We refer the reader to the examples folder of the AutoMed release for the code that generates the transformation pathways from `uni_s1` and `uni_s2` to `uni_zb`.

The following queries over the global schema `uni_zb` return, respectively, the set of all people, the set of females, the set of males, the set of departments, the names of all people, and the names of all departments:

```
<<person>>
<<male>>
<<female>>
<<dept>>
[n | {p,n}<-<<person,name>>]
[n | {d,n}<-<<dept,dname>>]
```

Notice that, due to the way the transformation pathways have been defined in this example, data from both data sources is returned by these queries. To remove duplicates, we can use the `distinct` built-in function (it is of course possible to change the transformation pathways to retain only one copy of common data within the global schema):

```
distinct <<person>>
distinct <<male>>
distinct <<female>>
distinct <<dept>>
[n | {p,n}<- distinct <<person,name>>]
[n | {d,n}<- distinct <<dept,dname>>]
```

Here are two, more complex, queries that return the names of females:

```
[n | {f}<- distinct <<female>>; {f1,n} <- distinct <<person,name>>;
     (=) f f1]
```

and the names of females working in the 'CS-BBK' department:

```
[n | {f}<- distinct <<female>>; {f1,n} <- distinct <<person,name>>;
     (=) f f1; {f2,d} <- distinct <<person,dname>>; (=) f2 f1;
     (=) d 'CS-BBK']
```

8 Data Migration

The query processor and wrappers can also be used to migrate data from one or more source schemas S_1, ..., S_n to a target schema T (expressed in the target data source's data-source-oriented model), assuming that the target schema is linked to the source schema(s) by transformation pathways in the STR. In particular, T can be populated from S_1, ..., S_n as follows:

(a) create a `QueryReformulator` object, to generate views of each construct $c_1, \ldots, c_m$ of T as a view over the constructs of S_1, ..., S_n:
`QueryReformulator qr = new QueryReformulator(tgt,srcs);`

(b) create m IQL queries, each consisting of one of the schemes $c_1, \ldots, c_m$ of T; reformulate them and fragment-process them;

(c) evaluate each of the resulting queries, resulting in each case in an ASG representing a list of values, vs, to be inserted into that construct of T;

(d) create an IQL query *sub vs c* and call the wrapper's `insertIQL` method with this query; this will have the effect of generating an insertion request to the data source to insert values *vs* into its construct *c*.

References

[1] M. Boyd, P. McBrien, and N. Tong. The AutoMed schema integration repository. In *Proc. BNCOD'02, LNCS 2405*, pages 42–45, 2002.

[2] S. L. Peyton Jones *et al. Implementing Functional Programming Languages.* Prenctice Hall, 1992.

[3] A. Poulovassilis. The AutoMed Intermediate Query Language. Technical report, Automed Project, 2001.

A IQL Syntax

```
query ::=          expr
                 | Let VarToken Equal query In query
                 | query Append query
                 | query Difference query
                 | query expr

expr ::=           NumToken
                 | StrToken
                 | VarToken
                 | Any
                 | BoolToken
                 | Void
                 | scheme
                 | VarToken Colon scheme
                 | LSB query Bar quals RSB
                 | LSB RSB
                 | LSB seq RSB
                 | LCB seq RCB
                 | LRB query RRB
                 | Lambda pattern expr

seq ::=            seq Comma query
                 | query

quals ::=          qual SemiColon quals
                 | qual

qual ::=           query
                 | pattern LArrow query

pattern ::=        VarToken
                 | LCB pattern_seq RCB
```

```
pattern_seq ::=      pattern_seq Comma pattern
                   | pattern

scheme ::=           LDAB scheme_seq RDAB

scheme_seq ::=       scheme_element
                   | scheme_seq Comma scheme_element

scheme_element ::=   VarToken
                   | StrToken
                   | NumToken
                   | scheme

BoolToken =          "True" | "False"

StrToken =           \'[ ^ \']*\'

NumToken =           [0-9]+|([0-9]+)(".")([0-9]+)

VarToken =           "(+)" | "(-)" | "(*)" | "(/)" | "(&)" | "(#)"
                   | "(=)" | "(!=)" | "(<)" | "(>)" | "(<=)" | "(>=)"
                   | [A-Za-z_][A-Za-z0-9_$]*

Let = "let"
In = "in"
Equal = "="
Append = "++"
Difference = "--"
SemiColon = ";"
LArrow = "<-"
Comma = ","
Bar = "|"
LSB = "["
RSB = "]"
LRB = "("
RRB = ")"
LCB = "{"
RCB = "}"
Any = "Any"
Lambda = "lambda"
LDAB = "<<"
RDAB = ">>"
Colon = ":"
Void = "Void"
```

B Built-In Functions

B.1 Unary Operators

```
not      /* (not e) returns the negation of e */
count    /* (count xs) returns the length of the list xs */
sort     /* (sort xs) sorts the list xs */
distinct /* (distinct xs) removes duplicates from the list xs */
group    /* (group xs) groups a list of pairs xs on their first component */
         /* and returns a list of pairs whose second components are lists */
max      /* (max xs) returns the maximum of a list of numbers xs */
min      /* (min xs) returns the minimum of xs */
sum      /* (sum xs) returns the sum of xs  */
avg      /* (avg xs) returns the average of xs */
```

B.2 Binary Infix Operators

```
++       /* (xs ++ ys) appends the two lists xs and ys */
--       /* (xs -- ys) returns the monus (bag difference) of xs and ys */
```

B.3 Binary Prefix Operators

```
(+) (-) (*) (/)
(=) (!=) (>) (<) (>=) (<=)
and or

setUnion  /* (setUnion xs ys) appends xs and ys and then removes duplicates */
sub       /* (sub xs ys) returns whether xs is a sub-bag of ys */
intersect /* (intersect xs ys) returns the bag interection of xs and ys */
member    /* (member xs x) returns whether x is a member of xs */

map       /* (map f xs) applies a function f to each member of xs */
flatmap   /* (flatmap f xs) applies a list-valued function f to each member */
          /* of xs and then appends the resulting lists */
gc        /* (gc agFun xs) first groups the list of pairs xs on their */
          /* first component, and then applies the aggregation function */
          /* agFun to the second component- a list -of the resulting pairs */
```

B.4 3-ary Operator

```
if        /* if e1 e2 e3 returns e2 if e1 is True and e3 otherwise */
```

C ASG Constructors and Methods

C.1 Constructors

`public ASG(String s)`
where `s` is a text IQL query.

`public ASG(File f)`
where `f` is a text file containing the text of an IQL query.

`public ASG(Cell n)`
where `n` is the root cell of a graph. This graph will be copied and packaged as an `ASG`.

C.2 Static Methods

`public static ASG toASGList(ArrayList list)`
generates an `ASG` list from an `ArrayList`. The `ArrayList` should be an array of `Cells`.

`public static ASG toASGTuple(ArrayList list)`
generates an `ASG` tuple from an `ArrayList`. The `ArrayList` should be an array of `Cells`.

`public static ArrayList fromASGList(ASG g)`
if `g` represents a list this returns an `ArrayList` of its `Cells`.

`public static ArrayList fromASGList(Cell n)`
analogous to the above where `n` is the root of the graph.

`public static ArrayList fromASGTuple(ASG g)`
if `g` represents a tuple this returns an `ArrayList` of its `Cells`.

`public static ArrayList fromASGTuple(Cell n)`
analogous to the above where `n` is the root of the graph.

`public static String fromASG(ASG g)`
returns the text IQL of `g`.

`public static String fromASG(Cell n)`
returns the text IQL of the abstract syntax graph of which `n` is the root.

`public static String formattedString(ASG g)`
if `g` is an `ASG` list the string returned has each element separated by carriage returns for more legible output.

`public static String formattedString(Cell n)`
analogous to the above where `n` is the root of the graph.

`public static ArrayList listOfSchemes(ASG g)`
returns a list of the schemes used in the `ASG` as `String` objects.

`public static ASG emptyQuery()`
returns the `ASG` representation of the query `Nil`.

`public static ASG voidQuery()`
returns the `ASG` representation of the query `Void`.

`public static replaceAllSchemaRefs(ASG g, Schema s)`
replaces all references to `Schemas` within `g` by references to `s`.

`public static replaceNullSchemaRefs(ASG g, Schema s)`
replaces all references to a null `Schema` object in `g` with references to `s`.

C.3 Non-Static Methods

`public Cell root()`
returns the root cell of the `ASG`.

`public String toString()`
returns the text IQL string of the `ASG`.

`public String formattedString()`
given that the `ASG` represents a list the string returned has each element separated by carriage returns for more legible output.

`public ArrayList listOfSchemes()`
returns a list of the schemes used in the `ASG` as `String` objects.

`public ASG copyOfASG()`
returns a copy of the `ASG`.

D `QueryReformulator` Constructors and Methods

`public QueryReformulator(Schema source, Schema[] targets)`
constructs a `QueryReformulator` object containing view definitions for the constructs of `source` in terms of the constructs of `targets` assuming the appropriate transformation pathways exist.

`public void reformulate(ASG g)`
reformulates the query represented by the `ASG g` according to the view definitions it has stored.

`public Schema source()`
returns the source schema.

`public Schema[] targets()`
returns an array of the target schemas.

E `Evaluator` Constructors and Methods

`public Evaluator(FunctionTable ft)`
Constructs an `Evaluator` object that will use the `ft` for the evaluation of built-in functions.

`public void evaluate(ASG g)`
Evaluates `g` to normal form. Does not return a new `ASG` but instead changes `g`.

`public void evaluate(Cell root)`
Analogous to the above. Evaluates the graph of which `root` is the root to normal form.

`public void weakHeadNF(ASG g)`
Evaluates `g` to weak head normal form. Does not return a new `ASG` but instead changes `g`.

`public void weakHeadNF(Cell root)`
Analogous to the above. Evaluates the graph of which `root` is the root to weak head normal form.

F `FunctionTable` Constructors and Methods

`public FunctionTable()`
Constructs a `FunctionTable` object representing an empty table.

`public FunctionTable(BuiltInFunction[] fs, String[] names)`
Constructs a `FunctionTable` object. Mappings from strings in `names` to their corresponding functions in `fs` are added to the table.

`public BuiltInFunction getFunction(String name)`
Returns the `BuiltInFunction` corresponding to the string `name`.

`public void addFunction(String name, BuiltInFunction f)`
Adds a string and function pair to the table.

G `BuiltInFunction` Methods

`public int getArity()`
Returns the arity of the function.

`public Cell perform(Cell[] args, Evaluator e)`
Returns a `Cell` object that is the root of a graph that represents the result of applying the built-in function to the arguments `args`. Uses the `Evaluator e` to evaluate any sub-queries that need to be evaluated.