Chapter 4

Object Query Language

4.1 Introduction

In this chapter, we describe an object query language named OQL, which supports the ODMG data model. It is complete and simple. It deals with complex objects without privileging the set construct and the select-from-where clause.

We first describe the design principles of the language in Section 4.2, then we introduce in the next sections the main features of OQL. We explain the input and result of a query in Section 4.3. Section 4.4 deals with object identity. Section 4.5 presents the path expressions. In Section 4.6, we show how OQL can invoke operations and Section 4.7 describes how polymorphism is managed by OQL. Section 4.8 concludes this part of the presentation of the main concepts by exemplifying the property of operators composition.

Finally, a formal and complete definition of the language is given in Section 4.9. For each feature of the language, we give the syntax, its semantics, and an example. Alternate syntax for some features are described in Section 4.10, which completes OQL in order to accept any syntactical form of SQL. The chapter ends with the formal syntax, which is given in Section 4.11.

4.2 Principles

Our design is based on the following principles and assumptions:

- OQL relies on the ODMG object model.
- OQL is very close to SQL 92. Extensions concern object-oriented notions, like complex objects, object identity, path expressions, polymorphism, operation invocation, late binding.
- OQL provides high-level primitives to deal with sets of objects but is not restricted to this collection construct. It also provides primitives to deal with structures, lists, arrays, and treats such constructs with the same efficiency.
- OQL is a functional language where operators can freely be composed, as long as the operands respect the type system. This is a consequence of the fact that the result of any query has a type which belongs to the ODMG type model, and thus can be queried again.
- OQL is not computationally complete. It is a simple-to-use query language which provides easy access to an ODBMS.

- Based on the same type system, OQL can be invoked from within programming languages for which an ODMG binding is defined. Conversely, OQL can invoke operations programmed in these languages.
- OQL does not provide explicit update operators but rather invokes operations defined on objects for that purpose, and thus does not breach the semantics of an ODBMS which, by definition, is managed by the "methods" defined on the objects.
- OQL provides declarative access to objects. Thus OQL queries can be easily optimized by virtue of this declarative nature.
- The formal semantics of OQL can easily be defined.

4.3 Query Input and Result

As a stand-alone language, OQL allows querying denotable objects starting from their names, which act as entry points into a database. A name may denote any kind of object, i.e., atomic, structure, collection, or literal.

As an embedded language, OQL allows querying denotable objects which are supported by the native language through expressions yielding atoms, structures, collections, and literals. An OQL query is a function which delivers an object whose type may be inferred from the operator contributing to the query expression. This point is illustrated with two short examples.

The schema defines the types Person and Employee. These types have the extents Persons and Employees respectively. One of these persons is the chairman (and there is an entry-point Chairman to that person). The type Person defines the name, birthdate, and salary as attributes and the operation age. The type Employee, a subtype of Person, defines the relationship subordinates and the operation seniority.

```
select distinct  x.age
from Persons x
where x.name = "Pat"
```

This selects the set of ages of all persons named Pat, returning a literal of type set<integer>.

```
select distinct struct(a: x.age, s: x.sex)
from Persons x
where x.name = "Pat"
```

This does about the same, but for each person, it builds a structure containing age and sex. It returns a literal of type set<struct>.

```
select distinct struct(name: x.name, hps:  (select y
                                            from x.subordinates as y
                                            where y.salary >100000))
```

```
from Employees x
```

This is the same type of example, but now we use a more complex function. For each employee we build a structure with the name of the employee and the set of the employee's highly paid subordinates. Notice we have used a select-from-where clause in the select part. For each employee x, to compute hps, we traverse the relationship subordinates and select among this set the employees with a salary superior to $100,000. The result of this query is therefore a literal of the type set<struct>, namely:

```
set<struct (name: string, hps: bag<Employee>)>
```

We could also use a select operator in the from part:

```
select struct (a: x.age, s: x.sex)
from (select y from Employees y where y.seniority = "10") as x
where x.name = "Pat"
```

Of course, you do not always have to use a select-from-where clause:

```
Chairman
```

retrieves the Chairman object.

```
Chairman.subordinates
```

retrieves the set of subordinates of the Chairman.

```
Persons
```

gives the set of all persons.

4.4 Dealing with Object Identity

The query language supports both objects (i.e., having an OID) and literal (identity equals their value), depending on the way these objects are constructed or selected.

4.4.1 Creating Objects

To create an object with identity a type name constructor is used. For instance, to create a Person defined in the previous example, simply write

```
Person(name: "Pat", birthdate: "3/28/56" , salary: 100,000)
```

The parameters in parentheses allow you to initialize certain properties of the object. Those which are not explicitly initialized are given a default value.

You distinguish such a construction from the construction expressions that yield objects without identity. For instance,

```
struct (a: 10, b: "Pat")
```

creates a structure with two fields.

If you now return to the example in Section 4.3, instead of computing literals, you can build objects. For example, assuming that these object types are defined:

```
typedef set<integer> vectint;
interface stat{
attributes
   attribute Short a;
   attribute Char c;
};
typedef   bag<stat> stats;
```

you can carry out the following queries:

```
vectint(select distinct.age
         from Persons
         where name = "Pat")
```

which returns an object of type vectint and

```
stats(select stat (a: age, s: sex)
       from Persons
       where name = "Pat")
```

which returns an object of type stats.

4.4.2 Selecting Existing Objects

The extraction expressions may return:

- A collection of objects with identity, e.g., select x from Persons x where x.name ="Pat" returns a collection of persons whose name is Pat.
- An object with identity, e.g., element (select x from Persons x where x.passport_number=1234567) returns the person whose passport number is 1234567.
- A collection of literals, e.g., select x.passport_number from Persons x where x.name="Pat" returns a collection of integers giving the passport numbers of people named Pat.
- A literal, e.g., Chairman.salary.

Therefore the result of a query is an object with or without object identity: some objects are generated by the query language interpreter, and others produced from the current database.

4.5 Path Expressions

As explained above, one can enter a database through a named object, but more generally as long as one gets an object, one needs a way to *navigate* from it and reach the right data one needs. To do this in OQL, we use the "." (or indifferently "->") notation which enables us to go inside complex objects, as well as to follow simple relationships. For example, we have a Person p and we want to know the name of the city where this person's spouse lives.

Example:

```
p.spouse.address.city.name
```

This query starts from a Person, gets his/her spouse, a Person again, goes inside the complex attribute of type Address to get the City object whose name is then accessed.

This example treated 1-1 relationship, let us now look at n-p relationships. Assume we want the names of the children of the person p. We cannot write p.children.name because children is a list of references, so the interpretation of the result of this query would be undefined. Intuitively, the result should be a collection of names, but we need an unambiguous notation to traverse such a multiple relationship and we use the select-from-where clause to handle collections just as in SQL.

Example:

```
select c.name
from p.children c
```

The result of this query is a value of type Bag<String>. If we want to get a Set, we simply drop duplicates, like in SQL by using the distinct keyword.

Example:

```
select distinct c.name
from p.children c
```

Now we have a means to navigate from an object to any object following any relationship and entering any complex subvalues of an object. For instance, we want the set of addresses of the children of each Person of the database. We know the collection named Persons contains all the persons of the database. We have now to traverse two collections: Persons and Person.children. Like in SQL, the select-from operator allows us to query more than one collection. These collections then appear in the from part. In OQL, a collection in the from part can be derived from a previous one by following a path which starts from it.

Example:

```
select c.address
from Persons p,
    p.children c
```

This query inspects all children of all persons. Its result is a value whose type is Bag<Address>.

4.5.1 Predicate

Of course, the where clause can be used to define any predicate which then serves to select only the data matching the predicate. For example, we want to restrict the previous result to the people living on Main Street, and having at least two children. Moreover we are only interested in the addresses of the children who do not live in the same city as their parents.

Example:

```
select c.address
from Persons p,
    p.children c
where p.address.street = "Main Street" and
     count(p.children) >= 2 and
     c.address.city != p.address.city
```

4.5.2 Join

In the from clause, collections which are not directly related can also be declared. As in SQL, this allows computation of *joins* between these collections. This example selects the people who bear the name of a flower, assuming there exists a set of all flowers called Flowers.

Example:

```
select p
from Persons p,
   Flowers f
where p.name = f.name
```

4.6 Method Invoking

OQL allows us to call a method with or without parameters anywhere the result type of the method matches the expected type in the query. The notation for calling a method is exactly the same as for accessing an attribute or traversing a relationship, in the case where the method has no parameter. If it has parameters, these are given between parentheses. This flexible syntax frees the user from knowing whether the property is stored (an attribute) or computed (a method, such as age in the following example). This example returns a bag containing the age of the oldest child of all persons with name "Paul".

Example:

```
select max(select c.age from p.children c)
from Persons p
where p.name = "Paul"
```

Of course, a method can return a complex object or a collection and then its call can be embedded in a complex path expression. For instance, if oldest_child is a method defined on the class Person which returns an object of class Person, the following example computes the set of street names where the oldest children of Parisian people are living.

Example:

```
select p.oldest_child.address.street
from Persons p
where p.lives_in("Paris")
```

Although oldest_child is a method we *traverse* it as if it were a relationship. Moreover, lives_in is a method with one parameter.

4.7 Polymorphism

A major contribution of object orientation is the possibility of manipulating polymorphic collections, and thanks to the *late binding* mechanism, to carry out generic actions on the elements of these collections. For instance, the set Persons contains objects of classes Person, Employee, and Student. So far, all the queries against the Persons extent dealt with the three possible classes of the elements of the collection.

If one wants to restrict a query on a subclass of Person, either the schema provides an extent for this subclass which can then be queried directly, or else the superclass extent can be filtered to select only the objects of the subclass, as shown in the *class indicator* example.

A query is an expression whose operators operate on typed operands. A query is correct if the types of operands match those required by the operators. In this sense, OQL is a typed query language. This is a necessary condition for an efficient query optimizer. When a polymorphic collection is filtered (for instance Persons), its elements are statically known to be of that class (for instance Person). This means that a property of a subclass (attribute or method) cannot be applied to such an element, except in two important cases: late binding to a method or explicit class indication.

4.7.1 Late Binding

Give the activities of each person.

Example:

```
select p.activities
from Persons p
```

where activities is a method which has three incarnations. Depending on the kind of person of the current p, the right incarnation is called. If p is an Employee, OQL calls the operation activities defined on this object, or else if p is a Student, OQL calls the operation activities defined for Students, or else, p is a Person and OQL calls the method activities of the type Person.

4.7.2 Class Indicator

To go down the class hierarchy, a user may explicitly declare the class of an object that cannot be inferred statically. The evaluator then has to check at runtime that this object actually belongs to the indicated class (or one of its subclasses). For example, assuming we know that only students spend their time in following a course of study, we can select those Persons and get their grade. We explicitly indicate in the query that these Persons are in Student:

Example:

```
select ((Student)p). grade
from Persons p
where  "course of study" in p.activities
```

4.8 Operator Composition

OQL is a purely functional language. All operators can be composed freely as long as the type system is respected. This is why the language is so simple and its manual so short. This philosophy is different from SQL, which is an ad-hoc language whose composition rules are not orthogonal. Adopting a complete orthogonality allows OQL to not restrict the power of expression and makes the language easier to learn without losing the SQL syntax for the simplest queries. However, when very specific

SQL syntax does not enter in a pure functional category, OQL accepts these SQL pec u-liarities as possible syntactical variations. This is explained more specifically in Section 4.10.

Among the operators offered by OQL but not yet introduced, we can mention the set operators (union, intersect, except), the universal (for all) and existential quantifiers (exists), the sort and group by operators, and the aggregation operators (count, sum, min, max, and avg).

To illustrate this free composition of operators, let us write a rather complex query. We want to know the name of the street where employees live and have the smallest salary on average, compared to employees living in other streets. We proceed by steps and then do it as one query. We use OQL define instruction to evaluate temporary results.

Example:

1. Build the extent of class Employee (assuming that it is not supported directly by the schema and that in this database only objects of class Employee have “has a job” in their activities field):

```
define Employees as
    select (Employee) p from Persons p
    where "has a job" in p.activities
```

2. Group the employees by street and compute the average salary in each street:

```
define salary_map as
    select street, average_salary:avg(select e.salary from parti-
tion)
    from Employees e
    group by street: e.address.street
```

   The result is of type Bag<struct(street: string, average_salary:float)>. The group by operator splits the employees into partitions, according to the criterion (the name of the street where this person lives). The select clause computes, in each partition, the average of the salaries of the employees belonging to the partition.

3. Sort this set by salary:

```
define sorted_salary_map as
     select s from salary_map s order by s.average_salary
```

   The result is now of type List<struct(street: string, average_salary:float)>.

4. Now get the smallest salary (the first in the list) and take the corr e-sponding street name. This is the final result.

```
first(sorted_salary_map).street
```

Example as a single query:

```
first( select street, average_salary: avg(select e.salary from partition)
        from (select (Employee) p from Persons p
                where "has a job" in p.activities ) as e
        group by street : e.address.street
        order by average_salary).street
```

4.9 Language Definition

OQL is an expression language. A query expression is built from typed operands composed recursively by operators. We will use the term *expression* to designate a valid query in this section. An expression returns a result which can be an object or a literal.

OQL is a typed language. This means that each query expression has a type. This type can be derived from the structure of the query expression, the schema type declarations and the type of the named objects and literals. Thus queries can be parsed at compile time and type checked against the schema for correctness.

For each query expression, we give the rules that allow to (1) check for type correctness and (2) deduct the type of the expression from the type of the subexpressions.

For collections, we need the following definition: Types t_1, t_2,..., t_n are compatible if elements of these types can be put in the same collection as defined in the object model section.

Compatibility is recursively defined as follows:

(1) t is compatible with t

(2) if t is compatible with t' then

set(t) is compatible with set(t')
bag(t) is compatible with bag(t')
list(t) is compatible with list(t')
array(t) is compatible with array(t')

(3) if there exist t such that t is a supertype of t_1 and t_2, then t_1 and t_2 are compatible.

This means in particular that:

- literal types are not compatible with object types.
- atomic literal types are compatible only if they are the same.
- structured literal types are compatible only if they have a common ancestor.
- collections literal types are compatible if they are of the same collection and the types of their members is compatible.
- atomic object types are compatible only if they have a common ancestor.
- collections object types are compatible if they are of the same collection and the types of their members is compatible.

Note that if t_1, t_2,..., t_n are compatible, then there exists a unique t such that:

. (1) $t > t_i$ for all i's

. (2) for all t' such that t'!=t and $t' > t_i$ for all i's, t' > t

This t is denoted lub(t_1, t_2,..., t_n).

The examples are based on the schema described in Chapter 3.

4.9.1 Queries

A query is a query expression with no bound variables

4.9.2 Named query definition

define [query] id(x_1, x_2,..., x_n) as e(x_1, x_2,..., x_n)

where id is an identifier, e is an OQL expression and x_1, x_2,..., x_n are free variables in the expression e, has the following semantics: this records the definition of the function with name id in the database schema.

id cannot be a named object, a method name, a function name, or a class name in that schema, otherwise there is an error.

Once the definition has been made, each time we compile or evaluate a query and encounter a function expression, if it cannot be directly evaluated or bound to a function or method, the compiler/interpreter replaces id by the expression e. Thus this acts as a view mechanism.

Query definitions are persistent, i.e., they remain active until overridden (by a new definition with the same name) or deleted, by a command of the form:

delete definition id(x_1, x_2,..., x_n)

Query definitions cannot be overloaded, i.e., if there is a definition of id with n parameters and we redefined id with p parameters, p different from n, this is still interpreted as a new definition of id and overrides the previous definition.

Example

```
define age(x) as
    select p.age
    from Persons p
    where p.name=x
define does() as
    select p
    from Persons p
    where p.name = "Doe"
```

4.9.3 Elementary Expressions

4.9.3.1 Atomic Literals

If l is an atomic literal, then l is an expression whose value is the literal itself. Literals have the usual syntax:

- Object Literal: nil
- Boolean Literal: false, true
- Integer Literal: sequence of digits, e.g, 27
- Float Literal: mantissa/exponent. The exponent is optional, e.g., 3.14 or 314.16e-2
- Character Literal: character between single quotes, e.g., 'z'
- String Literal: character string between double quote, e.g.,"a string"

4.9.3.2 Named Objects

If e is an object name, then e is an expression. It returns the entity attached to the name. The type of e is the type of the named object as declared in the database schema.

Example:

```
Students
```

This query returns the set of students. We have assumed here that there exists a name Students corresponding to the extent of objects of the class Student.

4.9.3.3 Iterator Variable

If x is a variable declared in a from part of a select-from-where, then x is an expression whose value is the current element of the iteration over the corresponding collection.

If x is declared in the From part of a select-from-where expression by a statement of the form

```
e as x
```

or

```
e x
```

or

```
x in e,
```

where e is of type collection(t), then x is of type t.

4.9.3.4 Named Query

If define q as e is a query definition expression, then q is an expression. The type of q is the type of e.

Example:

```
Doe
```

This query returns the student with name Doe. It refers to the query definition expression declared in Section 4.9.2.

4.9.4 Construction Expressions

4.9.4.1 Constructing Objects

If t is a type name, p_1, p_2,...p_n are properties of this type with respective type t_1, t_2,..., t_n, if e_1, e_2,...., e_n are expressions of type t_1, t_2,..., t_n, then t(p_1: e_1, p_2: e_2,...., p_n: e_n) is an expression of type t.

This returns a new object of type t whose properties p_1, p_2,...,p_n are initialized with the expression e_1, e_2,...,e_n. The type of e_i must be compatible with the type of p_i.

If t is a type name of a collection and e is a collection literal, then t(e) is a collection object. The type of e must be compatible with t.

Examples:

```
Employee (name: "Peter", boss: Chairman)
```

This creates an Employee object.

```
vectint (set(1,3,10))
```

This creates a set object (see the definition of vectint in Section 4.4.1).

4.9.4.2 Constructing Structures

If p_1, p_2,...p_n are properties names, if e_1, e_2,...,e_n are expressions with respective type t_1, t_2,..., t_n, then struct(p_1: e_1, p_2: e_2,...., p_n: e_n) is an expression of type struct(p_1: t_1, p_2:t_2,...., p_n: t_n). It returns the structure taking values e_1, e_2,...,e_n on the properties p_1, p_2,...p_n.

Note that this dynamically creates an instance of the type struct(p_1: t_1, p_2: t_2, ..., p_n: t_n) if t_i is the type of e_i.

Example:

```
struct(name: "Peter", age: 25);
```

This returns a structure with two attributes name and age taking respective values Peter and 25.

See also abbreviated syntax for some contexts in Section 4.10.1.

4.9.4.3 Constructing Sets

If e_1, e_2,..., e_n are expressions of compatible types t_1, t_2,..., t_n, then set(e_1, e_2,..., e_n) is an expression of type set(t), where t = lub(t_1, t_2,..., t_n). It returns the set containing the elements e_1, e_2,..., e_n. It creates a set instance.

Example:

```
set(1,2,3)
```

This returns a set consisting of the three elements 1,2, and 3.

4.9.4.4 lConstructing Lists

If e_1, e_2,..., e_n are expressions of compatible types t1, t2,..., tn , then list(e_1, e_2,..., e_n) or simply (e_1, e_2,..., e_n) are expressions of type list(t), where t = lub(t_1, t_2,..., t_n). They return the list having elements e_1, e_2,..., e_n. They create a list instance.

If min, max are two expressions of integer or character types, such that min < max, then list(min. max) or simply (min. max) is an expression of value: list(min, min+1,... max-1, max).

The type of list(min. max) is list(int) or list (char) depending of the type of min.

Example:

```
list(1,2,2,3)
```

This returns a list of four elements.

Example:

```
list(3 .. 5)
```

This returns the list (3,4,5)

4.9.4.5 Constructing Bags

If e_1, e_2,..., e_n are expressions of compatible types t_1, t_2,..., t_n, then bag(e_1, e_2,..., e_n) is an expression of type bag(t), where t = lub(t_1, t_2,..., t_n). It returns the bag having elements e_1, e_2,..., e_n. It creates a bag instance.

Example:

```
bag(1,1,2,3,3)
```

This returns a bag of five elements.

4.9.4.6 Constructing Arrays

If e_1, e_2,..., e_n are expressions of compatible types t_1, t_2,..., t_n, then array(e_1, e_2,..., e_n) is an expression of type array(t), where t = lub(t_1, t_2,..., t_n). It returns an array having elements e_1, e_2,..., e_n. It creates an array instance.

Example:

```
array(3,4,2,1,1)
```

This returns an array of five elements.

4.9.5 Atomic Type Expressions

4.9.5.1 Unary Expressions

If e is an expression and <op> is a unary operation valid for the type of e, then <op> e is an expression. It returns the result of applying <op> to e.

Arithmetic unary operators:	+, -, abs
Boolean unary operator:	not

Example:

```
not true
```

This returns false

If <op> is +, - or abs, and if e is of type integer or float, then <op>e is of type e

If e is of type boolean, then not e is of type boolean.

4.9.5.2 Binary Expressions

If e_1 and e_2 are expressions and <op> is a binary operation, then e_1<op>e_2 is an expression. It returns the result of applying <op> to e_1 and e_2.

Arithmetic integer binary operators:	+, -, *, /, mod (modulo)
Floating point binary operators:	+, -, *, /
Relational binary operators:	=, !=, <, <=, >, >=

These operators are defined on all atomic types.

Boolean binary operators:	and, or

Example:

```
count(Students) - count(TA)
```

This returns the difference between the number of students and the number of TAs.

if <op> is +, -, * or / and e_1 and e_2 are of type integer or float, then e_1 <op> e_2 is of type float if e_1 or e_2 is of type float and integer otherwise.

If <op> is =, !=, <, <=, >, or >=, and e_1 and e_2 are of compatible types (here types integer and float are considered as compatible), then e_1 <op> e_2 is of type boolean.

If <op> is and or or, and e_1 and e_2 are of type boolean, then e_1 <op> e_2 is of type boolean.

Because OQL is a declarative query language, its semantics allows for a reordering of expression for the purpose of optimization. Boolean expressions are evaluated in an order which was not necessarily the one specified by the user but the one chosen by the query optimizer. This introduces some degree of non determinism in the semantics of a boolean expression:

(1) the evaluation of a boolean expressions stops as soon as we know the result (i.e. when evaluating an and clause, we stop as soon as the result is false and when evaluating an or clause, we stop as soon as the result is true)

(2) some clauses can generate a run time error and depending in their order in evaluation, they will or will not be evaluated.

For instance

```
p.age = 20 or p.spouse.age <20
```

If it is evaluated against an object of class Person of age 20 but with no spouse, will return true if it is evaluated in this order or will return an error if the optimizer has changed the order of evaluation.

4.9.5.3 String Expressions

If s_1 and s_2 are expressions of type string, then s_1 || s_2, and $s_1 + s_2$ are equivalent expressions of type string whose value is the concatenation of the two strings.

If c is an expression of type character, and s an expression of type string, then c in s is an expression of type boolean whose value is true if the character belongs to the string, else false.

If s is an expression of type string, and i is an expression of type integer, then s[i] is an expression of type character whose value is the i+1th character of the string.

If s is an expression of type string, and low and up are expressions of type integer, then s[low:up] is an expression of type string whose value is the substring of s from the low+1th character up to the up+1th character.

If s is an expression of type string, and pattern a string literal which may include the wildcard characters: “?” or “_”, meaning any character, and “*” or “%”, meaning any substring including an empty substring, then s like pattern is an expression of type boolean whose value is true if s matches the pattern, else false.

Example:

```
'a nice string' like '%nice%str_ng'    is true
```

4.9.6 Object Expressions

4.9.6.1 Comparison of Objects

If e_1 and e_2 are expressions which denote objects of compatible object types (objects with identity), then $e_1 = e_2$ and $e_1 != e_2$ are expressions which return a boolean. The second expression is equivalent to not($e_1 = e_2$). Likewise $e_1 = e_2$ is true if they designate the same object.

Example:

```
Doe = element(select s from Students s where s.name = "Doe")
```

is true.

4.9.6.2 Comparison of Literals

If e_1 and e_2 are expressions which denote literals of the compatible literal types (objects without identity), then $e_1 = e_2$ and $e_1 != e_2$ are expressions which return a boolean. The second expression is equivalent to not($e_1 = e_2$). Likewise, $e_1 = e_2$ is true if the value e_1 is equal to the value e_2.

4.9.6.3 Extracting an Attribute or Traversing a Relationship from an Object

If e is an expression of a type (literal or object) having an attribute or a relationship p of type t, then e.p and e->p are expressions of type t. These are alternate syntax to extract property p of an object e.

If e happens to designate a deleted or a nonexisting object, i.e., nil, an attempt to access the attribute or to traverse the relationship raises an exception.

4.9.6.4 Applying an Operation to an Object

If e is an expression of a type having a method f without parameters and returning a result of type t then e->f and e.f are expressions of type t. These are alternate syntax to apply an operation on an object. The value of the expression is the one returned by the operation or else the object nil, if the operation returns nothing.

Example:

```
jones->number_of_students
```

This applies the operation number_of_students to jones.

4.9.6.5 Applying an Operation with Parameters to an Object

If e is an expression of an object type having a method f with parameters of type t_1, t_2,..., t_n and returning a result of type t, if e_1, e_2,..., e_n are expressions of type t_1, t_2,..., t_n, then e->f(e_1, e_2,..., e_n) and e.f(e_1, e_2,..., e_n) are expressions of type t that apply operation f with parameters e_1, e_2,..., e_n to object e. The value of the expression is the one returned by the operation or else the object nil, if the operation returns nothing.

In both cases, if e happens to designate a deleted or a nonexisting object, i.e., nil, an attempt to apply an operation to it raises an exception.

Example:

```
Doe->apply_course("Math", Turing)->number
```

This query calls the operation apply_course on class Student for the object Doe. It passes two parameters, a string and an object of class Professor. The operation returns an object of type Course and the query returns the number of this course.

4.9.7 Collection Expressions

4.9.7.1 Universal Quantification

If x is a variable name, e_1 and e_2 are expressions, e_1 denotes a collection, and e_2 is an expression of type boolean, then for all x in e_1: e_2 is an expression of type boolean. It returns true if all the elements of collection e_1 satisfy e_2 and false otherwise.

Example:

```
for all x in Students: x.student_id > 0
```

This returns true if all the objects in the Students set have a positive value for their student_id attribute. Otherwise it returns false.

4.9.7.2 Existential Quantification

If x is a variable name, if e_1 and e_2 are expressions, e_1 denotes a collection, and e_2 is an expression of type boolean, then exists x in e_1: e_2 is an expression of type boolean. It returns true if there is at least one element of collection e_1 that satisfies e_2 and false otherwise.

Example:

```
exists x in Doe.takes: x.taught_by.name = "Turing"
```

This returns true if at least one course Doe takes is taught by someone named Turing.

If e is a collection expression, then exists(e) and unique(e) are expressions which return a boolean value. The first one returns true if there exists at least one element in the collection, while the second one returns true if there exists only one element in the collection.

Note that these operators accept the SQL syntax for nested queries like

select ... from col where exists (select ... from col_1 where predicate)

The nested query returns a bag to which the operator exists is applied. This is of course the task of an optimizer to recognize that it is useless to compute effectively the intermediate bag result.

4.9.7.3 Membership Testing

If e_1 and e_2 are expressions, e_2 is a collection, and e_1 is an object or a literal having the same type or a subtype than the elements of e_2, then e_1 in e_2 is an expression of type boolean. It returns true if element e_1 belongs to collection e_2.

Example:

```
Doe in Does
```

This returns true.

```
Doe in TA
```

This returns true if Doe is a Teaching Assistant.

4.9.7.4 Aggregate Operators

If e is an expression which denotes a collection, if <op> is an operator from {min, max, count, sum, avg}, then <op>(e) is an expression.

Example:

```
max (select salary from Professors)
```

This returns the maximum salary of the Professors.

if e is of type collection(t), where t is integer or float, then <op>(e) where <op> is an aggregate operator different from “count”, is an expression of type t.

if e is of type collection(t), then count(e) is an expression of type integer.

4.9.8 Select From Where

The general form of a select statement is as follows:

select [distinct] f(x_1, x_2,..., x_n, x_{n+1}, x_{n+2},..., x_{n+p})
from x_1 in e_1(x_{n+1}, x_{n+2},..., x_{n+p})
    x_2 in e_2(x_1, x_{n+1}, x_{n+2},..., x_{n+p})
    x_3 in e_3(x_1, x_2, x_{n+1}, x_{n+2},..., x_{n+p})
    ...
    x_n in e_n(x_1, x_2,..., x_{n-1}, x_{n+1}, x_{n+2},..., x_{n+p})
[where p(x_1, x_2,., x_n, x_{n+1}, x_{n+2},..., x_{n+p})]
[order by f_1(x_1, x_2,., x_{n+p}), f_2(x_1, x_2,., x_{n+p}),..., f_q(x_1, x_2,., x_{n+p})]

Or

select [distinct] f(x_1, x_2,..., x_n, x_{n+1}, x_{n+2},..., x_{n+p})
from e_1(x_{n+1}, x_{n+2},..., x_{n+p}) as x_1
    e_2(x_1, x_{n+1}, x_{n+2},..., x_{n+p}) as x_2
    e_3(x_1, x_2, x_{n+1}, x_{n+2},..., x_{n+p}) as x_3
    ...
    e_n(x_1, x_2,..., x_{n-1}, x_{n+1}, x_{n+2},..., x_{n+p}) as x_n
[where p(x_1, x_2,., x_n, x_{n+1}, x_{n+2},..., x_{n+p})]
[order by f_1(x_1, x_2,., x_{n+p}), f_2(x_1, x_2,., x_{n+p}),..., f_q(x_1, x_2,., x_{n+p})]

x_{n+1}, x_{n+2},..., x_{n+p} are free variables that have to be bound to evaluate the query.

The e_i's have to be of type collection, p has to be of type boolean and the f_i's have to be of a sortable type, i.e., an atomic type.

The result of the query will be a collection of t, where t is the type of the result of f.

The semantics of the query are as follows:
Assuming x_{n+1}, x_{n+2},..., x_{n+p} are bound to X_1, X_2,..., X_n, the query is evaluated as follows.
(1) the result of the from clause is a bag of elements of the type struct(x_1: X_1, x_2: X_2,..., x_n:X_n) where
X_1 ranges over the collection bagof(e_1(X_{n+1}, X_{n+2},..., X_{n+p}))
X_2 ranges over the collection bagof(e_2(X_1, X_{n+1}, X_{n+2},..., X_{n+p}))
X_3 ranges over the collection bagof(e_3(X_1, X_2, X_{n+1}, X_{n+2},..., X_{n+p}))
...
X_n ranges over the collection bagof(e_n(X_1, X_2,..., X_{n-1}, X_{n+1}, X_{n+2},..., X_{n+p}))
where bagof(C) is defined as follows, for a collection C:
if C is a bag: C
if C is a list: the bag consisting of all the elements of C
if C is a set: the bag consisting of all the elements of C
(2) Filter the result of the from clause by retaining only those tuples (X_1,
X_2,..., X_n)

that satisfy the predicate p(X_1, X_2,..., X_{n-1}, X_n, X_{n+1}, X_{n+2},..., X_{n+p})

(3) if the key word “order by” is there, sort this collection lexicographically using the functions f_1, f_2,..., f_q. and transform it in a list. Lexicographic order by a set of function is performed as follows: first sort according to function f_1, then for all the elements having the same f_1 value sort them according to f_2, etc.

(4) apply to each one of these tuples the function

f(X_1, X_2,..., X_{n-1}, X_n, X_{n+1}, X_{n+2},..., X_{n+p}).

If f is just “*” then keep the result of step (3) as such.
(5) if the key word “distinct” is there, then eliminate the eventual duplicates and obtain a set or a list without duplicate.
Note: to summarize the type of the result of a select from where is as follows:

- It is always a collection,
- the collection type does not depend of the collections specified in the from clause
- the collection type depends only of the form of the query: if we use “order by” we get a list, if we use the “distinct” key word without “order by”, we get a set and if neither “sort by” nor “distinct” are used, we get a bag.

Example:

```
select couple(student: x.name, professor: z.name)
   from Students as x,
       x.takes as y,
       y.taught_by as z
where z.rank = "full professor"
```

This returns a bag of objects of type couple giving student names and the names of the full professors from which they take classes.

Example:

```
select  *
from Students as x,
   x.takes as y,
   y.taught_by as z
where z.rank = "full professor"
```

This returns a bag of structures, giving for each student "object", the section object followed by the student and the full professor "object" teaching in this section:

```
bag< struct(x: Student, y: Section, z: Professor) >
```

Syntactical variations are accepted for declaring the variables in the *from* part, exactly as with SQL. The *as* keyword may be omitted. Moreover, the variable itself can be omitted too, and in this case, the name of the collection itself serves as a variable name to range over it.

Example:

```
select couple(student: Students.name, professor: z.name)
from Students,
   Students.takes y,
   y.taught_by z
where z.rank = "full professor"
```

In a select-from-where query, the *where* clause can be omitted, with the meaning of a true predicate.

4.9.9 Group-by Operator

If *select_query* is a select-from-where query, *partition_attributes* is a structure expression and *predicate* a boolean expression, then

select_query group by *partition_attributes* is an expression and

select_query group by *partition_attributes* having *predicate*

is an expression.

The Cartesian product visited by the select operator is split into partitions. For each element of the Cartesian product, the partition attributes are evaluated. All elements which match the same values according to the given partition attributes, belong to the same partition. Thus the partitioned set, after the grouping operation, is a set of structures: each structure has the valued properties for this partition (the valued *partition_attributes)*, completed by a property which is conventionally called *partition* and which is the bag of all objects matching this particular valued partition.

If the partition attributes are att_1: e_1, att_2: e_2,..., att_n: e_n, then the result of the grouping is of type

```
set< struct(att_1: type_of(e_1), att_2: type_of(e_2), ..., att_n: type_of(e_n),
                  partition: bag< type_of(grouped elements) >)>
```

The type of grouped elements is defined as follows.

If the *from* clause declares the variables v_1 on collection col_1, v_2 on col_2, ..., v_n on col_n, the grouped elements is a structure with one attribute, v_k, for each collection having the type of the elements of the corresponding collection partition:

```
bag< struct(v_1: type_of(col_1 elements),..., v_n: type_of(col_n elements))>
```

If a collection col_k has no variable declared the corresponding attribute has an internal system name.

This partitioned set may then be filtered by the predicate of a *having* clause. Finally, the result is computed by evaluating the *select* clause for this partitioned and filtered set.

The having clause can thus apply aggregate functions on *partition*; likewise the select clause can refer to *partition* to compute the final result. Both clauses can refer also to the partition attributes.

Example:

```
select  *
from  Employees e
group by  low:     salary < 1000,
          medium:  salary >= 1000 and salary < 10000,
          high:    salary >= 10000
```

This gives a set of three elements, each of which has a property called *partition* which contains the bag of employees that enter in this category. So the type of the result is

```
set<struct(low: boolean, medium: boolean, high: boolean,
                partition: bag<struct(e: Employee)>)>
```

The second form enhances the first one with a *having* clause which enables you to filter the result using aggregative functions which operate on each partition.

Example:

```
select  department,
    avg_salary: avg(select e.salary from partition)
from Employees e
group by  department: e.deptno
having avg(select e.salary from partition) > 30000
```

This gives a set of couples: department and average of the salaries of the employees working in this department, when this average is more than 30000. So the type of the result is

```
bag<struct(department: integer, avg_salary: float)>
```

Note that to compute the average salary, we have used a shortcut notation allowed by the Scope Rules defined in Section 4.9.15. The fully developed notation would read

```
avg_salary: avg(select x.e.salary from partition x)
```

4.9.10 Order-by Operator

If *select_query* is a select-from-where or a select-from-where-group_by query, and if e_1, e_2,..., e_n are expressions, then *select_query* order by e_1, e_2,..., e_n is an expression. It returns a list of the selected elements sorted by the functions e_1, and inside each subset yielding the same e_1, sorted by e_2,..., and the final subsub...set, sorted by e_n.

Example:

```
select p from Persons p order by p.age, p.name
```

This sorts the set of persons on their age, then on their name and puts the sorted objects into the result as a list.

Each sort expression criterion can be followed by the keyword **asc** or **desc**, specifying respectively an ascending or descending order. The default order is that of the previous declaration. For the first expression, the default is ascending.

Example:

```
select * from Persons  order by age desc, name asc, department
```

4.9.11 Indexed Collection Expressions

4.9.11.1 Getting the i*th* Element of an Indexed Collection

If e_1 is an expression of type list(t) or array(t) and e_2 is an expression of type integer, then $e_1[e_2]$ is an expression of type t. This extracts the e_2+1 element of the indexed collection e_1. Notice that the first element has the rank 0.

Example:

```
list (a,b,c,d) [1]
```

This returns b.

Example:

```
element (select x
           from Courses x
           where x.name = "Math" and x.number ="101").requires[2]
```

This returns the third prerequisite of Math 101.

4.9.11.2 Extracting a Subcollection of an Indexed Collection

If e_1 is an expression of type list(t) (resp. array(t)), and e_2 and e_3 are expressions of type integer, then $e_1[e_2:e_3]$ is an expression of type list(t) (resp. array(t)). This extracts the subcollection of e_1 starting at position e_2 and ending at position e_3.

Example:

```
list (a,b,c,d) [1:3]
```

This returns list (b,c,d).

Example:

```
element (select x
           from Courses x
           where x.name="Math" and x.number="101").requires[0:2]
```

This returns the list consisting of the first three prerequisites of Math 101.

4.9.11.3 Getting the First and Last Elements of an Indexed Collection

If e is an expression of type list(t) or array(t), <op> is an operator from {first, last}, then <op>(e) is an expression of type t. This extracts the first and last element of a collection.

Example:

```
first(element(select x
              from Courses x
              where x.name="Math" and x.number="101").requires)
```

This returns the first prerequisite of Math 101.

4.9.11.4 Concatenating Two Indexed Collections

If e_1 and e_2 are expressions of type list(t_1) and list(t_2) (resp. array(t_1) and array(t_2)) where t_1 and t_2 are compatible, then e_1+e_2 is an expression of type list(lub(t_1, t_2)) (resp. array(lub(t_1, t_2))). This computes the concatenation of e_1 and e_2.

```
list (1,2) + list(2,3)
```

This query generates list (1,2,2,3).

4.9.12 Binary Set Expressions

4.9.12.1 Union, Intersection, Difference

If e_1 is an expression of type set(t_1) or bag(t_1) and e_2 is an expression of type set(t_2) or bag(t_2) where t_1 and t_2 are compatible types, if <op> is an operator from {union, except, intersect}, then e_1 <op> e_2 is an expression of type set(lub(t_1, t_2)) if both expressions are of type set, bag(lub(t_1, t_2))) if any of them is of type bag. This computes set theoretic operations, union, difference, and intersection on e_1 and e_2, as defined in Chapter 2.

When the operand's collection types are different (bag and set), the set is first converted into a bag and the result is a bag.

Examples:

```
Student except TA
```

This returns the set of students who are not Teaching Assistants.

```
bag(2,2,3,3,3) union bag(2,3,3,3)
```

This bag expression returns bag(2,2,3,3,3,2,3,3,3).

```
bag(2,2,3,3,3) intersect bag(2,3,3,3)
```

The intersection of two bags yields a bag that contains the minimum for each of the multiple values. So the result is bag(2,3,3,3).

```
bag(2,2,3,3,3) except bag(2,3,3,3)
```

This bag expression returns bag(2).

4.9.12.2 Inclusion

If e_1 and e_2 are expressions which denote sets or bag of compatible types and if <op> is an operator from {<, <=, >, >=}, then e_1 <op> e_2 is an expression of type boolean.

When the operands are different kinds of collections (bag and set), the set is first converted into a bag.

e_1 < e_2 is true if e_1 is included in e_2 but not equal to e_2

e_1 <= e_2 is true if e_1 is included in e_2

Example:

```
set(1,2,3) < set(3,4,2,1)  is true
```

4.9.13 Conversion Expressions

4.9.13.1 Extracting the Element of a Singleton

If e is an expression of type collection(t), element(e) is an expression of type t. This takes the singleton e and returns its element. If e is not a singleton this raises an exception.

Example:

```
element(select x from Professors x where x.name = "Turing")
```

This returns the professor whose name is Turing (if there is only one).

4.9.13.2 Turning a List into a Set

If e is an expression of type list(t), listtoset(e) is an expression of type set(t). This converts the list into a set, by forming the set containing all the elements of the list.

Example:

```
listtoset (list(1,2,3,2))
```

This returns the set containing 1, 2, and 3.

4.9.13.3 Removing Duplicates

If e is an expression of type col(t), where col is set or bag, then distinct(e) is an expression of type set(t) whose value is the same collection after removing the duplicated elements. If e is an expression of type col(t), where col is either list or array, then distinct(e) is an expression of type col(t) obtained by keeping for each element of the list, its first occurrence.

Examples:

```
distinct(list(1, 4, 2, 3, 2, 4, 1))
```

This returns list(1, 4, 2, 3).

4.9.13.4 Flattening Collection of Collections

If e is a collection-valued expression, flatten(e) is an expression. This converts a collection of collections of t into a collection of t. So flattening operates at the first level only.

Assuming the type of e to be col_1<col_2<t>>, the result of flatten(e) is:

- If col_2 is a set (resp. a bag), the union of all col_2<t> is done and the result is a set<t> (resp. bag<t>).
- If col_2 is a list or an array and col_1 is a list or an array, the concatenation of all col_2<t> is done following the order in col_1 and the result is col_2<t>, which is thus a list or an array. Of course duplicates, if any, are maintained by this operation.
- If col_2 is a list or an array and col_1 is a set (resp. a bag), the lists or arrays are converted into sets (resp. bags), the union of all these sets (resp. bags) is done and the result is a set<t> (resp. bag<t>).

Examples:

```
flatten(list(set(1,2,3), set(3,4,5,6), set(7)))
```

This returns the set containing 1,2,3,4,5,6,7.

```
flatten(list(list(1,2), list(1,2,3)))
```

This returns list(1,2,1,2,3).

```
flatten(set(list(1,2), list(1,2,3)))
```

This returns the set containing 1,2,3.

4.9.13.5 Typing an Expression

If e is an expression of type t and t' is a type name, and t and t' are comparable (either t>=t' or t<=t'), then (t')e is an expression of type t'. This expression has two impacts

(1) at compile time, it is a statement for the interpreter/compiler type checker to notify that e should be understood as of type t

(2) at run time it asserts that e is indeed of type t and will return the result of e if it is of type t, or an exception if it is of a type t' not less than c.

This mechanism allows the user to execute queries which would otherwise be rejected as incorrectly typed. For instance

```
select s.salary
from Student s
where s in (select sec.assistant from Sections sec)
```

Because s is restricted in the where clause to Teaching Assistants that teach a section, this query will indeed return the salaries of these people. However the type checker has no means to check that the s in Student have indeed always a salary field and the query will be rejected at compile time.

If we write

```
select ((Employee) s).salary
from Student s
where s in (select sec.assistant from Sections sec)
```

Then the type checker knows that s has to be of Employee type and the query is accepted as type correct. Note that at run time, each occurrence of s in the select clause will be checked for its type.

4.9.14 Function Call

If f is a function of type (t_1, t_2,..., t_n -> t), if e_1, e_2,..., e_n are expressions of type t_1, t_2,..., t_n , then f(e_1, e_2,..., e_n) is an expression of type t whose value is the value returned by the function, or the object nil, when the function does not return any value. The first form calls a function without a parameter, while the second one calls a function with the parameters e_1, e_2,..., e_n.

OQL does not define in which language the body of such a function is written. This allows one to extend the functionality of OQL without changing the language.

4.9.15 Scope Rules

The *from* part of a select-from-where query introduces explicit or implicit variables to range over the filtered collections. An example of an explicit variable is

```
select ... from Persons p ...
```

while an implicit declaration would be

```
select ... from Persons ...
```

The scope of these variables spreads over all the parts of the select-from-where expression including nested sub-expressions.

The *group by* part of a select-from-where-group_by query introduces the name *partition* along with possible explicit attribute names which characterize the partition. These names are visible in the corresponding *having* and *select* parts, including nested sub-expressions within these parts.

Inside a scope, you use these variable names to construct path expressions and reach properties (attributes and operations) when these variables denote complex objects. For instance, in the scope of the first from clause above, you access the age of a person by p.age.

When the variable is implicit, like in the second from clause, you directly use the name of the collection by Persons.age.

However, when no ambiguity exists, you can use the property name directly as a shortcut, without using the variable name to open the scope (this is made implicitly), writing simply: age. There is no ambiguity when a property name is defined for one and only one object denoted by a visible variable.

To summarize, a name appearing in a (nested) query is looked up as follows:

- a variable in the current scope, or
- a named query introduced by the *define* clause, or
- a named object, i.e., an entry point in the database, or
- an attribute name or an operation name of a variable in the current scope, when there is no ambiguity, i.e., this property name belongs to only one variable in the scope.

Example:

Assuming that in the current schema the names Persons and Cities are defined.

```
select  scope1
from    Persons,
        Cities c
where exists(select  scope2 from children as child)
        or count (select scope3, (select scope4 from partition)
                     from  children p,
                          scope5 v
                     group by  age: scope6
                  )
```

In *scope1*, we see the names: Persons, c, Cities, all property names of class Person and class City as long as they are not present in both classes, and they are not called “Persons”, “c”, nor “Cities”.

In *scope2*, we see the names: child, Persons, c, Cities, the property names of the class City which are not property of the class Person. No attribute of the class Person can be accessed directly since they are ambiguous between “child” and “Persons”.

In *scope3*, we see the names: age, partition, and the same names from scope1, except “age” and “partition”, if they exist.

In *scope4*, we see the names: age, partition, p, v, and the same names from scope1, except “age”, “partition”, “p” and “v”, if they exist.

In *scope5*, we see the names: p, and the same names from scope1, except “p”, if it exists.

In *scope6*, we see the names: p, v, Persons, c, Cities, the property names of the class City which are not property of the class Person. No attribute of the class Person can be accessed directly since they are ambiguous between “child” and “Persons”.

4.10 Syntactical Abbreviations

OQL defines an orthogonal expression language, in the sense that all operators can be composed with each other as long as the types of the operands are correct. To achieve this property, we have defined a functional language with simple operators such as “+” or composite operators such as “select from where”, “group_by”, and “order_by” which always deliver a result in the same type system and which thus can be recursively operated with other operations in the same query.

In order to accept the whole DML query part of SQL, as a valid syntax for OQL, we have added ad-hoc constructions each time SQL introduces a syntax which cannot be considered in the category of true operators. This section gives the list of these constructions that we call “abbreviations,” since they are completely equivalent to a functional OQL expression. At the same time, we give the semantics of these constructions, since all operators used for this description have been previously defined.

4.10.1 Structure Construction

The structure constructor has been introduced in Section 4.9.4.2. An alternate syntax is allowed in two contexts: select clause and group-by clause. In both contexts, the SQL syntax is accepted, along with the one already defined.

select *projection* {, *projection*} ...

select ... **group by** *projection* {, *projection*}

where *projection* is one of the forms:

1. expression **as** identifier
2. identifier: expression
3. expression

This is an alternate syntax for

struct(identifier: expression {, identifier: expression})

If there is only one *projection* and the syntax (3) is used in a select clause, then it is not interpreted as a structure construction but rather the expression stands as is. Furthermore, a (3) expression is only valid if it is possible to infer the name of the corresponding attribute (the identifier). This requires that the expression denotes a path expression (possibly of length one) ending by a property whose name is then chosen as the identifier.

Example:

```
select p.name, salary, student_id
from Professors p, p.teaches
```

This query returns a bag of structures:

```
bag<struct(name: string, salary: float, student_id: integer)>
```

4.10.2 Aggregate Operators

These operators have been introduced in Section 4.9.7.4. SQL adopts a notation which is not functional for them. So OQL accepts this syntax, too. If we define *aggregate* as one of **min**, **max**, **count**, **sum** and **avg**,

select count(*) from ... is equivalent to
count(select * from ...)

select *aggregate*(query) from ... is equivalent to
aggregate(select query from ...)

select *aggregate*(distinct query) from ... is equivalent to
aggregate(distinct(select query from ...)

4.10.3 Composite Predicates

If e_1 and e_2 are expressions, e_2 is a collection, e_1 has the type of its elements, if *relation* is a relational operator (=, !=, <, <=, > , >=), then e_1 *relation* some e_2 and e_1 *relation* any e_2 and e_1 *relation* all e_2 are expressions whose value is a boolean.

The two first predicates are equivalent to

exists x in e_2: e_1 *relation* x

The last predicate is equivalent to

for all x in e_2: e_1 *relation* x

Example:

10 < some (8,15, 7, 22)   is true

4.10.4 String Literal

OQL accepts single quotes as well to delineate a string (see Section 4.9.3.1), like SQL does. This introduces an ambiguity for a string with one character which then has the same syntax as a character literal. This ambiguity is solved by context.

4.11 OQL BNF

The OQL grammar is given using a rather informal BNF notation.

- { symbol } is a sequence of 0 or n symbol(s).
- *[*symbol*]* is an optional symbol. Do not confuse this with the separators [].
- **keyword** is a terminal of the grammar.
- xxx_name is the syntax of an identifier.
- xxx_literal is self-explanatory, e.g., "a string" is a string_literal.
- bind_argument stands for a parameter when embedded in a programming language, e.g., $3i.

The non-terminal query stands for a valid query expression. The grammar is presented as recursive rules producing valid queries. This explains why most of the time this non- terminal appears on the left side of ::=. Of course, each operator expects its "query" operands to be of the right types. These type constraints have been introduced in the previous sections.

These rules must be completed by the priority of OQL operators which is given after the grammar. Some syntactical ambiguities are solved semantically from the types of the operands.

4.11.1 Grammar

4.11.1.1 Axiom (see 4.9.1, 4.9.2)

query_program ::= {define_query;} query
define_query ::= **define** identifier **as** query

4.11.1.2 Basic (see 4.9.3)

query ::= **nil**
query ::= **true**
query ::= **false**
query ::= integer_literal
query ::= float_literal
query ::= character_literal
query ::= string_literal
query ::= entry_name
query ::= query_name
query ::= bind_argument[1]
query ::= from_variable_name
query ::= (query)

4.11.1.3 Simple Expression (see 4.9.5)

query ::= query + query[2]
query ::= query - query
query ::= query * query
query ::= query / query
query ::= - query
query ::= query **mod** query
query ::= **abs** (query)
query ::= query || query

4.11.1.4 Comparison (see 4.9.5)

query ::= query comparison_operator query
query ::= query **like** string_literal
comparison_operator ::= =
comparison_operator ::= !=
comparison_operator ::= >
comparison_operator ::= <
comparison_operator ::= >=
comparison_operator ::= <=

4.11.1.5 Boolean Expression (see 4.9.5)

query ::= **not** query
query ::= query **and** query
query ::= query **or** query

---

1. A bind argument allows one to bind expressions from a programming language to a query when embedded into this language (see chapters on language bindings).
2. The operator + is also used for list and array concatenation.

4.11.1.6 Constructor (see 4.9.4)

query ::= type_name (*[*query*]*)
query ::= **type_name** (identifier: query {,identifier: query})
query ::= **struct** (identifier: query {, identifier: query})
query ::= **set** (*[*query {, query}*]*)
query ::= **bag** (*[*query {,query}*]*)
query ::= **list** (*[*query {,query}*]*)
query ::= (query, query {, query})
query ::= *[***list***]*(query .. query)
query ::= **array** (*[*query {,query}*]*)

4.11.1.7 Accessor (see 4.9.6, 4.9.11, 4.9.14, 4.9.15)

query ::= query dot attribute_name
query ::= query dot relationship_name
query ::= query dot operation_name(query {,query})
dot ::= **.** | **->**
query ::= * query
query ::= query **[**query**]**
query ::= query **[**query:query**]**
query ::= **first** (query)
query ::= **last** (query)
query ::= function_name(*[*query {,query}*]*)

4.11.1.8 Collection Expression (see 4.9.7, 4.10.3)

query ::= for **all** identifier **in** query: query
query ::= **exists** identifier **in** query: query
query ::= **exists**(query)
query ::= **unique**(query)
query ::= query **in** query
query ::= query comparison_operator quantifier query
quantifier ::= **some**
quantifier ::= **any**
quantifier ::= **all**
query ::= **count** (query)
query ::= **count** (*)
query ::= **sum** (query)
query ::= **min** (query)
query ::= **max** (query)
query ::= **avg** (query)

4.11.1.9 Select Expression (see 4.9.8, 4.9.9, 4.9.10)

query ::= **select** *[* **distinct** *]* projection_attributes
    from variable_declaration {, variable_declaration}
    *[***where** query*]*
    *[***group by** partition_attributes*]*
    *[***having** query*]*
    *[***order by** sort_criterion {, sort_criterion}*]*
projection_attributes ::= projection {, projection}
projection_attributes ::= *
projection ::= query
projection ::= identifier: query
projection ::= query **as** identifier
variable_declaration ::= query *[[* **as** *]* identifier*]*
partition_attributes ::= projection {, projection}
sort_criterion ::= query *[*ordering*]*
ordering ::= **asc**
ordering ::= **desc**

4.11.1.10 Set Expression (see 4.9.12)

query ::= query **intersect** query
query ::= query **union** query
query ::= query **except** query

4.11.1.11 Conversion (see 4.9.13)

query ::= **listtoset** (query)
query ::= **element** (query)
query ::= **distinct**(e)
query ::= **flatten** (query)
query ::= (class_name) query

4.11.2 Operator Priorities

The following operators are sorted by decreasing priority. Operators on the same line have the same priority and group left-to-right.

() **[]** **.** **->**
not **-** (unary) **+** (unary)
in
***** **/** **mod** **intersect**
+ **-** **union** **except** **||**
< **>** **<=** **>=** **< some** **< any** **< all** (etc. ... for all comparison operators)
= **!=** **like**
and **exists** **for all**
or
.. **:**
,
(identifier) this is the cast operator
order
having
group by
where
from
select