EyeDB Object Query Language

Version 2.8.0

January 2006

Published by SYSRA
30, avenue Général Leclerc
91330 Yerres - France

home page: http://www.eyedb.org

Contents

The Object Query Language

.

1 Introduction

In this chapter, we present the EyeDB Object Query Language which supports the EyeDB object model. It is based on the ODMG 3 Object Query Language OQL.

We first describe the design principles of the language in Section 2, then we present in Section 3 the main differences between EyeDB OQL and ODMG OQL. The language concepts are presented in Section 4. In Section 5, we introduced the language syntax. A quick reference manual of OQL is given in Section 6.

In this chapter, OQL denotes the EyeDB Object Query Language while the standard ODMG 3 Object Query Language will be denoted as ODMG OQL.

2 Principles

The principles of OQL are close to the principles of ODMG OQL introduced in the book *Object Database Standard, ODMG 3* by Rick Cattell and al.

Our design is based on the following principles and assumptions:

- OQL relies on the EyeDB Object Model.
- OQL is based on ODMG OQL close to SQL 92. Extensions to SQL 92 concern object-oriented notions, like complex objects, object identify, path expressions, polymorphim, operation invocation and late binding. Extensions to ODMG OQL concern function definitions, selection statements, iteration statement, assignment operator (see Section 3).
- OQL provides high level primitives to deal with sets of objects, structures, lists and arrays.
- OQL is a functional language where operators can freely be composed as long as the operands respect the type system.
- OQL is computationally complete.
- OQL can be invoked from within programming languages for which an EyeDB binding is defined (currently C++ and Java). Conversely, OQL can invoke operations programmed in this language.

3 OQL vs. ODMG 3 OQL

OQL implements all the ODMG OQL functionalities with a few exceptions concerning the `select` clause. In order to accept the whole DML (Data Manipulation Language) query part of SQL as a valid syntax for ODMG OQL, ad-hoc constructions have been added to ODMG OQL each time SQL introduces a syntax that cannot be considered in the category of true operators.

For instance, the following construct is a valid ODMG OQL construct:

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

This construct is not currently a valid OQL construct. The alternate valid OQL form is:

```
select struct(name: p.name, salary: p.salary) from Professors p
```

In the same way, ODMG OQL accepts SQL forms of the agregate operators `min, max, count, sum` and `avg`, for instance:

```
select count(*) from Persons
select max(e.salary) from Employees e
```

These constructs are not currently valid OQL constructs. The alternate valid OQL forms are:

```
count(select p from Persons p)
max(select e.salary from Employees e)
```

In the same way, the `select *` clause is not currently implemented in OQL, neither the implicit `select` clause (i.e. without explicit identifier). For instance, the following constructs are not OQL valid constructs, although there are valid ODMG OQL constructs:

```
select * from Person
select name from Person
```

There is no alternate OQL valid form for the first construct. The alternate OQL valid forms for the second construct is:

```
select Person.name
select p.name from Person p
select p.name from Person as p
select p.name from p in Person
```

On the other hand, OQL provides a few extensions to ODMG OQL:

- assignment operators,
- four regular expression operators,
- selection statements, `if/else`,
- iteration statements, `while`, `do/while`, two forms of `for`,
- function definition statements, `function`,
- the `eval` and `unval` operators,
- identifier operators, `isset`, `unset`, `refof`, `valof`, `&`, `*`, `push`, `pop`,
- exception management operators, `throw`,
- type information operators, `classof`, `typeof`
- miscellaneous operators, `structof`, `bodyof`
- builtin and library functions.

For instance, the following constructs are valid OQL constructs:

```
for (x in l)
{
  if (classof x != "Person")
    throw "Person type expected";
  if (x->name ~ "^john")
  {
     ok := true;
     break;
  }
}

function fib(n) {
  if (n < 2)
     return n;
  return fib(n-1) + fib(n-2);
}

for (n := 0, v := 0; n < 15; n++)
  v += fib(n);

function swap(x, y) {
   v := *x;
   *x := *y;
   *y := v;
}
```

```
i := "ii"; j := "jj";
swap(&i, &j);

function get_from(classname, attrname) {
   return eval "select x." + attrname + " from " + classname + " x";
}

names := get_from("Person", "name");
```

These extensions make OQL computationally complete.

Some of the ODMG OQL functionnalities or specificities are not yet implemented:

1. the `group by/having` operator,
2. the `order by` operator is more restrictive than in the ODMG specifications,
3. contrary to ODMG OQL, it is necessary to put parenthesis to call a function or method with takes no arguments,
4. contrary to ODMG OQL, the `||` operator does means string concatenation. It is the logical or operator. This will be changed in a future version.

4 Language Concepts

The basic entity in OQL is the atom. An atom is the result of the evaluation of any expression. Atoms are manipulated through expressions.

Although OQL is not fully an expression language as some valid constructs, such as flow controls, are not expressions, the expression is the basic concept of OQL. The non-expression constructs, such as selection and iteration statements, control the flow (or evaluation) of expressions. An expression is built from typed operands composed recursively by operators.

OQL is a typed language: each atom has a type. This type can be an OQL builtin type or can be derived from the schema type declarations.

OQL binds the EyeDB object model by providing a mapping between OQL builtin types and the EyeDB object model types. As in the EyeDB object model, OQL supports both object entities (with a unique object identifier) and literal entities (with no identifier). The concept of object/literal is orthogonal to the concept of type, this means that any type may have object instances and literal instances. For instance, one can have literal or object integers, literal or object collections. For instance, the following constructs produces a literal integer:

```
1; // OQL interpreter generates a literal integer
first(select x.age from Person x); // database produces a literal integer
                                   // bound to an atom of type integer
```

while the followings produce respectively a `Person` object and a literal collection of `Person` objects:

```
first(select x from Person x);
select x from Person x;
```

An EyeDB object is always bound in OQL to an atom of type `oid` or of type `object`. A literal is bound to the corresponding direct type in OQL using a trivial mapping. For instance, a literal entity of the EyeDB type `integer` is bound to an OQL atom of the OQL type `integer`; while an object entity of the EyeDB type `Person` is bound to an OQL atom of type `oid` or `object`.

We introduce now the OQL builtin types and the way that they are generated. OQL includes 15 builtin types as follows:

- `integer`
- `string`
- `float`
- `char`
- `boolean`
- `identifier`
- `set`
- `bag`
- `array`
- `list`

- struct
- oid
- object
- null
- nil

Some of these atoms can be expressed as terminals of the OQL grammar - for instance `integer`, `float`, `string` - others are generated using syntaxic constructions such as function calls or specific constructions - for instance `list`, `bag`, `struct`.

We will introduced first the syntax of the atoms which can be expressed as terminals, then the way to produce non terminal atoms.

5 Language Syntax

5.1 Terminal Atom Syntax

To express the syntax of terminal atoms, we use the standard regular expression notation.

Integer Atom

Integers are coded on 64 bits.
The syntax for the integer type is one of the followings:

`[0-9]+`	*decimal base*
`0x[0-9a-fA-F]+`	*hexadecimal*
`0[0-7]+`	*octal*

The domain for the integer type is as follows:

Minimal Value	*Maximal Value*
-9223372036854775808	9223372036854775807

A few examples:

```
13940     // integer expressed in the decimal base
0x273f1   // integer expressed in the hexadecimal base
0x273F1   // integer expressed in the hexadecimal base
0100      // integer expressed in the octal base
```

Float Atom

The syntax for floating point atoms is one of the following regular expressions:

```
[0-9]+\.[0-9]+?
[0-9]+?\.[0-9]+
[0-9]+\.[0-9]+?(e|E)[+-]?[0-9]+([fF]|[lL])?
[0-9]+?\.[0-9]+(e|E)[+-]?[0-9]+([fF]|[lL])?
```

The domain for the float type is as follows:

Minimal Value	*Maximal Value*
4.94065645841246544e-324	1.79769313486231570e+308

A few examples:

```
1.
1.23
.3
0.3039
1e+10
2.e+112
```

```
1.2e-100
.234e-200
.234e-200f
.234e-200F
```

String Atom

The syntax for the string type is as follows:
`\"([^"]|\\\")*\"`

The following escape sequences are interpreted:

Escape Sequence	*Name*	*ASCII Name*
\a	alert	BEL
\b	backspace	BS
\f	form feed	FF
\n	newline	NL (LF)
\r	carriage return	CR
\t	horizontal tab	HT
\v	vertical tab	VT
\\	backslash	\
\"	double quote	”
\’	single quote	’
\ooo	octal number	\ooo

A few examples:

```
"hello"
"hello \"world\""
"this is a multi-lines\ntext\n"
"this text contains escape sequences: \007\v\f\n''
```

Char Atom

The syntax for the char type is one of the followings:
’ascii character’
`’\[0-7+]’`
`’\(x|X)[0-9a-fA-F+]’`
`’\a’`
`’\b’`
`’\f’`
`’\n’`
`’\r’`
`’\t’`
`’\v’`

A few examples:

```
'a'
'b'
'\n'
'\a'
'\007'
'\x50'
'\x5F'
```

Boolean Atom

The syntax for a boolean atom is one of the followings:
`true`
`false`

Identifier Atom

The syntax for an identifier atom is as follows:
`[a-zA-Z\$_#][a-zA-Z\$_0-9#]*`

This means that an identifier must start whith a letter, a "_", a "$" or a "#" which may be followed by letters, digits, "_", "$" and "#" characters.

For instance, the following words are some valid identifiers:

```
a
alpha
beta1
alpha_beta
$a
oql$maxint
oql#2
$
#
_1
```

Note that identifiers beginning by `oql$` or `oql#` are reserved for special used by the interpreter.

Oid Atom

The syntax for an `oid` is as following:
`[0-9]+:[0-9]+:[0-9]+:oid`

Note that `oid` atoms are not typed directly by the user, but are produced by the database via the OQL interpreter.
The following words are some syntaxically valid atom `oid`s:

```
123.2.33373:oid
82727272.1.292828282:oid
```

Object Atom

The syntax for an atom `object` is as following:
`[0-9a-fA-F]+:obj`

Note that object atoms are not typed directly by the user, but are produced by OQL interpreter.
The following words are some syntaxically valid atom `object`s:

```
38383:obj
ea954:obj
```

Null Atom

The `null` atom denotes an unitialized value. Its type depends on the context. It can denote a unitialized integer, float, char, string or oid.

The syntax for a null atom is one of the followings:
`null`
`NULL`

Nil Atom

The `nil` atom denotes the empty atom.
The syntax for a nil atom is as follows:
`nil`

5.2 Non Terminal Atom Production

The other atoms - `set`s, `bag`s, `array`s, `list`s and `struct`s - are non terminal atoms. This means that they cannot be generated using a simple lexical construct.

List, Set, Bag and Array Atoms

To construct a collection atom - `list`, `set`, `bag` or `array` -, one may use the function *collection()* where *collection* denotes the collection type, for instance:

```
set(1, 2, 3)
list(1, "hello", "world")
array(2, 3, list(3893, -2, 'a'), 22)
bag(2, 2, 3, 4, 5, 12)
```

This is the simple way to construct such atoms, but as any other atoms, a collection atom may be produced by the OQL interpreter as the evaluation of a complex expression, for instance:

```
select x from Person x
```

produces an atom `bag` of objects.

Struct Atom

The most direct way to construct a `struct` atom is as follows:
`struct({`*identifier*`:`*expr*`})`

For instance:

```
struct(a: 1)
struct(format: text, s: "this is the text")
struct(name: "john", age: 32, spouse: first(select Person))
```

5.3 Keywords

Any programming language has its own set of reserved words (keywords) that cannot be used as identifiers. For instance, the keyword "`if`" cannot be used as a variable in a C program.
OQL also has its own set of keywords. But OQL is one part among others in the information system: for instance, there are an Object Model, an Object Definition Language (ODL) and Language bindings. The Object Model does not introduce any keyword, while ODL has its own set of keywords which are different from the OQL keywords. For instance, a class can include an attribute whose name is "`if`" as it is not a ODL keyword. If one wants to access this attribute in OQL, using for instance the path expression "`x.if`", we will get a syntax error. This is not acceptable.
We introduce in OQL (and in ODL) a way to neutralize any keyword: the token "`@`" used as a prefix keyword neutralizes the keyword and makes it a valid identifier. For instance, "`x.@if`", denotes the attribute "`if`" of the instance "`x`".
More generaly, "`@`*identifier*" denotes the identifier "*identifier*" whether "*identifier*" is a keyword or not.

OQL introduces the following keywords:

add	all	append	array	as
asc	bag	bodyof	break	by
char	classof	contents	define	delete
desc	distinct	do	element	else
empty	eval	except	exists	false
float	for	from	function	group
having	ident	if	import	in
int	intersect	is	isset	like
list	mod	new	nil	not
oid	order	pop	print	push
refof	return	scopeof	select	set
string	struct	structof	suppress	then
throw	to	true	typeof	union
unset	unval	valof	where	while

5.4 Comments

In OQL, comments are identical to the C++:

- all characters after the token `//` until the end of the current line are ignored by the interpreter,
- all characters between the tokens `/*` and `*/` are ignored.

For instance:

```
1 + 2;        // this is a comment
a := "hello"; /* this is another
                 comment */
```

5.5 Statements

A valid OQL construct is composed of a sequence of statements. Main of the statements are expression statements.

A statement can be one of the following:
- an expression statement,
- a selection statement, `if/else`,
- an iteration statement, `while`, `do/while`, `for`,
- a function definition statement, `function`
- a jump statement, `break`, `return`
- a compound statement,
- an empty statement.

The OQL expression sub-grammar is very close from the C grammar. The OQL grammar for the flow controls statement - `if/else`, `while`, `do/while`, `for` - is identical to the C grammar. The common operators of OQL and C have the same associativity and precedence.

5.6 Expression Statements

An expression statement is an expression following by a semicolon. Expressions are built from typed operands composed recursively by operators.

The syntax of an expression statement is as follows:
expr `;`
where *expr* denotes any expression.

There are three main kinds of expressions: atomic expressions, unary expressions and binary expressions. Atomic expressions are composed of one terminal atom and no operators, unary expressions are composed of one operand and one operator, binary expressions are composed of two operands and one operator.

We divide the OQL expression family into several semantical sub-families according to their operators as follows:

- atomic expressions,
- arithmetic expressions,
- assignment expressions,
- auto increment & decrement expressions,
- comparison expressions,
- logical expressions,
- conditional lists,
- expression sequences,
- array deferencing,
- identifier expressions,
- path expressions,
- function call,
- method invocation,
- eval/unval operators,
- set expressions,
- object creation,
- object deletion,
- collection expressions,
- exception expressions,
- function definition expressions,
- conversion expressions,
- type information expressions
- query expressions,
- miscellenaous expressions

In the following sub-sections, we introduce all the OQL expression types using the following template presentation:

1. we present first, in an unformal way, the syntax and the semantics of the operators,

2. general formal information is presented in a first table:
   - operator(s)
   - type
   - syntax
   - operand types
   - functions

3. in an optionnal second table, we present all the valid operand combination and their result type. A comment about the function performed is added if necessary. This table is skipped in case of the operand type combination is unique or trivial.

4. in a last table, we introduce a few examples. The examples manipulating database objects use the schema that can be found in the directory
   `$EYEDBROOT/examples/common`:

```
// person.odl

enum CivilState {
  Lady = 0x10,
  Sir  = 0x20,
  Miss = 0x40
};

class Address {
  attribute string street;
  attribute string<32> town;
  attribute string country;
};

class Person {
  attribute string name;
  attribute int age;
  attribute Address addr;
  attribute Address other_addrs[];
  attribute CivilState cstate;
  attribute Person * spouse inverse Person::spouse;
  attribute set<Car *> cars inverse owner;
  attribute array<Person *> children;

  int change_address(in string street, in string town,
                     out string oldstreet, out string oldtown);

  static int getPersonCount();
  index on name;
};

class Car {
  attribute string brand;
  attribute int num;
  Person *owner inverse cars;
};

class Employee extends Person {
  attribute long salary;
};
```

Expression types are gathered so to minimize the number of tables in this document.

5.7 Atomic Literal Expressions

Atomic literal expressions are expressions composed of a single terminal (or token or lexical unit) without any operator. They are also called primary expressions. These expressions have already been introduced in Section 5.1.

General Information	
Operator	*no operator*
Type	unary
Syntax	*terminal atom*
Operand Types	`integer, float, char, string, boolean, identifier, null, nil, oid,`
Result Type	same type as the operand

Expression Examples	
expression	*result*
`1`	`1`
`2.`	12.
`'a'`	`'a'`
`"hello"`	`"hello"`
`alpha`	*value of* `alpha`
`true`	`true`
`83283.1.29292:oid`	*an error is raised* in case of the oid is invalid. Otherwise the result is the input oid: `83283.1.29292:oid`

5.8 Arithmetic Expressions

Arithmetic expressions gather the expressions used for any arithmetic computation: addition, multiplication, substraction, division, modulo, shift left and right, bitwise or, bitwise exclusive, bitwise and, bitwise complement. This Section introduced these operators with a special focus on the additive operator which is a multi-purpose operator.

Additive Expression

The additive operator (i.e. `+`) is used for arithmetic addition of integers, floating point numbers and characters, and is also used for string concatenation, list or array concatenation and set or bag union. Its functionality depends on the type of its operands: it is a polymorphic operator. Note that the choice of its functionality is done at evaluation time, not at compile time. That means that the functionality of an expression such as `x + y` is unknown until the evaluation time. Depending on the dynamic type of the operands `x` and `y`, it can be a simple arithmetic addition, a string or list or array concatenation, a set or bag union or it can raise an error.
When used as a arithmetic operator and when the two operands have not the same type, one of the operands can be automatically promote to the type of the second one. The *promotion* mechanism is the same as in the C or C++ languages: `integer` may be promoted to `float`, `char` may be promoted to `float` or `integer`.

General Information	
Operator	`+`
Type	binary
Syntax	*expr* `+` *expr*
Commutative	yes
Operand Types	`integer, float, char, string, list, bag, set, array`
Result Type	see following table

Function	multi functions according to operands: arithmetic addition, string concatenation, list or array concatenation, set or bag union.

Possible Operand Combinations			
first operand type	*second operand type*	*result type*	
`integer`	`integer`	`integer`	
`integer`	`float`	`float`	
`char`	`char`	`integer`	
`char`	`integer`	`integer`	
`char`	`float`	`float`	
`float`	`float`	`float`	
`string`	`string`	`string`	*string concatenation*
`list`	`list`	`list`	*list concatenation*
`array`	`array`	`array`	*array concatenation*
`set`	`set`	`set`	*set union*
`bag`	`bag`	`bag`	*bag union*

Expression Examples	
expression	*result*
`1 + 2`	`3`
`1 + 2.`	`3.`
`2 + 2.3`	`4.3`
`'a' + 'b'`	`195`
`'a' + 1.2`	`98.200`
`"hello" + "world"`	`"helloworld"`
`list(1, 2, 3) + list(2, 3, 4)`	`list(1, 2, 3, 2, 3, 4)`
`set(1, 2, 3) + set(2, 3, 4)`	`set(1, 2, 3, 4)`
`1 + "hello"`	*raises an error*
`set(1, 2, 3) + list(2, 3, 4)`	*raises an error*

Multiplicative, Division and Minus Expressions

Multiplicative, division and minus expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. When operands have different types, promotionnal mechanisms are the same as for the additive operator.

General Information	

Operators	`-`
	`*`
	`/`

Type	binary
Syntaxes	*expr* `-` *expr* *expr* `*` *expr* *expr* `/` *expr*
Commutative	`-` : no `*` : yes `/` : no
Operand Types	`integer, float, char`
Result Type	see following table

Functions	- : substract * : multiply / : divide

Possible Operand Combinations		
first operand type	*second operand type*	*result type*
integer	integer	integer
integer	float	float
integer	char	integer
char	char	integer
char	integer	integer
char	float	float
float	float	float
float	integer	float
float	char	float

Expression Examples	
expression	*result*
1 - 2	-1
3 * 2.	6.
2 * 'a'	194
'a' * 'b'	9506
1 / 2	0
1 / 2.	.5000
1. / 2	.5000
1. / 2	.5000
"hello" * "world"	*raises an error*
1 - "hello"	*raises an error*

Shift, Mod, And, Or, XOr Expressions

Shift, modulo, and, or and xor expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. Operand types must be `integer` or `char` and the only possible type promotion is from `char` to `integer`.

General Information	
Operators	<< >> % & | ^
Type	binary
Syntaxes	*expr* << *expr* *expr* >> *expr* *expr* % *expr* *expr* & *expr* *expr* | *expr* *expr* ^ *expr*
Commutative	<< : no >> : no % : no & : yes | : yes

	`^` : yes
Operand Types	`integer, char`
Result Type	`integer`
Functions	`<<` : left shift `>>` : right shift `%` : modulo `&` : bitwise and `|` : bitwise or `^` : bitwise exclusive or

Expression Examples	
expression	*result*
`1 << 4`	`16`
`100 >> 2`	`25`
`100 % 13`	`9`
`0xf12 & 0xf`	`2`
`0xf12 | 0xf`	`3871`
`0xf12 ^ 0xf`	`3869`
`'b' % '9'`	`8`
`2 << 1.2`	*raises an error*
`2 % 3.4`	*raises an error*
`2.1 % 3`	*raises an error*

Sign Expressions

Sign expressions are the expressions using the unary operators `+` or `-`. The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. These unary operators accept only `integer`, `char` and `float` operands.

General Information	
Operators	`+` `-`
Type	unary
Syntaxes	`+` *expr* `-` *expr*
Operand Types	`integer, char, float`
Result Type	see following table
Functions	sign operator

Possible Operand Combinations	
operand type	*result type*
`integer`	`integer`
`float`	`float`
`char`	`integer`

Expression Examples	
expression	*result*
`+12`	`12`
`-100`	`-100`
`-123.4`	`-123.4`
`+'a'`	`97`
`-'a'`	`-97`

`+"hello"`	*raises an error*
`-null"`	*raises an error*

Complement Expressions

The complement operator performs a bitwise complement on its operand. The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. This operator accepts only `integer` and `char` operands.

General Information	
Operator	`~`
Type	unary
Syntax	`~` *expr*
Operand Types	`integer`, `char`
Result Type	`integer`
Functions	bitwise complement

Expression Examples	
expression	*result*
`~112`	`-113`
`~0`	`-1`
`~'a'`	`-98`
`~2.3`	*raises an error*
`~"hello"`	*raises an error*

5.9 Assignment Expressions

The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions except that the simple assignment operator in OQL is `:=` instead of `=` in C or C++. The left operand must be a left value. A left value is an OQL entity which is assignable: for instance any identifier or a valid path expression.
When the assignment is simple and when the left value is an identifier, no type checking on the right operand is done. For instance, `x := 10` and `x := "hello"` are always valid expressions. In the case of the left value is a path expression, the OQL interpreter checks that the type of the second operand matches the expected type of the first one. For instance if `p` denotes a `Person` instance, `p->age := 32` is certainly valid while `p->age := "hello"` raises a type check error.
When the assignment is combined with another operation (for instance, the `-=` operator), the left operand must be initialized and the interpreter checks that the left and right operand can be combined through the other operator.

For instance, the following constructs are valid:

```
a := 10;
a -= 20;

a := "hello";
a += " world";

p := first(select Person);
p.name := "johnny";

first(select Person.age = 0).name := "baby";
```

while these ones produce errors:

```
a := "hello";
a -= 20; // raises the error: operation 'string + integer' is not valid

a := list(1, 2);
a *= 2; // raises the error: operation 'list * integer' is not valid

unset b;
```

```
b += 20; // raises the error: uninitialized identifier 'b'

p := first(select Person);
p.age := "baby"; // raises the error: integer expected, got string
```

General Information	

Operators	`:=`
	`*=`
	`/=`
	`%=`
	`+=`
	`-=`
	`<<=`
	`>>=`
	`&=`
	`|=`
	`^=`

Type	binary
Syntaxes	*lvalue* `:=` *expr* *lvalue* `*=` *expr* *lvalue* `/=` *expr* *lvalue* `%=` *expr* *lvalue* `+=` *expr* *lvalue* `-=` *expr* *lvalue* `<<=` *expr* *lvalue* `>>=` *expr* *lvalue* `&=` *expr* *lvalue* `|=` *expr* *lvalue* `^=` *expr*
Commutative	no
Operand Types	*leftvalue* on the left side and *any type* on the right side
Result Type	the type of the right operand
Functions	perform an operation and assignment

Expression Examples	
expression	*result*
`a := 24`	`24`
`a += 12`	`36`
`a /= 2`	`18`
`a ^= 100`	`118`
`first(select Person).age := 38`	`38`
`"hello" := 4`	*raises an error* (i.e. `"hello"` is not a leftvalue)
`8 := 5`	*raises an error* (i.e. `8` is not a leftvalue)
`unset a; a += 20`	*raises an error* (i.e. uninitialized identifier)

5.10 Auto Increment & Decrement Expressions

The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. The operand must be an initialized left value of type `integer`, `char` or `float`. In case of the operand is a `char` atom, the result type is an `integer`. Otherwise, the result type is the type of the operand.

General Information	
Operators	`++` `--`
Type	unary
Syntaxes	*expr* `--` *expr* `++` `++`*expr* `--`*expr*
Operand Type	*leftvalue* of type `integer`, `char` or `float`.
Result Type	see following table
Functions	*expr* `--` : post-decrementation *expr* `++` : post-incrementation `++`*expr* : pre-incrementation `--`*expr* : pre-incrementation

Possible Operand Combinations	
operand type	*result type*
`integer`	`integer`
`float`	`float`
`char`	`integer`

Expression Examples		
expression	*result*	
`a := 1; a++`	`1`	*initially* `a` equals `1`; the result of the evaluation is `1` but after the evaluation, `a` equals 2
`--a`	`0`	
`a++`	`0`	`a` equals `1` after the evaluation

5.11 Comparison Expressions

The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions except that the equal operator could be either `==` or `=`.

Equal and NotEqual Expressions

Operands may have any type at all. If the type of the operands differ (modulo the type promotion mechanisms for numbers), the result of the expression *operand1* `==` *operand2* is always `false` while the result of *operand1* `!=` *operand2* is always `true`.

General Information	
Operators	`==` `!=`
Type	binary
Syntaxes	*expr* `==` *expr* *expr* `!=` *expr*
Commutative	yes
Operand Types	*any type*
Result Type	`boolean`
Functions	equal not equal

When operands are number of different types, an automatic promotion is done to the more precise type.

Expression Examples	
expression	*result*
`1 == 1`	`true`
`1 == 1.0`	`true`
`1 != 2`	`true`
`1 == 2`	`false`
`1 == "hello"`	`false`
`"hello" == "hello"`	`true`
`list(1, 2, 3) == list(1, 2, 3)`	`true`
`set(1, 3, 2) == set(1, 2, 3)`	`true`

Less and Greater Expressions

The comparison operators `<, <=, >` and `>=` are multi-purpose operators: they are used for integer, floating point number and character comparison, but also for list or array term-to-term comparison and for set or bag inclusion. Their functionality depends on the type of its operands: they are polymorphic operators. Note that the choice of the functionality is done at evaluation time, not at compile time. That means that the functionality of an expression such as `x < y` is unknown until the evaluation time. Depending on the dynamic type of the operands `x` and `y`, it can be an arithmetic comparison (if `x` and `y` are numbers), a alpha-numeric comparison (if `x` and `y` are strings), a term-to-term ordered collection comparison (if `x` and `y` are lists or arrays) or a set or bag inclusion comparison (if `x` and `y` are sets or bags).

While arithmetic and alpha-numeric comparisons are trivial and do not need any suplementary explanations, the term-to-term ordered collection comparisons needs to be detailed.

The general algorithm for this functionnality is as follows:

1. let *l1* and *l2* two OQL ordered collections, containing respectively *l1_cnt* and *l1_cnt* atoms.
2. let *op* one of the following polymorphic comparison operators: `< <= > >=`,
3. *l1 op l2* is `true` if and only if all the following conditions are realized:
   (a) *l1* and *l2* must be of the same collection type,
   (b) *l1_cnt op l2_cnt* or *l1_cnt* equals *l2_cnt*
   (c) for each atom *l1[i]* with *i* in `[`*i*`,`*l1_cnt*`]`, *l1[i] op l2[i]*

For instance:
`list(1, 2) <= list(0, 2)` is `true`
`list(1, 2) <= list(3)` is `false`
`list(1, 2) <= list(3)` is `false`
`list("aaa", 4) < list("bbbb", 8)` is `true`
`list("aaa", 4, list(1, 2)) < list("b", 8, list(2, 3))` is `true`
`list(set(2, 4), 3) < list(set(4, 2, 3), 4)` is `true`
`list(2, 3) < list("hello", 2)` raises an error
`list(2, 3) < array(2, 4)` raises an error

Note that the fact that *l1* `<=` *l2* is `false` does not implie that *l1* `>` *l2* is `true`. Indeed, `list(2, 3) < list(1, 3, 2)` and `list(1, 3, 2) >= list(2, 3)` are `false`.

General Information	

Operators	`<`
	`<=`
	`>`
	`>=`

Type	binary
Syntaxes	*expr* `<` *expr* *expr* `<=` *expr* *expr* `>` *expr* *expr* `>=` *expr*

Commutative	no
Operand Types	`integer, float, char, string, list, array, set, bag`
Result Type	`boolean`
Functions	the function depends on the operands: `<` : less than *or* is included in `<=` : less than or equal *or* is included in or equal `>` : greater than *or* contains `>=` : greater than or equal *or* contains or equal

Possible Operand Combinations			
first operand type	*second operand type*	*result type*	*comments*
`integer`, `char`, `float`	`integer`, `char`, `float`	`boolean`	performs an arithmetic comparison
`string`	`string`	`boolean`	performs an alpha-numeric comparison
`set`	`set`	`boolean`	performs an inclusion comparison
`bag`	`bag`	`boolean`	performs an inclusion comparison
`set`	`bag`	`boolean`	performs an inclusion comparison: the `set` operand is converted to a `bag`
`bag`	`set`	`boolean`	performs an inclusion comparison: the `set` operand is converted to a `bag`
`list`	`list`	`boolean`	performs a term-to-term polymorphic (i.e. numeric, alpha-numeric or inclusion) comparison
`array`	`array`	`boolean`	performs a term-to-term polymorphic comparison

Note that in case of different operand types, an automatic promotion is done to the more precise type.

Expression Examples	
expression	*result*
`1 < 2`	`true`
`1 >= 2`	`false`
`2.  <= 2`	`true`
`"hello" < "world"`	`true`
`"hello" >= "world"`	`false`
`list(1, 2) < list(2, 3)`	`true`
`list(1, 2) < list(0, 3)`	`false`
`list(1, 2) < list(0, 3, 2)`	`false`
`list(0, 3, 2) >= list(1, 2)`	`false`
`list(1, 2) < list(2, 3, 3)`	`true`
`list(1, 2) < list(0)`	`false`
`set(1, 2) < set(2, 4, 44)`	`false`
`set(1, 2) < set(2, 1, 44)`	`true`
`"hello" >= 3`	*raises an error*
`list(1, 2) < array(2, 4, 44)`	*raises an error*
`set(1, 2) < bag(2, 1, 44)`	*raises an error*

Regular Expression Operators

OQL provides the ODMG OQL regular expression operator `like` plus four other ones. These four extra operators are based on the regular expression UNIX library. So, the syntax of the regular expression are the same as that used by the well known UNIX tools `grep`, `sed`, and so on. All the regular expression operators takes two string operands: the first one is the string to compare, the second one is the regular expression. So, these operators are not commutative. These operators provide the following functionalities:

`like` : ODMG OQL operator. Returns `true` if the first operand matches the regular expression. Otherwise `false` is returned. The regular expression is am SQL regular expression where, for instance, `%` and _ are wilcard characters.
`~` : This operator has the same functionnality as the `like` operator but the regular expression has the UNIX syntax.
`~` : Returns `true` if the first operand matches the regular expression in a case insensitive way. Otherwise `false` is returned.
`!~` : Returns `true` if the first operand does not match the regular expression. Otherwise `false` is returned.
`!~~` : Returns `true` if the first operand does not match the regular expression in a case insensitive way. Otherwise `false` is returned.

Note that the operator `like` *uses currently the UNIX form of regular expressions instead of the SQL form. It will become ODMG/SQL compliant in a next version.*

General Information	

Operators	`~`
	`~~`
	`!~`
	`!~~`
	`like`

Type	binary
Syntaxes	*expr* `~` *expr* *expr* `~~` *expr* *expr* `!~` *expr* *expr* `!~~` *expr* *expr* `like` *expr*
Commutative	no
Operand Types	`string`
Result Type	`boolean`
Functions	`~` : matches the regular expression `~` : matches the regular expression, case insensitive `!~` : does not match the regular expression `!~~` : does not match the regular expression, case insensitive `like` : matches the regular expression

Expression Examples	
expression	*result*
`"hello" ~ "LL"`	`false`
`"hello" ~~ "LL"`	`true`
`"hello" ~ "^LL"`	`false`
`"hello" ~ "^h"`	`true`
`"hello" !~ "^h"`	`false`
`"hello" ~ ".*ll.*"`	`true`
`".*ll.*" ~ "hello"`	`false` *because regular expression should be on the right*

5.12 Logical Expressions

OQL provide three logical expressions which can take two form each. The logical or operator is `||` or `or`. The logical and operator is `&&` or `and`. The logical not operator is `!` or `not`.

The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. Note that the ODMG operator "`||`" denotes the string concatenation.
As for `C` and `C++`, the OQL interpreter performs a lazy evaluation:

- *expr1* `||` *expr2*
  expr2 is not evaluated if *expr1* is evaluated to `true`.
- *expr1* `&&` *expr2*
  expr2 is not evaluated if *expr1* is evaluated to `false`.

General Information	
Operators	`||` `&&`
Type	unary, binary
Syntaxes	*expr* `||` *expr* *expr* `or` *expr* *expr* `&&` *expr* *expr* `and` *expr* `!` *expr* `not` *expr*
Operand Type	`boolean`
Result Type	`boolean`
Functions	logical or logical and logical not

Expression Examples	
expression	*result*
`true || false`	`true`
`false || false`	`false`
`true && false`	`false`
`1 == 2 || 3 == 4`	`false`
`1 == 2 || 3 == 3`	`true`
`1 == 2 or 3 == 3`	`true`
`1 == 2 || "hello" == "hello"`	`true`
`(1 == 2 || 2 == 2) && (a = "hello")`	returns `true` if `a` equals "hello". `false` otherwise
`1 || 3 == 3`	*raises an error*: boolean expected got integer
`!3`	*raises an error*: boolean expected got integer
`!(1 == 1)`	`false`
`not(1 == 1)`	`false`

5.13 Conditional Expression

The unique conditional expression operator is `?:`. The expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. The first operand must be an boolean and the two others may be of any type. Contrary to `C` and `C++`, the two last operands does not need to be of the same type.

General Information

Operator	`?:`
Type	ternary
Syntaxe	*expr* `?` *expr* `:` *expr*
Operand Types	first operand is boolean, others are any type
Result Type	type of the evaluated operand
Functions	conditional evaluation: evaluates and returns the second operand if the first operand is true; otherwise evaluates and returns the second operand

Expression Examples	
expression	*result*
`true ? "hello" : "world"`	`"hello"`
`true ? 2.3 : "world"`	`2.3`
`1+1 == 2 ? (a := 3.1415926535) : nil`	`3.1515926535`
`1 ? 3 : nil`	*raises an error*: boolean expected, got integer.

5.14 Expression Sequences

The expression sequence operator - also called comma sequencing - expression syntax, semantics, associativity and precedence are quite identical to the corresponding C and C++ expressions. This operator `,` takes two operands: it evaluates both of them and returns the second one.

General Information	
Operator	`,`
Type	binary
Syntaxe	*expr* `,` *expr*
Commutative	no
Operand Types	*any type*
Result Type	type of the second operand
Functions	evaluates the first operand, then the second one. Returns the evaluation of the second one.

Expression Examples	
expression	*result*
`true, "hello"`	`"hello"`
`a := 2, 4`	`4` (*note that* `a` equals `2`)
`b := 10, a := b+1`	`11`

5.15 Array Deferencing

OQL provides polymorphic single and range deferencing. The single deferencing is used to get one element in an ordered collection or a character in a string or one element in a non-collection array. The range deferencing is used to get several elements.
The deferencing of ordered collections is introduced in more details in Section 5.24.

Single Deferencing

The single deferencing operator takes as its first operand an atom of type `string`, an indexed (or ordered) collection (`list` or `array`) or a non-collection array. The second operand must be of type `integer`. Depending on the type of the first operand, the returned atom is as follows:

1. if the first operand is a string, the returned atom is the `#`*expr* character of the string where *expr* denotes the second operand. If *expr* is equal to the length to the string the character `'\000'` is returned. If it greater than the length the string an *out of bounds* error is raised.

2. for an ordered collection, the returned atom is the `#`*expr* item of the collection. If *expr* is greater than or is equal to the size of the collection, an *out of bounds* error is raised.

3. if the first operand is an non-collection array, the returned atom is the `#`*expr* item of the array. If *expr* is greater than or is equal to the size of the array, an *out of bounds* error is raised.

The single deferencing operator may be used as a left value, that means that a single deferencing expression is assignable. For instance the sequence of statements:

```
s := "hello";
s[1] := 'E';
s[4] := 'O';
```

set the variable `s` to `"hEllO"`.

The single deferencing operator may be used everywhere in a path expression. For instance, `first(select Person).other_addrs[2].street[3]` denotes the character `#3` of the `street` attribute in the `#2` `other_addrs` non-collection array attribute of the first `Person` instance.

General Information	
Operator	`[]`
Syntaxe	*expr* `[`*expr* `]`
Type	binary
Commutative	no
Operand Types	first operand: `string`, indexed collection (`list` or `array`) or non-collection array, second operand: `integer`
Result Type	`char` if first operand is a string, otherwise type of the returned item in the indexed collection or non-collection array.
Functions	`[`*expr* `]` : returns the character (or item in the indexed collection or in the non-collection array) number *expr*
Note	this operator may be used in the composition of a left value.

Expression Examples		
expression	*result*	
`"hello"[0]`	`'h'`	
`a := "hello"; a[1]`	`'e'`	
`a[3]`	`l`	
`a[6]`	*raises an error*	
`a[0] := 'H'`	`'H'`	`a` equals `"Hello"`
`list(1, 2, "hello", 4)[3]`	`"hello"`	
`list(1, 2, "hello", 4)[4]`	*raises an error*	
`first(select Person).name[2] := 'X'`	`'X'`	

Range Deferencing

The range deferencing operators, `[:]` and `[?]`, takes as their first operand an atom of type `string`, an indexed (or ordered) collection (`list` or `array`) or a non-collection array. The other operands must be of type `integer`. The `[?]` may have also an unordered collection (`set` or `bag` as its first operand.

The operator syntax and semantics are as follows:

- *expr* `[`*expr1*`:`*expr2*`]`
  1. if the first operand is a string, the returned atom is a list composed of the characters between the `#`*expr1* and the `#`*expr2* characters of the string. If *expr1* is less than zero or or if *expr2* is greater than the length of the string an *out of bounds* error is raised.
  2. for an ordered collection, the returned atom is a list composed of the items between the `#`*expr1* and the `#`*expr2* items of the collection. If *expr1* is less than zero or if i *expr2* iss greater than or is equal to the size of the collection, an *out of bounds* error is raised.
  3. if the first operand is an non-collection array, the returned atom is a list composed of the items between the `#`*expr1* and the `#`*expr2* items of the array. If *expr1* is less than zero or if i *expr2* iss greater than or is equal to the size of the collection, an *out of bounds* error is raised.
- *expr* `[?]`
  1. if the first operand is a string, the returned atom is a list composed of all the characters of the string, including the last character `'\000'`.
  2. for an ordered or an unordered collection, the returned atom is a list composed of all the items of the collection. If the collection is a list, the list itself is returned. If the collection is an array, this operator has the same functionnality as the `listtoarray` library function.
  3. if the first operand is an non-collection array, the returned atom is a list composed of the all the items of the array.

Contrary to the single deferencing operator, the range deferencing operator cannot be used as a left value.

The range deferencing operators may be used everywhere in a path expression. For instance, `first(select Person).children[?].name[?]`, denotes the list of all characters of the `name` attribute in all the `children` of the first `Person` instance.

General Information	
Operators	`[:]` `[?]`
Syntaxes	*expr* `[`*expr* `:`*expr* `]` *expr* `[?]`
Type	ternary or unary
Operand Types	first operand: `string` or indexed collections (`list` or `array`), second operand and third operand: `integer`
Result Type	a list of `char` if first operand is a string, otherwise a list of returned items in the indexed collection or non-collection array.
Functions	`[`*expr1*`:`*expr2*`]` : returns a lits of characters (or items in collection) indexed from *expr1* to *expr2* `[?]` : returns a list of all characters (or items in collection)

Expression Examples	
expression	*result*
`"hello"[0:2]`	`list('h', 'e', 'l')`
`"hello"[?]`	`list('h', 'e', 'l', 'l', 'o', '\000')`
`list(1, 2, "hello", 4)[2:3]`	`list("hello", 4)`
`array(1, 2, "hello", 4)[?]`	`list(1, 2, "hello", 4)`

`first(select Person).name[?]`	`list('j', 'o', 'h', 'n', '\000')`
`list(select class.type = "user")[0:4].name`	`list("Employee", "Address", "Person")`

5.16 Identifier Expressions

We call an identifier expression an unary or binary expression whose operands must be identifiers. There are height identifier operators: `::`, `isset`, `unset`, `&` (identical to `refof`), `*` (identical to `valof`), `scopeof`, `push` and `pop`.

As all these operators take identifiers as their operands, we skip the second table (operand combinations) while introducing these operators.

`::` Operator

The `::` unary/binary operator (called scope operator) is used to define a global or particular scope for a variable.
The unary version of this operator denotes a global scope. For instance, `::alpha` denotes the global variable `alpha`. In the body of a function, identifiers denote local variables; outside the body of a function identifiers denote global variables, that means that, in this context, the global scope operator is not mandatory. If one wants to use a global variable in the body of a function, the global scope operator is mandatory. Refer to the Function Definition Statement Section for more information about local function variables.

The binary version of this operator denotes a particular scope. For instance, `Person::checkName` denotes the class attribute or method of the class `Person`.

note: class (or static) attributes are not currently well supported by the OQL interpreter. Class attributes are only supported in some specific query expressions (refer to the Query Expression Section).

General Information	
Operator	`::`
Syntax	`::` *identifier* `identifier::identifier`
Type	unary and binary
Operand Types	`identifier`
Result Type	value of the identifier if used as a right value; identifier reference if used as a left value
Function	defines a global or particular scope for the identifier.

Expression Examples	
expression	*result*
`::a`	the value of the global variable `a`
`::alpha := 1`	sets the value of the global variable `alpha` to `1`, returns `1`
`Person::checkName("wayne")`	calls the class method `checkName` in the class `Person`
`2::alpha`	*raises an error*

`isset` Operator

The `isset` operator is used to check whether a variable is already set or not. It returns `true` is the variable is set, `false` otherwise.

General Information

Operator	`isset`
Syntax	`isset` *identifier*
Type	unary
Operand Type	`identifier`
Result Type	`boolean`
Function	evaluated to `true` if the identifier is set, `false` otherwise

Expression Examples	
expression	*result*
`isset oql$variables`	`true`
`isset a`	returns `true` if `a` is set, `false` otherwise
`isset 1`	*raises an error*

`unset` Operator

The `unset` operator is used to unset an variable. It returns the `nil` atom.

General Information	
Operator	`unset`
Syntax	`unset` *identifier*
Type	unary
Operand Type	`identifier`
Result Type	`nil`
Function	unset the identifier

Expression Examples	
expression	*result*
`unset a`	`nil`
`unset ::a`	`nil`
`unset 2`	*raises an error*

`refof` Operator

The `&` (identical to `refof`) operator is used to get the reference of an identifier. This operator is essentially used when one calls a function or method which updates one or more given parameters. For instance, let the function `swap(x, y)` which swaps the value of its two parameters. One needs to give the references of the variables that one wants to swap. For instance:

```
i := "ii";
j := "jj";

swap(&i, &j);
```

After the call to `swap`, the variable `i` equals `jj` while the variable `j` equals `ii`.
The reverse operator `*` (described following section) is used in the swap function.

General Information	
Operator	`refof` `&`
Syntax	`refof` *identifier* `&` *identifier*
Type	unary

Operand Type	`identifier`
Result Type	`identifier`
Function	evaluates the expression to the identifier reference; returned an identifier atom

Expression Examples	
expression	*result*
`&alpha`	`alpha`
`refof alpha`	`alpha`

`valof` Operator

The `*` (identical to `valof`) operator is used to get the value of the identifier pointed by a reference. For instance, after the two following expressions:

```
alpha := 1;
ralpha := &alpha;
```

`*ralpha` equals `1`.

This operator may be used in the composition of a left value, for instance:

```
alpha := 1;
ralpha := &alpha;
*ralpha := 2; // now, alpha equals 2
*ralpha += 8; // now, alpha equals 10
```

But this operator is essentially used in the body of functions or methods which update one or more given parameters, for instance, the function `swap` described in the previous section is as follows:

```
function swap(x, y) {
    v := *x;
   *x := *y;
   *y := v;
}
```

General Information	
Operator	`valof` `*`
Syntax	`valof` *identifier* `*` *identifier*
Type	unary
Operand Type	`identifier`
Result Type	value of the identifier
Function	returns the value of the identifier denotes by the operand

Expression Examples	
expression	*result*
`*alpha`	if `alpha` value is an atom identifier x, returns the value of x, otherwise an error is thrown
`x := 12; alpha := &x; *x`	`*x` returns `12`

`scopeof` Operator

The `scopeof` operator returns the string `"global"` or `"local"` depending whether the identifier is global or local.

General Information	
Operator	`scopeof`
Syntax	`scopeof` *identifier*
Type	unary
Operand Type	`identifier`
Result Type	`string`
Function	returns the scope of the identifier.

Expression Examples	
expression	*result*
`scopeof ::alpha`	returns `"global"` for any `alpha` if it set; otherwise an error is thrown
`scopeof alpha`	returns `"global"` or `"local"` depending on the context.

`push` Operator

The `push` operator is used to push an identifier on a new local table. This operator is rarely used.

General Information	
Operator	`push`
Syntax	`push` *identifier* `push` *identifier* `:=` *expr*
Type	unary and binary
Operand Types	first operand `identifier`, optionnal second operand: *any type*
Result Type	identifier or any type in case of an assignment
Function	push the identifier on to the symbol table stack. An assignment can be performed at the same time. Returns the identifier or the value of the expression assignment.

Expression Examples	
expression	*result*
`push a`	pushes `a` on a new local symbol table.
`push a := 10`	pushes `a` on a new local symbol table and assigns its value to `10`

`pop` Operator

The `pop` operator is used to pop an identifier from a local table. It is used after a push. For instance:

```
a := "hello";
a;              // a equals "hello"

push a := 10;
a;              // a equals 10
```

```
pop a;
a;              // a equals "hello"
```

General Information	
Operator	`pop`
Syntax	`pop` *identifier*
Type	unary
Operand Type	`identifier`
Result Type	the type of the value of the identifier
Function	pop the identifier from the symbol table stack

Expression Examples	
expression	*result*
`pop a`	returns the value of `a` if it is set; otherwise an error is returned

5.17 Path Expressions

The path expression operator `->` (identical to `.`) is used to *navigate* from an object and read the right data one needs. This operator enables us to go inside complex objects, as well as to follow simple relationships. For instance, if `p` denotes a `Person` instance, `p.spouse` denotes the `spouse` attribute of this person.
The more complex expression `p.spouse.address.street` denotes the `street` in the `address` of `spouse` of the person `p`. This notation is very intuitive because it looks like the well known `C`, `C++` and `Java` syntaxes.

The path expression operator may composed a left value, for instance:

```
p.spouse.name := "mary";
```

set the `name` of the `spouse` of the person `p` to `mary`.

This operator may be combined with the array deferencing operators, for instance:

```
p.spouse.name[2];
p.spouse.name[2] := 'A';
p.spouse.other_addrs[2].street[3] := 'C';
p.spouse.children[?];
p.spouse.children[?].name;
```

The path expression operator may be also used to navigate through `struct` atom, for instance: `(struct(a : 1, b : "hello")).b` returns `"hello"`. Note that because of the precedence of operators, parenthesis are necessary around the literal `struct` construct.
Finally, the path operator may be applied to a collection; in this case a collection of the same type of this operand is returned. For instance:
`(select Person).name` returns a `bag` of `string`.
`(select distinct Person).age` returns a `set` of `int`.
Note that the path expression operator is used frequently in the query expressions as shown in a next section.

General Information	
Operators	`.` `->`
Syntaxes	*expr* `.` *expr* *expr* `->` *expr*
Type	binary
Operand Types	first operand: `oid` or `object`, second operand: identifier

Result Type	type of the attribute denoted by the second operand
Functions	returns the attribute value denoted by second operand of the object denoted by the first operand The first operand must denote an EyeDB instance (object or literal) of an agregat including the attribute denoted by the second operand.
Note	these two operators are identical

Expression Examples		
expression	*result*	*comments*
`p->name`	*the value of attribute* `name` *in the object denoted by* `p`	`p` must denote an EyeDB instance (object or literal) of an agregat including the attribute `name`
`first(select x Person x from x.lastname = "wayne")->lastname`	`"wayne"`	

5.18 Function Call

OQL allows one to call an OQL function with or without parameters. The operator for function call is `()`.
A function call may be the first term of a path expression, for instance: `first(select Person)->name`.
Contrary to the method invocation, there are no function overloading mechanisms: that means, that one cannot have differents functions with the same name and a different signature. To take benefit of the overloading mechanisms, one must use methods.
Note: contrary to the ODMG 3 specifications, one currently needs to use parenthesis to invoke a method even if the method has no arguments.

General Information	
Operator	`()`
Syntaxe	*expr* `(`*expr_list*`)`
Type	n-ary
Operand Types	first operand: `identifier`, other operands: *any type*
Returned type	type of the returned atom by the function call
Functions	calls the OQL function denoted by the first operand using the other operands as arguments. The number of operands must be equal to the number of arguments of the OQL function plus one

Expression Examples	
expression	*result*
`fact(10)`	`3628800`
`fact(fact(3))`	`720`
`toUpper("hello world")`	`"HELLO WORLD"`
`toUpper("hello") + "world"`	`"HELLOworld"`
`interval(1, 5)`	`list(1, 2, 3, 4, 5)`
`swap(&i, &j)`	`nil`
`first(select Person).spouse.name`	`"mary"`

5.19 Method Invocation

Instance Method Invocation

OQL allows one to call a instance method with or without parameters. The method can be written in C++ or in OQL. As in C++, method calls use a combination of the path expression operator and the function call operator.

As in `C++` or `Java`, methods can be overloaded: that means that one can have differents methods with the same name and a different signature or differents methods with the same name and the same signature in a class hierarchy. The choice of the method to invoke is done at evaluation time not at compile time. For instance let two methods `Person Person::f(in int, in int)` and `int Person::f(in float, in string)`, the method to be invoked in the expression `p->f(x, y)` is decided at evaluation time according to the true types of `x` and `y`:

```
p := first(select Person);

x := 1; y := 2;

p->f(x, y); // X::f(in int, in int) is invoked
p->f(x, y)->name; // this is valid because p->f(x, y) returns a Person

x := 1.3; y := "hello";

p->f(x, y); // X::f(in float, in string) is invoked
p->f(x, y)->name; // this is not valid because p->f(x, y) returns an integer
```

A major contribution of object orientation is the possibility of manipulating polymorphic objects and thanks to the late binding mechanism to carry out generic actions on the elements of these objects.
For instance, let the two methods `void Person::doit(in int)` and `void Employee::doit(in int)`, the method to be invoked in the expression `p->doit(x)` is decided at evaluation time according to the true type of `p`:

```
p := new Person();
p->doit(1); // Person::doit(in int) is invoked

p := new Employee();
p->doit(1); // Employee::doit(in int) is invoked
```

To invoke a method, the following conditions must be realize:

1. the object or oid on which the method is applied must be an instance of a class, for instance `X`.
2. the name of the invoked method, the number and the type of arguments must be compatible with an existing method in the class `X`,
3. the result type must match the expected type in the expression.

For instance, let the methods `int compute(in int, int float)` and `int compute(in int, in float, in int[], out string)` in the class `X`. To invoke the first method on an instance of `X`, one needs to apply the method `compute` to an instance of `X` with one integer and one float, for instance:

```
x := new X();

x.compute(1, 2.3);
x.compute(a := fact(10), float(fib(10)));
```

To invoke the second method on an instance of `X`, one needs to apply the method `compute` to an instance of `X` with an integer, a float, an ordered collection of integer and a reference to a variable, for instance:

```
x.compute(1, 23.4, list(1, 2, 3, 4), &a);
```

The following table shows the mapping (which defines the compatibility) between the ODL and the OQL types.

ODL/OQL Mapping	
ODL Type	OQL Type
`in int16`	`integer`
`out int16`	`identifier`
`inout int16`	`identifier` initialized to a*n* `integer`

`in int32`	`integer`
`out int32`	`identifier`
`inout int32`	`identifier` initialized to a*n* `integer`
`in int64`	`integer`
`out int64`	`identifier`
`inout int64`	`identifier` initialized to a*n* `integer`
`in byte`	`char`
`out byte`	`identifier`
`inout byte`	`identifier` initialized to a `char`
`in char`	`char`
`out char`	`identifier`
`inout char`	`identifier` initialized to a `char`
`in string`	`string`
`out string`	`identifier`
`inout string`	`identifier` initialized to a `string`
`in float`	`float`
`out float`	`identifier`
`inout float`	`identifier` initialized to a `float`
`in oid`	`oid`
`out oid`	`identifier`
`inout oid`	`identifier` initialized to a `oid`
`in object *`	`oid` of any class
`out object *`	`identifier`
`inout object *`	`identifier` initialized to a*n* `oid` of any class
`in X *` (`X` *denotes a class instance*)	`oid` of class `X`
`out X *` (`X` *denotes a class instance*)	`identifier`
`inout X *` (`X` *denotes a class instance*)	`identifier` initialized to a `oid` of class `X`
`in X *[]` (`X` *denotes a class instance*)	ordered collection of `oid` of class `X`
`out X *[]` (`X` *denotes a class instance*)	`identifier`
`inout X *[]` (`X` *denotes a class instance*)	`identifier` initialized to a*n* ordered collection of `oid` of class `X`
`in X[]` (`X` *denotes any ODL type*)	*ordered collection* of atoms bound to `X`
`out X[]` (`X` *denotes any ODL type*)	`identifier`
`inout X[]` (`X` *denotes any ODL type*)	`identifier` initialized to a*n* *ordered collection* of atoms bound to `X`

Note: contrary to the ODMG 3 specifications, one currently needs to use parenthesis to invoke a method even if the method has no arguments.

General Information	
Operators	`.` `->`
Syntaxes	*expr* `.` *expr* `(`*expr_list*`)` *expr* `->` *expr* `(`*expr_list*`)`
Type	n-ary
Operand Types	first operand: `oid` or `object`, second operand: identifier, other operands: any type
Result Type	type of the atom returned by the method call
Functions	invokes the method denoted by the second operand applied to the object denoted by the first operand, using the other operands as arguments.

	The first operand must denote an EyeDB instance (object or literal) of an agregat including the method whose name is the second operand. The number of arguments and the type of arguments must match one of the methods included in the class of the object denoted by the first operand.
Note	these two operators are identical

Expression Examples		
expression	*result*	*comments*
`p->getOid()`	*the value of the oid of object denoted by* `p`	as `getOid()` is a native method of the class `object`, each object can call this method
`img->compute(1, 2.3)`	*the value returned by the method call*	the first operand must denote an EyeDB instance (object or literal) of an agregat including the method whose name is `compute`
`first(select Person.name = "wayne").getSpouse()`	*the value returned by the method call*	

Class Method Invocation

OQL allows one to call a class method with or without parameters. The method can be written in C++ or in OQL. As in C++, method calls use a combination of the scope operator and the function call operator. To invoke a class method, the following conditions must be realize:

1. the name of the invoked method, the number and the type of arguments must be compatible with an existing method in the class `X`,

2. the result type must match the expected type in the expression.

The overloading and the late binding mechanisms are the same as for the instance method invocations.

General Information	
Operator	`::`
Syntaxe	`identifier::identifier(`*expr_list*`)`
Type	n-ary
Operand Types	first operand: `identifier`, second operand: `identifier`, other operands: *any type*
Result Type	type of the atom returned by the method call
Functions	invokes the class method denoted by the second operand applied to the class denoted by the first operand, using the other operands as arguments. The first operand must denote an EyeDB class of an agregat including the class method whose name is the second operand. The number of arguments and the type of arguments must match one of the class methods included in the class denoted by the first operand.

Expression Examples		
expression	*result*	*comments*
`EyeDB::getVersion()`	`2.8.0`	`getVersion()` is a native static method of the class `EyeDB`
`Person::checkName("johnny")`	*the value returned by the method call*	the class method `checkName` must exist in the class `Person` and must take one and only one input string argument.

5.20 Eval/Unval Operators

`eval` Operator

One major feature of OQL is that one can invoke its evaluator using the `eval` operator. This allows us to build OQL constructs at runtime and perform their evaluation. This is very useful, for instance, when we want to build a query expression where the projection or the `from` reference sets are is unknown. For instance, the following function allows us to retrieve the values of the attribute *attrname* in the class *classname*:

```
function getValues(classname, attrname) {
  cmd := "select x." + attrname + " from " + classname + " x";
  return (eval cmd);
}
```

General Information	
Operator	`eval`
Syntaxe	`eval string`
Type	unary
Operand Types	`string`
Functions	calls the OQL evaluator on the string operand. The string operand can contain any OQL valid construct: an expression, a statement or a sequence of statements.

Expression Examples	
expression	*result*
`eval "10"`	`10`
`eval "a := 100"`	result is `100`; the variable `a` is set to `100`
`eval "a := \"hello\"; b := a + \"world\""`	result is `"hello world"`; the variable `a` is set to `"hello"`; the variable `b` is set to `"hello world"`

`unval` Operator

The `unval` is the inverse of the `unval` in the sense that it takes any valid OQL expression and returns the string representation; the comments and, when not necessary, the spaces and tabulations are skipped. For instance, the construct `unval a := 10` returns `"(a:=10)"`.

General Information	
Operator	`unval`
Syntax	`unval` *expr*
Type	unary

Operand Types	*any type*
Functions	returns the string expression

Expression Examples	
expression	*result*
`unval 10`	`"10"`
`unval alpha += 10 - beta + 1`	`"(alpha:=(alpha+((10-beta)+1)))"`
`eval unval alpha := "hello"`	returns `"hello"`; `alpha` is set to `"hello"`

5.21 Set Expressions

OQL allows us to perform the following operations on `set`s and `bag`s: union, intersection, difference and inclusion. The operands can be sets or bags. For all these operators, when the operand's collection types are different (`bag` and `set`), the set is first converted to a bag and the result is a bag.

`union` Operator

The `union` operator performs the union of two `set`s or `bag`s. This operator has the same precedence as the logical or operator.

General Information	
Operator	`union`
Syntax	*expr* `union` *expr*
Type	binary
Operand Types	`set` or `bag`
Result Type	`set` if both two operands are of type `set`, `bag` otherwise
Functions	returns the union of the two operands.

Expression Examples	
expression	*result*
`set(1, 2) union set(2, 3)`	`set(1, 2, 3)`
`set(1, 2) union bag(2, 3)`	`bag(1, 2, 2, 3)`
`list(1, 2) union bag(2, 3)`	*raises an error*

`intersect` Operator

The `intersect` operator performs the intersection of two `set`s or `bag`s. This operator has the same precedence as the logical and operator.

General Information	
Operator	`intersect`
Syntax	*expr* `intersect` *expr*
Type	binary
Operand Types	`set` or `bag`
Result Type	`set` if both two operands are of type `set`, `bag` otherwise
Functions	returns the intersection of the two operands.

Expression Examples

expression	*result*
`set(1, 2) intersect set(2, 3)`	`set(2)`
`set(1, 2) intersect bag(2, 3)`	`bag(2)`
`bag(1, 2, 2, 3) intersect bag(2, 3, 2)`	`bag(2, 2, 3)`
`list(1, 2) intersect bag(2, 3)`	*raises an error*

`except` Operator

The `except` operator performs the difference between two `set`s or `bag`s. This operator has the same precedence as the logical or operator.

General Information	
Operator	`except`
Syntax	*expr* `except` *expr*
Type	binary
Operand Types	`set` or `bag`
Result Type	`set` if both two operands are of type `set`, `bag` otherwise
Functions	returns the difference of the two operands.

Expression Examples	
expression	*result*
`set(1, 2) except set(2, 3)`	`set(1)`
`set(1, 2) except bag(2, 3)`	`bag(1)`
`set(1, 2, 10) except bag(12)`	`bag(1, 2, 10)`
`list(1, 2) except bag(2, 3)`	*raises an error*

Inclusion Operators

The inclusion operators for `set`s and `bag`s are the comparison operators *less than/greater than* introduced in a previous section.

General Information	

Operator	`<`
	`<=`
	`>`
	`>=`

Syntax	*expr* `<` *expr* *expr* `<=` *expr* *expr* `>` *expr* *expr* `>=` *expr*
Type	binary
Operand Types	`set` or `bag`
Result Type	`boolean`
Functions	*coll1* `<` *coll2* : returns `true` if and only if *coll1* is included in *coll2* but not equal to *coll2* *coll1* `>` *coll2* : returns `true` if and only if *coll2* is included in *coll1* and not equal to *coll1* *coll2* `<=` *coll1* : returns `true` if and only if *coll1* is included in *coll2* or equal to *coll2* *coll1* `>=` *coll2* : returns `true` if and only if *coll2* is included in *coll1* or equal to *coll1*

Expression Examples

expression	*result*
`set(1, 2) < set(2, 3)`	`false`
`set(1, 2) < set(2, 3, 1)`	`true`
`set(1, 2) < bag(2, 3, 1)`	`true`
`set(1, 2) <= bag(2, 1)`	`false`
`set(1, 2) >= bag(2, 1)`	`false`

5.22 Object Creation

OQL allows us to create persistent or transient objects using the `new` operator. The general syntax for an object creation is as follows:
[`new`] [`<`[*expr*]`>`] *class_name*`({`*path_expression* `:` *expr* `})`

1. the operator `new` is optionnal: when the operator is missing, the construct is called an implicit `new` construct. When the optionnal following construct "`<`[*expr*]`>`" is not used, there is no functionnal differences between using `new` or not.
2. the optionnal construct after the `new` operator indicates the target location of the object to create:
   (a) if this construct is omitted, the object will be a persistent object created in the current default database,
   (b) if this construct is under the form `<`*expr*`>`, the OQL interpreter expects for a database object handle as the result of the expression evaluation. This database will be used as the target location. For instance:

   ```
   new < oql$db > Person();
   ```

   will create a `Person` instance in the database pointed by `oql$db`, which is in fact the current database.
   (c) if this construct is under the form `<>`, the object will be a transient object.
3. the *class_name* indicates the name of a valid user type in the context of the current database.
4. the *path_expression* indicates an attribute name or a sequence of attributes using the optional array operator, for instance the following path expressions are valid for an object construction:

   ```
   name
   lastname
   addr.street
   addr.town[3]
   spouse.name
   ```

5. the *expr* behind *path_expression* is any OQL expression as soon as its result type matches the expected type of the *path_expression*.
6. the order of evaluation of the expressions is in the {*path_expression* `:` *expr*} sequence is from left to right.
7. the expression returns the oid of the created object.

For instance:

- `new Person()` creates a person with all its attributes unitialized,
- `Person()` creates a person with all its attributes unitialized,
- `new Person(name : "john")` creates a person with its attribute `name` initialized to `john`,
- `Person(name : "john")` creates a person with its attribute `name` initialized to `john`,
- `new Person(name : "john", age : 32, spouse : new Person(name : "mary"))` creates a person named `mary` and a person named `john`, age `32` whose `spouse` is the person `mary`.

The `new` operator can also be used to create basic type object. Note that in this case, the operator is mandatory. The syntax for basic type creation is as follows:
`new` [`<`[*expr*]`>`] *basic_type* `(`*value*`)`.

1. where *basic_type* denotes an ODL basic type. It may be one of the following type: `int32`, `int16`, `int32`, `char`, `byte`, `float` or `oid`. Note that the type `string` is not allowed here.

2. the *value* must be an atomic value of an OQL type mapped from the ODL basic type

For instance:

- `new int(2)`
- `new float(2.3)`
- `new float(2)`
- `new char('a')`
- `new oid(first(select Person))`

Finally, the `new` operator can be also used to create collections. The syntax for collection creation is as follows:
[`new`] [`<`[*expr*]`>`] *coll_type*`<` *class_name* [, *coll_name]*`>` `(`[*collection of elements*]`)`

1. the `new` operator is optionnal,
2. where *coll_type* denotes type of the collection: `set`, `bag`, `array` or `list`,
3. the *class_name* denotes the name of the class of the elements of the collection, for instance `Person*`, `Car*`,
4. *coll_name* is an optionnal string which denotes the name of the collection to create,
5. the optionnal *collection of elements* within parenthesis contains the elements (generally oids) to insert initially in the created collection,
6. the expression returns the oid of the created collection.

For instance:

- `new set<Person *>()` creates an empty `set` of persons,
- `new set<Person *>(list(select Person))` creates a `set` containing all the persons in the database,
- `new set<Person *, "all babies">(list(select Person.age < 1))` creates a `set` named `all babies` containing all the persons whose age is less than `1`.
- `new array<Car *>()` creates an empty `array` of cars.
- `new array<int>(list(1, 2, 3, 4)` creates an `array` of integers initially containing `1, 2, 3, 4`.
- `new set<int *> (list(new int(2), new int(10)))` creates a `set` of integer objects containing initially two integer objects.

General Information	
Operator	`new`
Syntax	`new` [`<`[*expr*]`>`] *class_name*`(`{*path_expression* `:` *expr* }`)`
Type	n-ary
Operand Types	*any type*
Result Type	`oid` or `object`
Functions	creates an persistent or transient object

Expression Examples	
expression	*result*

`john := new Person(name: "john",`
`lastname: "wayne",`
`age : fib(10));`
	returns the oid of the created `Person` instance

`new Person(name: "mary",`
`lastname: "poppins",`
`addr.town : "jungle",`
`addr.street[0] : 'a',`
`addr.street[1] : 'b',`
`spouse : john,`
`spouse.age : 72`
`);`
	returns the oid of the created `Person` instance

5.23 Object Deletion

The `delete` unary operator is used to delete persistent objects.

General Information	
Operator	`delete`
Syntax	`delete` *expr*
Type	unary
Operand Types	`oid` or `object`
Result Type	the operand type
Functions	delete a transient or persistent object

Expression Examples	
expression	*result*
`delete first(select Person)`	the oid of the deleted `Person` instance
`delete new Person()`	the oid of the deleted `Person` instance
`for (x in (select Person) delete x`	the oids of the deleted `Person` instances

5.24 Collection Expressions

OQL introduces a few operators for object collection manipulation: one of them is the array deferencing operator "`[]`" (5.15) that is overloaded for ordered collection manipulation.
Some them are ODMG OQL compliant, the others are EyeDB extensions. Object collections may be persistent or transient, orderered or not. These operators allows us to make the following kind of operations:

1. gets the contents of a collection: operator `contents`,
2. get an element at a given position in an ordered collection: operator `[]` (ODMG compliant),
3. get elements at some given positions in an ordered collection: operators `[:]` (ODMG compliant) and `[?]`,
4. checks if an element is in a collection: operator `in` (ODMG compliant),
5. add an element in an unordered collection: operator `add/to`
6. suppress an element from an unordered collection: operator `suppress/from`,
7. set or suppress an element in an ordered at a given position: operator `[]`,
8. set or suppress elements in an ordered at a given position: operators `[:]` and `[?]`,
9. append an element in an ordered collection: operator `append/to`,
10. checks if a given condition is realized for at least one element in a collection: operator `in` (ODMG compliant),
11. checks if a given condition is realized for all elements in a collection: operator `for/all` (ODMG compliant),

12. checks if a given condition is realized for a given number range of elements in a collection: operator extended `for`.

In all the following examples, the OQL variables `p0` denotes the first `Person` instance in the database: `p0 := first(select Person)`.

Important Note: although they are reference in the following descriptions of collection operators, the object collections `list` are not implemented in the current version of EyeDB.

`contents` Operator

The `contents` unary operator is used to give the contents of a given ordered or unordered object collection. It returned an OQL collection of the same type of the object collection: `set`, `bag`, `array` or `list`.

General Information	
Operator	`contents`
Syntax	`contents` *expr*
Type	unary
Operand Types	`oid` or `object` collection
Result Type	a collection of objects
Functions	returns the contents of an object collection

Expression Examples	
`contents(p0.children)`	an `array` of `Person` `oid`s
`select contents(x.children) from Person x`	returns a `list` of `array`s of `Person` `oid`s.
`contents(list(1, 2, 3))`	*raises an error*: `oid or object expected, got list`

`in` Operator

The `in` operator is used to check if a given element is in ordered or unordered object collection.

General Information	
Operator	`in`
Syntax	*expr* `in` *expr*
Type	binary
Operand Types	first operand: *any type*, second operand: `oid` or `object` collection
Result Type	`boolean`
Functions	returns `true` if the first operand belongs to the collection pointed by the second operand; `false` otherwise

Expression Examples	
`first(select Person.name = NULL) in p0.children`	returns true if the first `Person` instance whose `name` is unitialized is in the `array` of `children` of the first `Person`
`first(select Car.brand = "renault") in p0.cars`	returns true if the first `Car` instance whose `brand` equals `renault` is in the `set` of `cars` of the first `Person`

`add/to` Operator

The `add/to` operator is used to add an element in an unordered collection (`set` or `bag`).

General Information	
Operator	`add/to`
Syntaxes	`add` *expr* `to` *expr*
Type	binary
Operand Types	first operand: *any type*, second operand: `oid` or `object` unorderered collection (`set` or `bag`)
Result Type	type of the first operand
Functions	adds the first operand to the non-indexed collection (i.e. bag or set) pointed by the second operand; returns the first operand.

Expression Examples	
`add new Car(num : 100) to p0.cars`	returns the created `Car` oid
`add new Person(name : "john") to p0.children`	*raises an error*: `cannot used non indexed insertion in an array`
`add new Car() to new set<Car *>()`	returns the just created car; but we have *lost* the oid of the just created `set` of cars!
`add new Person() to (c := new bag<Person *>())`	returns the just created person; the created bag has been kept in the OQL variable `c`

`[]` Operator

The polymorphic `[]` operator is used to set or get an element in an ordered collection (`array` or `list`) at a given position: it can be used in a right or left value.

General Information	
Operator	`[]`
Syntaxes	*expr* `[` *expr* `]`
Type	binary
Operand Types	first operand: collection `array` or `list`, second operand: `integer`
Result Type	the type of the element,
Functions	gets the element in the collection pointed by the first operand at the position pointed by the second operarand. If used at a left value, gets a reference to that element.

Expression Examples	
`p0.children[0]`	returns the child at position #0 in `p0.children` collection. Returns `nil` if there is no child at this position
`p0.children[0] := Person(name : "john")`	returns the created `Person` oid
`p0.children[12039] := Person(name : "henry")`	returns the created `Person` oid. This expression is valid in any cas as the collection `array`s automatically increased its size
`(array<Person *>())[0] := new Car(num : 100)`	returns the just created person; but the created array has been "*lost*" as it is not tied to any instance and as we did not bind it to any OQL variable

`(c := array<Person *>())[0] := new Car(num : 100)`	returns the just created person; the created array has been kept in the OQL variable `c`
`p0.cars[1] := Car(num : 100)`	*raises an error*: `array expected, got set`

`[:]` Operator

The polymorphic `[:]` operator is used to set or get some elements in an ordered collection (`array` or `list`) at some given positions: it can be used in a right or left value. When used in a right value, the returned atom is a `set` of `struct` with the two attributes `index` and `value`. In each `struct` element returned, the value of `index` is the position of the element, the value of `value` is the element value. Note the returned `struct` elements are not ordered according to the element postions; it is why a `set` is returned. When used as a left value, the returned atom is a `set` of references on the elements .

General Information	
Operator	`[:]`
Syntaxes	*expr* `[` *expr* `:` *expr* `]`
Type	ternary
Operand Types	first operand: collection `array` or `list`, second and third operands: `integer`
Result Type	the type of the element,
Functions	gets the elements in the collection pointed by the first operand at the position range pointed by the second and third operarands. If used at a left value, gets references to that elements.

Expression Examples	
`p0.children[0:1]` Returns `nil` if there is no child at these positions	returns a set of `struct` including the children and the position of the children position #0 and #1 in the `p0.children` collection. For instance: `set(struct(index : 0, value : 3874.33.293847:oid), struct(index : 1, value : 2938.33.1928394:oid))`
`p0.children[0:4] := Person(name : "john")` returns the created `Person` oid	Sets all the children at the position #0 to #4 to a new `Person` instance.
`p0.children[12000:12039] := Person(name : "henry")`	returns the created `Person` oid. This expression is valid in any cas as the collection `array`s automatically increased its size
`(array<Person *>(list(Person())))[0]`	returns the just created person within the just created array. But the array is "*lost*" as it is not tied to any instance and as we did not bind it to any OQL variable
`(x := array<Person *>(list(Person())))[0]`	returns the just created person; the created array has been kept in the OQL variable `c`

`[?]` Operator

The polymorphic `[?]` operator is used to set or get all the elements in an ordered collection (`array` or `list`). It can be used in a right or left value. When used in a right value, the returned atom is a `set` of `struct` with the two attributes `index` and `value`. In each `struct` element returned, the value of `index` is the position of the element, the value of `value` is the element value. Note the returned `struct` elements are not ordered according to the element postions; it is why a

`set` is returned.
When used as a left value, the returned atom is a `set` of references on the elements .

General Information	
Operator	`[?]`
Syntaxes	*expr* `[?]`
Type	unary
Operand Type	collection `array` or `list`,
Result Type	a set of `struct` or a set of references
Functions	gets all the elements in the collection pointed by the first operand If used at a left value, gets references to that elements.

Expression Examples	
`p0.children[?]` Returns `nil` if the collection is empty	returns a set of `struct` including the children and the position of all the children in the `p0.children` collection. For instance: `set(struct(index : 0, value : 3874.33.293847:oid), struct(index : 1, value : 2938.33.1928394:oid))`
`p0.children[?] := Person(name : "john")` returns the created `Person` oid	Sets all the children to a new `Person` instance.
`(array<Person *>(list(Person(), Person())))[?]`	returns a `set` of `struct` including the just created `Person` instances in the just created array.

`append/to` Operator

The `append/to` operator is used to append an element to an ordered collection (`list` or `array`).

General Information	
Operator	`append`
Syntaxes	`append` *expr* `to` *expr*
Type	binary
Operand Types	first operand: *any type*, second operand: `oid` or `object` denoting an ordered collection
Result Type	*any type*
Functions	appends the element denoted by the first operand to the indexed collection (i.e. list or array) denoted by the second operand.

Expression Examples	
`append Person() to p0.children`	the created `Person` instance
`append Car()to p0.cars`	*raises an error*: `array or list expected, got set<Person*>`

`suppress/from` Operator

The `suppress/from` operator is used to suppress an element from an ordered collection (`set` or `bag`).

General Information	
Operator	`suppress/from`
Syntaxes	`suppress` *expr* `from` *expr*
Type	binary
Operand Types	first operand: *any type*, second operand: `oid` or `object` collection
Result Type	type of the first operand
Functions	suppress the first operand from the non-indexed collection (i.e. bag or set) pointed by the second operand; returns the first operand.

Expression Examples	
`suppress (select Car.num = 1000) from p0.cars`	the suppressed car if it was found in the collection; otherwise, *raises an error*
`suppress new Car() from p.cars`	*raises an error*: `item '71238.13.3959935:oid' not found in collection`
`suppress p0 from p0.children`	*raises an error*: `cannot used non indexed suppression in an array`

`empty` Operator

The `empty` operator is used to empty an ordered or an unordered collection.

General Information	
Operator	`empty`
Syntax	`empty` *expr*
Type	unary
Operand Types	`oid` or `object` collection
Result Type	`nil`
Functions	empty the collection pointed by the operand

Expression Examples	
`empty(first (select Person).children)`	`nil`
`empty(first (select Person).cars)`	`nil`
`empty new set<Car *>(list(new Car()))`	`nil`; this expression creates a collection of `Car` containing initially a new `Car`, and empty it!

`in` Operator

The `in` operator is used to check if a given condition is realized for at least one element in an ordered or unordered collection.

General Information	
Operator	`in`
Syntax	*identifier* `in` *expr* `:` *expr*
Type	ternary
Operand Types	first operand: `identifier`, second operand: `oid` or `object` collection, third operand: boolean
Result Type	`boolean`

Functions	returns `true` if it exists in the collection pointed by the second operand an element for which the third operand is evaluated to `true`.

Expression Examples	
`x in p0.children: x.name = "mary"`	`true` or `false`
`x in p0.cars: x.num < 100 and x.num >= 90`	`true` or `false`

`for` Operator

The `for/all` operator is used to check if a given condition is realized for all elements in an ordered or unordered collection. This operator is ODMG compliant.

General Information	
Operator	`for/all`
Syntaxes	`for all` *identifier* `in` *expr* `:` *expr*
Type	ternary
Operand Types	first operand: `identifier`, second operand: `oid` or `object` collection, third operand: boolean
Result Type	`boolean`
Functions	returns `true` if for all items contained in the collection pointed by the second operand the third operand is evaluated to `true`.

Expression Examples	
`for all x in p0.children: x.name == "john"`	`true` or `false`
`for all x in p0.cars: x.num % 10`	`true` or `false`

The `for`/*cardinality* operator is used to check if a given condition is realized for a given number range of elements in an orderer or unordered collection. Note that this operator is the generalisation of the `in` and `for/all` operators:
`for <0:$>` is equivalent to `in`
`for <$>` is equivalent to `for all`

General Information	
Operator	`for`*cardinality*
Syntaxes	`for <` *expr* `:` *expr* `>` *identifier* `in` *expr* `:` *expr* `for <` *expr* `>` *identifier* `in` *expr* `:` *expr*
Type	5-ary
Operand Types	first and optional second operands: `integer` or `$`, where `$` denotes the collection cardinality, third operand: `oid` or `object`, fourth operand: `identifier`, fifth operand: `boolean`
Result Type	`boolean`
Functions	returns `true` if the number of items in the collection pointed by the third operand for which the third operand is evaluated to `true` is in the interval `[`first operand`,` second operand`]`.

Expression Examples	
`for <0:4> x in p0.children:  x.name == "john"`	`true` if at most 4 children have their name equals to `john`
`for <4> x in p0.children:  x.name == "john"`	`true` if one an only one children have its name equals to `john`
`for <0:$> x in p0.cars:  x.num = 10`	equivalent to `in`
`for <$> x in p0.cars:  x.num = 10`	equivalent to `for/all`

5.25 Exception Expressions

Currently, EyeDB OQL does not provide full support for exception management as there is no `try/catch` operator. Nevertheless, the `throw` operator allows us to raise an error message, for instance:

```
if (!check(p))
  throw "variable p is not correct".
```

The `throw` operator stops the current thread of statements and returns the error message at the uppest level. In the following code:

```
a := 1;
throw "this is an error";
b := 2;
```

the variable `a` will be assigned to `1`, but the variable `b` will not be assigned to `2` as the `throw` expression deroutes the normal thread of statements. The `throw` operator is often used in the body of functions,

General Information	
Operator	`throw`
Syntax	`throw` *expr*
Type	unary
Operand Type	`string`
Result Type	`nil`
Functions	raises an error message

Expression Examples	
`throw "error message"`	`nil`
`throw "error #1:  " + msg`	`nil`

5.26 Function Definition Expressions

As introduced previously, OQL supports functions. There are two types of functions definition syntax: function definition expression and function definition statements. The first ones, exposed in this section, are more restrictive than the second ones, as their definition can contain only one expression. The second ones contain an unbounded sequence of statements. The function definition expressions are ODMG compatible and are called *Named Query Definition* in the ODMG standard. To define such a function, one must use the operator `define/as`. The general syntax for a definition function expression is:

`define` *identifier* [`(`*arglist*`)`] `as` *expr*.

For instance:

```
define div2(x) as x/2;
div2(10); // returns 5

define pimul(x) as x * 3.1415926535;
pimul(23); // returns 72.256631

define getOnePerson(name) as first(select Person.name = name);
getOnePerson("john"); // returns an oid or nil
```

As the last operand of the `define/as` operator is any OQL expression, it can be a sequence of expressions by using the comma sequencing operator. Therefore, the following construct is valid:

```
define getit(age) as ::getit_call++,
                     list_of_persons := select Person.age >= age,
                     (count(list_of_persons) > 0 ? list_of_persons[0] : nil);

getit(10); // returns the first Person whose age is greater or equal to
              10, or nil
getit_call; // equals 1
list_of_persons; // raises an error: uninitialized identifier 'list_of_persons

getit(20);
getit_call; // equals 2
```

Several comments about this code:

1. `::getit_call` denotes a global variable, while `list_of_persons` denotes a variable local to the function: the variable scoping has previously been introduced in the identifier expressions, and will be explained again in the function definition statements Section.

2. as there are no iteration expression (for instance a `for` or `while` expression) and as a function definition expression can contain only one expression, one cannot use a function definition expression with iteration statements. To use an iteration statement, one needs to use a function definition statement.

3. when one needs to define a function with a sequence of expressions, it may be easier and more readable to use a function definition statement instead of a function definition expression. The *statement* version of the previous function is:

   ```
   function getit(age) {
     ::getit_call++;
     list_of_persons := select Person.age >= age;
     if (count(list_of_persons))
       return list_of_persons[0];
   }
   ```

   which is - for `C`, `C++` or `Java` programmers - a more natural construct.

Finally, the function definition allows us for recursive definitions, for instance:

```
define fib(n) as (n < 2 ? n : fib(n-2) + fib(n-1));
fib(10); // returns 55
```

General Information	
Operator	`define/as`
Syntax	`define` *identifier* [*arglist*] `as` *expr*
Type	n-ary
Operand Type	first operand: `identifier`, last operand: *any type*, other operands: `identifier`
Result Type	`identifier`
Functions	defines a function

Expression Examples	
`define Persons as select Person`	`Persons`
`define fact(n) as (n < 2 ? n : n * fact(n-1))`	`fact`

5.27 Conversion Expressions

The conversion unary operators `string`, `int`, `char`, `float`, `oid` and `ident` allows us to make the following conversions:

Operator	*From*	*To*	*Returned Atom*
`string`	*any type*	`string`	the string representation of the operand
`int`	`int`	`int`	the `int` operand
	`char`	`int`	the operand casted to an `int`
	`float`	`int`	the operand casted to an `int`
	`string`	`int`	the operand converted to an `int`
`char`	`char`	`char`	the `char` operand
	`int`	`char`	the operand casted to a `char`
	`float`	textttchar	the operand casted to a `char`
	`string`	`char`	the operand converted to a `char`
`float`	`float`	`float`	the `float` operand
	`char`	`float`	the operand casted to a `float`
	`int`	`float`	the operand casted to a `float`
	`string`	`float`	the operand converted to a `float`
`oid`	`oid`	`oid`	the `oid` operand
	`string`	`oid`	the `string` operand converted to an `oid`
`ident`	`ident`	`ident`	the `ident` operand
	`string`	`ident`	the `string` operand converted to an `ident`

These operators are used to perform an explicit conversion such as convert the string `"123"` to the integer `123`, or to perform an explicit cast for numbers such as casting the integer `10` to the float `10.0`. These operators evaluate first their operand before performing the conversion. If the operand type is valid, no error is raised even if its format is not valid, for instance: `int "alpha"` returns `0`, while `oid "aoaoai"` returns `NULL`. Note that because of the precedence of these operators, parenthesis are necessary to make a conversion of a non-primary operand. For instance, `string 1+2` is not valid: you should use `string (1+2)`.

`string` operator

The `string` operator evaluates its operand and returns its string representation.

General Information	
Operator	`string`
Syntax	`string` *expr*
Type	unary
Operand Type	any type
Result Type	`string`
Function	returns the string representation of any atom

Expression Examples	
`string 123.3`	`"1203.300000"`
`string 'a'`	`"a"`
`string first(select Person)`	`"71211.13.1486847:oid"`
`string &alpha`	`"::alpha"`
`string list(1, 2, 3+2)`	`"list(1, 2, 5)"`
`string (list("hello", 30) + list(10))`	`"list("hello", 30, 10)"`
`string (1+3)`	`"4"`
`string 1+3`	*raises an error*

`int` operator

The `int` operator evaluates its operand and converts or casts it to an integer.
If the operand is the string, it converts it using the `atoi` C function. If the string is not a valid integer, it returns a `0`.
If the operand is a char or float, it casts it to an integer.
If the operand is an integer, it returns it.

General Information	
Operator	`int`
Syntax	`int` *expr*
Type	unary
Operand Type	`int`, `char`, `float` or `string`
Result Type	`int`
Function	returns the integer conversion or cast of the operand

Expression Examples	
`int 123.3`	`123`
`int 12`	`12`
`int 'a'`	`97`
`int "123"`	`123`
`int ("123" + "12")`	`12312`
`int alpha`	the value of `alpha` converted or casted to an integer
`int list(1, 2, 3)`	*raises an error*

`char` operator

The `char` operator evaluates its operand and converts or casts it to a `char`.
If the operand is the string of length one, it returns the character of this string. If the string has several characters, it returns a `'\000`.
If the operand is a integer or float, it casts it to a character.
If the operand is a character integer, it returns it.

General Information	
Operator	`char`
Syntax	`char` *expr*
Type	unary
Operand Type	`int`, `char`, `float` or `string`
Result Type	`char`
Function	returns the character conversion or cast of the operand

Expression Examples	
`char 123.3`	{
`char 'a'`	`'a'`
`char alpha`	the value of `alpha` converted or casted to a character
`char "a"`	`'a'`
`char "hello"`	`'^@'`
`char list(1, 2, 3)`	*raises an error*

`float` operator

The `float` operator evaluates its operand and converts or casts it to a float.
If the operand is the string, it converts it using the `atof` C function. If the string is not a valid float, it returns a `0.0`.
If the operand is a integer or float, it casts it to a float.
If the operand is a float, it returns it.

General Information	
Operator	`float`

Syntax	`float` *expr*
Type	unary
Operand Type	`int`, `char`, `float` or `string`
Result Type	`float`
Function	returns the float conversion or cast of the operand

Expression Examples	
`float 123.0`	`123.0`
`float 123.3`	`123.3`
`float 'a'`	`97.000`
`float "123.0000000"`	`123.0`
`float ("123." + "12")`	`123.12`
`float "hello"`	`0.0`
`float alpha`	the value of `alpha` converted or casted to a float
`float list(1, 2, 3)`	*raises an error*

`oid` operator

The `oid` operator evaluates its string operand and returns the corresponding oid. If the string does not denote a valid oid, the `NULL` oid is returned.
If the operand is an oid, it returns it.

General Information	
Operator	`oid`
Syntax	`oid` *expr*
Type	unary
Operand Type	`oid` or `string`
Result Type	`oid`
Function	returns the oid denoted by the string operand

Expression Examples	
`oid "234.34.33:oid"`	`234.34.33:oid`
`oid 234.34.33:oid`	`234.34.33:oid`
`oid first(select Person)`	returns the first person oid
`oid 'a'`	*raises an error*

`ident` operator

The `ident` operator evaluates its string operand and returns the corresponding identifier. If the operand is an identifier, it returns it.

General Information	
Operator	`ident`
Syntax	`ident` *expr*
Type	unary
Operand Type	`ident` or `string`
Result Type	`string`
Function	returns the identifier denoted by the string operand

Expression Examples	
`ident "alpha"`	`alpha`
`ident "alpha#1x"`	`alpha#1x`
`ident "alpha" := 123`	`123`, `alpha` has been assigned to `123`
`valof &(ident "alpha")`	`123`
`ident 'a'`	*raises an error*

5.28 Type Information Expressions

OQL provides two type information unary operators: `typeof` and `classof`. The first one takes any operand type, while the second one takes an oid or an object operand. Note that because of the precedence of these operators, parenthesis are necessary to get type information about a non-primary operand. For instance, `typeof 1+2` is not valid: you should use `typeof (1+2)`.

`typeof` operator

The `typeof` operator is used to get the type of any OQL atom. It evaluates its operand and returns the string type of its operand. For instance: `typeof 1` returns `"int"` while `typeof "hello"` returns `"string"`.

General Information	
Operator	`typeof`
Syntax	`typeof` *expr*
Type	unary
Operand Type	any type
Result Type	`string`
Function	returns the type of the atom

Expression Examples	
`typeof "alpha"`	`"string"`
`typeof (1+20.)`	`"float"`
`typeof list(1, 2, 3)`	`"list"`
`typeof first(select Person)`	`"oid"`
`typeof 1+3049`	*raises an error*
`typeof alpha`	type of the value of `alpha`
`typeof &alpha`	`ident`

`classof` operator

`typeof` operator

The `classof` operator is used to get the class of any oid or object. It evaluates its operand and returns the string class of its operand. For instance: `classof first(select Person)` returns `"Person"` while `typeof new Car()` returns `"Car"`.

General Information	
Operator	`classof`
Syntax	`classof` *expr*
Type	unary
Operand Type	`oid` or `object`
Result Type	`string`
Function	returns the class of the operand

Expression Examples

`classof first(select class)`	`"basic_class"`
`classof (c := new Car(num : 10))`	`"Car"`
`classof NULL`	`""`
`classof first(select Person).spouse`	`"Person"` or `NULL`
`classof 1`	*raises an error*

5.29 Query Expressions

The `select` operator is used to perform queries in a database. The general syntax of this operator is as follows:

`select` [`distinct`] *projection* [`from` *fromList* [`where` *predicat*] [`order by` *orderExprList* [`asc|desc`]]

1. `distinct` means that duplicates must be eliminated,
2. *projection* is an expression using the variables defined in the *fromList*,
3. *fromList* is a sequence of comma-separated items under one of the following forms:
   var `in` *expr*
   expr `as` *var*
   expr var
   where *expr* is an expression of type `collection` or is a class name, and *var* is the name of a variable.
4. *predicat* is a boolean expression using the variables defined in the *fromList*,
5. *orderExprList* is a comma-separated list of sortable expressions (i.e. atomic type).
6. `asc` means that the order should be performed in an ascendant way (the default) and `desc` means the inverse.

ODMG vs. EyeDB OQL Query Expressions

As explained in the Section *OQL vs. ODMG 3 OQL*, there are a few differences between ODMG OQL and EyeDB OQL query expressions:

- the `having/group` clause is not supported in the current implementation.
- in the current implementation, one cannot use implicit `from` clause (i.e. `from` clause without variables). ODMG OQL supports constructs such as: `from Person` without any variable. This implementation does not.
- contrary to ODMG OQL, the `from` clause is optionnal, A `select` expression which does not use the `from` is called an implicit `select` expression.
- in a `from` clause such as "`x in expr`", "`expr`" can be the name of a class. In this case, the interpreter understand this as the extent of this class. ODMG OQL does not support that.
- the SQL specific agregate operators `min(*)`, `max(*)`, `count(*)`, `sum(*)` and `avg(*)` are not supported
- the `order by` clause is more restrictive than in the ODMG OQL specifications (see below).
- the `select *` is not supported in the current implementation.

The general `select` syntax

Rather than introducing the `select` operator in a formal way by using a lot mathematical symbols, we introduce it first, in an unformal way, and then, through query examples.

The unformal description of the query process is as follows:

1. The `from` clause determine the sets of objects on which the query will be applied. For instance, in the `from` clause, "`x in Person, y in x.children`", the sets on which the query will be applied are all the `Person` instances bound to the variable `x` and the children of these instances, bound to the variable `y`.
2. These sets of objects are filtered by retaining only the objects that satisfy the predicat in the `where` clause. These result objects are gathered into a `bag`. If no `where` clause is there, all objects are retained.
3. If an `order by` clause is present, a sort is performed using to the following process:
   (a) each order expression must be of a sortable type: number (`int`, `char` or `float`) or `string`. If not, an error is raised.

(b) the `bag` is ordered into a `list` according to the first order expression. Then identical atoms are ordered again using the second order expression and so on.

(c) there is a restriction in the current implementation: each expression in the *orderExprList* must be present in the *projection* expression. If not present, an error is raised.

4. The *projection* expression is evaluated for each object in the `collection` and the results of these evaluations are gathered into a `bag`.

5. If the keyword `distinct` is there, the eventual duplicates are eliminated.

6. Finally, if the `order by` clause is present, the result `bag` is converted to a `list`; if the `distinct` keyword is there without an `order by`, the `bag` is converted to a `set`; and if neither `order by` nor `distinct` are used, we get a `bag`.

The following table presents several examples:

Simple `select/from` Examples
`select x from Person x` returns a `bag` containing all the `Person` `oid`s in the database
`select x.name from Person x` returns a `bag` containing the name of every `Person` instance in the database
`select struct(name: x.name, age: x.age) from Person x` returns a `bag` containing `struct` elements including the name and age of every `Person` instance in the database
`select list(x, x.name) from Person x where x.name ~ "h"` returns a `bag` of `list` elements containing the `oid` and the name of every `Person` instances whose name matches the regular expression `"h"`
`select list(x, x.name) from Person x where x.name ~ "h" order by x.name` same as previous example, but the result is a `list` ordered by the name of the persons
`select x from Person x where x.spouse.name = "john" or x.age < 10` returns a bag of `Person` instances whose spouse name is equal to `"john"` or the age is less than `10`
`select x from Person x order by x.name` current implementation restriction: *raises an error*: `x.name not found in projection`

Arrays and collections in query expressions

OQL provide supports for performing direct queries through non-collection array and collection attributes without explicit joins. To perform such queries, one must use the operators `[]`, `[?]` or `[:]`. All the operators may be used for non-collection arrays. For collections (`list`, `set`, `bag` and `array`), only the operator `[?]` is valid.
The operator `[]` denotes an element in a non-collection array.
The operator `[:]` denotes a range of elements in a non-collection array.
The operator `[?]` denotes all elements in a non-collection array or in a collection. The following table presents several examples:

Array and Collection based Query Examples
`select x.name from Person x where x.name[0] = 'j'` returns all the `Person` instances whose name begins with a `'j'`
`select x from Person x where x.other_addrs[0].street ~~ "par."` returns all the `Person` instances whose first `other_addrs` street matches the regular expression `"par."`
`select x from Person x where x.other_addrs[?].street ~~ "par.."` returns all the `Person` instances whose any `other_addrs` street matches the regular expression `"par.."`
`select x from Person x where x.other_addrs[1:3].street ~~ "par.."` returns all the `Person` instances whose the first, second or third `other_addrs` street matches the regular expression `"par.."`
`select x from Person x where x.children[?].name = "johnny"`

returns all the persons whose one of its children is called `"johnny"`
`select x from Person x where x.cars[?].num < 100 or x.children[?].name = "mary"` returns all the persons whose one of its cars has a number less than `100` or a child called `"mary"`
`select x from Person x where x.children[1].name = "johnny"` although the `children` is a collection array, an error is raised. This is a current limitation of the implementation that will disapear soon

The Implicit `select` syntax

An implicit `select` expression is a `select` expression with neither a `from` nor an explicit `where` clause. In fact, the `where` clause may be included in the *projection* expression.
This particular syntax has the advantage to be more compact and more simple than the general syntax, but some queries cannot be performed:

1. queries performing an explicit join,
2. queries having `or` or `and` in their `where` clause.

The following table presents several examples:

Simple Implicit `select` Examples
`select 1` returns `1`
`select Person` returns a `bag` containing all `Person` instances in the database
`select Person.name` returns a `bag` containing the name of every `Person` instances in the database
`select Person.name = "john"` returns a `bag` containing the `oid`s of every `Person` instances whose name is equal to `"john"`
`(select distinct Person.name = "john").age` returns a `set` containing the age of every `Person` instances whose name is equal to `"john"`
`select Person.name = "john" or Person.age = 10` *raises an error*: `use select/from/where clause`
`select Person.name order by Person.name` returns a `list` containing of the sorted names of all `Person` instances

Querying the schema

As every abstraction is an object, one can perform queries on the schema and classes. For instance, to get the `oid`s of all the existing classes in a schema:
`select schema` which is equivalent to `select class`.
In the EyeDB object model, the class `class` has the following native attributes (some are inherited from the `object` class):

1. `class` (inherited from `object`) is the class of this class,
2. `protection` (inherited from `object`) is the protection object of this class,
3. `type` is the type of the class: `"system"` or `"user"`.
4. `name` is the name of this class,
5. `parent` is the parent class of this class,
6. `extent` is the extent collection of this class: contains all the object instances of this class,
7. `components` is the collection of components of this class: contains the constraints, the methods, the triggers, the index.

Queries can be performed according to one or more of the previous attributes.

Query Schema Examples
`select class.name = "Person"` returns a `bag` containing the class whose name is `"Person"`
`select class.type = "user"` returns all the user classes
`select x from class x where x.name ~ "P" and x.type = "user"` returns the user classes whose name matches the given regular expression
`select class.parent.name = "Person"` returns a `bag` containing the sub-classes of the class `Person`

How queries are optimized?

EyeDB queries implementation make an heavy use of index. Index may be used as terminal index or as non-terminal index: terminal index are those used at the end of a path expression, and the other are those used inside a path expression.

For instance, assuming that each attribute in our schema is indexed, the query `select Person.name = "john"` uses the index on the attribute `name` as a terminal index.

The query `select Person.spouse.age < 23` uses the index on the attribute `age` as a terminal index, and the index on the indirect attribute `spouse` as a non-terminal index, while the query `select Person.spouse = 6252.3.48474:oid` uses the index on the `spouse` attribute as a terminal index.

The query `select Person.children[?].spouse.name = "mary"` uses the index of the attributes `name`, `spouse` and of the literal collection attribute `children`.

The queries with a `where` clause containing logical `and` constructs are optimized so to take advantage of the index. For instance, assuming that there is an index on the `name` attribute and no index on the `age` attribute, the query `select x from Person x where x.age = 100 and x.name = "john"` will perform first the query on the `name` attribute and then filter the result according to the given predicat on the `age` attribute.
If the two expressions around the logical `and` operator have an index or if none of the two expression has a index, the interpreter performs first the left part and then the second part. So, in this case, the order of the two expressions around the `and` is important.

Finally, the query `select Person.name` reads directly the index of the `name` attribute, instead of reading the name of each `Person` instance.

Nevertheless, currently, there are some constructs that the OQL interpreter does not interpret cleverly and where index are not used. This is unfortanely the case of the join constructs such as:
`select x from Person x, x.spouse y where y.name = "mary"`.
This construct will not make use of the index on the `spouse` attribute (but it will use the index of the `name` attribute). Fortunately, in a lot of cases, join queries may be replaced by a path expression query, for instance:
`select x from Person x where x.spouse.name = "mary"` is the alternate form the previous join query.

This alternate form (path-expression oriented) is the preferred one, because:
- it is more intuitive,
- it is more *object oriented* (please forget relationnal!),
- it is more compact,
- it uses index properly.

Of course, the last reason is not a good reason as a proper query implementation should use index whatever the used syntax. The implementation will be improved in a next version.

Note that query optimisations are fragile and do not believe that the OQL interpreter knows your data better than you. So, the two following simple rules should be applied:

1. when path expression oriented queries are enough, do not use join constructs,

2. in a `and` expression within a `where` clause, put the expression which denotes the most little set of instances on the left side of the `and`. In some particular cases, the OQL interpreter knows how to optimize the `and`, but in most of the cases, it does not.

Some experimental hints can be added to an `and` expression in the `where` clause to help the interpreter to do the things better, but there are currentlty too much experimental to be documented here.

5.30 Miscellenaous Expressions

A few OQL operators cannot be easily classified in one of the previous categories. We choose to classify them in the miscellenaous operators. These operators are: `bodyof`, `structof`, `[!]` and `import`.

`bodyof` operator

The `bodyof` operaror is used to get the body of an OQL function. For instance, let the function `fib`: `define fib(n) as (n < 2 ? n : fib(n-2) + fib(n-1))`. The expression `bodyof fib` will return: `"fib(n) ((n<2)?n:(fib((n-2))+fib((n-1))))"`. This operator could be applied to expression-functions (i.e. functions defined with the `define/as` operator) or statement-functions (i.e. functions defined with the `function` operator).

General Information	
Operator	`bodyof`
Syntax	`classof` *expr*
Type	unary
Operand Type	`ident` denoting a function
Result Type	`string`
Function	returns the body of the function

Expression Examples	
`bodyof is_int`	`"is_int(x) ((typeof x)=="integer")"`
`bodyof first`	`"first(l) { if ((((!is_list(l))&&(!is_array(l)))) return l; start:=0; f:=nil; for (x in l) if ((start==0)) { start:=1; f:=x; break; }; ; return f; }"`
`bodyof 1`	*raises an error*

`structof` operator

The `structof` is used to get the meta-type of a `struct` atom. The meta-type of a `struct` atom is the list of the attributes of this `struct`. For instance, the meta-type of `struct(a : 1, b : "hello")` is the list composed of the two attribute `a` and `b`. The following expression `structof struct(a : 1, b : "hello")` returns `list("a", "b")`.

General Information	
Operator	`structof`
Syntax	`structof` *expr*
Type	unary
Operand Type	`struct`
Result Type	a `list` of `string`s
Function	returns the meta-type of the operand

Expression Examples	
`structof struct(alpha : 1, beta : 2)`	`list("alpha", "beta")`
`structof first(select struct(x: x.firstname) from Person x)`	`list("x")`
`structof 1`	*raises an error*

`[!]` operator

The `[!]` is used to get the length (or size) of an ordered or unordered collection, a string or a struct. For instance, `"hello"[!]` returns `5`, while `list(1, 2, 3)[!]` returns `3`. Note that this operator is far more efficient than the `strlen`

library function.

General Information	
Operator	`[!]`
Syntax	*expr* `[!]`
Type	unary
Operand Type	`string`, `collection` or `struct`
Result Type	a `int`
Function	returns length of the operand

Expression Examples	
`(select Person)[!]`	the number of person instances
`(struct(a: 1, b:2, c: "hello"))[!]`	`3`
`("hello"+"world")[!]`	`10`
`"hello"+"world"[!]`	*raises an error*

`import` operator

The `import` operator is used to import an OQL file in the current OQL session. Its operand is a string which denote the absolute path (i.e. beginning with a `/`) or relative path (i.e. not beginning with a `/`) of the file to import. When the path is relative, the OQL interpreter will look in every directories pointed by the EyeDB configuration variable `oqlpath`. By default, `oqlpath` is equal to `%root%/etc/oql`. If the file name has no `.oql` extension, the OQL interpreter will automatically adds one.

General Information	
Operator	`import`
Syntax	`import` *expr*
Type	unary
Operand Type	`string`
Result Type	a `string`
Function	import the file

Expression Examples	
`import "stdlib"`	`"/usr/local/eyedb/etc/so/stdlib.oql"`
`import "roudoudou"`	*raises an error*: `cannot find file 'roudoudou'`

5.31 Selection Statements

The selection statement is based on the `if/else` constructs.

Syntax: `if (` *cond_expr* `)` *statement1* [`else` *statement2*]
where *cond_expr* is a boolean expression, and *statement1* and *statement2* may be any statement: an expression statement, an iteration statement, a compound statement, a selection statement, a function definition statement, a jump statement or an empty statement.

Semantics: if the boolean expression *cond_expr* is evaluated to `true`, the statement *statement1* is executed. Otherwise, if an `else` part is there, the statement *statement2* is executed. The statements 1 and 2 may be any statement: an expression statement, iteration statement, compound statement, selection statement, function definition statement, jump statement or empty statement.
Note that a selection statement does not return any atom

The following table presents several examples of `if/else` statements:

if/else Statement Examples
`if (true) a := 1;` the variable `a` is assigned to `1`
`if (1) b := 2;` an error is raised: `boolean expected for condition`
`if (check(10) > 54) {a := 1; b := 2;} else {c := 2; d := 2}` here compound statements are used, because several expression statements need to be executed
`if ((check(10) > 54 || alpha < 2) && beta > 2.3 ) {callme(2);}` use of a composite conditional expression
`if (a == 1) b := 2; else if (b == 3) {c := 2; return 4;} else if (check(1)) return 2; else return 3;` selection statements are combined

5.32 Iteration Statements

The iteration statements are based on the following constructs:

- `while`
- `do/while`
- `for` C, C++, Java form
- `for` collection form

`while` statement

Syntax: `while (` *cond_expr* `)` *statement*
where *cond_expr* is a boolean expression, and *statement* any statement: an expression statement, an iteration statement, a compound statement, a selection statement, a function definition statement, a jump statement or an empty statement.

Semantics: The statement is executed while the boolean expression `cond_expr` is evaluated to `true`. Note that a `while` statement does not return any atom

The following table presents several examples of `while` statements:

while Statement Examples
`while (true) a++;` `a` is increment definitively!
`while (n--) a++;` an error is raised: `boolean expected, got integer`
`while (n-- > 0) a++;` this is better
`while (n++ <= 100 || stop) {if (!perform(a++)) break; check(a);}` note the usage of a compound statement and of the `break`
`while (name != "john") {l := (select Person.name = name); name := get_name();}` *no comments*

`do/while` statement

Syntax: `do` *statement* `while (` *cond_expr* `)`
where *cond_expr* is a boolean expression, and *statement* any statement: an expression statement, an iteration statement, a compound statement, a selection statement, a function definition statement, a jump statement or an empty statement.

Semantics: The statement is executed at least once. Then while the boolean expression `cond_expr` is evaluated to `true`, the statement is executed. Note that a `do/while` statement does not return any atom

The following table presents several examples of `do/while` statements:

`do/while` Statement Examples
`do a++; while (true);` `a` is increment definitively!
`do a=+; while (n--);` an error is raised: `boolean expected, got integer`
`do a++; while (n-- > 0);` this is better
`do {if (!perform(a++)) break; check(a);} while (n++ <= 100 || stop);` note the usage of a compound statement and of the `break`
`do {l := (select Person.name = name); name := get_name();} while (name != "john");` *no comments*

C-`for` statement

Syntax: `for (` [*expr1*] `;` [*cond_expr*] `;` [*expr2*] `)` *statement*
where *cond_expr* is a boolean expression, *expr1* and *expr2* are any expressionss and *statement* any statement: an expression statement, an iteration statement, a compound statement, a selection statement, a function definition statement, a jump statement or an empty statement.

Semantics: The expression *expr1* is evaluated. While the boolean expression `cond_expr` is evaluated to `true`, the statement is executed and the expression *expr2* is evaluated.
Note that a C-`for` statement does not return any atom

The following table presents several examples of C-`for` statements:

C-`for` Statement Examples
`for (x := 0; x < 100; x++) a++;` increment `a` an hundred times
`for (x := 0, y := 1; x < 100 && check(y); x++) {y := get(y); if (y == 9999) break;}` a more complex example
`for (x := 100; x; x--) perform(x);` raises an error: `boolean expected, got integer`
`for (x := 0;;) doit();` note that there is neither a conditionnal expression, nor a second expression
`for (;;) doit();` same as `while(true)`

collection-`for` statement

Syntax: `for (` *var* `in` *expr* `)` *statement*
where *cond_expr* is a boolean expression, *var* denotes the name of a variable, *expr* is an expression of type collection and *statement* any statement: an expression statement, an iteration statement, a compound statement, a selection statement, a function definition statement, a jump statement or an empty statement.

Semantics: For each element in the collection denoted by *expr*, the variable *var* is assigned to this element and the statement is executed. Note that a collection-`for` statement does not return any atom

The following table presents several examples of collection-`for` statements:

collection-`for` Statement Examples

`for (x in list(1, 2, 3)) a += x;` increments `a` with `1`, `2` and `3`
`for (x in (select Person)) names += x.name;` concatenates all the person names
`for (x in (select Person.name = "john")) if (x.age < 10 || x.spouse.age < 10) throw "cannot mary children!!";` a moralistic example
`for (x in 1) doit();` raises an error: `boolean expected, got integer`

5.33 Jump Statements

There are two jump statements based on the keywords `break` and `return`.

`break` Statement

The syntax for `break` statement is:
`break` *statement*
where *statement* is an optional integer expression statement indicating the break level. If not specified, the break level is `1`. The `break` statement may appear only within an iteration statement: `while`, `do/while`, `for` statements.

`break` *Statement Examples*	
Statement	*Comments*
`break;`	*raises an error*: `break operator <<break; >> : level 1 is too deep.`
`break 3;`	*raises an error*: `break operator <<break; >> : level 3 is too deep.`
`for (x := 0; ; x++) if (x == 30) break;`	break the loop when `x` is equal to `30`
`while (true) {x++; if (!check(x)) break;}`	break the loop when `check(x)` is not true
`while (true) {x++; if (!check(x)) break 2;}`	*raises an error*: `break operator <<break; >> : level 2 is too deep.`
`while (true) {x++; while (x < 100) if (!check(&x)) break 2;}`	break the two loop when `check(&x)` is not true

`return` Statement

The syntax for `return` statement is: `return` [*statement*]
where *statement* is an optional expression statement indicating the atom to return. The atom to return may be of any type. If not specified, no atom is returned. The `return` statement may appear only within a function definition statement.

`return` *Statement Examples*	
Statement	*Comments*
`return;`	*raises an error*: `return operator <<return; >> : return must be performed in a function`
`function fact(n) {return n * fact(n-1);}`	ok.
`function f(x) {if (g(x)) return x+1; return x+2;}`	ok.
`function f(b) {if (b) return list(1, 2, 3); return bag(1);}`	ok.
`function f(b) {if (b) return list(1, 2, 3); return bag(1);}`	ok.
`define as f(x) (if (x) return x)`	*raises an error*: `syntax error`

5.34 Function Definition Statements

As introduced previously, OQL supports functions. There are two sorts of functions definition syntax: function definition expression and function definition statements. The first ones, exposed in Section 5.26 can contain only one expressions. That means that they cannot include neither selection, neither statements, neither jump statements. Furthermore, as only one expression is allowed, functions that need several expressions, one must use the comma sequencing operaot to seperate the expressions, thus making this code not always readable.
We introduce here the more general form of function definitions which overrides the limitations of the previous form. The general form of function defintion statements is: `function` *identifier* `(`[*arglist*]`)` *compound_statement*

1. *identifier* denotes any valid OQL identifier, except a keyword
2. *arglist* is an optional comma-separated list of identifiers optionally followed, for default arguments, by a "`?`" and an *expr*, for instance:
   (*var1*`,` *var2*`,` *var3*`?` *expr*`,` *var4*`?` *expr*`)`
3. *compound_statement* is a optionnal semicolon-separated list of statements surrounded by braces.

For instance:

```
function f(x, y, z ? oql$maxint) {
  if (x > y)
    throw "error #1";
  return x - y * 2 / z;
}
```

Argument Types/Return Type

Functions are not typed. That means that neither the return type nor the argument types may be given. It is why there is no function overloading mechanisms. To take benefit of the overloading mechanisms, one must use methods.
Nevertheless, it is possible to add type checking by using library functions such as `is_int`, `is_string`... combined with the `assert` or the `assert_msg` library functions. For instance, to check that the first argument is an integer and the second one a collection:

```
function doit(n, coll) {
   assert_msg(is_int(n), "doit: argument #1: integer expected");
   assert_msg(is_coll(coll), "doit: argument #2: collection expected");
   // body of the function
}
```

The `assert_msg` check that its first argument is equal to `true`, otherwiser an exception containing the second argument string is thrown:

```
doit(1, list(1, 2, 3)); // ok

doit(1.2, list(1)); // raises the error:
                    // assertion failed: 'doit: argument #1: integer expected'
```

Arguments `in`, `out` and `inout`

Furthermore, one cannot specify that an argument is an input argument (`in`), an output argument (`out`) or an input/output argument (`inout`). In a function call, expressions and variables are always passed by value not by reference, this means that the call to "`perform(x, y)`" cannot modify neither `x` nor `y`. (*In fact, yes it can! It is explained below. But forget it for now*).
So, to modify variable through a function call, one needs to give the reference (or address) of this variable, not its value. In this case, the function must execute specific code dealing with address variables instead of their values.
The `refof` operator, introduced in a previous section, gives the reference of an identifier. Remember that the expression `refof x` returns the identifier `x`. To make a function call using references one must do: `swap(refof x, refof y)` or the equivalent more compact form `swap(&x, &y)`.
Contrary to C++, reference manipulation is not transparent in OQL: to access the value of a reference, one must use the `valof` operator (i.e. `*` operator). The `swap` function which swaps its two `inout` arguments has already been introduced:

```
function swap(rx, ry) {
    v := *rx;
  *rx := *ry;
  *ry := v;
}
```

The arguments have been prefixed by `r` to indicate that they are references. So, the function call `swap(&x, &y)` will swap properly the variables `x` and `y`.
One can add type checking in the `swap` function, as follows:

```
function swap(rx, ry) {
  assert_msg(is_ident(rx), "swap: argument #1 identifier expected");
  assert_msg(is_ident(ry), "swap: argument #2 identifier expected");
    v := *rx;
  *rx := *ry;
  *ry := v;
}
```

Return Value

By default, a statement-oriented function returns no atom. To make a function returning an atom, one must use the `return` statement previously introduced. As a function has no specified returned type, it may contained several `return` statements returning atom of different types:

```
function perform(x) {
  if (x == 1)
    return "hello";
  if (x == 2)
    return list(1, 2, 3) + list(4, 20);
  if (x == 3)
    return 2;
  if (x == 4)
    return 'a';
}

alpha := perform(1); // alpha is equal to "hello"
alpha := perform(3); // alpha is equal to 2
alpha := perform(8); // alpha is equal to nil
```

Default Arguments

OQL provides support for default arguments in a function definition statement. The syntax for a default argument is: "`var ?` *expr*" or "`var :=` *expr*".
As in C and C++, the arguments with a default value must not followed by any argument with default values. For instance, `function f(x, y, z := "alpha")` is valid while `function f(x, y, z := "alpha", t)` is not valid.

Unval Arguments

Sometimes, it is interesting to prevent the evaluation of some input arguments. For instance, let the function `if_then_else` which takes three arguments:

1. `cond`: a boolean expression,
2. `then_expr`: expression of any type; is evaluated and returned if and only if the condition is evaluated to `true`
3. `else_expr`: expression of any type; is evaluated and returned if and only if the condition is evaluated to `false`

It is clear that the following function definition:

```
function if_then_else(cond, then_expr, else_expr) {
   if (cond)
     return then_expr;
   return else_expr;
```

is not correct as, although it returns the correct expression, the `then_expr` *and* the `else_expr` will be evaluated. For instance, `if_then_else(x < 10, ::a := 2, ::b := 3)` will return `2` if `x` is less than `10`, otherwise it will return `3`, but in any case, `a` will be assigned to `2` and `b` will be assigned to `3`.
So, one needs a way to tell the interpreter that we do not want to evaluate the second and the third argument. The special character | before an argument means that this argument must not be evaluated. In this case, this argument is substitued by the string representation of the expression. For instance, let the function `if_then_else`:

```
function if_then_else(cond, |then_expr, |else_expr) {
   // ...
}
```

when performing the call "`if_then_else(x < 10, ::a := 2, ::b := 3)`":

1. the value of `cond` in the body of the function will be `true` or `false`,
2. the value of `then_expr` in the body of the function will be `"::a:=2"`
3. the value of `else_expr` in the body of the function will be `"::b:=3"`

The correct implementation of this function is as follows:

```
function if_then_else(cond, |then_expr, |else_expr) {
   if (cond)
     return eval then_expr;
   return eval else_expr;
}
```

Scope of Variables

In the body of a function defintion, every variable on the left side of an assignment has a local scope except if this variable is prefixed by the global scope operator `::`. That means, that after the following statement sequence:

```
a := 2;

function doit() {
   a := 1;
}
```

the variable `a` is still equal to `2`. While after:

Recursivity

```
a := 2;

function doit() {
   ::a := 1;
}
```

the variable `a` is equal to `1`.

Particularity

One can define a statement-oriented function inside the body of a statement-oriented function, for instance:

```
function compute(amount_in_euro, usdollar_per_euro) {

  function euro2usdollar(euro, usd ? usdollar_per_euro) {
      return euro * usd;
  }

  x := euro2usdollar(euro * 1.24);
  x += euro2usdollar(1000);

  return x * .120;
}
```

Note that the function defined in the body of the function `compute` has a global scope, that means that after one execution of `compute` the function is available at the global level of the OQL session. It is possible, that in a future version, the functions defined in the body of a function definition will have a local scope.

The `oql$functions` Variable

The `oql$functions` value is a list whose elements are the name of all the OQL functions defined in the current OQL session. Each you add a user function, this variable is updated. At the beginning of a session, the value of textttoql$functions is:

```
list(is_int, is_char, is_double, is_string, is_oid, is_num, is_bool, is_bag,
     is_set, is_array, is_list, is_coll, is_struct, is_empty, void, assert,
     assert_msg, min, max, first, last, cdr, count, interval, sum, avg, is_in,
     distinct, flatten, flatten1, tolower, toupper, tocap, toset, tolist,
     tobag, toarray, listtoset, bagtoset, arraytoset, listtobag, settobag,
     arraytobag, bagtolist, settolist, arraytolist, bagtoarray, settoarray,
     listtoarray, strlen, substring, forone, forall, delete_from, get_from,
     ifempty, null_ifempty, getone)
```

For instance, to put the complete definition of all these functions into the variable `functionString`:

```
functionString := "";
for (x in oql$functions)
  functionString += "FUNCTION " x + ": " +  bodyof x + "\n";
```

The next section provides a few statement-oriented and expression-oriented function definitions.

6 Quick Reference Manual

This OQL quick reference manual presents consise information about the builtin and library functions and methods, the special variables, and the `eyedboql` tool. The standard library source code is presented and it provides a quick reference card containing all the language constructs.

6.1 Builtin and Library Functions and Methods

OQL provides a few builtin and library functions. The builtin functions are written in C++ and cannot be overriden while the library functions are written in OQL and may be redefined by the user. The code for the library functions can be found in the section *The Standard Library Package*. The EyeDB system classes `object`, `database`, `connection` and `EyeDB` contains builtin class and instance methods that can be accessed from OQL. Some of these methods are briefly introduced in this section.

Type Predicat Functions

`is_int(x)` : returns `true` if `x` is an `int`
`is_double(x)` : returns `true` if `x` is a `double`; otherwise, returns `false`
`is_string(x)` : returns `true` if `x` is a `string`; otherwise, returns `false`
`is_oid(x)` : returns `true` if `x` is an `oid`; otherwise, returns `false`
`is_num(x)` : returns `true` if `x` is an number (`int`, `float` or `char`); otherwise, returns `false`
`is_bool(x)` : returns `true` if `x` is a `bool`; otherwise, returns `false`
`is_bag(x)` : returns `true` if `x` is a `bag`; otherwise, returns `false`
`is_set(x)` : returns `true` if `x` is a `set`; otherwise, returns `false`
`is_array(x)` : returns `true` if `x` is a `array`; otherwise, returns `false`
`is_list(x)` : returns `true` if `x` is a `list`; otherwise, returns `false`
`is_coll(x)` : returns `true` if `x` is a collection; otherwise, returns `false`
`is_struct(x)` : returns `true` if `x` is a `struct`; otherwise, returns `false`
`is_empty(x)` : returns `true` if `x` is `nil`; otherwise, returns `false`

Collection Conversion Functions

The collection conversion functions take one collection argument and convert this collection to another collection type and returns the converted collection.

`toset(coll)` : converts `coll` to a `set`
`tolist(coll)` : converts `coll` to a `list`
`tobag(coll)` : converts `coll` to a `bag`
`toarray(coll)` : converts `coll` to a `array`
`listtoset(coll)` : checks that `coll` is a `list` then converts `coll` to a `set`
`bagtoset(coll)` : checks that `coll` is a `bag` then converts `coll` to a `set`
`arraytoset(coll)` : checks that `coll` is a `array` then converts `coll` to a `set`
`listtobag(coll)` : checks that `coll` is a `list` then converts `coll` to a `bag`
`settobag(coll)` : checks that `coll` is a `set` then converts `coll` to a `bag`
`arraytobag(coll)` : checks that `coll` is a `array` then converts `coll` to a `bag`
`bagtolist(coll)` : checks that `coll` is a `bag` then converts `coll` to a `list`
`settolist(coll)` : checks that `coll` is a `set` then converts `coll` to a `list`
`arraytolist(coll)` : checks that `coll` is a `array` then converts `coll` to a `list`
`bagtoarray(coll)` : checks that `coll` is a `bag` then converts `coll` to a `array`
`settoarray(coll)` : checks that `coll` is a `set` then converts `coll` to a `array`
`listtoarray(coll)` : checks that `coll` is a `set` then converts `coll` to a `array`

Sort Functions

These functions are used to sort collection of sortable atom of homogeneous types: `int`, `char`, `float` or `string`.

`sort(coll)` : `coll` must a be a collection of homogeneous sortable atoms; sorts and returns this collection
`rsort(coll)` : `coll` must a be a collection of homogeneous sortable atoms; reverse sorts and returns this collection
`isort(coll, idx)` : `coll` must a be a collection of `list` or `array` of homogeneous sortable atoms; `idx` must be of `int` type;

sorts the collection of collections according to the #idx element of the inner collection

`risort(coll, idx)` : same as previous function, but perform a reverse sort

Collection Miscelleanous Functions

`first(coll)` : returns the first element of `coll`
`car(coll)` : returns the first element of `coll`
`last(coll)` : returns the last element of `coll`
`cdr(coll)` : returns all elements of `coll` but the first
`getn(coll, n)` : returns at most n elements of `coll`
`count(coll)` : returns the count of elements of `coll`
identical to `coll[!]`, but less efficient
`sum(coll)` : returns the sum of the numbers of `coll`
`avg(coll)` : returns the float average of the numbers `coll`
`distinct(coll)` : eliminates duplicates of `coll`
`flatten(coll)` : recursive flattening of `coll`
`flatten1(coll)` : one level flattening of `coll`
`min(coll)` : returns the minimal number of `coll`
`max(coll)` : returns the maximal number of `coll`
`forone(coll, f, data)` : if `f(e, data)` for one element `e` of `coll`, returns `true`; otherwise returns `false`;
`forall(coll, f, data)` : if `f(e, data)` for all element `e` of `coll`, returns `true`; otherwiser returns `false`;

String Function Functions

`tolower(str)` : converts (and returns) string `str` into lowercase
`toupper(str)` : converts (and returns) string `str` into uppercase
`tocap(str)` : converts the first character and each character following a _ of `str` into an uppercase
`strlen(str)` : returns the length of `str`; same as `str[!]`, but less efficient
`substring(str, from, len)` : returns the sub-string of `str` from the #`from` to the `from+len` characters; same as `str[from:from+len]` but less efficient

Query Functions

`delete_from(class)` : deletes all the instances of a given class
`get_from(class)` : gets all the instances of a given class

Useful Functions

`assert(cond)` : throws an exception is `cond` is not `true`
`assert_msg(cond, msg)` : throws the exception message `msg` if `cond` is not `true`
`interval(from, to)` : returns a `list` composed of the number from `from` to `to`

Native Methods of the Class `object`

The native methods of the class `object` allows us to perform a few action such as getting the oid of an instance `getOid()`, getting the database of an instance `getDatabase()` or converts the instance to its string representation `toString()`. For instance, to apply this last method to the first `Person` instance: `first(select Person).toString()`.

All the native methods of the class `object` are instance methods.

`oid getOid()` : returns the `oid` of the object
`string toString()` : returns the string representation of the object
`database *getDatabase()` : returns the `database` instance of the object
`void setDatabase(in database *)` : changes the `database` of the object

```
void store()                    : stores the object in the database
object *clone()                 : clones the object; returns the clone
int getCTime()                  : returns the creation time of the object (seconds from 1/1/1970)
int getMTime()                  : returns the last modification time of the object
string getStringCTime()         : returns the string representation of the creation time of the object
string getStringMTime()         : returns the string representation of the creation time of the object
bool isRemoved()                : returns true if the object is removed; false otherwise
bool isModify()                 : returns true if the object is modified; false otherwise
```

Native Methods of the Class `connection`

All the native methods of the class `connection` are instance methods. an object obtained using the `new` operator, They can be applied on a `database` object that can be either the current connection `oql$db->getConnection()` either an object obtained using the `new` operator, for instance: `new <> connection()`.

```
void open()             : opens a new connection with default host and port
void open(in string host, in string port) : opens a new connection using host and port
void close()            : closes the connection
```

Native Methods of the Class `database`

The following methods are the instance methods of the class `database`: They can be applied on a `database` object that can be either `oql$db` either an object obtained using the `new` operator, for instance: `new <> database(name : "foo")`.

```
void open(in connection *conn,
          in int mode)              : opens a new database using the connection
                                      conn and the open flag mode mode
void open(in connection *conn,
          in int mode,
          in string userauth,
          in string passwdauth)     : opens a new database using the connection conn,
                                      the open flag mode mode and the authentication
                                      userauth/passwdauth
void close()                        : closes the database
connection *getConnection()         : returns the connection tied to the database
int getOpenMode()                   : returns the open flag mode of the database
int getVersionNumber()              : returns the version number of the database
string getVersion()                 : returns the string version of the database

void removeObject(in oid)           : removes the object whose oid is given

void transactionBegin()             : begins a new transaction
void transactionBegin(in string mode) : begins a new transaction in mode mode
void transactionCommit()            : commits the current transaction
void transactionAbort()             : abort the current transaction

bool isInTransaction()              : returns true if a transaction is in progress;
                                      false otherwiser
```

Native Methods of the Class `EyeDB`

All the native methods of the class `EyeDB` are class methods.

```
string getConfigValue(in string s) : returns the string value of the configuration variable s
int getVersionNumber()             : returns the EyeDB current version number
string getVersion()                : returns the EyeDB current stringversion
string getArchitecture()           : returns the architecture of the current server
string getDefaultCompiler()        : returns the C++ compiler used to compile the current server
```

6.2 Special Variables

The following variables are predefined or have a special meaning:

`oql$variables` : list containing the name of all variables
`oql$functions` : list containing the name of all functions
`oql$result` : the result atom of the last statement
`oql$db` : `object` atom instance of the class `database` denoting the current database
`oql$minint` : the minimal integer `0x8000000000000000`
`oql$maxint` : the maximal integer `0x7FFFFFFFFFFFFFFF`
`oql$minfloat` : the minimal float `4.94065645841246544e-324`
`oql$maxfloat` : the maximal float `1.79769313486231570e+308`

6.3 The `eyedboql` Tool

`eyedboql` is a tool that allows you to execute interactively OQL statements. This tool is similar to the `Oracle sqlplus` and `Sybase isql` well known tools.

Running `eyedboql`

The simplest way to run `eyedboql` is to type `eyedboql` as follows (assuming that `$` is your shell prompt):

```
$ eyedboql
Welcome to eyedboql.
  Type '\help' to display the command list.
  Type '\copyright' to display the copyright.
?
```

The string "`? `" is the default prompt for `eyedboql`.

In an `eyedboql` session, anything you type is interpreted as an OQL construct (or a part of an OQL construct), Nevertheless, if the first word of an input line begins with the escape character "
", this word is interpreted as an `eyedboql` internal command. There are about 30 internal commands, but you need to know only of few of them to use `eyedboql`.

The purpose of the main internal commands is:
- to create or delete databases,
- to open a database,
- to begin, commit or abort a transaction,
- to set the current user and password,
- to execute the contents of a file,
- to display the string representation of an object,
- to display the HTML representation of an object in a WEB browser,
- to change the prompts and the escape character.

Executing Statements

`eyedboql` allows us to execute OQL statements in an interactive way. For instance, to perform the addition `1+3`:

```
$ eyedboql
Welcome to eyedboql.
  Type '\help' to display the command list.
  Type '\copyright' to display the copyright.
? 1+3;
= 4
?
```

The string "`= `" preceedes the result atom (if any) of your statement; in the current example, the result atom is the evaluation of the expression statement `1+3;`.

Execution Process

Each complete statement typed is sent to the OQL interpreter, A complete statement has a special meaning: it is any sequence of characters:
a. that end with a semi-colon and

b. which parenthesis are balanced and
c. which brakets are balanced and
d. which braces are balanced.

While the statement is not complete, `eyedboql` prompts the "second prompt" ("`>>` " by default) after each newline. Once the statement is complete, it is sent to the OQL interpreter, then the atom result is display (if any) after the string "`=` " and the main prompt "`?` " is displayed again.

For instance, the input sequence of characters "`1+`*newline*`3`*newline*`;`" gives:

```
? 1+
>> 3
>> ;
= 4
```

while the input sequence "{ `a := 1+3;`*newline*`c := 2+94;`*newline*`d := a+c}`" gives:

```
? { a := 1+3;
>> c := 2+94;
>> d := a+c}
?
```

Note that no "`=` *result atom*" is echoed because a compound statement does not return any atom.

This last example:

```
? while (true) {
>>   a++;
>>   b++;
>> }
>>
>> ;
?
```

shows the necessity of typing a semicolon after the `while` statement although a `while` statement does not need to end by a semi-colom in the OQL specifications.

Getting Started

By default in an `eyedboql` session, the database `EYEDBDBM` is opened in read-only mode. To use another database, one must use either the command line option `-db` either the
`open` internal command.

To start to play with `eyedboql` one needs to know the following five internal commands:

1. `open` *database* [`rw`|`ro`] `local` `trsless`: opens the *database* in read-write (`rw`) or read-only (`ro` or no option) mode. If `local` is there, *database* is opened in a local mode. If `trsless` is there *database* is opened in transaction-less mode. For instance "
   `open foo rw`" opens the database `foo` is read-write mode.

2. `print` [*sequence of oids*] *other options*: if a sequence of oids is given: loads the object corresponding to each `oid` and displays its string representation,
   if no sequence of oids is given: loads the object corresponding to each `oid` returned by the last statement and displays its string representation.
   For instance, "`select Person;`" will return a `bag` containing the `oid` of each `Person` instance.
   The internal command "
   `print`" typed after that will loads and displays of the corresponding `Person` instances.
   The *other options* are one or more space-separated of the followings:

`full`	: loads and displays object using the full recursive mode
`ctime`	: displays the creation time of the object
`mtime`	: displays the last modification time of the object
`contents`	: displays the contents of collections
`native`	: displays the native attributes
`all`	: means "`ctime mtime contents native`"

For instance “
`print full contents`” will load and display the objects and their collection contents in a full recursive mode.

3. `commit`: commits the current transaction

4. `abort`: aborts the current transaction

5. `quit` or *^D*: quits the current `eyedboql` session

Note that a transaction is started automatically before the first complete statement of the sesssion of before the complete statement immediately following
`commit` or
`abort` internal command.

Here is a commented example showing the use of these internal commands:

run `eyedboql`:

```
$ eyedboql
Welcome to eyedboql.
  Type ‘\help’ to display the command list.
  Type ‘\copyright’ to display the copyright.
```

open the database `person` *in read-write mode*:

```
? \open person rw
```

get the first person whose name is ”john” and display it:

```
? john := first(select Person.name = "john");
= 66373.12.4008447:oid
? \print
66373.12.4008447:oid Person = {
        name = "john";
        age = 32;
        addr Address = {
                street = "clichy";
                town = "Paris";
                country = NULL;
        };
        cstate = Sir;
        *spouse 66891.12.2738687:oid;
        cars set<Car*> = set {
                name       = "";
                count      = 4;
                dimension  = 1;
                reference  = true;
                magorder   = 4;
        };
        children array<Person*> = array {
                name       = "";
                count      = 0;
                range      = [0,0[;
                dimension  = 1;
                reference  = true;
                magorder   = 4;
        };
        x = NULL;
};
```

change the name of john to ”JOHNNY”:

```
? john.name := "JOHNNY";
= "JOHNNY"
```

retrieve the person whose name is "JOHNNY" and compares it to `john` *using* `assert` *: all is fine, no error is raised!*

```
? assert(john = first(select Person.name = "JOHNNY"));
```

abort the transaction and look for the person whose name is "JOHNNY": no person is returned! this is ok as the transaction was aborted:

```
? \abort
? select Person.name = "JOHNNY";
= list()
```

change the name of john to "JOHNNY" again and commit the transaction:

```
? john.name := "JOHNNY";
= "JOHNNY"
? \commit
```

then retrieve again the person whose name is "JOHNNY" and compare it to `john` *using* `assert`*: all is fine, no error is raised!*

```
? assert(john = first(select Person.name = "JOHNNY"));
```

quit `eyedboql` *session*

```
? \quit
$
```

We are going to conclude this section by this important note:
as introduced previously, the current transaction will be committed (resp. aborted) by a
`commit` (resp.
`abort`) command.
If you quit `eyedboql` before committing (resp. aborting) the transaction, it will be automatically aborted, so all your changes will be lost.
This is the default behaviour. This behaviour can be changed by using the
`commitonclose` internal command.

Command Line Options

The `eyedboql` command line options are as follows:

Program Options:

-d <name>, –database=<name¿	Database name
-r, –read	Open database in read mode
-w, –read-write	Open database in read/write mode
-s, –strict-read	Open database in strict read mode
-l, –local	Open database in local mode
-c <command>, –command=<command>	OQL command to execute
-p, –print	Display all the objects loaded
–full	Full recursive mode is used to display objects
–commit	Commits the current transaction on close
-i, –interact	Enter interpreter after executing file or commands
-e, –echo	Echo each command
–admin	Open database in admin mode
-h, –help	Display this message
<file¿	File(s) to execute

Common Options:

-U <user>|@, –user=<user>|@	User name
-P [<passwd>], –passwd[=<passwd>]	Password
–host=<host>	eyedbd host
–port=<port>	eyedbd port
–inet	Use the tcp_port variable if port is not set
–dbm=<dbmfile>	EYEDBDBM database file
–conf=<conffile>	Configuration file
–logdev=<logfile>	Output log file
–logmask=<mask>	Output log mask
–logdate=on|off	Control date display in output log

–logtimer=on|off	Control timer display in output log
–logpid=on|off	Control pid display in output log
–logprog=on|off	Control progname display in output log
–error-policy=<value>	Control error policy: status|exception|abort|stop|echo
–trans-def-mag=<magorder>	Default transaction magnitude order
–arch	Display the client architecture
-v, –version	Display the version
–help-eyedb-options	Display this message

For instance, to execute the statement "`delete_from(Person)`" on the database `person`:

```
$ eyedboql -d person -w -c "delete_from(Person)"
$
```

To execute the command "`persons := (select Person)`" and then enter the interactive mode of `eyedboql`:

```
$ eyedboql -d person -w -c "persons := (select Person)" -i
Welcome to eyedboql.
  Type '\help' to display the command list.
  Type '\copyright' to display the copyright.
?
```

To execute the file `mylib.oql`:

```
$ eyedboql -d person -w mylib.oql
```

6.4 The Standard Library Package

The `stdlib.oql` file contains a few basic library functions. It can be found in the directory *libdir*`/eyedb/oql`. It is automatically imported when opening an OQL session. The following code, extracted from this file, provides an interesting way to understand OQL.

```
//
// minimal and maximal integer values
//

oql$maxint := 0x7FFFFFFFFFFFFFFF;
oql$minint := 0x8000000000000000;

nulloid := 0:0:0:oid;
NULLOID := 0:0:0:oid;

//
// type predicat functions
//

define is_int(x)    as (typeof x == "integer");
define is_char(x)   as (typeof x == "char");
define is_float(x)  as (typeof x == "float");
define is_string(x) as (typeof x == "string");
define is_oid(x)    as (typeof x == "oid");
define is_object(x) as (typeof x == "object");
define is_num(x)    as (is_int(x) || is_float(x) || is_char(x));
define is_bool(x)   as (typeof x == "bool");
define is_bag(x)    as (typeof x == "bag");
define is_set(x)    as (typeof x == "set");
define is_array(x)  as (typeof x == "array");
define is_list(x)   as (typeof x == "list");
define is_coll(x)   as (is_list(x) || is_array(x) || is_set(x) || is_bag(x));
define is_struct(x) as (typeof x == "struct");
define is_empty(x)  as (typeof x == "nil");

//
// void(x): evaluates argument and returns nil
//
```

```
define void(x) as (x, nil);

function assert(|cond) {
  r := eval cond;
  if (!r)
    throw "assertion failed: '" + cond + "'";
}

function assert_msg(|cond, msg) {
  r := eval cond;
  if (!r)
    throw "assertion failed: '" + msg + "'";
}

//
// min(l): returns the minimal integer in a collection
//

function min(l) {
  msg := "function min(" + (string l) + "): ";
  if (!is_coll(l))
    throw msg + "collection expected";

  m := oql$maxint;

  for (x in l) {
    if (x != null) {
      if (!is_num(x))
        throw (msg + "numeric expected");
      if (x < m)
        m := x;
    }
  }

  return m;
}

//
// max(l): returns the maximal integer in a collection
//

function max(l) {
  msg := "function max(" + (string l) + "): ";
  if (!is_coll(l))
    throw msg + "collection expected";
  m := oql$minint;
  for (x in l) {
    if (x != null) {
      if (!is_num(x))
        throw (msg + "numeric expected");
      if (x > m)
        m := x;
    }
  }
  return m;
}

//
// first(l): returns the first element in a list or array
//

function first(l) {
```

```
  if (!is_coll(l)) // if (!is_list(l) && !is_array(l))
    throw "function first: collection expected";

  start := 0;
  f := nil;
  for (x in l)
    if (start == 0) {
      start := 1;
      f := x;
      break;
    }
  return f;
}

car := &first;

//
// last(l): returns the last element in a list or array
//

function last(l) {
  if (!is_coll(l)) // if (!is_list(l) && !is_array(l))
    throw "function last: list or array expected";

  f := nil;
  for (x in l)
    f := x;
  return f;
}

//
// cdr(l): returns all elements in a collection except the first one
//

function cdr(l) {
  if (!is_coll(l)) // if (!is_list(l) && !is_array(l))
    throw "function cdr: list or array expected";

  r := list();
  n := 0;
  for (x in l) {
    if (n != 0)
      r += x;
    n++;
  }
  return r;
}

//
// getn(l, n): returns all elements in a collection
//

function getn(l, n) {
   rl := list();
   cnt := 0;

   for (x in l) {
     if (cnt++ >= n)
       break;
     rl += x;
   }

   return rl;
```

```
}

//
// getrange(l, f, nb): returns all elements in a collection from element from
//
// identical to l[f:f+nb]
//

function getrange(l, f, nb) {
  if (!is_list(l) && !is_array(l))
    throw "function getrange: list or array expected";

   rl := list();
   cnt := 0;

   max := f + nb;

   for (x in l) {
     if (cnt >= max)
       break;

     if (cnt >= f)
       rl += x;

     cnt++;
   }

   return rl;
}

//
// count(l): returns element count of a collection
//

function count(l) {
  if (typeof l == "nil")
    return 0;
  if (!is_coll(l))
    throw "function count: collection expected, got " + typeof(l);
  return l[!];
}

//
// interval(x, y): constructs an integer list bounded by 'x' and 'y'
//

function interval(x, y) {
  n := x-1;
  l := list();
  while (n++ < y)
    l += n;
  return l;
}

//
// sum(l): returns the sum of collection elements
//

function sum(l) {
  if (!is_coll(l))
    throw "function sum: collection expected";
  n := 0;
  for (x in l)
```

```
    n += x;
  return n;
}

//
// avg(l): returns the average of collection elements
//

function avg(l) {
  if (!is_coll(l))
    throw "function avg: collection expected";
  return float(sum(l))/count(l);
}

//
// is_in(l, z): returns true in element 'z' is in collection 'l'
//

function is_in(l, z) {
  for (x in l)
    if (x == z)
      return true;

  return false;
}

//
// distinct(l): returns distinct elements in a collection
//

function distinct(l) {
  if (is_list(l)) ll := list();
  else if (is_bag(l)) ll := bag();
  else if (is_array(l)) ll := array();
  else if (is_set(l)) ll := set();
  else throw "function distinct: collection expected";

  for (x in l)
    if (!is_in(ll, x))
      ll += x;

  return ll;
}

//
// flatten(l): full recursive flatten function
//

function flatten(l) {
  if (!is_coll(l))
    return l;
  ll := list();
  for (x in l)
    if (is_coll(x))
      ll += flatten(x);
    else
      ll += x;

  return ll;
}

//
// flatten1(l): 1-level recursive flatten function
```

```
//

function flatten1(l) {
  if (!is_coll(l))
    return l;

  ll := list();
  for (x in l)
    ll += x;
  return ll;
}

//
// tolower(s): returns lower case string
//

function tolower(s) {

  n := 0;
  x := "";
  delta := 'a' - 'A';

  while (s[n] != '\000') {
    if (s[n] >= 'A' && s[n] <= 'Z')
      x += string(char(s[n] + delta));
    else
      x += string(s[n]);
    n++;
  }

  return x;
}

//
// toupper(s): returns upper cased string
//

function toupper(s) {

  n := 0;
  x := "";
  delta := 'A' - 'a';

  while (s[n] != '\000') {
    if (s[n] >= 'a' && s[n] <= 'z')
      x += string(char(s[n] + delta));
    else x += string(s[n]);
    n++;
  }

  return x;
}

//
// tocap(s): returns capitalized word string
//

function tocap(s) {

  n := 1;
  x := "";
  delta := 'A' - 'a';
```

```
  s := tolower(s);

  if (s[0] >= 'a' && s[0] <= 'z')
    x += string(char(s[0] + delta));

  while (s[n] != '\000') {
    if (s[n] == '_')
      x += string(char(s[++n] + delta));
    else
      x += string(s[n]);
    n++;
  }

  return x;
}

//
// Collection Conversion Functions
//

//
// General Conversion Functions
//

function toset(l) {
  if (!is_coll(l))
    throw ("function toset: collection expected, got " + typeof(l));

  if (!is_set(l)) {
    s := set();
    for (x in l)
      s += x;
    return s;
  }

  return l;
}

function tolist(l) {
  if (!is_coll(l))
    throw ("function tolist: collection expected, got " +  typeof(l));

  if (!is_list(l)) {
    s := list();
    for (x in l)
      s += x;
    return s;
  }

  return l;
}

function tobag(l) {
  if (!is_coll(l))
    throw ("function tobag: collection expected, got " +  typeof(l));

  if (!is_bag(l)) {
    s := bag();
    for (x in l)
      s += x;
    return s;
  }
```

```
  return l;
}

function toarray(l) {
  if (!is_coll(l))
    throw ("function toarray: collection expected, got " +  typeof(l));

  if (!is_array(l)) {
    s := array();
    for (x in l)
      s += x;
    return s;
  }

  return l;
}

//
// toset family Conversion Functions
//

function listtoset(l) {
  if (!is_list(l))
    throw ("function listtoset: list expected, got " + typeof(l));
  return toset(l);
}

function bagtoset(l) {
  if (!is_bag(l))
    throw ("function bagtoset: bag expected, got " + typeof(l));
  return toset(l);
}

function arraytoset(l) {
  if (!is_array(l))
    throw ("function arraytoset: array expected, got " + typeof(l));
  return toset(l);
}


//
// tobag family Conversion Functions
//

function listtobag(l) {
  if (!is_list(l))
    throw ("function listtobag: list expected, got " + typeof(l));
  return tobag(l);
}

function settobag(l) {
  if (!is_set(l))
    throw ("function settobag: set expected, got " + typeof(l));
  return tobag(l);
}

function arraytobag(l) {
  if (!is_array(l))
    throw ("function arraytobag: array expected, got " + typeof(l));
  return tobag(l);
}

//
```

```
// tolist family Conversion Functions
//

function bagtolist(l) {
  if (!is_bag(l))
    throw ("function bagtolist: bag expected, got " + typeof(l));
  return tolist(l);
}

function settolist(l) {
  if (!is_set(l))
    throw ("function settolist: set expected, got " + typeof(l));
  return tolist(l);
}

function arraytolist(l) {
  if (!is_array(l))
    throw ("function arraytolist: array expected, got " + typeof(l));
  return tolist(l);
}

//
// toarray family Conversion Functions
//

function bagtoarray(l) {
  if (!is_bag(l))
    throw ("function bagtoarray: bag expected, got " + typeof(l));
  return toarray(l);
}

function settoarray(l) {
  if (!is_set(l))
    throw ("function settoarray: set expected, got " + typeof(l));
  return toarray(l);
}

function listtoarray(l) {
  if (!is_list(l))
    throw ("function listtoarray: list expected, got " + typeof(l));
  return toarray(l);
}

//
// strlen(s): same as s[!]
//

function strlen(s) {
  len := 0;
  while (s[len] != '\000')
    len++;
  return len;
}

//
// substring(str, f, len)
//

function substring(str, f, len) {
  s := "";
  n := 0;
  max := str[!] - f;
  while (n < len && n < max) {
```

```
    s += string(str[n+f]);
    n++;
  }

  return s;
}

//
// forone(l, fpred, data): returns true if and only if the function 'fpred'
// returns true for at least one element 'x' in list 'l'
//

function forone(l, fpred, data) {
  for (x in l)
    if (fpred(x, data)) return true;
  return false;
}

//
// forone(l, fpred, data): returns true if and only if the function 'fpred'
// returns true for all elements 'x' in list 'l'
//

function forall(l, fpred, data) {
  for (x in l)
    if (!fpred(x, data)) return false;
  return true;
}

//
// delete_from(cls): delete all instances of class 'cls'
//

function delete_from(|cls) {
  for (x in (eval "select " + cls))
    delete x;
}

//
// delete_(coll): delete contents of collection coll
//

function delete_(coll) {
  for (x in coll)
    delete x;
}

//
// get_from(cls): returns all instances of class 'cls'
//

function get_from(|cls) {
  eval "select " + cls;
}

//
// generates an unused global symbol
//

function gensym() {
  prefix := "::oql#_#_#";
  for (i := 0; ; i++) {
    varname := prefix + string(i);
```

```
    if (!(eval "isset " + varname)) {
      eval varname + " := 0";
      return ident(varname);
    }
  }
}

//
// expression-like for-each function
//

function foreach_expr(|x, |coll, |expr, colltyp ? "list") {
  varname := "_#_#_R_#_#_";

  statement := "push " + varname + " := " + colltyp + "(); " +
    "for (" + x + " in " + coll + ") " +
    "{" + varname + " += " + expr + ";}" +
    "pop " + varname;

  return eval statement;
}

//
// expression-like for-C function
//

function for_expr(|start, |cond, |end, |expr, colltyp ? "list") {
  varname := "_#_#_R_#_#_";

  statement := "push " + varname + " := " + colltyp + "(); " +
    "for (" + start + "; " + cond + "; " + end + ")" +
    "{" + varname + " += " + expr + ";}" +
    "pop " + varname;

   return eval statement;
}

//
// expression-like while-C function
//

function while_expr(|cond, |expr, colltyp ? "list") {
  varname := "_#_#_R_#_#_";

  statement := "push " + varname + " := " + colltyp + "(); " +
    "while (" + cond + ")" +
    "{" + varname + " += " + expr + ";}" +
    "pop " + varname;

   return eval statement;
}

//
// expression-like do/while-C function
//

function do_while_expr(|expr, |cond, colltyp ? "list") {
  varname := "_#_#_R_#_#_";

  statement := "push " + varname + " := " + colltyp + "(); " +
    "do {" + varname + " += " + expr + ";}" +
    "while (" + cond + ");" +
    "pop " + varname;
```

```
   return eval statement;
}

function extentof(|classname) {
  return (select one class.name = classname).extent;
}

function countof(|classname) {
  return (select one class.name = classname).extent.count;
}

function objectcount(db := oql$db) {
     objcnt := 0;
     db->transactionBegin();
     for (cl in (select <db> x from class x where
                 x.type != "system" and x.name !~ "<"))
        objcnt += cl.extent.count;
     db->transactionCommit();
     return objcnt;
}

function ifempty(x, y) {
  if (is_empty(x))
    return y;
  return x;
}

function null_ifempty(x) {
  return ifempty(x, null);
}

function getone(x) {
  if (is_empty(x))
    return null;
  return first(flatten(x));
}

//
// database and transaction management
//

function open_db(db_name_or_id, strmode, user := null, passwd := null) {

  if (strmode == "r")
    mode := DBREAD;
  else if (strmode == "rw")
    mode := DBRW;
  else if (strmode == "rlocal")
    mode := DBREAD|DBOPENLOCAL;
  else if (strmode == "rwlocal")
    mode := DBRW|DBOPENLOCAL;
  else
    throw "invalid open mode: r, rw, rlocal or rwlocal expected, got " +
      strmode;

  if (is_int(db_name_or_id))
    db := new<> database(dbid : db_name_or_id);
  else
    db := new<> database(dbname : db_name_or_id);

  if (user == null)
    db.open(oql$db.getConnection(), mode);
```

```
  else
    db.open(oql$db.getConnection(), mode, user, passwd);

  return db;
}

function set_default(db) {
  db->setDefaultDatabase();
}

function begin(db := oql$db) {
  db->transactionBegin();
}

function begin_params(trsmode, lockmode, recovmode, magorder, ratioalrt, wait_timeout, db := oql$db) {
  db->transactionBegin(trsmode, lockmode, recovmode, magorder, ratioalrt, wait_timeout);
}

function commit(db := oql$db) {
  db->transactionCommit();
}

function abort(db := oql$db) {
  db->transactionAbort();
}

//
// miscellaneous
//

function print_function(f) {
  print "function " + (bodyof f) + "\n";
}

function print_functions() {
  cnt := 0;
  for (f in oql$functions) {
    if (cnt > 0) print "\n";
    print_function(f);
    cnt++;
  }
}

function print_variable(v) {
   print string(v) + " = " + string(eval string(v)) + ";\n";
}

function print_variables() {
  for (v in oql$variables) {
    print_variable(v);
    cnt++;
  }
}

function print_classes(system := false) {
  if (system)
    l := (select list(x, x.name) from class x order by x.name);
  else
    l :=  (select list(x, x.name) from class x where x.type = "user" and x.name !~ "<" order by x.name);

  for (c in l) {
    cls := c[0];
    clsname := c[1];
```

```
    print "class " + clsname;
    if (cls.parent != NULL && (system || cls.parent.type != "system"))
       print " extends " + cls.parent.name;
    print "\n";
  }
}

function print_obj(o, flags := 0) {
   print o->toString(flags);
}

function print_objs(l, flags := 0) {
   for (o in l)
     print_obj(o, flags);
}

//
// contents_ expression
//

function contents_(coll) {
  r := list();
  for (x in coll) {
   for (s in contents(x))
     r += s;
  }

  return r;
};

function println(s) {
  print(s+"\n");
}

function bench(|cmd) {
  t0 := time_stamp::local_time_stamp();
  r := eval cmd;
  t1 := time_stamp::local_time_stamp();
  us := t1->substract(t0).usecs;
  println("Elapsed time: " + string(us/1000.) + " ms");
  return r;
}

;
```

6.5 OQL Quick Reference Card

The following table presents all the OQL statements, expression types and the operators. For the operators common to C++ and OQL, the precedence and associativity is the same.

Quick Reference Card		
Statements		
expression statement	*expr* ;	
selection statement	`if` (*cond_expr*) *statement* [`else` *statement*]	
jump statements	`break` [*expr*] ; `return` [*expr*] ;	
iteration statements	`while` (*cond_expr*) *statement* `do` *statement* `while` (*cond_expr*) `for` ([*expr*] ; [*cond_expr*] ; [*expr*]) *statement* `for` (*var* `in` *expr*) *statement*	
compound statement	{ *statement* }	
function definition statement	`function` *identifier* ([*arglist*]) *compound_statement*	
empty statement	;	
Arithmetic Expressions		
add	+	*expr* + *expr*
substract	-	*expr* - *expr*
multiply	*	*expr* * *expr*
divide	/	*expr* / *expr*
shift left	<<	*expr* << *expr*
shift right	>>	*expr* >> *expr*
modular	%	*expr* % *expr*
bitwise and	&	*expr* & *expr*
bitwise inclusive or	|	*expr* | *expr*
bitwise xor	^	*expr* ^ *expr*
complement	~	~ *expr*
Assignment Expressions		
simple assignment	:=	*lvalue* := *expr*
add and assign	+=	*lvalue* += *expr*
substract and assign	-=	*lvalue* -= *expr*
multiply and assign	*=	*lvalue* *= *expr*
divide and assign	/=	*lvalue* /= *expr*
shift left and assign	<<	*lvalue* <<= *expr*
shift right and assign	>>	*lvalue* >>= *expr*
inclusive OR and assign	|=	*lvalue* |= *expr*
exclusive OR and assign	&	*lvalue* &= *expr*
modulo and assign	%=	*lvalue* %= *expr*
exclusive OR and assign	^	*lvalue* ^= *expr*
Auto Increment & Decrement Expressions		
post increment	++	*lvalue*++
post decrement	--	*lvalue*--
pre increment	++	++*lvalue*
pre increment	--	--*lvalue*
Logical Expressions		
logical and	&&	*expr* && *expr*
logical and	`and`	*expr* `and` *expr*
logical or	||	*expr* || *expr*
logical or	`or`	*expr* `or` *expr*
Comparison Expressions		
not	!	! *expr*
not	`not`	`not` *expr*
equal	=	*expr* = *expr*
equal	==	*expr* == *expr*
not equal	!=	*expr* != *expr*
less than	<	*expr* < *expr*
less	<=	*expr* <= *expr*
greater	>	*expr* > *expr*

greater than	`>=`	*expr* `>=` *expr*
match regular expression	`~`	*expr* `~` *expr*
match regular expression case insensitive	`~~`	*expr* `~~` *expr*
not match regular expression	`!~`	*expr* `!~` *expr*
not match regular expression case insensitive	`!~~`	*expr* `!~~` *expr*
match regular expression	`like`	*expr* `like` *expr*
Conditionnal Expressions		
conditionnal expression	`? :`	*expr* `?` *expr* `:` *expr*
Expression Lists		
comma sequencing	`,`	*expr* `,` *expr*
Array Expressions		
subscripting	`[]`	*expr* `[`*expr* `]`
interval subscriptiong	`[:]`	*expr* `[`*expr* `:`*expr* `]`
Path Expressions		
member selection	`.`	*expr* `.` *expr*
member selection	`->`	*expr* `->` *expr*
Function Call		
function call	`()`	*expr* `(`*expr_list*`)`
Method Invocation		
member selection	`()`	*expr* `->`*expr* `(`*arglist*`)`
Eval/Unval Operators		
eval	`eval`	`eval` *expr*
no eval	`unval`	`unval` *expr*
Identifier Expressions		
scope	`::`	`::` *identifier*
is set	`isset`	`isset` *identifier*
unset	`unset`	`unset` *identifier*
reference of	`&`	`&` *identifier*
	`refof`	`refof` *identifier*
value of	`*`	`*` *identifier*
value of	`valof`	`valof` *identifier*
scope of	`scopeof`	`scopeof` *identifier*
push onto symbol table	`push`	`push` *identifier*
push onto symbol table and assign	`push`	`push` *expr*
pop from symbol table	`pop`	`pop` *identifier*
Set Expressions		
union	`union`	*expr* `union` *expr*
intersection	`intersect`	*expr* `intersect` *expr*
except	`except`	*expr* `except` *expr*
include	`<`	*expr* `<` *expr*
include or equal	`<=`	*expr* `<=` *expr*
contain	`>`	*expr* `>` *expr*
contain or equal	`>=`	*expr* `>=` *expr*
Object Creation		
new	`new`	[`new`] *new_construct*
new	`new`	`new<` *opt_expr* `>` *new_construct*
Object Deletion		
delete	`delete`	`delete` *expr*
Collection Expressions		
contents	`contents`	`contents` *expr*
is in	`in`	*expr* `in` *expr*
add to collection	`add to`	`add` *expr* `to` *expr*
suppress from collection	`suppress from`	`suppress` *expr* `from` *expr*
set element in or get element from an indexed collection	`[]`	*expr* `[` *expr* `]`
append to an indexed collection	`append/to`	`append` *expr* `to` *expr*
empty collection	`empty`	`empty` *expr*
exists in collection	`in`	`exists` *identifier* `in` *expr* `:` *expr*
for all in collection	`for all`	`for all` *identifier* `in` *expr* `:` *expr*

for some in collection	`for`	`for <` *expr* `,` *expr* `> in` *expr* `:` *expr*
Exception Expressions		
throw exception	`throw`	`throw` *expr*
Function Definition		
define function	`define as`	`define` *identifier* [*arglist* `as` *expr*
Conversion		
string conversion	`string`	`string(` *expr* `)`
integer conversion	`int`	`int(` *expr* `)`
character conversion	`char`	`char(` *expr* `)`
float conversion	`float`	`float(` *expr* `)`
identifier conversion	`ident`	`ident(` *expr* `)`
oid conversion	`oid`	`oid(` *expr* `)`
Query Expressions		
database query	`select`	`select` *expr* [`from` {*expr* [`as`] *identifier*} [`where` *expr*]] [`order by` {*expr*}] `select` *expr* [`from` {*identifier* `in` *expr*} [`where` *expr*]] [`order by` {*expr*}]
Type Information Expressions		
class of	`classof`	`classof` *expr*
typeof of	`typeof`	`typeof` *expr*
Miscellenaous Expressions		
structure of	`structof`	`structof` *expr*
body of	`bodyof`	`bodyof` *expr*
length pf	`[!]`	*expr* `[!]`
import package	`import`	`import` *expr*