Object Definition Language

//Objectives
– Portability of database schemas across ODBMS
– Interoperability of ODBMSs from multiple vendors
//Development principles
– All semantic constructs of ODMG object model
– Specification language (not full programming language)
– Program language independence
– OMG IDL (Interface Definition Language) compatibility
– Practical and short time implementable
Type Specification

A type is defined by specifying its interface in ODL

Top-level BNF

type definition ::= interface <type_name> [:<supertype_list>] 

\{ 
 \[<type_property_list>] 
 \[<property_list>] 
 \[<operation_list>] 
\};

Any list may be omitted if not applicable

Type Characteristics Specification

Type characteristics
- Supertypes
- Extent naming
- Key(s)

Example

interface Professor :Person 
\{ 
 extent professors;
 keys faculty_id, soc_sec_no;
 <property_list>
 <operation_list> 
\};

Notes
- No more than one extent or key definition. Each attribute or relationship traversal name in key definition should be specified in the property list
- Extent naming and key definition may appear in any order
- Supertype, extent naming and key definition may be omitted if not applicable
Property List

//BNF

<property_list ::= <property_spec>; | <property_spec><property_list>
<property_spec>::= <attribute_spec> | <relationship_spec>

//Structured types have bracketed list of field-type pairs associated with them.
//Enumerated types have bracketed lists of values.
//Relationship have inverses.
//An element from another class is indicated by <class>::
//Form a set type with Set<type>.

Attribute Specification: Example

interface Professor {
  extent professors;
  keys faculty_id, soc_sec_no;

  attribute string name;
  attribute integer faculty_id;
  attribute integer soc_sec_no;
  attribute Struct<integer number, string street, Ref<City> city> address;
  attribute Enum {male, female} gender;

  <operation_list>
}
interface City {
  extent cities;
  key city_code;
  attribute integer city_code;
  attribute string name;
}
### Property List (2)

#### Relationship specification
- Definition: to define a traversal path for a relationship
  - Designation of target type
  - Information about inverse traversal path
- Example
  ```java
  interface Professor {
    extent professors;
    keys faculty_id, soc_sec_no;

    <attribute_list>;

    relationship Set<Student> advisees inverse Student::advisor;
    relationship Set<TA> teaching_assitants inverse TA::works_for;
    relationship Department department inverse Department::faculty
  }
  ```

### Operation List

#### BNF
```
operation_list ::= <operation_spec>; | <operation_spec><operation_list>
<operation_spec> ::= <return_type> <operation_name>
                   ([<argument_list>]) [<exception_raised>]
... 
<exception_raised> ::= raises(<exception_list>)
... 
```

#### Example
  ```java
  interface Professor {
    <type_property_list>
    <attribute_list>
    <relationship_list>

    grant_tenure() raises(ineligible_for_tenure);
    hire (in Professor);
    fire (in Professor) raises(no_such_employee);
  }
  ```
Object Query Language

- Relies on ODMG object model
- OQL is very close to SQL-92. Extensions concern object-oriented notions.
- High level primitives to deal with sets of objects, structures and lists.
- OQL is a functional language where operators can freely be composed, as long as the operands respect the type system.
- Not computationally complete.
- No explicit update operators (instead use operations defined on objects).
- Declarative access. Thus OQL queries can be easily optimised by virtue of this declarative nature.
- Formal semantics can easily be defined

Language Description

- A query: a (possibly empty) set of query definition expressions followed by an expression. The result of a query is an object with or without identity.

Notation:

- q: query name  
- a: atom  
- e: expression  
- t: type name  
- p: property name  
- f: operation name  
- x: variable
Language Description (2)

// Query definition expressions are of the form:

define q as e

// Example

define Joe as element (select x from x in Students where x.name = “Joe”)

// Elementary expressions: a variable x, an atom, a named object or a query name q.

27, nil, Students, Joe

Language Description (3)

// Construction expressions
- Object: t(p₁: e₁, ..., pₙ: eₙ)
  type of eᵢ must be compatible with pᵢ
  Employee (name:”Peter”, boss:”Paul”)
- Structure: struct(p₁: e₁, ..., pₙ: eₙ)
  struct (name:”Peter”, age:25)
- Set: set(p₁: e₁, ..., pₙ: eₙ)
  set(1,2,3)
- Bag: bag( e₁, ..., eₙ)
  bag(1,1,2,3,3)
- List: list( e₁, ..., eₙ)
  list(1,1,2,3,3)
- Array: array( e₁, ..., eₙ)
  array(1,1,2,3,3)

// Arithmetic expressions
- unary expressions: <op>e
  not(true)
- Binary expressions: e₁ <op> e₂
  count(Students) - count(TA)
Language Description (4)

Collection expressions

- Universal quantification: for all x in e₁: e₂
  
  for all x in Students: x.student_id > 0

- Existential quantification: exists x in e₁: e₂
  
  exists x in Doe.takes: x.taught_by.name = "Turing"

- Membership testing: e₁ in e₂
  
  Joe in TA

- Select From Where:
  
  select e from x₁ in e₁, ..., xₙ in eₙ where e'

  select distinct e from x₁ in e₁, ..., xₙ in eₙ where e'

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

- Sort-by: sort x in e by e₁, ..., eₙ
  
  sort x in Persons by x.age, x.name

- Unary set operator:
  
  <op> (e)

  <op> ∈ {min, max, count, sum, avg}

  max(select x.salary from x in Professors)

Language Description (5)

Collection expressions (cont.)

- Group_by:
  
  group x in e by (p₁: e₁, ..., pₙ: eₙ)
  
  with (p₁: e₁, ..., pₙ: eₙ)

  group x in Employees
  
  by (low: x.salary < 1000,
       medium: x.salary >= 1000 and x.salary < 10000,
       high: x.salary >= 10000)

  returns

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

  group e in Employees
  
  by (department: e.deptno)
  
  with (avg_salary: avg(select x.salary from x in partition))

  returns

  set<struct(department: integer, avg_salary: float)>
Indexed Collection Expressions

- **Get the i-th Element:** \( e_i[ e_2 ] \)  
  \[ \text{list}\{a,b,c,d\} \]  
  \[ 1 \]  
  \[ \text{element}\{\text{select } x \} \]  
  \[ \text{from } x \text{ in } \text{course} \]  
  \[ \text{where } x.\text{name} = \text{"math" and } x.\text{number} = \text{"101"}.\text{requires}[2]\} \]

- **Extracting a subcollection:** \( e_i[ e_2 : e_3 ] \)  
  \[ \text{list}\{a,b,c,d\}[1:3] \]  
  \[ \text{element}\{\text{select } x \} \]  
  \[ \text{from } x \text{ in } \text{course} \]  
  \[ \text{where } x.\text{name} = \text{"math" and } x.\text{number} = \text{"101"}.\text{requires}[1:2]\} \]

- **Last and first:**  
  \[ \text{first}(e), \text{last}(e) \]  
  \[ \text{first}\{\text{element}\{\text{select } x \} \} \]  
  \[ \text{from } x \text{ in } \text{course} \]  
  \[ \text{where } x.\text{name} = \text{"math" and } x.\text{number} = \text{"101"}.\text{requires}\} \]

- **Concatenating:**  
  \[ e_1 + e_2 \]  
  \[ \text{list}\{1,2\}+\text{list}\{2,3\} \]

Binary set expressions:

\[ e_1 <\text{op}> e_2 \]  
\[ <\text{op}> \in \{\text{union, except, intersect}\} \text{ Student except TA} \]

\[ \text{bag}\{2,2,3,3,3\} \text{ union bag}\{2,3,3,3\} = \text{bag}\{2,2,3,3,3,2,3,3,3\} \]

\[ \text{bag}\{2,2,3,3,3\} \text{ intersect bag}\{2,3,3,3\} = \text{bag}\{2,3,3,3\} \]

\[ \text{bag}\{2,2,3,3,3\} \text{ except bag}\{2,3,3,3\} = \text{bag}\{2\} \]

Structure expressions:

\[ e \rightarrow p, e.p \]

\[ \text{Joe.name} \]

\[ \text{Joe->name} \]
### Language Description (8)

#### Conversion expressions
- **Extracting the element of a singleton:** \(\text{element}(e)\)
  \[
  \text{element}(\text{select } x \text{ from } x \text{ in Professors where } x. \text{name} = "\text{Turing}")
  \]
- **List to set:** \(\text{listtoset}(e)\)
  \[
  \text{listtoset}(\text{list}(1,2,3,2))
  \]
- **Flattening:** \(\text{flatten}(e)\)
  \[
  \text{flatten}([1,2,3], [3,4,5,6], [7]) = [1,2,3,4,5,6,7],
  \text{flatten}([1,2],[1,2,3]) = [1,2,1,2,3],
  \text{flatten}([1,2],[1,2,3]) = [1,2,3]
  \]
- **Typing and expression:** \((t)e\)
  \[
  \text{select } ((\text{Employee}) s).\text{salary}
  \text{from } s \text{ in Students}
  \text{where } s \text{ in } (\text{select sec.assistant from sec in Sections})
  \]
- **Operation expressions:** \(e \rightarrow f, e.f\)
  \[
  \text{jones} \rightarrow \text{number_of_students}
  \]

#### Examples

- **select** distinct x.age
  from x in Persons
  where x.name = "Pat"
  - returns a literal of type Set<integer>

- **select** distinct struct(a:x.age, s:x.sex)
  from x in Persons
  where x.name = "Pat"
  - returns a literal of type Set<struct>

- **select** distinct
  
  struct(name:x.name, hps:( select y
  from y in x.subordinates
  where y.salary > 100000))
  
  from x in Employees
  - returns a literal of type Set<struct(name:string, hps:bag<Employee>>
Examples (2)

```csharp
// select struct (a:x.age, s:x.sex) from x in (select y from y in Employees where y.seniority = "10") where x.name = "Pat"
// returns a literal of type bag<struct>

Chairman
// returns the Chairman object (just the one, presumably!)

Chairman.subordinates
// returns the set of subordinates of the Chairman

Persons
// returns the set of all persons
```

Examples (3)

Consider the DreamHome application (an application you have become experts in, hopefully!):

// To get a set of all staff:
staff

// To get a set of all branch managers:
branch.offices.ManagedBy

// To get a set of all staff who live in London:
```
define Londoners as
   select x from x in staff
   where x.address.city = "London"
select x.name from x in Londoners
```

This returns a literal of type set<string>
Examples (4)

To get a structured set containing name, sex and age for all staff who live in London:

```sql
select struct (n:x.name, s:x.sex, a:x.age)
from x in staff
where x.address.city = "london"
```

This returns a literal of type set<struct>

Object Identity

- Mutable Object has an OID
- Literal: identity = their value
- Creating an object:
  - To create an object with identity: a type name constructor is used.
    Person (name: "Pat", birthdate: "3/28/56", salary: 100,000)
  - Build objects from a query:
    ```
    retirer (   select struct(n:x.name, a:x.age, s:x.sex)
                from x in persons
                where x.age > 60)
    ```
  - Objects without identity are created using struct:
    ```
    struct(a: 10, b: "pat")
    ```
Objects in SQL3

OQL extends C++ with database concepts, while SQL3 extends SQL with OO concepts.

Systems using the SQL3 philosophy are called object-relational

All major relational vendors have something of this kind, allowing any class to become the type of a column:

- Informix Data Blades
- Oracle Cartridges
- Sybase Plug-Ins
- IBM/DB2 Extenders

Two Levels of SQL3 Objects

1. For tuples of relations = “row types”
2. For columns of relations = “types”
   » But row types can also be used as column types.

References: Row types can have references

- If T is a row type, then \texttt{REF(T)} is the type of a reference to a T object.
- Unlike OO systems, refs are values that can be seen by queries.
Example of Row Types

CREATE RO TYPE BarType (  
    name CHAR(20) UNIQUE,  
    addr CHAR(20)  
);  
CREATE ROW TYPE BeerType (  
    name CHAR(20) UNIQUE,  
    manf CHAR(20)  
);  
CREATE ROW TYPE MenuType (  
    bar REF(BarType),  
    bee REF(BeerType),  
    price FLOAT  
);
Dereferencing

\( A \rightarrow B \) = the B attribute of the object referred to by reference A.

Example: Find the beers served by Joe.

```sql
SELECT beer \rightarrow name
FROM Sells
WHERE bar \rightarrow name = 'Joe’s Bar';
```

ADTs in SQL3

Allow types with methods in columns of a relation.

Intended application: data that doesn’t fit relational model well, e.g., locations, signals, images, etc.
The type itself is usually a multi-attribute tuple.
Type declaration:

```sql
CREATE TYPE <name> (  
    attributes  
    method declarations or definitions 
);  
```

Methods defined in a PL/SQL-like language
CREATE TYPE BeerADT (  
    name CHAR(20),  
    manf CHAR(20),  
    FUNCTION newBeer(  
        : n CHAR(20),  
        :m CHAR(20)  
    )  
    RETURNS BeerADT;  
    :b BeerADT; /* local decl. */  
BEGIN  
    :b := BeerADT(); /* built-in constructor */  
    :b.name := :n;  
    :b.manf := :m;  
    RETURN :b;  
END;  
FUNCTION getMinPrice(:b BeerADT)  
    RETURNS FLOAT;  
BEGIN  
    SELECT MIN(price) INTO :p  
    FROM Sells  
    WHERE beer = :b.name;  
    RETURN :p;  
END;