3.1 ODL: OBJECT DEFINITION LANGUAGE

Object Definition Language (ODL) is the specification language defining the interface to object types conforming to the ODMG Object Model. Often abbreviated by the acronym ODL.This language's purpose is to define the structure of an Entity-relationship diagram.

Class Declarations

interface < name > {elements = attributes, relationships, methods }

Element Declarations

- attribute < type > < name > ;
- **relationship < rangetype > < name >** ;
- float gpa(in: Student) raises(noGrades) float = return type.
- in: indicates Student argument is read-only. Other options: out, inout.

Relationships

- use inverse to specify inverse relationships
- at most one' semantics remain
- multiplicity
  - if many-many between C and D, then use Set<D> and Set<C>, respectively
  - if many-one from C to D, then use D in C and Set<C> in D
  - if many-one from D to C, then use C in D and Set<D> in C
  - if one-one between C and D, then use D and C, respectively

Datatypes

- Basis

-Atomic: integer, float, character, character string, boolean, and enumeration o classes

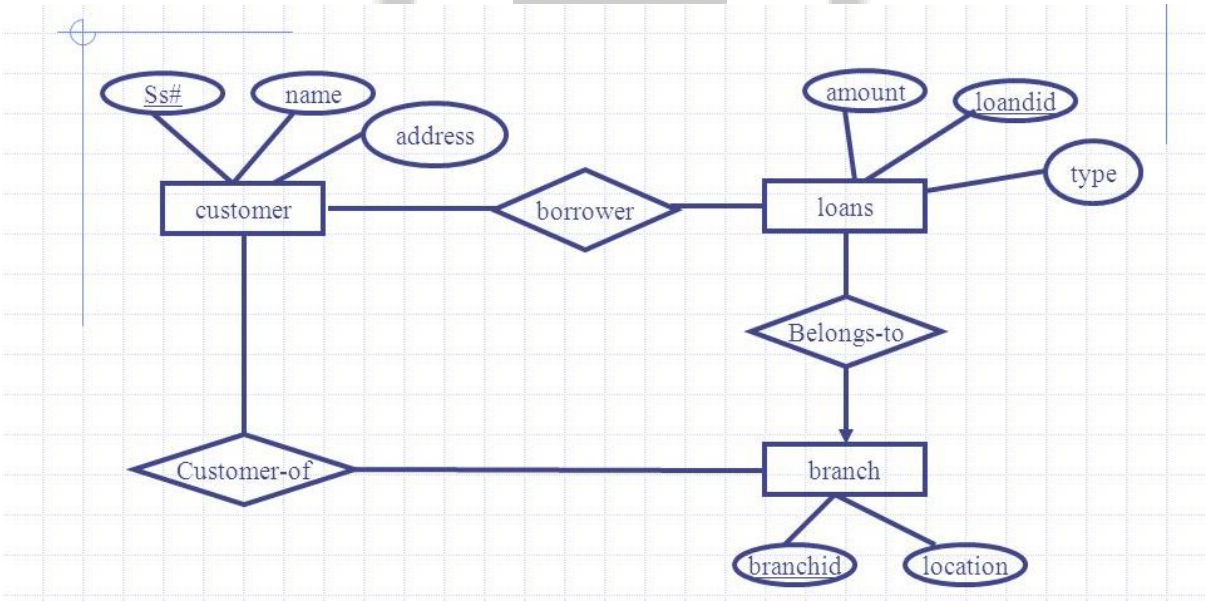
- type constructors (can be composed to create complex types)
  - set: Set<T>
  - bag: Bag<T>
  - list: List<T> (sequential access)
  - array: Array<T,i> (random access)
  - dictionary: Dictionary<T,S>

- structures

- Rules for types and relationships
  - Type of a relationship is either a class type or a (single use of a) collection type constructor applied to a class type' [FCDB]
  - Type of an attribute is built starting with atomic type(s)' [FCDB] **Relationship types cannot involve the following**
    - atomic types (e.g., Set<integer>),
    - structures (e.g., Struct N {Movie field1, Star field2}, or
    - two applications of collection types (e.g., Set<Array<Star, 10>>) **Similarities between E/R and ODL**
      - both support all multiplicities of relationships
      - both support inheritance

Differences between E/R and ODL:

- Attributes are single-valued in E/R; can be multivalued in ODL
- Keys required in E/R, optional in ODL
- Can only model one key in E/R; can model all (or any subset) in ODL
- Multiway relationships supported in E/R; only binary relationships supported in ODL
- No multiple inheritance in E/R; ODL supports multiple inheritance



Keys: ss#, loanid, branched

Cardinality constraint: each loan belongs to a single branch.

3.2 OQL - OBJECT QUERY LANGUAGE

OQL is the way to access data in an O2 database. OQL is a powerful and easy-to-use SQL-like query language with special features dealing with complex objects, values and methods.

SELECT, FROM, WHERE SELECT

<list of values>

FROM <list of collections and variable assignments>

WHERE <condition>

The SELECT clause extracts those elements of a collection meeting a specific condition. By using the keyword DISTINCT duplicated elements in the resulting collection get eliminated. Collections in FROM can be either extents (persistent names - sets) or expressions that evaluate to a collection (a set). Strings are enclosed in double-quotes in OQL. We can rename a field by if we prefix the path with the desired name and a colon.

Example Query 1

Give the names of people who are older than 26 years

old: SELECT SName: p.name

FROM p in People

WHERE p.age > 26 (hit Ctrl-D)

Dot Notation & Path Expressions

We use the dot notation and path expressions to access components of complex values.

Let variables t and ta range over objects in extents (persistent names) of Tutors and TAs (i.e., range over objects in sets Tutors and TAs).

ta.salary -> real

t.students -> set of tuples of type tuple(name: string, fee: real) representing students

t.salary -> real

Cascade of dots can be used if all names represent objects and not a collection.

Example Query 2

Find the names of the students of all tutors:

```
SELECT s.name
FROM Tutors t, t.students s
```

Here we notice that the variable t that binds to the first collection of FROM is used to help us define the second collection s. Because students is a collection, we use it in the FROM list, like t.students above, if we want to access attributes of students.

Subqueries in FROM Clause

Example Query 3

Give the names of the Tutors which have a salary greater than $300 and have a student paying more than $30:

```
SELECT t.name
FROM ( SELECT t FROM Tutors t WHERE t.salary > 300 ) r, r.students s WHERE s.fee > 30
```

Subqueries in WHERE Clause

Example Query 4

Give the names of people who aren't TAs:

```
SELECT p.name
FROM p in People
WHERE not ( p.name in SELECT t.name FROM t in TAs )
```

Set Operations and Aggregation

The standard O2C operators for sets are + (union), * (intersection), and - (difference). In OQL, the operators are written as UNION, INTERSECT and EXCEPT , respectively.

Example Query 5

Give the names of TAs with the highest salary:

```
SELECT t.name
FROM t in TAs
WHERE t.salary = max ( select ta.salary from ta in TAs )
```

GROUP BY

The GROUP BY operator creates a set of tuples with two fields. The first has the type of the specified GROUP BY attribute. The second field is the set of tuples that match that attribute. By default, the second field is called PARTITION.

Example Query 6

Give the names of the students and the average fee they pay their Tutors:

```
SELECT sname, avgFee: AVG(SELECT p.s.fee
FROM partition p)  FROM t in Tutors, t.students s
GROUP BY sname: s.name
```

Embedded OQL

Instead of using query mode, you can incorporate these queries in your O2 programs using the "o2query" command:

```
run body {  o2 real total_salaries;
o2query( total_salaries, "sum ( SELECT ta->get_salary \
FROM ta in TAs )" );
printf("TAs combined salary: %.2f\n", total_salaries);
};
```

The first argument for o2query is the variable in which you want to store the query results. The second argument is a string that contains the query to be performed. If your query string takes up several lines, be sure to backslash (\) the carriage returns.