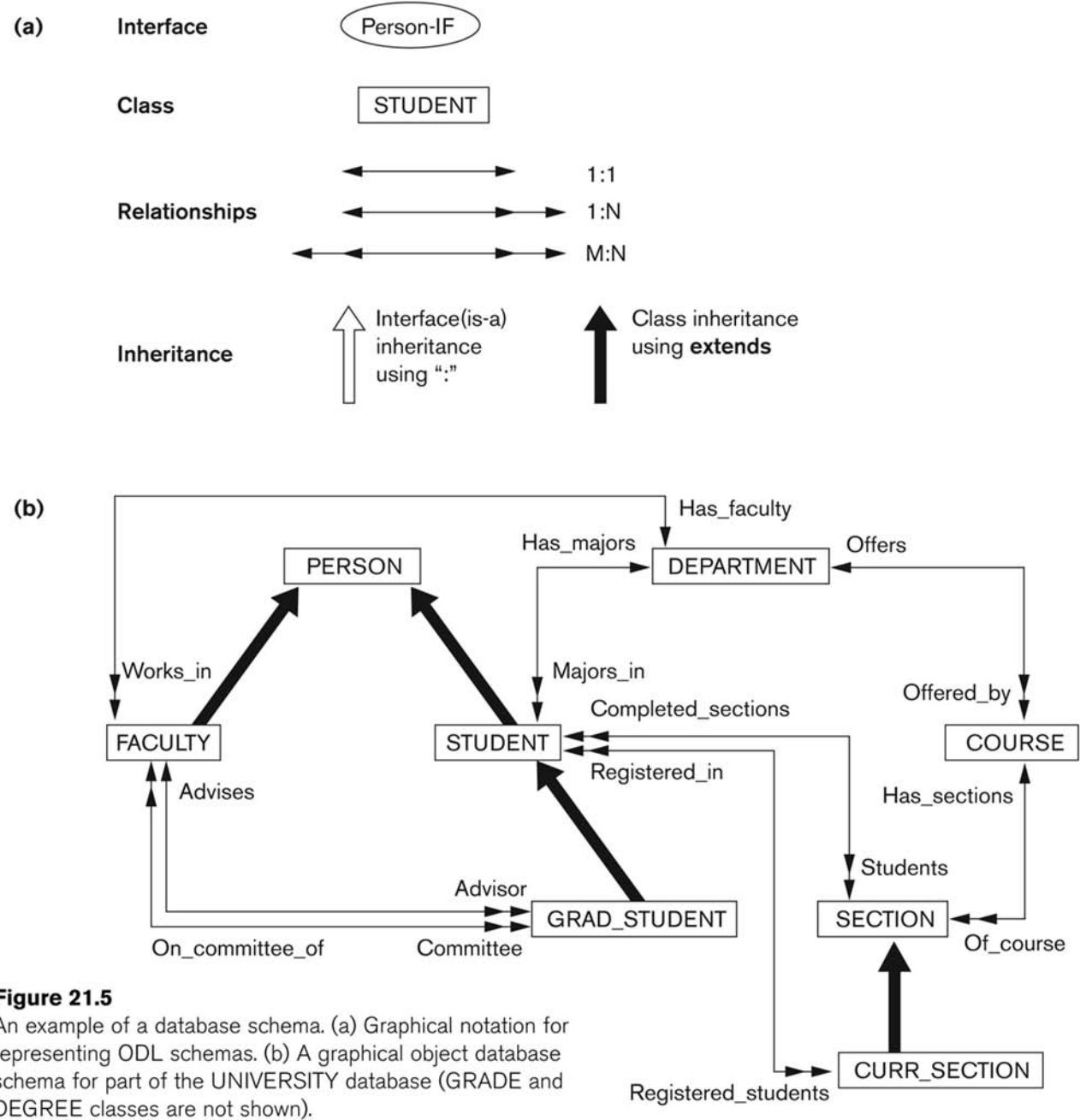

Figure 21.5
An example of a database schema. (a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).

Figure 21.6
Possible ODL schema for the UNIVERSITY database of Figure 21.5(b) (continued).

```
class PERSON
(  extent       PERSONS
   key          Ssn )
{  attribute    struct Pname {   string   Fname,
                                 string   Mname,
                                 string   Lname  }    Name;
   attribute    string                                Ssn;
   attribute    date                                  Birth_date;
   attribute    enum Gender{M, F}                     Sex;
   attribute    struct Address {  short    No,
                                  string   Street,
                                  short    Apt_no,
                                  string   City,
                                  string   State,
                                  short    Zip  }     Address;
   short        Age();   };

class FACULTY extends PERSON
(  extent       FACULTY )
{  attribute    string          Rank;
   attribute    float           Salary;
   attribute    string          Office;
   attribute    string          Phone;
   relationship DEPARTMENT      Works_in inverse DEPARTMENT::Has faculty;
   relationship set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
   relationship set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
   void         give_raise(in float raise);
   void         promote(in string new rank);   };

class GRADE
(  extent       GRADES )
{
   attribute    enum GradeValues{A,B,C,D,F,I, P} Grade;
   relationship SECTION Section inverse SECTION::Students;
   relationship STUDENT Student inverse STUDENT::Completed_sections;    };

class STUDENT extends PERSON
(  extent       STUDENTS )
{  attribute    string          Class;
   attribute    Department      Minors_in;
   relationship Department Majors_in inverse DEPARTMENT::Has_majors;
   relationship set<GRADE> Completed_sections inverse GRADE::Student;
   relationship set<CURR_SECTION> Registered_in inverse CURR_SECTION::Registered_students;
   void         change_major(in string dname) raises(dname_not_valid);
   float        gpa();
   void         register(in short secno) raises(section_not_valid);
   void         assign_grade(in short secno; in GradeValue grade)
                   raises(section_not_valid,grade_not_valid);   };
```

Figure 21.6 (continued)
Possible ODL schema for the UNIVERSITY database of Figure 21.5(b).

```
class DEGREE
{  attribute      string      College;
   attribute      string      Degree;
   attribute      string      Year;    };

class GRAD_STUDENT extends STUDENT
(  extent         GRAD_STUDENTS  )
{  attribute  set<Degree>      Degrees;
   relationship Faculty_advisor inverse FACULTY::Advises;
   relationship  set<FACULTY> Committee inverse FACULTY::On_committee_of;
   void          assign_advisor(in string Lname; in string Fname)
                    raises(faculty_not_valid);
   void          assign_committee_member(in string Lname; in string Fname)
                    raises(faculty_not_valid);  };

class DEPARTMENT
(  extent         DEPARTMENTS
   key            Dname )
{  attribute      string      Dname;
   attribute      string      Dphone;
   attribute      string      Doffice;
   attribute      string      College;
   attribute      FACULTY     Chair;
   relationship   set<FACULTY> Has_faculty inverse FACULTY::Works_in;
   relationship   set<STUDENT> Has_majors inverse STUDENT::Majors_in;
   relationship   set<COURSE> Offers inverse COURSE::Offered_by;  };

class COURSE
(  extent         COURSES
   key            Cno )
{  attribute      string      Cname;
   attribute      string      Cno;
   attribute      string      Description;
   relationship set<SECTION> Has_sections inverse SECTION::Of_course;
   relationship <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };

class SECTION
(  extent         SECTIONS  )
{  attribute      short       Sec_no;
   attribute      string      Year;
   attribute      enum Quarter{Fall, Winter, Spring, Summer}
                              Qtr;
   relationship set<Grade> Students inverse Grade::Section;
   relationship COURSE Of_course inverse COURSE::Has_sections;  };

class CURR_SECTION extends SECTION
(  extent         CURRENT_SECTIONS  )
{  relationship set<STUDENT> Registered_students
                    inverse STUDENT::Registered_in
   void          register_student(in string Ssn)
                    raises(student_not_valid, section_full);  };
```

Object Query Language (OQL)

- OQL is ODMG's query language
- OQL works closely with programming languages such as C++
- Embedded OQL statements return objects that are compatible with the type system of the host language
- OQL's syntax is similar to SQL with additional features for objects

Simple OQL Queries

- Basic syntax: SELECT…FROM…WHERE…
  - SELECT d.name
  - FROM d in departments
  - WHERE d.college = ‘Engineering’;
- An **entry point** to the database is needed for each query, it can be any **named persistent object**
- An **`extent`** name (e.g., departments in the above example) may serve as an entry point

Iterator Variables

- Iterator variables are defined whenever a collection is referenced in an OQL query
- In the previous example d serves as an iterator and ranges over each object in the collection
- Syntactical options for specifying an iterator:
  - `d in departments`
  - `departments d`
  - `departments as d`

Data Type of Query Results

- The data type of a query result can be any type defined in the ODMG model
- A query does not have to follow the `SELECT … FROM … WHERE …` format
- A persistent name on its own can serve as a query whose result is a reference to the persistent object.
  Example:
  - departments; CS_department;
    whose types are set<Departments>, Department resp.

Path Expressions

- A **path expression** is used to specify a path to attributes and objects in an entry point
- A path expression starts at a persistent object name, or at an iterator variable
- The name will be followed by zero or more relationship or attribute names, connected using the **dot notation**
  - CS_department.Chair; (returns a Faculty object)
  - CS_department.Chair.Rank; (returns a string)
  - CS_department.Has_faculty;
    (returns a set<Faculty> object)

- CS_department.Has_faculty.Rank should give the ranks of the CS dpt. Faculty 
- The object returned would have an ambiguous type: set<string> or bag<string>
- We need to use an iterator variable
  select distinct F.Rank
  from F in CS_department.Has_faculty;
- distinct → set<string> duplicate elimination
- Example of an interator variable defined in the from clause to range over a restricted collection

- In general, an OQL query can return a result with a complex structure specified in the query itself, using **struct**
  Example: CS_department.Chair.Advises;
  returns an object of type set<GRAD_STUDENT>
- Retrieve the names and a list of previous degrees of each graduate student: degrees is defined by an embedded query

```
select struct ( name : struct (lname:S.name.Lname,
                               fname:S.name.Fname),
                degrees : (select struct (deg:D.Degree,
                                          yr:D.Year, clg:D.College)
                           from D in S.Degrees) )
from S in CS_department.Chair.Advises;
```

- The iterators S, D range over the corresp. collections

- Attributes, relationships and operation names can be used interchangeably within path expressions, as long as the OQL type system is not violated.
- **select** struct(lname:S.name.Lname,fname:S.name.Fname,gpa:S.gpa)
  from S **in** CS_department.Has_majors
  where S.Class="senior"
  order by gpa **desc**, lname **asc**, fname **asc**;
- Retrieve the names and the GPA of all senior students majoring in CS, ordered by GPA

Views as Named Objects

- The **define** keyword in OQL is used to specify an identifier for a **named query**
- The name should be unique; if not, the results will replace an existing named query
- Once a query definition is created, it will persist until deleted or redefined
- A view definition can include parameters (arguments)

An Example of an OQL View

- A view (== named query) to retrieve the set of students minoring in a given department:

```
define    has_minor(deptName) as
select    S
from      S in STUDENTS
where     S.Minors_in.Dname=deptName
```

- has_minor can now be used in OQL queries:
- **has_minor('Computer Science');**
- Returns a set of students minoring in CS
- Model inverse rels. that are not used frequently.

Single Elements from Collections

- An OQL query returns a collection
- OQL's `element` operator can be used to return a single element from a singleton collection that contains one element:

```
element(select d
        from   d in DEPARTMENTS
        where  d.dname = 'Computer Science');
```

- If the collection is empty or has more than one elements, an **exception** is raised
- Since a dpt. name is unique across all dpts. the result should be one department.

Collection Operators, Aggregate Functions

- OQL supports a number of aggregate operators that can be applied to query results
- The aggregate operators operate over a collection and include
  - `min, max, count, sum, avg`
- `count` returns an integer type
- `min, max, sum, avg`
  return the same type as the operand collection type

Examples of OQL Aggregate Operators

- The number of students minoring in CS:

```
count(S in has_minor('Computer Science'));
```

- The average GPA of all senior students majoring in Business:

```
avg (select s.gpa
     from   s in STUDENTS
     where  s.class = 'senior' and
       s.Majors_in.Dname ='Business');
```

- Aggregate operators can be applied to any collection of the appropriate type and can be used in any part of the query:

```
select   D.Dname
from     D in DEPARTMENTS
where    count(D.Has_majors) > 100;
```

- Retrieve all dept. names that have more than 100 majors.

Membership and Quantification

- OQL provides membership and quantification operators that return a Boolean type, T/F
  - (e **`in`** c)
    returns true if e is a member of the collection c
  - (**`for all`** e **`in`** c: b)
    returns true if all e elements of collection c satisfy b
  - (**`exists`** e **`in`** c: b)
    returns true if at least one e in collection c satisfies b

An Example of Membership

- Retrieve the names of all students who completed DB1:

```
select s.Pname.Fname, s.Pname.Lname
from   s in STUDENTS
where  'DB1' in
  (select c.Cname
   from   c in
  s.Completed_sections.Section.of_course);
```

Queries returning T/F results

- Is Jeremy a CS student?
- `Jeremy in has_minor('Computer Science'));`
- Are all CS grad. Students advised by CS faculty?
- ```
  for all G in
  (select   S
   from     S in GRAD_STUDENTS
   where    S.Majors_in.Dname="CS")
   :  G.Advisor in CS_DEPARTMENT.Has_faculty;
  ```
- An illustration of inheritance: S.Majors_in
  ```

An exists query

- Does any graduate CS major have a GPA >= 4?
-
```
exists G in
(select   S
 from     S in GRAD_STUDENTS
 where    S.Majors_in.Dname="CS")
 :  G.Gpa >= 4;
```

Ordered Collection Expressions

- Collections that are lists or arrays allow retrieving their **first**, **last**, and **ith** elements
- OQL provides additional operators for extracting a sub-collection and concatenating two lists
- Query expressions that involve lists or arrays can invoke these operations
- OQL also provides operators for ordering the results

An Example of Ordered Collection

- Retrieve the last name of the faculty member who earns the highest salary:
- (assuming there is only one such person)

```
first (select struct
            (lname: f.Pname.Lname,
                    salary:f.Salary)
from        f in FACULTY
order by     f.Salary desc);
```

Another Example of Ordered Collection

- Retrieve the top three CS majors, based on GPA

```
(select struct
        (lname:f.Pname.Lname,gpa:s.Gpa)
from      s in CS_department.Has_majors
order by gpa desc)[0:2];
```