Airline DB

June 2009  FP for DB  Case studies 1

airports & flight connections
-- airlines as abstract entities

data Airline = BA | UA | NZ deriving ( Eq, Show)
allAirlines :: [Airline]
allAirlines = [BA, UA, NZ]

type AirlineName = String
airlineName :: Airline -> AirlineName
airlineName BA = "British Airways"
airlineName UA = "United Airlines"
airlineName NZ = "Air New Zealand"

-- airports as abstract entities

data Airport = LHR | JFK | DEN | LAX | AKL
deriving ( Eq, Show)
allAirports :: [Airport]
allAirports = [LHR, JFK, DEN, LAX, AKL]

type AirportName = String
type Country = String
type AirportInfo = (AirportName, Country)

airportInfo :: Airport -> AirportInfo
airportInfo LHR = ("London Heathrow", "England")
airportInfo JFK = ("J F Kennedy", "United States")
airportInfo DEN = ("Denver", "United States")
airportInfo LAX = ("Los Angeles Int", "United States")
airportInfo AKL = ("Auckland", "New Zealand")

airportName :: Airport -> AirportName
airportName x = firstOf2 (airportInfo x)

airportCountry :: Airport -> Country
airportCountry x = secondOf2 (airportInfo x)
-- flights as abstract entities (airline, source, destination)

data Flight = BA1 | UA1 | UA123 | UA987 | UA234 | UA842 | NZ2
deriving (Eq, Show)
allFlights :: [Flight]
allFlights = [BA1, UA1, UA123, UA987, UA234, UA842, NZ2]

flightInfo :: Flight -> (Airline, Airport, Airport)
flightInfo BA1 = (BA, LHR, JFK)
flightInfo UA1 = (UA, LHR, JFK)
flightInfo UA123 = (UA, JFK, DEN)
flightInfo UA987 = (UA, LHR, LAX)
flightInfo UA234 = (UA, DEN, LAX)
flightInfo UA842 = (UA, LAX, AKL)
flightInfo NZ2 = (NZ, LAX, AKL)

flightAirline :: Flight -> Airline
flightAirline f = firstOf3 (flightInfo f)

flightSource :: Flight -> Airport
flightSource f = secondOf3 (flightInfo f)

flightDest :: Flight -> Airport
flightDest f = thirdOf3 (flightInfo f)

-- codes of the airports located in the United States

allAirports = [LHR, JFK, DEN, LAX, AKL]

airportInfo LHR = ("London Heathrow", "England")
airportInfo JFK = ("J F Kennedy", "United States")
airportInfo DEN = ("Denver", "United States")
airportInfo LAX = ("Los Angeles Int", "United States")
airportInfo AKL = ("Auckland", "New Zealand")

airportCountry x = secondOf2 (airportInfo x)

[p | p <- allAirports, airportCountry p = "United States"]
-- all airports flown to/from by a given airline

\[ \text{allFlights} = \{ \text{BA1, UA1, UA123, UA987, UA234, UA842, NZ2} \} \]

\[
\begin{align*}
\text{flightInfo} & : \text{Flight} \rightarrow (\text{Airline, Airport, Airport}) \\
\text{flightInfo} \text{ BA1} & = (\text{BA, LHR, JFK}) \\
\text{flightInfo} \text{ UA1} & = (\text{UA, LHR, JFK}) \\
\text{flightInfo} \text{ UA123} & = (\text{UA, JFK, DEN}) \\
\text{flightInfo} \text{ UA987} & = (\text{UA, LHR, LAX}) \\
\text{flightInfo} \text{ UA234} & = (\text{UA, DEN, LAX}) \\
\text{flightInfo} \text{ UA842} & = (\text{UA, LAX, AKL}) \\
\text{flightInfo} \text{ NZ2} & = (\text{NZ, LAX, AKL}) \\
\text{flightSource} \ f & = \text{secondOf3} (\text{flightInfo} \ f) \\
\text{flightDest} \ f & = \text{thirdOf3} (\text{flightInfo} \ f)
\end{align*}
\]

\[
\begin{align*}
\text{serves} & : \text{Airline} \rightarrow [\text{Airport} ] \\
\text{serves} \ x & = \\
& \{ [\text{flightSource} \ f | f \leftarrow \text{allFlights}, \text{flightAirline} \ f = x] \} \uplus \\
& \{ [\text{flightDest} \ f | f \leftarrow \text{allFlights}, \text{flightAirline} \ f = x] \}
\end{align*}
\]

-- all airports from where an airline flies to more than one destination

\[
\begin{align*}
\text{hubs} & : \text{Airline} \rightarrow [\text{Airport} ] \\
\text{hubs} \ x & = \\
& \{ p | p \leftarrow \text{allAirports}, \ f1 \leftarrow \text{allFlights}, \text{flightAirline} \ f1 = x, \text{flightSource} \ f1 = p, \ f2 \leftarrow \text{allFlights}, \text{flightAirline} \ f2 = x, \text{flightSource} \ f2 = p, \text{flightDest} \ f1 \neq \text{flightDest} \ f2 \}
\end{align*}
\]
-- all airports reachable from a given airport on a given airline

getthere :: Airline -> Airport -> [Airport]
getthere x y =
    dests ++ [y' | d <- dests, y' <- getthere x d]
where dests = [flightDest f | f <- allFlights,
    flightAirline f == x, flightSource f == y]

relational airline DB

<table>
<thead>
<tr>
<th>LINE</th>
<th>ID</th>
<th>NAME</th>
<th>PORT</th>
<th>CODE</th>
<th>NAME</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BA</td>
<td>British</td>
<td></td>
<td>LHR</td>
<td>Heathrow</td>
<td>England</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONNECT</th>
<th>ORIG</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>LHR</td>
<td>VIE</td>
</tr>
<tr>
<td>704</td>
<td>BA</td>
<td></td>
</tr>
</tbody>
</table>
-- airports located in the United States

[ p | p <- allAirports, 
    airportCountry p = “United States”]

Π (σ PORT (COUNTRY = ‘United States’)) ID

select ID from PORT
where COUNTRY = “United States”

-- airports served by a given airline

serves x =
[flightSource f | f <- allFlights, flightAirline f == x]
++ [flightDest f | f <- allFlights, flightAirline f == x]

Π (σ ((LINE ▶ PORT) ▶ CONNECT)
    (CODE = ORIG or CODE = DEST))
    NAME

select distinct PORT.NAME
from LINE, PORT, CONNECT
where ID = L-ID
and (CODE = ORIG or CODE = DEST)
and LINE.NAME = x
**-- airports from where an airline flies to more than one destination**

hubs :: Airline -> [Airport]
hubs x = [p | p <- allAirports, f1 <- allFlights, flightAirline f1 == x, flightSource f1 == p, f2 <- allFlights, flightAirline f2 == x, flightSource f2 == p, flightDest f1 /= flightDest f2]

A ::= Π (σ (CONNECT (L-ID = x))) (ORIG, DEST)
returns all connection pairs for x - but R/Algebra does not provide tools for grouping or counting

select ORIG from CONNECT where L-ID = x
group by ORIG having count (*) > 1

**-- all airports reachable from a given airport on a given airline**

select DEST from CONNECT where L-ID = x and ORIG = y (Blue AKL) -> LHR
getthere \( x, y = \) dests ++ \([y' \mid d \leftarrow \text{dests}, y' \leftarrow \text{getthere } x, d]\) where dests = \([\text{flightDest } f \mid f \leftarrow \text{allFlights}, \text{flightAirline } f = x, \text{flightSource } f = y]\)

SQL> select * from GRAPH;

<table>
<thead>
<tr>
<th>ORIG</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKL</td>
<td>LHR</td>
</tr>
<tr>
<td>LHR</td>
<td>JFK</td>
</tr>
<tr>
<td>LHR</td>
<td>LAX</td>
</tr>
<tr>
<td>JFK</td>
<td>VIE</td>
</tr>
<tr>
<td>VIE</td>
<td>WAW</td>
</tr>
</tbody>
</table>

SQL> select level, dest from graph
where prior dest = orig
start with orig = 'AKL';

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LHR</td>
</tr>
<tr>
<td>2</td>
<td>JFK</td>
</tr>
<tr>
<td>3</td>
<td>VIE</td>
</tr>
<tr>
<td>4</td>
<td>WAW</td>
</tr>
<tr>
<td>2</td>
<td>LAX</td>
</tr>
</tbody>
</table>

3 examples mastered by your predecessors
Employees, managers, projects

```haskell
-- employees
data Employee = E1 | E2 | E3 | E4 deriving (Eq, Show)
allEmployees :: [Employee]
allEmployees = [E1, E2, E3, E4]

type EmployeeName = String
type EmployeeSalary = Int
type EmployeeInfo = (EmployeeName, EmployeeSalary)

employeeInfo E1 = ("Karin", 30000)
employeeInfo E2 = ("John", 25000)
employeeInfo E3 = ("Mary", 22000)
employeeInfo E4 = ("Peter", 20000)
```
-- employees report to their supervisors
reportsto :: Employee -> [Employee]
reportsto E1 = []
reportsto E2 = [E1]
reportsto E3 = [E2]
reportsto E4 = [E1]

-- employees work on projects
data Project = Red | Blue deriving (Eq, Show)
allProjects :: [Project]
allProjects = [Red, Blue]

-- an employee may work on one or more project
workson :: Employee -> [Project]
workson E1 = [Red, Blue]
workson E2 = [Red]
workson E3 = [Red]
workson E4 = [Blue]

-- find all managers
-- (i.e. employees reported to = the whole tree except leaves)
managers :: [Employee]
managers = [x | emp <- allEmployees, x <- reportsto emp]

-- for a given employee find his manager, his manager's manager, and so on
manages :: Employee -> [Employee]
manages x = reportsto x ++ [man|m <- reportsto x, man <- manages m]
-- find all the employees who work on a given project

team :: Project -> [Employee]
team x = [emp | emp <- allEmployees, project <- workson emp, project == x]

-- find the names of all the managers whose employees work on a given project

names :: Project -> [EmployeeName]
names x = [getFirst (employeInfo manager) | manager <- managers, teammember <- team x, manager == teammember]
where getFirst (x, _) = x

Modules, prerequisites, teachers
data Prof = WS | HL | SP | AT deriving (Eq, Show)
allProfs :: [Prof]
allProfs = [WS, HL, SP, AT]

type ProfName = String
type ProfRoom = String
type ProfInfo = (ProfName, ProfRoom)

profInfo :: Prof -> ProfInfo
profInfo WS = ("Wayne Smith", "WHE110")
profInfo HL = ("Henry Long", "WHE115")
profInfo SP = ("Steve Pirx", "WHE 101")
profInfo AT = ("Andy Thue", "WHE 300")

profName :: Prof -> ProfName
profName a = firstOf2 (profInfo a)

profRoom :: Prof -> ProfRoom
profRoom a = secondOf2 (profInfo a)

data Subject = ADT | DM | EBUS | FP | IM deriving (Eq, Show)
allSubjects :: [Subject]
allSubjects = [ADT, DM, EBUS, FP, IM]

type ID = String
type Title = String

subjectInfo :: Subject -> (ID, Title, ProfName)
subjectInfo ADT = ("103020", "Abstract Data Types", "Wayne Smith")
subjectInfo DM = ("345730", "Data Management", "Wayne Smith")
subjectInfo EBUS = ("195640", "eBusiness", "Henry Long")
subjectInfo FP = ("338313", "Functional Programming", "Steve Pirx")
subjectInfo IM = ("672943", "Information Management", "Andy Thue")

idNr :: Subject -> ID
idNr b = firstOf3 (subjectInfo b)
title :: Subject -> Title
title b = secondOf3 (subjectInfo b)
subProf :: Subject -> ProfName
subProf b = thirdOf3 (subjectInfo b)
data PreSubject = ADT1 | FP1 | DM1 | FP2 | IM1 deriving (Eq, Show)
allPreSubjects :: [PreSubject]
allPreSubjects = [ADT1, DM1, FP1, FP2, IM1]

preInfo :: PreSubject -> (ID, ID)
preInfo ADT1 = ("103020", "672943")
preInfo FP1 = ("338313", "345730")
preInfo FP2 = ("338313", "103020")
preInfo DM1 = ("345730", "672943")
preInfo IM1 = ("672943", "195640")

subId :: PreSubject -> ID
subId c = firstOf2 (preInfo c)

reqSubId :: PreSubject -> ID
reqSubId c = secondOf2 (preInfo c)

firstOf2, secondOf2 :: (String, String) -> String
firstOf2 (x, y) = x
secondOf2 (x, y) = y

firstOf3, secondOf3, thirdOf3 :: (String, String, String) -> String
firstOf3 (x, y, z) = x
secondOf3 (x, y, z) = y
thirdOf3 (x, y, z) = z

-- all subjects taught by a given professor
allSubProf :: Prof -> [Title]
allSubProf p = [title b | b <- allSubjects, subProf b == profName p]

-- prerequisite for a subject
reqSub :: ID -> [ID]
reqSub p = [reqSubId c | c <- allPreSubjects, subId c == p]

-- subjects with no prerequisites
noReqSub :: [ID]
oReqSub = [idNr b | b <- allSubjects, reqSub (idNr b) == []]

June 2009  FP for DB  Case studies 25
--subjects that have more than one pre-requisite

moreOne :: [ID]
moreOne = [IdNr b | b <- allSubjects,
            c1 <- allPreSubjects, IdNr b == subjId c1,
            c2 <- allPreSubjects, IdNr b == subjId c2,
            reqSubId c1 /= reqSubId c2]
data Person = A | B | C | D | E | F | G deriving (Eq, Ord, Enum, Show)
allPersons :: [Person]
allPersons = [A, B, C, D, E, F, G]

type ID = Int
type SURNAME = String
type BIRTHPLACE = String
type SEX = String
type FATHERID = ID
type MOTHERID = ID

type PersonInfo = (ID, SURNAME, BIRTHPLACE, SEX, FATHERID, MOTHERID)
personInfo :: Person->PersonInfo
personInfo A = (1,"Schmidt","Linz","Female",2,3)
personInfo B = (2,"Huber","Linz","Male",4,5)
personInfo C = (3,"Huber","Wien","Female",6,7)
personInfo D = (4,"Grossvater vaeterlichseits","Traun","Male",0,0)

----------

data Education = Ed1|Ed2|Ed3|Ed4|Ed5 deriving (Eq,Ord,Enum,Show)
allEducations::[Education]
allEducations=[Ed1,Ed2,Ed3,Ed4,Ed5]
type DEGREE = String
type DISCIPLINE = String
type UNIVERSITY = String
type AWARDYEAR = Int

type EducationInfo = (Person,DEGREE,DISCIPLINE,UNIVERSITY,AWARDYEAR)
educationInfo :: Education->EducationInfo
educationInfo Ed2 = (A,"PhD","Computing","JKU",2001)
educationInfo Ed5 = (C,"BSc","Informatics","TU Wien",1982)

personeducation :: Education->Person
personeducation x = firstOf5(educationInfo x)
degree :: Education->DEGREE
degree x = secondOf5(educationInfo x)
discipline :: Education->DISCIPLINE
discipline x = thirdOf5(educationInfo x)
university :: Education->UNIVERSITY
university x = fourthOf5(educationInfo x)
awardyear :: Education->AWARDYEAR
awardyear x = fifthOf5(educationInfo x)

data Employment = Em1|Em2|Em3|Em4|Em5|Em6 deriving (Eq,Ord,Enum,Show)
allEmployments::[Employment]
allEmployments=[Em1,Em2,Em3,Em4,Em5,Em6]
type COMPANY = String
type FROM = Int
type TO = Int
type POSITION = String
type SALARY = Int
type EmploymentInfo = (Person,COMPANY,FROM,TO,POSITION,SALARY)

employmentInfo :: Employment->EmploymentInfo

personemployment :: Employment->Person
personemployment x = firstOf6(employmentInfo x)

company :: Employment->COMPANY
company x = secondOf6(employmentInfo x)

from :: Employment->FROM
from x = thirdOf6(employmentInfo x)

to :: Employment->TO
to x = fourthOf6(employmentInfo x)

position :: Employment->POSITION
position x = fifthOf6(employmentInfo x)

salary :: Employment->SALARY
salary x = sixthOf6(employmentInfo x)

ins::Person->[Person]->[Person]
ins []=[x]
ins x:y:ys = \{x==y \Rightarrow x:y:ys
\&\& otherwise \Rightarrow y:ins x ys

member::[Person]->Person->Bool
member[] y = False
member(x:xs)y=(x==y)||member xs y

distinct::[Person]->[Person]
distinct[]=[]
distinct(x:xs) = \{member (distinct xs)x = (distinct xs)
\&\& otherwise \Rightarrow ins x(distinct xs)

namesOf::[Person]->[SURNAME]
namesOf [] = []
namesOf (x:xs) = surname x : namesOf xs

firstOf5 (a,b,c,d,e) = a
secondOf5 (a,b,c,d,e) = b
thirdOf5 (a,b,c,d,e) = c
fourthOf5 (a,b,c,d,e) = d
fifthOf5 (a,b,c,d,e) = e

firstOf6 (a,b,c,d,e,f) = a
secondOf6 (a,b,c,d,e,f) = b
thirdOf6 (a,b,c,d,e,f) = c
fourthOf6 (a,b,c,d,e,f) = d
fifthOf6 (a,b,c,d,e,f) = e
sixthOf6 (a,b,c,d,e,f) = f
-- Persons at a specific University after a specific AwardYear

personsAtUniversityWithAwardYearAfter::UNIVERSITY->AWARDYEAR->[Person]
personsAtUniversityWithAwardYearAfter u a = distinct[personeducation ed|ed<-allEducations, university ed == u, awardyear ed >= a]

-- Grandparents of a specific person

personWithID::Database.ID->Person
personWithID i = head[p|p<-allPersons, i == Database.id p]

parentsOf::Person->[Person]
parentsOf p = [personWithID(fatherid p), personWithID(motherid p)]

grandParentsOf::Person->[Person]
grandParentsOf p = parentsOf(head[q|q<-allPersons, Database.id q == fatherid p]) ++ parentsOf(head[r|r<-allPersons, Database.id r == motherid p])

-- Colleagues of a specific person

employmentsOfPerson::Person->[Employment]
employmentsOfPerson p = [em|em<-allEmployments, personemployment em == p]

employmentsWithOfCompanyWithinTime::COMPANY->FROM->TO->[Employment]
employmentsWithOfCompanyWithinTime c f t = [em|em<-allEmployments, company em == c, (((f<=from em)&&(from em<t))||((f<=to em)&&(to em<=t)))]

colleaguesOfPersonEmployment::[Employment]->[Person]
colleaguesOfPersonEmployment[] = []
colleaguesOfPersonEmployment(x:xs)=[personemployment em|em<- employmentsWithOfCompanyWithinTime (company(x)) (from(x)) (to(x)), personemployment em /= personemployment x] ++ colleaguesOfPersonEmployment xs

colleaguesOfPerson::Person->[Person]
colleaguesOfPerson p = colleaguesOfPersonEmployment(employmentsOfPerson p)
Abrial’s Binary Model

KNOWLEDGE ::= 

ELEMENTARY FACTS
• John Doe was born in London on 19 Nov 1962
• The car with a number plate B1 BYE is a Ferrari

SIMPLE RULES
• Every man has necessarily two parents of whom he is the child
• A person has sometimes a spouse and if X is the spouse of Y then Y is the spouse of X
• A car has (if any) only one owner. Conversely, an owner may have zero, one or several cars

COMPLEX RULES
• The sex of a person is not subject to any change
• A single person who marries may not be single again in the future
• A person may not be, at a given time, in two different places

DEDUCTIVE RULES
• if \( x > y \) then \( \text{BIG} := x \) else \( \text{BIG} := y \)
• \( \text{square()} = \text{twice (twice ())} \)

WHEN THE MODEL DOES NOT KNOW A FACT OR A LAW ABOUT REALITY THIS DOES NOT MEAN THAT THIS FACT OR LAW DOES NOT EXISTS,

CONSEQUENCE:

IF

THE MODEL HAS EXACTLY THE SAME KNOWLEDGE OF TWO OBJECTS IT DOES NOT FOLLOW THEY ARE ONE AND THE SAME OBJECT.

THEREFORE

AN OBJECT ENTERING THE ‘PERCEPTION FIELD’ OF THE MODEL MUST IDENTIFY ITSELF AS either NEW OBJECT or ALREADY KNOWN OBJECT

THE DESCRIPTION OF AN OBJECT INSIDE THE MODEL IS GIVEN VIA THE CONNECTIONS (access functions) IT HAS WITH OTHER OBJECTS
person_of_sex (MALE) = {JOHN, PETER}
person_of_sex (FEMALE) = {JANE, MARY}
age (JOHN) = (27)
person_of_age (50) = {PETER, MARY}
child (PETER) = {JANE}
parent (JANE) = {PETER, MARY}...

CATEGORIES

JOHN, JANE, PETER, MARY are PERSONs
27, 50, 20 are NUMBERs
MALE, FEMALE are SEXes

THUS, THE STRUCTURE OF THE EXAMPLE CAN BE ABSTRACTED INTO:

AND FURTHER STILL INTO:
CONNECTIONS MAY THEMSELVES REQUIRE SOME INFORMATION

EXAMPLE: PETER was_invited_by (PAUL and JANE) to PARIS on 15Jul1993

THIS CAN BE DESCRIBED BY BUILDING A NEW CATEGORY - INVITATION AND THE FOLLOWING STRUCTURE

```plaintext
defn CATEGORIES
    PERSON = cat
    JOHN = generate PERSON
    x ← generate PERSON
    kill JOHN, kill x
```

a person has exactly one sex, one age, two parents, zero or one spouse and any number of children

r1 = rel (PERSON, SEX, sex = fun(1, 1), of_sex = fun(0, ∞))
r2 = rel (PERSON, NUMBER, age = fun(1, 1), of_age = fun(0, ∞))
r3 = rel (PERSON, PERSON, spouse = fun(0, 1), spouse)
r4 = rel (PERSON, PERSON, parent = fun(2, 2), child = fun(0, ∞))
declare
{
  member () ⇒ entity
  student () ⇒ member
  staff () ⇒ member
  course () ⇒ entity
  event () ⇒ entity
  tutorial () ⇒ event
  lecture () ⇒ event

  fn (member) ⇒ string
  sn (member) ⇒ string
  sex (member) ⇒ string

  course (student) ⇒ course
  tutorial (student) ⇒ tutorial
  mark (student, course) ⇒ integer
  field (student) ⇒ string

  title (course) ⇒ string
  lecture (course) ⇒ lecture

  day (event) ⇒ string
  slot (event) ⇒ string
  room (event) ⇒ string

  course (staff) ⇒ course
  phone (staff) ⇒ integer
  qual (staff) ⇒ string
  staff (tutorial) ⇒ staff

derived functions

\[
\text{define} \\
\{ \\
\text{staff(course)} \mapsto \text{staff such that} \\
\quad \text{some } c \in \text{course (staff)} \\
\quad \text{has } c = \text{course} \quad -- \text{inverse of} \\
\text{teacher (student)} \mapsto \text{staff (course (student))} \\
\text{tutor (student)} \rightarrow \text{staff (tutorial (student))} \\
\} -- \text{combinations of inverse, composition, recursion, transitivity}
\]

derived functions are represented by algorithms accepting arguments to compute results

retrievals

\[
-- \text{get the names of all members} \\
\text{for each } m \text{ in member} \\
\text{get fn(m), sn(m)}
\]

\[
-- \text{get surnames of all female students} \\
\text{for each } s \text{ in student} \\
\text{such that } \text{sex(s)} = 'F' \\
\text{get sn(s)}
\]
retrievals

-- get the names of those students that take a course on FDB
for each s in student
  such that
    some c in course (s)
    has title (c) = 'FDB'
get sn(s)

retrievals

-- get the titles of courses taught by Stefan
for the s in staff
  such that fn (s) = 'Stefan'
    for each c in course (s) get title(c)
-- error handling procedure is called if more than one Stefan exists
**updating - insertion**

*a new m in member*

-- creates a new member entity, adds it to the extent of member type, associates it with the variable m

*a new s in student*

-- creates a new entity, which is included in the extents of both student and member entity types

**updating - new record**

*for a new s in student*

let fn(s) = ‘Mary’
let sn(s) = ‘Jones’
let sex(s) = ‘F’
let field(s) = ‘Comp’
updating - change values

for the s in student such that
fn(s) = 'Mary' and sn(s) = 'Jones'
let tutorial(s) = the t in tutorial such that
day(t) = 'Mon' and slot(t) = '09,10' and room(t) = 'm101'

updating - adding rules

for the s in student such that
fn(s) = 'Mary' and sn(s) = 'Jones'
include course(s) = {
    the c1 in course such that title(c1) = 'Haskell'
    the c2 in course such that title(c2) = 'Prolog'
}

-- similarly exclude
constraint unique-id on
fn(member), sn(member) → unique

constraint must-be-supplied on
sex(member) → total  -- i.e. not partial

constraint must-differ on
student, staff → disjoint

constraint non-upd-sex on
sex(member) → fixed

constraint ris on
mark (student, course) →
some c in course(student)
has c = course