DATA, INFORMATION AND KNOWLEDGE IDENTIFICATION

- (2a) Identify the underlined sentences of the following paragraph as **data, information or knowledge**: "*This course of Knowledge Representation and Engineering is composed of three chapters: Introduction and Concepts, Knowledge Representation, and Knowledge Engineering. It's a six-credit course with two week hours for theory and two week hours for problems and practice. Like all the other subjects in the master, half of the practical hours will be off class. As the rest of subjects, KRE will be continuously evaluated. Continuous evaluation in KRE will consist of two theoretical-practical tests, and two practical work deliveries. The final mark will be calculated as 30% of the results of each one of the theoretical tests and 20% of each practical work. For second evaluation, there will be a single exam*."

Data (Information):

D1 – Knowledge Representation and Engineering (course name).
D2 – three (number of chapters)
D3 – Introduction and Concepts (chapter name)
D4 – Knowledge representation (chapter name)
D5 – Knowledge Engineering (chapter name)
D6 – Six (credits of the course)
D7, D7' – two (week hours of theory), two (week hours of problems and practice)
D8, D8' – theory (sort of class), Problems and Practice (sort of class)
D9 – master (host of the course)
D10 – half (practical hours), half practical hours (off-line hours)
D11 – continuously evaluated (sort of evaluation of the course)
D12 – two (practical work deliveries)
D13, D13' – 30% (part of final mark corresponding to each theoretical test), theoretical test (means of evaluation)
D14, D14' – 20% (part of final mark corresponding to each practical work), practical work (means of evaluation)
D15 – single exam (second evaluation)

Knowledge:

Knowledge representation and Engineering is a course; Knowledge representation and engineering is composed of chapters; Introduction and concepts is one chapter; knowledge representation is one chapter; knowledge engineering is one chapter; KRE has six credits; KRE has two week hours for theory; KRE has two week hours for problems and practice; half of the practical hours of KRE are off class; half of the practical hours of other subjects in the master are off class; all the subjects in the master are continuously evaluated; continuous evaluation of KRE consists of a number of theoretical practical tests and a number of work deliveries; the final mark is calculated as 30% of each theoretical tests; the final mark is calculated as 20% of each practical work; the second evaluation is a single exam.

SORTS OF KNOWLEDGE

- (3h) Find out know-what and know-how knowledge in the paragraphs: "*Chronic disease treatment divides each disease in stages. Patients that have one chronic disease are classified in one of these stages. General practitioners base their decisions in the current stage of the patient and the time this patient has been in that stage. In general, a patient that is in a mild-moderate dangerous stage (MDS) is asked to modify his/her lifestyle (diet, salt intake reduction, moderate exercise), if the patient has been in a MDS for a significant period, he/she is prescribed with one drug to minimal dosage, while the patient is not improving the dosage is increased with fix increments. If a maximal dosage is reached, then a second drug to minimal dosage is prescribed. Patients can reach treatments with 4 drugs. Patients that arrive in highly dangerous stage (HDS) are directly prescribed with one drug and recommended lifestyle changes.*"

  Know-what knowledge (facts):

  K1 - Chronic disease treatment divides each disease in stages.
  K2 - Patients that have one chronic disease are classified in one of the stages of the disease.
  K3 - General practitioners base their decisions in the current stage of the patient and the time this patient has been in the stage.
  K5 – Lifestyle modification consists of diet, salt intake reduction, and moderate exercise.
  K7 – Patients can reach treatments with 4 drugs.

  Know-how knowledge (rules):

  K4 – If a patient is in mild-moderate dangerous stage, she's asked to modify his/her lifestyle.
  K4' – If a patient is in mild-moderate dangerous stage, she's prescribed with one drug to minimal dosage.
  K6 – If a treatment reaches maximal dosage of one drug, a second drug to minimal dosage is prescribed.
  K8 – If a patient arrives in a highly dangerous stage, she's directly prescribed with one drug.
  K8' - If a patient arrives in a highly dangerous stage, she's recommended lifestyle changes.

FIRST ORDER LOGIC (FOL)

- (4) Represent the following assertions in FOL: *men are not women; surgeons are doctors; adults can only be men and women; if a person marries another person, this one is also married to the first one; parents have children; marriage is only allowed between adults; a person cannot be married to two or more different persons*.

"Men are not women" $\forall x \quad man(x) \supset \neg woman(x)$

"Surgeons are doctors" $\forall x \quad surgeon(x) \supset doctor(x)$

"Adults can only be men or women" $\forall x \quad adult(x) \supset man(x) \vee woman(x)$

"If a person marries another person, this one is also married to the first one" $\forall x \forall y \quad married(x,y) \supset married(y,x)$

"Parents have children" $\forall x \exists y \quad parent(x) \supset haschild(x,y)$

"Marriage is only allowed between adults" $\forall x \forall y \quad married(x,y) \supset adult(x) \wedge adult(y)$

"A person cannot be married to two or more different persons" $\forall x \forall y \forall z \quad married(x,y) \wedge married(y,z) \supset (z = x)$

- (7) Given the predicates parent (x,y)="x is parent of y" and male(x)="x is a male" provide FOL expressions to represent the concepts: son; daughter; brother; uncle; grandfather; John has not children; John's sister has some children.

```
son(x,y) ≡ parent (y,x).
daughter(x,y) ≡ (parent(y,x) ∧ ¬male(x)).
brother(x,y) ≡ (∃z. (parent(z,x) ∧ parent(z,y) ∧ male(x))).
uncle(x,y) ≡ (∃z. ∃w. (parent(z,y) ∧ parent(w,z) ∧ parent(w,x) ∧ male(x))).
grandparent(x,y) ≡ (∃z. (parent(z,y) ∧ parent(x,z))).

"John has no children" ≡ (∀x.∀y. (parent(x,y) ⊃ x ≠ John))
                       ≡ (∀x. ¬parent(John,x))
"John has no siblings" ≡ (∃z.∀s. (parent(z,John) ∧ parent(z,s) ⊃ s = John))
"John's sister has some children" ≡ (∃z. ∃s. ∃c. (parent(z,John) ∧
parent(z,s) ∧ ¬male(s) ⊃ parent(s,c))).
```

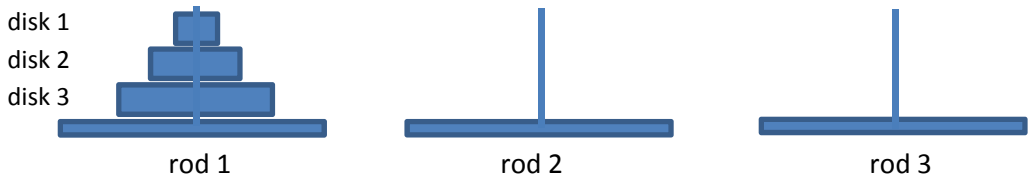
- (10d) In the world of blocks we have three shapes: Triangle, Square and Rectangle; three colors: White, Black, and Grey; and the possibility of having one block on top of another: On(x,y). Provide a description of the following worlds in FOL:

```
Square (x) ∧ Rectangle (y) ∧ Triangle (z) ∧
Grey (x) ∧ Black (y) ∧ White (z) ∧
On(x, y) ∧ On (y, z).
```

- (11) In the world of blocks defined in exercise 10 provide FOL expressions for the following facts:
  1. Triangles cannot have other blocks on top: ∀x.∀y Triangle(x) ⊃ ¬On(x,y)

2. All composition must have a Triangle at the very best top: $\forall x.\ \text{Triangle}(x) \vee (\exists y.\ \text{On}(y,x))$
3. All Intermediate blocks must be Grey: $\forall x.\ \forall y.\ \forall z.\ \text{On}(x,y) \wedge \text{On}(y,z) \supset \text{Grey}(y)$
4. Some intermediate block is Black: $\forall x.\ \forall y.\ \forall z.\ \text{On}(x,z) \wedge \text{On}(z,y) \supset \text{Black}(z) \vee (\exists w.\text{On}(w,x))$
5. Only White blocks are permitted: $\forall x.\text{White}(x) \wedge \neg\text{Grey}(x) \wedge \neg\text{Black}(x)$
6. There are not Black blocks immediately on top of White blocks: $\forall x.\ \forall y.\ \text{White}(x) \wedge \text{On}(y,x) \supset \neg\text{Black}(y)$

- (15) For the problem of Hanoi Towers provide FOL expressions describing: the initial configuration; disk d can be moved to rod r; disk d is moved to rod d.



Initial configuration ≡

$$\exists r1.\ \exists r2.\ \exists r3.\ \exists d1.\ \exists d2.\ \exists d3.\ \text{rod}(r1) \wedge \text{rod}(r2) \wedge \text{rod}(r3) \wedge \text{disk}(d1) \wedge \text{disk}(d2) \wedge \text{disk}(d3) \wedge \text{on}(d1,r1) \wedge \text{on}(d2,r1) \wedge \text{on}(d3,r1) \wedge \text{bigger}(d3,d2) \wedge \text{bigger}(d3,d1) \wedge \text{bigger}(d2,d1)$$

"disk d can be moved to rod r": $\text{movable}(d,r) \equiv$

$$\exists x.\forall y.\ \text{on}(d,x) \wedge (\text{on}(y,x) \supset \text{bigger}(y,d)) \wedge (\text{on}(y,r) \supset \text{bigger}(y,d))$$

"disk d moves to rod r" (the notion of time must be introduced to avoid inconsistencies):

$$\text{move}(d,r,t) \equiv$$

$$\text{Movable}(d,r,t) \supset (\forall x.\ (\text{rod}(x) \wedge x \neq r) \supset (\neg\text{on}(d,x,t+1) \wedge \text{on}(d,r,t+1)))$$

RULES

- (20) Propose a rule for the knowledge represented in the following sentences:

Grandmothers tell nice stories

$\text{tell\_nice\_stories}(X) \Leftarrow \text{grandmother}(X)$

I hate all people that own cats

$\text{I\_hate}(X) \Leftarrow \text{owns}(X,Y) \wedge \text{cat}(Y) \wedge \text{person}(X)$

Summer days are hotter than winter days

$\text{hotter}(X,Y) \Leftarrow \text{summer\_day}(X) \wedge \text{winter\_day}(Y)$

People that have both life and health insurances are full covered

$\text{full\_covered}(X) \Leftarrow \text{has\_life\_insurance}(X) \wedge \text{has\_health\_insurance}(X)$

All the subjects in the master have at least two different exams (assuming that the consequent of the rule can have several facts)

$\textit{has\_exam}(S,X) \wedge \textit{has\_exam}(S,Y) \wedge X \neq Y \Leftarrow \textit{is\_subject\_in\_master}(S)$

The friends of the friends are friends (assuming friendship commutative)

$friend(X,Z) \Leftarrow friend(X,Y) \wedge friend(Y,Z)$

The friends of the enemies are enemies

$enemy(X,Z) \Leftarrow enemy(Y,X) \wedge friend(Z,Y)$

There are six eye colors: amber, blue, brown, grey, green, and hazel.

$eyecolor(amber) \vee eyecolor(blue) \vee eyecolor(brown) \vee eyecolor(grey) \vee \ldots$

People with green eye color are more attractive than the rest

$more\_attractive(X,Y) \Leftarrow person(X) \wedge eyecolor(X,grey) \wedge person(Y) \wedge \neg eyecolor(X,grey)$

…

I hate John, if he registers to a subject I don't.

$hate(I,John) \wedge (\neg register(I,S) \Leftarrow register(John,S) \wedge subject(S))$

Some people in love get married. All just married people are in love. Some people stop loving their couple sometime after marriage. All married people that are not in love, get divorced.

$married(X,Y) \Leftarrow person(X) \wedge person(Y) \wedge conditionOfSome(X,Y)$

$inlove(X,Y) \Leftarrow just\_married(X,Y) \wedge person(X) \wedge person(Y)$

$\neg inlove(X,Y,T) \Leftarrow married(X,Y,T') \wedge person(X) \wedge person(Y) \wedge T=sometimeAfter(T')$

$divorced(X,Y) \Leftarrow married(X,Y) \wedge person(X) \wedge person(Y) \wedge \neg inlove(X,Y)$

RULES IN CLIPS

- (30) The Towers of Hanoi problem (see exercise 15) has an elegant recursive solution, but it also has a less well known iterative solution as follows. First, we arrange the pegs in a circle, so that clockwise we have rods A, B, C, and then A again. Disks are given the respective names 1, 2, and 3. Following this, assuming we never move the same disk twice, there will always only be one disk that can be legally moved, and we transfer it to the first rod it can occupy, moving it in a clockwise direction, if it is even, and counter-clockwise, if it is odd. Write a collection of production rules that implement this procedure. Initially, the working memory will have elements (on rod: A disk:i) for each disk and an element (solve). When your rules stop firing, you should have (on rod:C disk:i) for each disk and (done) in the working memory.

```
;; WORKING MEMORY
(on rod:1 disk:1)
(on rod:1 disk:2)
(on rod:1 disk:3)
(solve)
(lastmoved disk:0)

;; RULES
R1:  IF (on rod:r disk:d&{1v3})
        (lastmoved disk:{<> d})
        (solve)
        -(on rod:r disk:{<d})
        -(on rod:[(r mod 3)+ 2] disk:{<d}
     THEN
        MODIFY 1 (rod [(r mod 3) +2])
        MODIFY 2 (disk d)
```

```
R2:  IF (on rod:r disk:d&{2})
        (lastmoved disk:{<> d})
        (solve)
        -(on rod:r disk:{<d})
        -(on rod:[(r mod 3)+ 2] disk:{<d}
     THEN
        MODIFY 1 (rod [(r mod 3) +1])
        MODIFY 2 (disk d)

R3:  IF (solve)
        (lastmoved disk:d1)
        -(on rod:{<>3} disk:d2)
     THEN
        REMOVE 1
        REMOVE 2
        ADD (done)
```

- (31) A circular railway is composed of four train stations S1, S2, S3, and S4. A train T circulates from S1 to S2, from S2 to S3, from S3 to S4, and from S4 to S1, starting the cycle again. The train has a capacity for 10 passengers. Passengers arrive to and leave from stations. Passengers have a destination station where they want to arrive to. All the passengers arrive to a station, get into the next train with a vacancy and waits till the train arrives to the passenger's destination station. Represent this system with CLIPS.

```
(defglobal ?*num-stations* = 4) ; there are 4 stations
(defglobal ?*train-capacity* = 10) ; capacity of train is 10 passengers

(assert (train 0 stopped 1)) ; the train is empty and stopped at the first station

; ---------------------Rules for getting on/off the train:

(defrule get-on-train
       "a passenger steps on the train"
       (declare (salience 0)) ; medium priority -default 0
       ?train <- (train ?occupancy&:(< ?occupancy ?*train-capacity*) stopped ?station)
       ?person <- (person ?id ?station ?destination)
       =>
       (retract ?train ?person)
       (assert (passenger ?id ?destination) (train (+ ?occupancy 1) stopped ?station))
       (printout t "Passenger " ?id " gets on train." crlf))

(defrule get-off-train
       "a passenger steps off the train when arriving to destination"
```

```
        (declare (salience 1)) ; left leave before enter the train
        ?train <- (train ?occupancy stopped ?station)
        ?passenger <- (passenger ?id ?station) ; dest. is current station
        =>
        (retract ?train ?passenger)
        (assert (train (- ?occupancy 1) stopped ?station))
        (printout t "Passenger " ?id " gets off the train in station " ?station "."
crlf))

; ----------------------------Train moves between stations

(defrule train-leaves
        "the train goes to next station"
        (declare (salience -1)) ; lowest priority: train leaves only if nobody wants to
get in or if it's full
        ?train <- (train ?passengers stopped ?station)
        =>
        (retract ?train)
        (assert (train ?passengers moving ?station))
        (printout t "Train leaving station " ?station "." crlf))

(defrule train-arrives
        "the train arrives to next station"
        ?train <- (train ?passengers moving ?previous-station)
        =>
        (retract ?train)
        (bind ?current-station (+ (mod ?previous-station ?*num-stations*) 1))
        (assert (train ?passengers stopped ?current-station))
        (printout t "Train arriving at station " ?current-station "." crlf)
        (halt)) ; (or (read)) stop used to avoid train round uncontrolled.

; ---------------------------------Creating persons

(defglobal ?*person-id* = 1) ; counter to generate person IDs

(deffunction new-persons (?num)
        "creates num new persons willing to use the train"
        (loop-for-count ?num
        (assert (person ?*person-id* (+ (mod (random) ?*num-stations*) 1) (+ (mod
(random) ?*num-stations*) 1)))
        (bind ?*person-id* (+ 1 ?*person-id*))) ; next person-id
        (printout t ?num " new users created." ctrlf))
```

PRODUCTION SYSTEMS

- (28) In the world of blocks there's a group of blocks on a table, and we want to make a heap with all these blocks and with bigger blocks below smaller blocks. We count with a robot arm. Provide production systems for the following implementations:
  (a) WME = (block id:int size:int position:{table, robot-hand, heap })

```
; Robot takes the biggest block
IF    (block id: name size: s position: table)
      -(block position: table size:{> s})
      -(block position: robot-hand)
THEN
      MODIFY 1 (position robot-hand)

; Robot releases the block
IF    (block position: robot-hand)
THEN
      MODIFY 1 (position heap)
```

  (b) We are only allowed to have the type of WME (block id:int size:int position: {table, robot-hand, #position in the heap}.

```
; We'll need a WME as a counter of the position in the heap
(counter n: int)

; Robot takes the biggest block
IF    (block id: name size: s position: table)
      -(block position: table size:{> s})
      -(block position: robot-hand)

THEN
      MODIFY 1 (position robot-hand)

; Robot releases the block
IF    (block position: robot-hand)
      (counter n: i)
THEN
      MODIFY 1 (position i)
      MODIFY 2 (n [+ i 1])
```

  (c) We are allowed to have the following WME types: (block id:int size:int), (robot-hand block: int), (on block1: int, block2: int). Note: you should convert the blocks that the robot takes into used-blocks to avoid the robot to take them more than once.

```
; We'll need a WME as indicator of whether a block is eligible or not.

; Robot takes the biggest block
IF    (block id: name size: s)
      -(on block1: name)
      -(robot-hand)

THEN
      REMOVE 1
      ADD (block-used id: name size: s)
      ADD (robot-hand block: name)

; Robot releases the first block
IF    (robot-hand block: name)
      -(on)
THEN
      REMOVE 1
      ADD (on block1: name block2: heap)
```

```
; Robot releases subsequent blocks
IF    (robot-hand block: name)
      (on block1: name1)
      (block-used id: name1 size: s1)
      -(block-used size: {> s1})
THEN
      REMOVE 1
      ADD (on block1: name block2: name1)

; In order to recover all the WME (block) from the block-used facts
; we introduce the token (recover-blocks) which is removed when all
; the block-used WMEs have been converted to block facts.

IF    -(block)
THEN ADD (recover-blocks)

IF    (recover-blocks)
      (block-used id: name size: s)
THEN REMOVE 2
     ADD (block-used id: name size: s)

IF    (recover-blocks)
      -(block-used)
THEN REMOVE 1
```

Implement all the solutions in CLIPS.

(a)

```
(defrule robot-takes-block
  ?block <- (block ?name ?s table) ; there's a block on table such that
  (forall (block ?name1 ?s1 table) ; all the blocks on the table
     (block ?name1 ?s1&:(<= ?s1 ?s) table)); have a smaller size
  =>
  (retract ?block)
  (assert (block ?name ?s robot-hand))
  (printout t "Robot takes " ?name crlf))

(defrule robot-releases-block
  ?block <- (block ?name ?s robot-hand)
  =>
  (retract ?block)
  (assert (block ?name ?s heap))
  (printout t "Robot releases "?name crlf))

(deffunction make-block (?num)
  (if (> ?num 0) then
    (assert (block ?num ?num table)) (make-block (- ?num 1))))
```

(b)

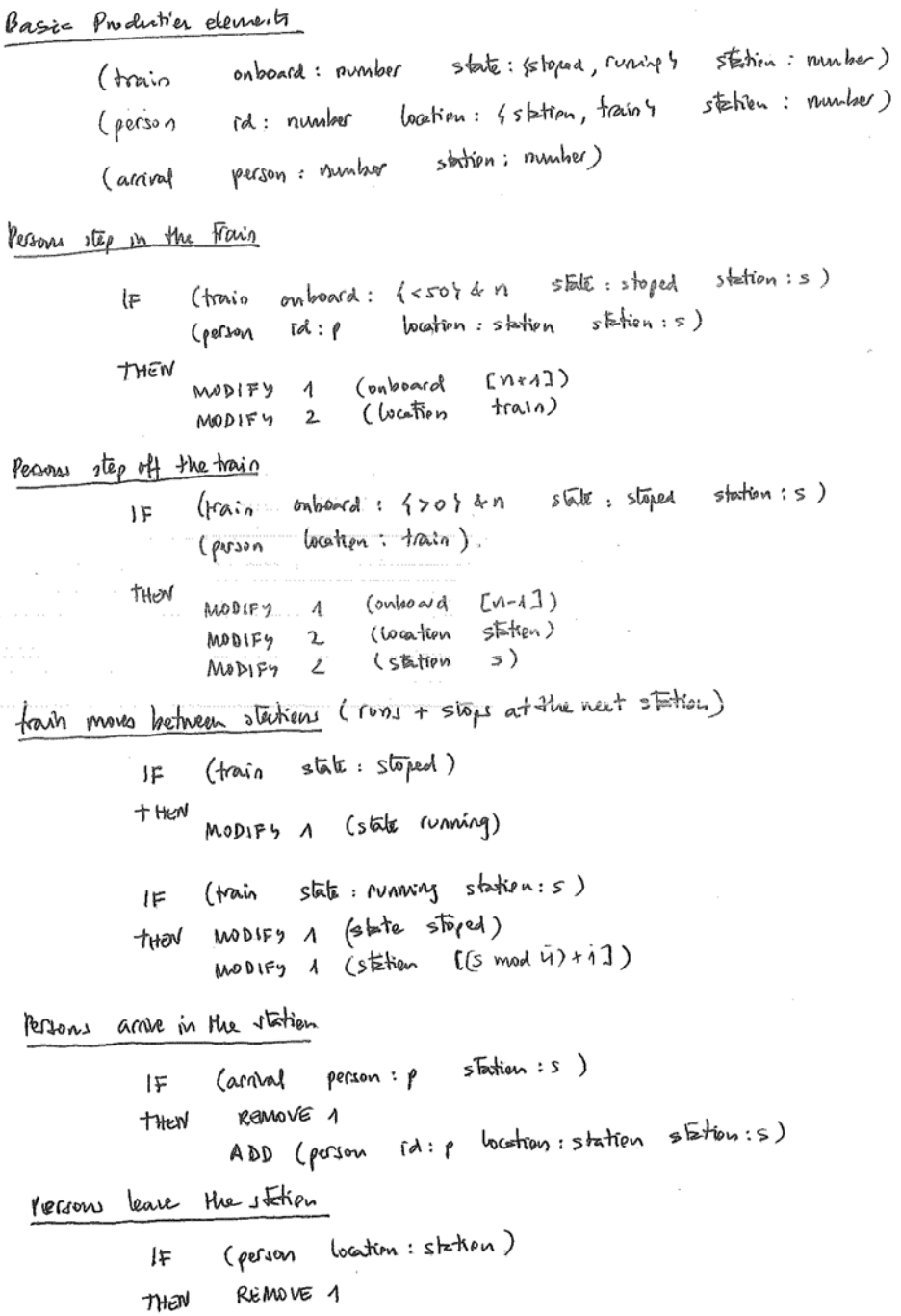
```
(defrule robot-takes-block
  ?block <- (block ?name ?s table) ; there's a block on table such that
  (forall (block ?name1 ?s1 table) ; all the blocks on the table
     (block ?name1 ?s1&:(<= ?s1 ?s) table)); have a smaller size
  (not (block ?name2 ?s2 robot-hand)); and robot-hand is free
  =>
  (retract ?block)
  (assert (block ?name ?s robot-hand))
  (printout t "Robot takes " ?name crlf))
```

```
(defrule robot-releases-block
  ?block <- (block ?name ?s robot-hand)
  ?count <- (counter ?i)
  =>
  (retract ?block ?count)
  (assert (block ?name ?s ?i) (counter (+ ?i 1)))
  (printout t “Robot releases “?name crlf))

(deffunction make-block (?num)
  (if (> ?num 0) then
    (assert (block ?num ?num table)) (make-block (- ?num 1))))
```

(c)

```
(defrule robot-takes-a-not-used-block
  ?block <- (block ?name ?s) ; there's a block on table such that
  (not (block ? ?s1&:(> ?s1 ?s))) ; it is the biggest one
  (not (on ?name ?)) ; not still used
  (not (robot-hand ?)) ; and robot hand is free
  =>
  (retract ?block)
  (assert (block-used ?name ?s) (robot-hand ?name))
  (printout t "Robot takes " ?name crlf))

(defrule robot-releases-first-block
  ?robot <- (robot-hand ?name) ; robot-hand has the first block
  (not (on ? ?))
  =>
  (retract ?robot)
  (assert (on ?name heap)) ; the first block is stacked on the heap
  (printout t "Robot releases "?name crlf))

(defrule robot-releases-subsequent-block
  ?robot <- (robot-hand ?name)
  (on ?name2 ?)
  ?block <- (block-used ?name2 ?s2)
  (not (block-used ? ?s&:(> ?s ?s2)))
  =>
  (retract ?robot)
  (assert (on ?name ?name2))
  (printout t "Robot releases " ?name crlf))

; in order to recover all the block facts from the block-used facts
; we introduce a recovery rule with lower priority

(defrule recover-blocks
  (declare (salience -10))
  ?used <- (block-used ?name ?size)
  =>
  (retract ?used)
  (assert (block ?name ?size))
)

(deffunction make-block (?num)
  (if (> ?num 0) then
    (assert (block ?num ?num)) (make-block (- ?num 1))))
```

- (31) A circular railway is composed of four train stations S1, S2, S3, and S4. A train T circulates from S1 to S2, from S2 to S3, from S3 to S4, and from S4 to S1, starting the cycle again. The train has a capacity for 30 passengers sited and 20 passengers standing. Passengers arrive to and leave from stations. Some passengers want to be seated, some others don't care. Passengers have a destination station where they want to arrive to. All the passengers arrive to a station, get into the next train with a vacancy of the sort wished (seated or don't care) and waits till the train arrives to the passenger's destination station. Represent this system with a production system.



- (32) In some academies, students are able to enroll in subjects that they want to attend and whose pre-requirement they have all been passed. Pre-requirement of a subject are a set of other subjects that need to be passed before registering to the first one.

Basic Production Elements

```
(want-to-attend subject: s)
(academic-record subject: s passed: {yes, no, enrolled})
(pre-requirement subject: r of: s)
```

Subjects without pre-requirements can be enrolled:

```
IF  (want-to-attend subject: s)
    -(academic-record subject: s passed: yes)
    -(pre-requirement subject: r of: s)
THEN
    REMOVE 1
    ADD (academic-record subject: s passed: enrolled)
```

Subjects with passed pre-requirements can also be enrolled:

```
IF  (want-to-attend subject: s)
    (pre-requirement subject: r of: s)
    (academic-record subject: r passed: yes)
THEN
    REMOVE 2
    ADD (considered subject: r of: s)

IF  -(want-to-attend subject: s)
    (considered subject: r of: s)
THEN
    REMOVE 2
    ADD (pre-requirement subject: r of: s)
```

Subjects without pre- and co-requirements can be enrolled:

```
IF  (want-to-attend subject: s)
    -(academic-record subject: s passed: yes)
    -(pre-requirement subject: r of: s)
    -(co-requirement subject: c of: s)
THEN
    REMOVE 1
    ADD (academic-record subject: s passed: enrolled)
```

Subjects with passed or enrolled co-requirements can also be enrolled:

```
IF  (want-to-attend subject: s)
    (co-requirement subject: c of: s)
    (academic-record subject: c passed: {yes | enrolled})
THEN
    REMOVE 2
    ADD (considered2 subject: c of: s)

IF  -(want-to-attend subject: s)
    (considered2 subject: c of: s)
THEN
    REMOVE 2
    ADD (co-requirement subject: c of: s)
```

FRAMES AND SCRIPTS

- (36. Classroom scheduler) Build a program that helps schedule rooms for classes of various sizes at a university, using the sort of frame technology (frames, slots, and facets). Slots of frames might be used to record when and where a class is to be held, the capacity of a room, etc., and IF-ADDED and other facets might be used to encode constraints as well as to fill in implied values when the KB is updated.

  In this exercise, we want to consider updating the KB in several ways: (1) asserting that a class of a given size is to be held in a given room at a given time; the system would either go ahead and add this to its schedule, or alert the user that it was not possible to do so; (2) asserting that a class of a given size is to be held at a given time, with the system providing a suitable room (if one is available) when queried; (3) asserting that a class of a given size is desired, with the system providing a time and place when queried.

(1) "Class C of size S is held in room R, at time T"

```
(create C (
    (:INSTANCE-OF Class-1)
    (:SIZE S)
    (:ROOM R)
    (:TIME T)))
```

```
(create Class-1 (
       (:SIZE unsigned-int)
       (:ROOM classroom)
       (:TIME (:IF-ADDED f1(v))))
```

f1 is the function that checks whether the assigned class room is of the correct size and free at the given time, and books it.

```
f1(v) = (lambda (v)
           (let ((size SELF:SIZE)(room SELF:ROOM)
                 (occupation room:OCCUPATION)(result v))
               (if (>= room:SIZE size)
                  (while (occupation)
                     (if (= occupation:TIME v)
                        (set result "unknown"))
                     (set occupation occupation:NEXT))
                  (set result "unknown"))
           result))
```

(2) "Class C of size S is held at time T"

```
(create C (
    (:INSTANCE-OF Class-2)
    (:SIZE S)
    (:TIME T)))
```

```
(create Class-2 (
       (:SIZE unsigned-int)
       (:ROOM (:IF-NEEDED f2(v))
       (:TIME time))
```

f2 is the function that searches for a free class room to hold the class at the given time, and books it. If the class room is already assigned, this class room is returned.

```
f2(v) = (lambda ()
           (let ((size SELF:SIZE)(time SELF:TIME))
               (for (room SCHOOL:CLASSROOMS)
                    (if (>= room:SIZE size)
                       (let ((free true) (occupation room:OCCUPATION))
                          (while (occupation)
                             (if (= occupation:TIME time)
```

```
                                    (set free false))
                                (set occupation occupation:NEXT))
                            (if (free) (let class room)))))
                    class))
```

(3) "Class C of size S is to be held"

(create C ((:INSTANCE-OF Class-3) (:SIZE S)))	(create Class-3 ((:SIZE (:IF-ADDED f3(v)))

f3 is the function that searches for a free class room at some time t, with capacity to hold this class, and books it. If the class is already assigned, this class room is not booked.

```
(create SCHOOL (
  (:CLASSROOMS CLASSROOM)))

(create CLASSROOM (
  (:SIZE unsigned-in)
  (:OCCUPATION OCCUPATION)))

(create OCCUPATION (
  (:TIME time)
  (:NEXT OCCUPATION)))
```

- (39) Provide a Script to represent the process of borrowing a book in a Library that could cover the following steps: enter the library, look for a book in the shelves, in the computer, or directly with the librarian, make the reservation, go home with the book (if it was found), and return the book after some time.

```
(BOOK-BORROWING
  <:IS-A library-act>
  <:PROPS {:library :book :shelf :computer :reservation}
  <:ROLES {:librarian :user}
  <:OPENING-CONDITIONS {(SELF:user:wants-a-book SELF:book)}>
  <:RESULTS {(add SELF:user:readings SELF:book)
             (add SELF:library:reservations SELF:reservation)}>
  <:SCENE {
     Entering {
        (set SELF:user:place SELF:library)
        (set SELF:scene Look-for-book)}
     Look-for-book {
        (or (set SELF:user:searches SELF:shelf SELF:book)
          (set SELF:user:asks SELF:comuter SELF:book)
          (set SELF:user:asks SELF_librarian SELF:book))
        (if (SELF:user:found SELF:book)
          (set SELF:scene Make-reservaton)
          (set SELF:scene Leave-library))}
     Make-reservation {
        (set SELF:reservation (reservation SELF:user (return-date)))
        (set SELF:book:state SELF:reservation)
        (set SELF:scene Leave-library)}
     Return-book {
        (set SELF:user:place SELF:library)
        (set SELF:book:state (free))}}>
)
```

FRAMES IN COOL

- (35 in COOL) Represent in COOL the following knowledge base:
  (a) Vehicles are means of transportation with wheels with many possible colors and there are companies building different models of vehicles.

```
(defclass COMPANY "vehicle builder" (is-a USER)
  (slot company-name (type STRING)) ; ex. "Renault", …
  (multislot list-of-models)        ; list of car model this company makes
)
(defclass VEHICLE "means of transportation" (is-a USER)
  (slot wheels (type INTEGER))      ; a vehicle has a number of wheels
  (multislot color)                 ; a vehicle can be of several colors
)
```

  (b) There are vehicles that are classic-cars. These cars are made by a company.

```
(defclass CLASSIC-CAR "vehicles that are classic cars" (is-a VEHICLE)
  (slot company)
  (slot model)
)
```

  (c) All classic-cars must have a company (i.e., company slot of CLASSIC-CAR is not optional).

```
; CLASS CONSISTENCY: company slot must contain an instance of COMPANY ...

(defmessage-handler CLASSIC-CAR put-company around (?company)
; check company exists when inserted in a CLASSIC CAR
  (if (not (any-instancep ((?c COMPANY)) (eq ?c ?company)))
    then ; not a company assigned to classic-car
      (printout t "Slot company of CLASSIC-CAR " ?company "  must be an
instance of COMPANY." crlf)
      (override-next-handler [nil]) ; assign the null object to company slot
    else
      (call-next-handler)
))

; NON OPTIONALITY: ... and cannot be empty

(defmessage-handler CLASSIC-CAR init around ()
; Impossible to create a CLASSIC-CAR with empty slot company
  (bind ?company (dynamic-get company))
  (if (any-instancep ((?c COMPANY)) (eq ?c ?company))
    then
      (call-next-handler)
    else
      (printout t "Cannot create CLASSIC-CAR " ?self " with empty company
slot." crlf)
))
```

  (d) Classic-cars have a model. If the company making the car doesn't have this model in its list of models, it is inserted.

```
; IF-ADDED: Insert CLASSIC-CAR model in the corresponding COMPANY list-of-
models if it does not exist

(deffunction insert-model-in-company (?model ?company)
  (bind ?list (send ?company get-list-of-models))
  (if (and (not (member$ ?model ?list)) (not (eq ?model nil)))
    then
```

```
      (send ?company put-list-of-models (insert$ ?list 1 ?model))
      (printout t "Model " ?model " inserted in list of models of company "
?company crlf)
))

(defmessage-handler CLASSIC-CAR init around ()
; update COMPANY list-of-models when a new CLASSIC-CAR is created
  (bind ?company (dynamic-get company))
  (if (any-instancep ((?c COMPANY)) (eq ?c ?company))
    then
      (call-next-handler)
      (insert-model-in-company (dynamic-get model) ?company)
    else
      (printout t "Cannot create CLASSIC-CAR " ?self " with empty company
slot." crlf)
))

(defmessage-handler CLASSIC-CAR put-model around (?model)
; update COMPANY list-of-models when a CLASSIC-CAR is assigned a model
  (call-next-handler)
  (insert-model-in-company ?model (dynamic-get company))
)
```

(e) Classic-cars have a factory price (the cost of producing the car) and a retail price (the cost to final client). Retail price is always 30% more than the factory-price, and the factory-price can change the production costs.

```
; IF-NEEDED: retail price of classic car is read-only and dynamically
calculated as +30% of factory price

(defclass CLASSIC-CAR (is-a VEHICLE)
  (slot company (type INSTANCE))
  (slot model)
  (slot factory-price)
  (slot retail-price (access read-only)); it is calculated +30% factory-price
)

(defmessage-handler CLASSIC-CAR get-retail-price around ()
; when retail-price is needed, it is calculated
  (bind ?fprice (dynamic-get factory-price))
  (call-next-handler)
  (if (not (eq ?fprice nil))
    then
      (* ?fprice 1.30)
    else
      ?fprice
))
```

(f) Classic-cars have a horsepower value that must be always in the range [50, 200].

```
; RANGE VALUES: the horsepower of classic cars is between 50 and 200

(defclass CLASSIC-CAR (is-a VEHICLE)
  (slot company (type INSTANCE))
  (slot model)
  (slot factory-price)
  (slot retail-price (access read-only)); it is calculated +30% factory-price
  (slot horse-power)
)

(defmessage-handler CLASSIC-CAR put-horse-power around (?hp)
```

```
; check that inserted horsepower value is in the range
  (if (or (< ?hp 50) (> ?hp 200))
    then
      (printout t "CLASSIC-CAR horsepower " ?hp " shoud be between 50 and
200." crlf)
      (override-next-handler nil)
    else
      (call-next-handler)
))
```

(g) Classic-cars have one single color that can be red, white, black, yellow, dark, or other. The information is stored codified, and recovered decodified.

```
; LIMITED VALUES: the possible colors of classic cars are Red, White, Black,
Yellow, Dark, Other

; CLASSIC-CAR inherits the multislot color of VEHICLE

(deffunction code-color (?color)
  (switch ?color
    (case "red" then R)
    (case "white" then W)
    (case "black" then B)
    (case "yellow" then Y)
    (case "dark" then D)
    (default O)
))

(deffunction decode-color (?code-list) ; color is a multislot
  (if (member$ R ?code-list) then "red"
   else (if (member$ W ?code-list) then "white"
   else (if (member$ B ?code-list) then "black"
   else (if (member$ Y ?code-list) then "yellow"
   else (if (member$ D ?code-list) then "dark"
   else (if (member$ O ?code-list) then "other"
   else "unknown")))))
))

(defmessage-handler CLASSIC-CAR put-color (?color)
  (override-next-handler (code-color ?color))
)

(defmessage-handler CLASSIC-CAR get-color ()
  (decode-color (override-next-handler))
)
```

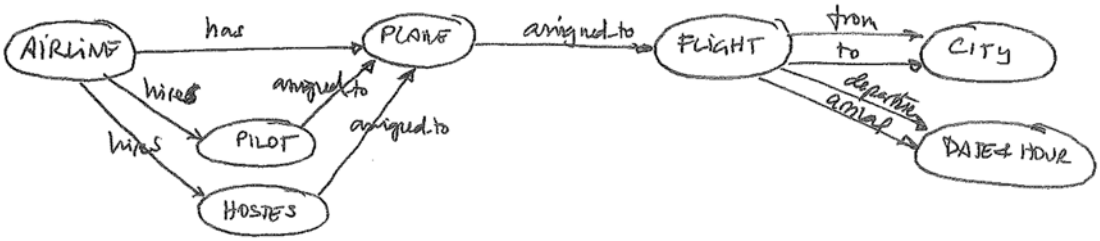
(h) Persons can buy and sell classic-cars with the corresponding exchange of money.

```
; MULTICLASS RELATIONSHIPS: A CLASSIC car is owned by a PERSON

(defclass PERSON (is-a USER)
  (slot person-name (type STRING))
  (slot money (type INTEGER) (default 0))
)

(defclass CLASSIC-CAR (is-a VEHICLE)
  (slot company (type INSTANCE))
  (slot model)
  (slot factory-price)
  (slot retail-price (access read-only) (type FLOAT));
  (slot horse-power)
```

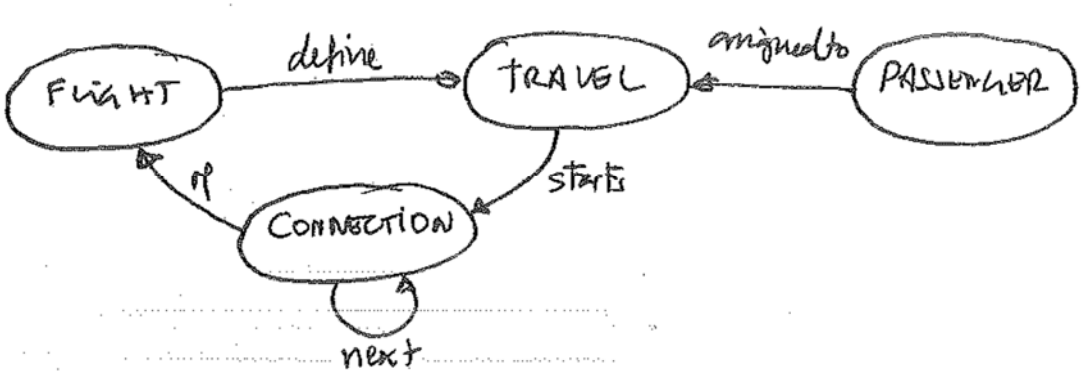
```
  (slot owner (type INSTANCE))
)

; MULTICLASS CONSISTENCY: When a PERSON sells/purchases a CLASSIC car he
earns/spends the retail-price

(defmessage-handler CLASSIC-CAR put-owner around (?customer)
  (bind ?seller (dynamic-get owner))
  (bind ?price (send ?self get-retail-price))
  (if (any-instancep ((?p PERSON)) (eq ?p ?customer))
    then
      (bind ?customer-money (send ?customer get-money))
      (if (> ?price ?customer-money)
        then ; the customer does not have enought money to buy the car
          (printout t "PERSON " ?customer " doesn't have enough money to buy
the car." crlf)
        else
          (call-next-handler)
          (send ?customer put-money (- ?customer-money ?price))
          (if (not (eq ?seller [nil]))
            then
              (send ?seller put-money (+ (send ?seller get-money) ?price))))
    else
      (printout t "A CLASSIC-CAR cannot be purchased by " ?self " being not a
PERSON." crlf)
)
(dynamic-get owner)
)
```

SEMANTIC NETWORKS

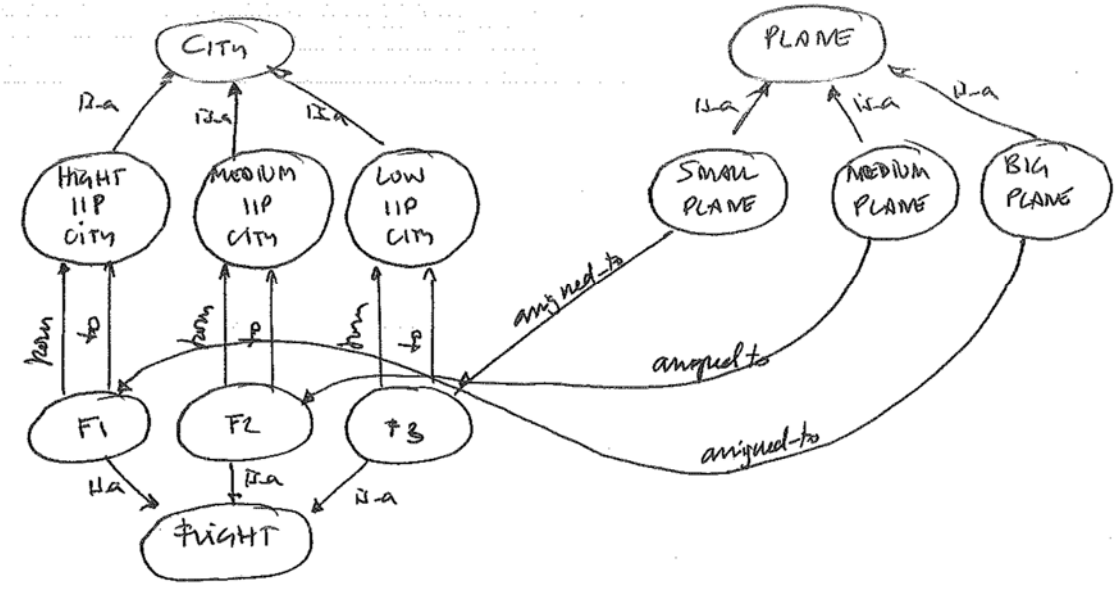
- (48) Airlines have planes and hire pilots and hostesses that are assigned to planes; planes are assigned to flights; flights are between two cities (from and to), they have two dates and hours assigned (departure and arrival);



- flights define travels, though travels can have several flights connected; passengers are assigned to travels;

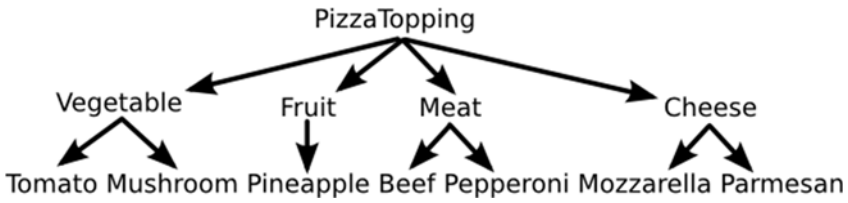


- cities have an intended influence population that can be high, medium or low; flights between two high populated cities require big planes; flights between two low populated cities require small planes; the rest of flights require medium planes.



ONTOLOGIES

- (55) Given the following class hierarchy about pizza toppings



a) Can something be meat but not a pizza topping?

No, since meat is a subclass of pizza topping.

b) Can something be both meat and a vegetable?

Yes, nothing prevents another class from being a subclass of both meat and vegetable.

c) Is beef a pizza topping?

Yes, since Meat is a subclass of PizzaTopping and Beef is a subclass of Meat.

d) In OWL, can we prevent something from being both a fruit and a vegetable? If so, how? If not, why not?

Yes, we can prevent something from being both a fruit and a vegetable by adding disjointness axioms between Fruit and Vegetable.

e) How are the class names in this ontology misleading? How would you rename the classes to make them less misleading?

All of the subclasses of "PizzaTopping" have misleading names. For example, according to the ontology all fruits are pizza toppings. However, this is somewhat different from the standard definition of pizza toppings. A better name for the subclasses would be to append "PizzaTopping" to each class name. For example, "meat" becomes "meatPizzaTopping", "fruit" becomes "fruitPizzaTopping" and "tomato" becomes "tomatoPizzaTopping".

- (56) Following with the pizza topping world, Consider the following segment of OWL code:

```
class(VegetarianPizza
Pizza
complementOf(restriction(hasTopping someValuesFrom Meat)))

class(TomatoPizza
Pizza
restriction(hasTopping someValuesFrom Tomato))
```

Using information from both the pizza topping hierarchy and the OWL code segment, answer the following questions:

a) Can a TomatoPizza contain meat?

Yes, since the only restriction for a tomato pizza is that it contains tomato.

b) Can a pizza be both a VegetarianPizza and a TomatoPizza?

Yes, if the pizza has tomato but doesn't have meat.

c) Are all TomatoPizzas VegetarianPizzas?

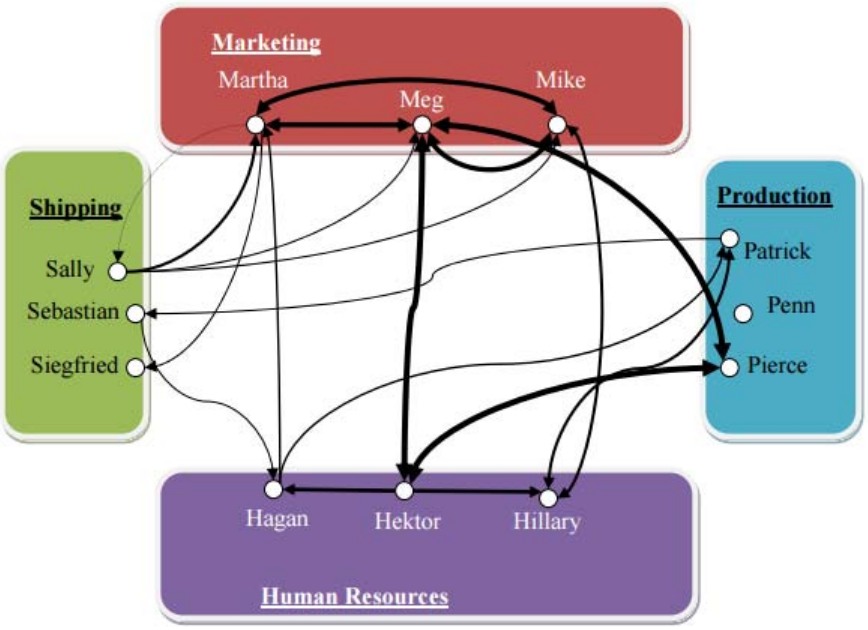
No, since some TomatoPizzas can have meat.

- (57) Define a new pizza called "BeefPizza" using a similar syntax style as used on the OWL code segment above. A BeefPizza has some Tomato toppings, some Beef toppings, but no other toppings.

```
class(BeefPizza
Pizza
restriction(hasTopping someValuesFrom Tomato)
restriction(hasTopping someValuesFrom Beef)
restriction(hasTopping allValuesFrom (Tomato or Beef)))
```

KNOWLEDGE AUDITING

(66) Given the following knowledge map on the interactions of the members in a company,



(a) ¿what can you deduce for the departments with regard to the way they work inside and with other departments?

(b) ¿what can you deduce from the people in the company?

- People in the shipping department do not work together. One of the members (Siegfried) is not productive. Other member (Sally) is oriented to the Marketing department.
- The work in the marketing department is highly cohesive (everybody works with everybody) and collaborative. Meg is the only person of this department working with the production department.
- In the production department, there's one person (Penn) not working at all.
- Hektor in the human resources department seems to be the boss, who delegates some interactions to Hagan and Hillary. It has a hierarchical structure. As a consequence, Meg and Pierce seem to be

the bosses of the Marketing and Production departments, respectively (since they mutually interact with Hektor, mainly).

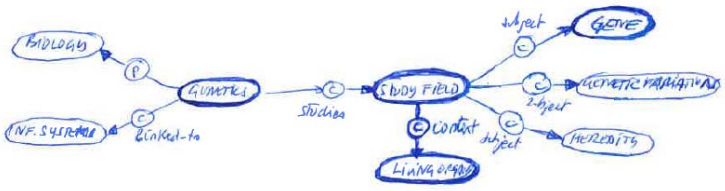
(c) Provide an example of business process that is not possible in this company (for example: “Martha detects a market need and discusses it with Mike, who decides to contact Hillary to analyze the possibilities to ask Patrick’s help to start producing a new product, and Sally for future shipping of this product”. Clearly this situation is not possible since there is not a communication from Mike to Sally. Notice that the rest of communications in the scenario depicted are possible).

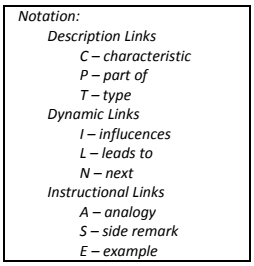
“Pierce is asked to start producing a new product, but they lack of know-how about this product and he asks Hector about the possibility of hiring an expert. In order to make a decision, Hector asks Hilary and Hagan to find out the expected success marketing the new product, and the costs of shipping it, respectively. The problem in this setting is that Hagan is not in contact with any person in the Shipping department. In fact, nobody in the Human Resources department is in contact with any person in the Shipping department.”

KNOWLEDGE ACQUISITION

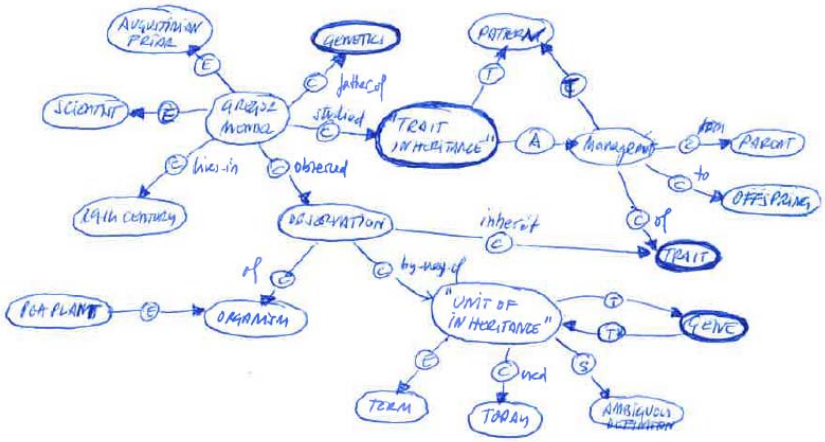
(72d) Use Wikipedia as a domain expert and surf it in a knowledge acquisition process with regard to the following domain areas: d) Choose your own topic of interest. Represent the knowledge captured as a semantic network.

(***Genetics***, *March 2017*): *"Genetics is the study of genes, genetic variation, and heredity in living organisms. It is generally considered a field of biology, but intersects frequently with many other life sciences and is strongly linked with the study of information systems.*

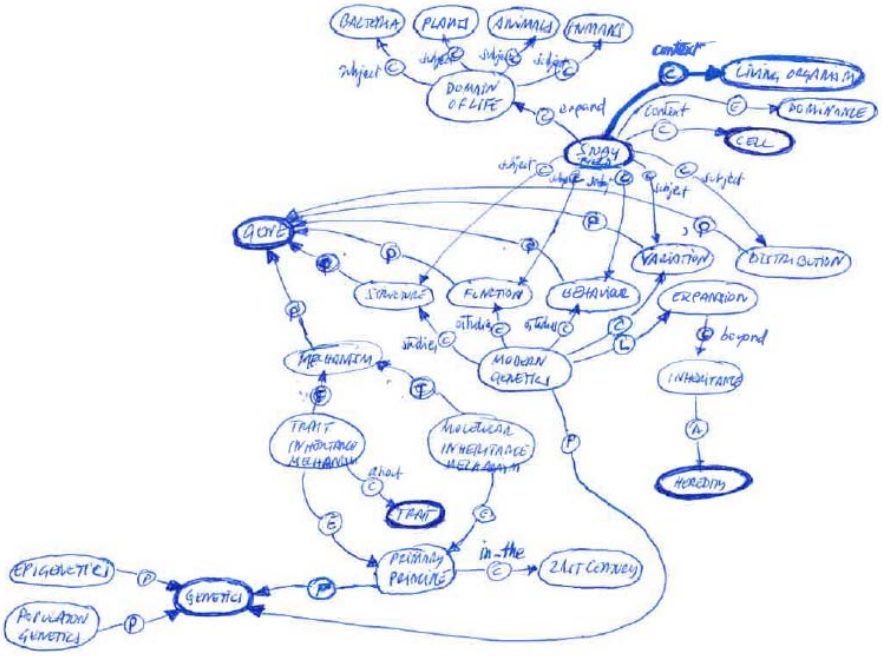




The father of genetics is Gregor Mendel, a late 19th-century scientist and Augustinian friar. Mendel studied "trait inheritance", patterns in the way traits are handed down from parents to offspring. He observed that organisms (pea plants) inherit traits by way of discrete "units of inheritance". This term, still used today, is a somewhat ambiguous definition of what is referred to as a gene.



Trait inheritance and molecular inheritance mechanisms of genes are still primary principles of genetics in the 21st century, but modern genetics has expanded beyond inheritance to studying the function and behavior of genes. Gene structure and function, variation, and distribution are studied within the context of the cell, the organism (e.g. dominance), and within the context of a population. Genetics has given rise to a number of subfields, including epigenetics and population genetics. Organisms studied within the broad field span the domain of life, including bacteria, plants, animals, and humans.



Genetic processes work in combination with an organism's environment and experiences to influence development and behavior, often referred to as nature versus nurture. The intracellular or extracellular environment of a cell or organism may switch gene transcription on or off. A classic example is two seeds of genetically identical corn, one placed in a temperate climate and one in an arid climate. While the average height of the two corn stalks may be genetically determined to be equal, the one in the arid climate only grows to half the height of the one in the temperate climate due to lack of water and nutrients in its environment."

<LEFT AS EXERCISE>

KNOWLEDGE REPRESENTATION IN THE WEB

(73a) Define DTD for each one of the following descriptions and use the web tools discussed in class to validate them : a) “Letters have a header a body and a footer. In the header there is a date (month, day, and year) and the information about the destination (optional person name, position, and full address –number, street/avenue, city, ZIP, optional state, country-), the body has an introduction of the sort formal (“Dear”), friendly (“Dear friend”, “Hi”, “Hello”), or nonexistent, and a free content made of paragraphs. The letter concludes with a footer with a closing sentence that can be formal (“Looking forward to knowing about you”, “Sincerely”, …), semiformal (“Regard”, “Best regards”, …), or informal (“Bye”, “Best regards”, “see you”, “ciao”, …), concluding with the sender, that can contain, the name, the name and the surname, or the name, surname, and affiliation of the sender.”

```
<!ELEMENT letter (header, body, footer)>
<!ELEMENT header (date, destination)>
<!ELEMENT date EMPTY>
<!ATTLIST date month (Jan|Feb|Mar|Apr|May|Jun|Jul|Ago|Sep|Oct|Nov|Dec) #REQUIRED>
<!ATTLIST date day CDATA #REQUIRED>
<!ATTLIST date year CDATA #REQUIRED>
<!ELEMENT destination (person?, address)>
<!ELEMENT person EMPTY>
<!ATTLIST person
     name CDATA #IMPLIED
     position CDATA #REQUIRED>
<!ELEMENT address EMPTY>
<!ATTLIST address
     number CDATA #REQUIRED
     street CDATA #REQUIRED
     city CDATA #REQUIRED
     zip CDATA #REQUIRED
     state CDATA #IMPLIED
     country CDATA #REQUIRED >
<!ELEMENT body (intro?, content)>
<!ELEMENT intro EMPTY>
<!ATTLIST intro
     type (formal|friendly) #REQUIRED
     name (Dear|Dear_Friend|Hi|Hello) #REQUIRED>
<!ELEMENT content (#PCDATA)>
<!ELEMENT footer ((formal|semiformal|informal), sender)>
<!ELEMENT formal EMPTY>
<!ATTLIST formal name (Looking_forward_to_knowing_about_you|Sincerely) #REQUIRED>
<!ELEMENT semiformal EMPTY>
<!ATTLIST semiformal name (Regards|Best_Regards) #REQUIRED>
<!ELEMENT informal EMPTY>
<!ATTLIST informal name (Bye|Best_regards|See_you|Ciao) #REQUIRED>
<!ELEMENT sender (name|(name,surname)|(name,surname,affiliation)) >
<!ELEMENT name (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

Example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE letter SYSTEM "73a Letter.dtd">
 <letter>
   <header>
    <date month="Feb" day="5" year="2017"/>
    <destination>
      <person name="John Smith" position="Director"/>
      <address number="15" street="Main St." city="Cox" zip="56 X89" country="UK"/>
    </destination>
```

```
  </header>
  <body>
    <intro type="formal" name="Dear"/>
    <content>
      Paragraph 1.
      Paragraph 2.
    </content>
  </body>
  <footer>
    <formal name="Sincerely"/>
    <sender>
      <name>Peter</name>
      <surname>White</surname>
      <affiliation>URV</affiliation>
    </sender>
  </footer>
</letter>
```

(75) Construct the XPath queries to recover the following information:

a) *The attribute name*: `/document/@name OR /document/data(@name)`
b) *The document elements below the node "linkList"*: `//linkList/document`
c) *The combination of "lastName", followed by the string ", ", and the "fistName":* `concat(/person/lastName/text(), ", ", /person/firstName/text())`
d) *The jobs with a priority with value equal to "critical" or "high":* `//job[@priority='critical' or @priority='high']`
e) *All persons with age<35*: `//person[@age<35]`
f) *The first three person elements*: `//person[position()<=3]`
g) *Persons whose first name begins with H*: `//person[starts-with(@firstName,'H')]`
h) *Person elements with attribute "firstName" a maximum 5 characters long*: `//person[string-length(@firstName)<5]`
i) *The sum of all numbers which round off to 34 (Result: 102.66)*: `sum(//number[round(text())=34])`
j) *The product siblings relatively from a given node "Lautsprecher":* `//product[@name="Lautsprecher"]/following-sibling::product`
k) *The product siblings containing 1 in category, relatively from a given node "Lautsprecher":* `//product[@name="Lautsprecher"]/following-sibling::product[@category="1"]`
l) *How many albums in the collection are by solo artist and how many are by bands (Result: 4 and 2). The name of the artist that goes with the cd Rudebox (Result: Robbie Williams). The name of the first listed cd of the last artist in the collection (Result: Bad):* `count(/collection/artist/cds/cd)` - `count(/collection/band/cds/cd)` - `//cd[text()='Rudebox']/../../name/text()` - `/collection/artist[last()]/cds/cd[1]/text()`
m) *Number of jobs with priority "low" (Result:* 2): `count(//job[@priority='low'])`
n) Select all job elements with an exceeded budget. That is to say the "availableBudget" is greater than the addition of all its work expenses (attributes "useBudget") (Result: element "cut Screens"): `//job[@availableBudget>sum(./work/data(@usedBudget))]`
o) Calculate the number of titles using its position (Result: 5): `???`

(76) A library contains bookcases and each bookcase is composed of shelves that can contain an undefined number of books and/or magazines. The DTD defining this structure is:

```
<!ELEMENT library (bookcase*)>
<!ELEMENT bookcase (shelf+)>
<!ELEMENT shelf (book* magazine*)>
<!ELEMENT book (author? title)>
<!ATTLIST book code ID #REQUIRED year CDATA #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
```

<!ELEMENT magazine (#PCDATA)>
<!ATTLIST magazine year CDATA #REQUIRED>

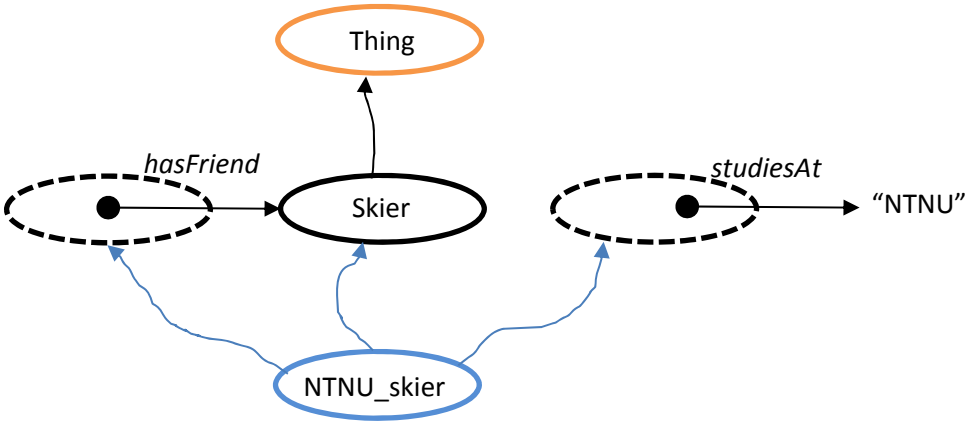
Provide XPath expressions for the following queries:

a) The first book in the library. //book[1]
b) The bookcase that contains the book with code 'A323'. //bookcase/book[@code='A323']/..
c) The title of all the books of Mark Twain in the library. //book/author[.='Mark Twain']/../title
d) The books that are published in year 1968 and are in the second shelf of any bookcase.
//self[2]/book[@year='1968']
e) The shelves that contain only magazines. //self[count(book)=0][count(magazine)>0]
f) Number of books in the library that have no author (anonymous).
count(//book[count(author)=0])

(80a) Given the following OWL DL ontology, define in plain text what their main class stands for:

a)
```
<owl:Class rdf:ID="ntnu_skier">
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:allValuesFrom>
 <owl:Class rdf:ID="skier"/>
 </owl:allValuesFrom>
 <owl:onProperty>
 <owl:ObjectProperty rdf:ID="hasFriend"/>
 </owl:onProperty>
 </owl:Restriction>
 </rdfs:subClassOf>
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:hasValue>
 <university rdf:ID="NTNU"/>
 </owl:hasValue>
 <owl:onProperty>
 <owl:ObjectProperty rdf:ID="studiesAt"/>
 </owl:onProperty>
 </owl:Restriction>
 </rdfs:subClassOf>
 <rdfs:subClassOf rdf:resource="#skier"/>
 </owl:Class>
```

Graphical representation:



Interpretation:

"An NTNU skier is anything that is a skier, whose friends are all skiers and who studies at NTNU (Norwegian University of Science and Technology)".