Example Exam for TI1600-A Multi-Agent Systemen

6 April 2010

This exam will test your knowledge and understanding of the material provided to you and presented in the lectures (slides), the book of Blackburn, Bos and Striegnitz *Learn Prolog Now!* (Chapters 1 to 5, Sections 9.1 & 9.2, Chapter 10, Section 11.2), as well as *The* GOAL *Programming Guide*. It is *not* allowed to use materials such as slides during the exam. You will have 3 hours (from 9 till 12) to complete the exam. You may provide your answers in Dutch as well as in English. It has 6 questions, for a total of 100 points. Please don't include irrelevant information: you will be marked down for this. Before you hand in your answers, please check that you have put your name and student number on top of every sheet you hand in.

Questions

Question 1 *15 points*

This assignment concerns Prolog.

(a) (10 points) Write a (set of) clause(s) that define the predicate `remove(X,L1,L2)` such that `L2` is a list obtained from the list `L1` by removing all occurrences of `X`. For example, the query `remove(3,[1,3,2,3,4],L2).` should yield `L2 = [1, 2, 4]` and `remove(3,[1,2,4],L2).` should yield `L2 = [1, 2, 4]`.

Solution:

```
remove(_,[],[]).
remove(X,[X|T],L) :- remove(X,T,L).
remove(X,[Y|T],[Y|L]) :- not(X=Y), remove(X,T,L).
```

(b) (5 points) Modify the (set of) clause(s) that define the predicate `remove(X,L1,L2)`, such that `L2` is a list obtained from the list `L1` by removing *only the first* occurrence of `X`. If `X` does not occur in `L1`, nothing should be removed. For example, the query `remove(3,[1,3,2,3,4],L2).` should yield `L2 = [1, 2, 3, 4]` and `remove(3,[1,2,4],L2).` should yield `L2 = [1, 2, 4]`.

Solution:

```
remove(_,[],[]).
remove(X,[X|L],L) :- !.
remove(X,[Y|T],[Y|L]) :- not(X=Y), remove(X,T,L).
```

Question 2 *15 points*

This assignment concerns Prolog. Binary trees are trees where all internal nodes have exactly two children. The smallest binary trees consist of only one leaf node. We will represent leaf nodes as `leaf(Int)`, where `Int` is an integer. That is, leaf nodes have integer labels. For example, `leaf(1)` and `leaf(5)` are leaf nodes, and therefore small binary trees. Given two binary trees `B1` and `B2` we can combine them into one binary tree using a predicate `tree/2` as follows: `tree(B1,B2)`.

So, from the leafs `leaf(1)` and `leaf(5)` we can build the binary tree `tree(leaf(1),leaf(5))`, from the binary trees `tree(leaf(1),leaf(5))` and `leaf(3)` we can build the binary tree `tree(tree(leaf(1),leaf(5)),leaf(3))`, from the binary trees `tree(leaf(1),leaf(5))` and `tree(leaf(7),leaf(8))` we can build the binary tree `tree(tree(leaf(1),leaf(5)), tree(leaf(7),leaf(8)))`, etc.

(a) (15 points) Write a (set of) clause(s) that define the predicate `sumTree(Tree,Sum)` such that `Sum` is the sum of the labels of the leafs of the tree `Tree`. For example, the query `sumTree(tree(tree(leaf(1),leaf(5)),leaf(3)),X).` should yield `X = 9`.

Solution:

```
sumTree(leaf(X),X).
sumTree(tree(T1,T2),Sum) :- sumTree(T1,Sum1),
 sumTree(T2,Sum2),
 Sum is Sum1 + Sum2.
```

Question 3 *10 points*

This assignment concerns Prolog. Assume we have a database of results of Unreal Tournament matches played by various teams. The pairings of which team played against which other were not arranged in any systematic way, so each team just played some other team(s). The results are in the program represented as facts like `won(team1,team2), won(team3,team1), won(team4,team2)`, where `won(T1,T2)` represents that team `T1` won the match (and team `T2` lost). In this assignment, we consider a set of clauses that should define a predicate `class(Team,Category)` that ranks teams into categories. We have just three categories:

- **great**: every team that won all its games
- **ok**: any team that won some games and lost some
- **improvable**: any team that lost all its games

For example, given the results stated above, team3 and team4 are great, team1 is ok, and team2 is improvable.

The following knowledge base is a *faulty* implementation of the predicate `class/2`.

```
won(team1,team2).
won(team3,team1).
won(team4,team2).

class(X,ok) :- won(X,_), won(_,X).
class(X,great) :- not(won(_,X)), won(X,_).
class(X,improvable) :- won(_,X), not(won(X,_)).
```

The query `class(X,great)` should yield `X = team3` and `X = team4`, but this knowledge base yields `false`. The following trace is produced:

```
[trace]  ?- class(X,great).
   Call: (7) class(_G335, great) ? creep
^  Call: (8) not(won(_G403, _G335)) ? creep
   Call: (9) won(_G403, _G335) ? creep
   Exit: (9) won(team1, team2) ? creep
^  Fail: (8) not(won(_G403, _G335)) ? creep
   Redo: (7) class(_G335, great) ? creep
false.
```

The queries `class(team3,great).` and `class(team4,great).` do correctly yield `true`.

(a) (5 points) Explain why this program yields the wrong answer when posing the query `class(X,great).`

(b) (5 points) Propose a modification of the program to yield the correct solution.

Solution: Prolog uses negation as failure, which means that the goal `not(won(TeamA,TeamB))` fails if `won(TeamA,TeamB)` succeeds. If `TeamB` is not instantiated, `won(TeamA,TeamB)` will succeed if Prolog can derive `won(TeamA,TeamB)` for some instantiation of `TeamA` and `TeamB` (in this case `won(team1, team2)` as indicated by the trace), and consequently the goal `not(won(TeamA,TeamB))` fails. If the clauses are swapped, `X` is instantiated using the first goal (e.g., `X = team3`), and then `not(won(TeamA,team3))` succeeds.

```
%
 classCorrect(X,ok) :- won(X,_), won(_,X).
 classCorrect(X,great) :- won(X,_), not(won(_,X)).
 classCorrect(X,improvable) :- won(_,X), not(won(X,_)).
```

Question 4 *10 points*

This question concerns the agent programming language GOAL.

(a) (5 points) Explain the difference between the `goal(<fact>)`, `a-goal(<fact>)`, and `goal-a(<fact>)` operators.

> **Solution:** The `goal(<fact>)` operator holds if the `<fact>` follows from one of the goals in the goal base (in combination with the knowledge base), `a-goal(<fact>)` operator holds if `<fact>` follows from one of the goals but is not believed to be true by the agent, and `goal-a(<fact>)` operator holds if `<fact>` follows from one of the goals and is also believed to be true by the agent.

(b) (5 points) Explain the difference between executing a rule of the form `if bel(p(X)) then insert(r(X))` and a rule of the form `forall bel(p(X)) then insert(r(X))`. Also explain what difference it makes to put either of these rules in the **program** section of the **main** module or in that of the **event** module.

> **Solution:** The differences are that
>
> - all instances of action rules of the form `forall bel(p(X)) then insert(r(X))` that make `p(X)` true are executed while only one instance is executed for rules of the form `if bel(p(X)) then insert(r(X))`.
> - the order of the rules in the **program** section of the **main** module by default is linear which means that the first applicable rule is fired whereas in the **event** module by default all applicable rules are fired.

Question 5 *25 points*

This question concerns the agent programming language GOAL.

```
main: deliveryAgent {
  knowledge {
    ordered(C,P) :- order(C,Y), member(P,Y).
    loaded_order(C) :- order(C,O), loaded(O).
    loaded([P]) :- in(P, truck).
    loaded([P|L]) :- in(P,truck), loaded(L).
    empty :- not(in(P,truck)).
    packed :- setOf(P,in(P,truck),L), length(L,X), X>=2.
    delivered_order(C) :- order(C,O),loc(C,X),orderloc(O,X).
    orderloc([H|T], X) :- loc(H,X), orderloc(T,X).
    orderloc([], X).
  }
  beliefs {
    home(a). loc(p1,a). loc(p2,a). loc(p3,a). loc(p4,a). loc(truck,a).
    order(c1,[p1,p2]). order(c2,[p3,p4]).
    loc(c1,b). loc(c2,c).
  }
  goals {
    delivered_order(c1). delivered_order(c2).
  }
  main module{
   program{
    if bel(home(X)) then goto(X).

    if goal(delivered_order(C)) then deliverOrder .
  }
  module deliverOrder[focus=select]{
    program {
      if goal(delivered_order(C)), bel(ordered(C, P), loc(C, X)),
         not(bel(in(P, truck))), not(bel(loc(truck, X))) then load(P).
      if goal(delivered_order(C)), bel(loc(truck, X), loaded_order(C), loc(C, Y))
         then goto(Y).
      if goal(delivered_order(C)), bel(loc(truck, X),loc(C, X),in(P, truck),ordered(C, P))
         then unload(P).
      if bel(empty, home(Y)) then goto(Y).
    }
  }
  actionspec{
    load(P){
      pre{ not(packed), loc(truck,X), loc(P,X) }
      post{ in(P,truck), not(loc(P,X)) }
    }
    goto(Y){
      pre{ loc(truck,X), not(X=Y) }
      post{ loc(truck,Y), not(loc(truck,X))}
    }
    unload(P){
      ...
    }
  }
```

```
}
```

(a) (10 points) Explain which actions the GOAL agent may perform *next*, given the agent program listed above.

Solution: The GOAL agent can enter the module `deliverOrder` (in two ways), and then will perform either `load(p1)`, `load(p2)`, `load(p3)`, or `load(p4)`.

(b) (10 points) Complete the action specification for the action `unload(P)` for unloading a package `P`. Only use the predicates that are already available in the given agent program.

Solution:

```
unload(P){
  pre{ in(P,truck), loc(truck,X) }
  post{ loc(P,X), not(in(P,truck)) }
}
```

(c) (5 points) Explain which goals will be part of the attention set if the module is activated given the mental state that is part of the agent program above.

Solution:
The attenion set consists either of the goal `delivered_order(c1)` or `delivered_order(c2)`.

Question 6 *25 points*

This question concerns the agent programming language GOAL.

```
main: coffeeMaker {
   knowledge {
       requiredFor(coffee, water).
       requiredFor(coffee, grounds).
       requiredFor(espresso, coffee).
       requiredFor(grinds, beans).
       canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
   }
   beliefs {
       have(water). have(beans).
       canMake(maker, [coffee, espresso]).
   }
   goals {
       have(coffee).
   }
   main module{
     program {
       % if we need to make something, then make it (the action's precondition
       % checks if we have what it takes, literally)
       if goal(have(P)) then make(P).
     }
   }
   event module{
     ...
   }
   actionspec {
```

```
    make(Prod) {
        pre { forall(requiredFor(Prod, Req), have(Req)) }
        post { have(Prod) }
    }
}
```

(a) (15 points) Provide a rule that will make the agent above ask (send a question) any other agent what it can make, but only asks this if the agent does not already know what the other agent can make. Only use the predicates that are already available in the given agent program.

Solution:

```
if bel(agent(A), not(me(A)), not(canMake(A, _)))
   then sendonce(A, ?canMake(A, _)).
```

(b) (5 points) Explain in which module you would insert the rule you provided.

(c) (5 points) Explain the difference between the three possible moods of a message. That is, explain the difference between an agent `agent1` that performs `send(agent2, :<fact>)`, `send(agent2, ?<fact>)`, and `send(agent2, !<fact>)`.

Solution: The different moods represent different message types: `:` represents indicative message types (e.g. "The door is open"), `?` represents interrogative message type (e.g. "Is the door open?"), and `!` represents imperative message type ("Open the door"). The main difference is the way these messages are stored (both by the sending and the receiving agent). They result respectively in the following facts being inserted into the mailbox:

- `sent(agent2, <fact>)`,
- `sent(agent2, int(<fact>))`,
- `sent(agent2, imp(<fact>))`,
- `received(agent1, <fact>)`,
- `received(agent1, int(<fact>))`,
- `received(agent1, imp(<fact>))`.

End of exam