Massachusetts Institute of Technology

16.410-13 Principles of Autonomy and Decision Making

Assignment #1, tutorial. Due: LEC #4

Problem PS.1.1.1: A simple forward-chaining rule set

You are given the following database of assertions:

```
(P a b)
(P a c)
(P c d)
(P d e)
(P d f)
```

and the following rules:

```
(R1 IF   (P ?x ?y)
         (P ?y ?z)
    THEN (G ?x ?z))

(R2 IF   (G ?x ?y)
         (P ?y ?z)
    THEN (H ?x ?z))
```

Part 1: New rules

In the space below, write all the (new) assertions that will be added to the database by a forward-chaining system. Be sure to write the assertions as lists, i.e. in the same format as the input assertions shown above.

Part 2: Triggering

1. How many triggered rule instances are found before the first rule is fired?
2. How many of these triggered rule instances are found again after the first rule is fired?

Problem PS.1.1.2: IF-THEN Forward Chaining

The following problems refer to general properties of forward-chaining systems using only deduction rules, that is, rules that have only IF and THEN clauses. For this problem, assume the rules do not use STOP assertions anywhere.

Part 1: Order

Given a rule set with only IF-THEN rules, indicate which of the following statements are true.

1. Nothing that an IF-THEN rule adds to the database can make a rule that is triggered once become untriggered.
2. In our default forward-chainer, assuming no infinite looping, the order of the rules will never affect which assertions end up being added to the database.
3. The following rule can cause infinite looping `IF ?x THEN ?x`
4. The following rule can cause infinite looping `IF THEN (Hello)`
5. The following rule can cause infinite looping `IF ?x THEN (P ?x)`

Part 2: Conflict

Given a rule set with only IF-THEN rules and assuming no infinite looping, which of the following conflict-resolution strategies will produce the same set of new assertions as the strategy implemented in our default forward-chaining system?

1. Fire all triggered rule instances.
2. Fire the triggered rule instance derived from the most recently added assertion (settle ties randomly)
3. Fire the triggered rule instance derived from the least recently added assertion (settle ties randomly)
4. Fire a random triggered rule instance.

True/False problems do not have a Check button.

Problem PS.1.1.3: Finding the way

Impressed with the argument that all intelligent behavior may arise from rule-based processing, you decide to implement a rule-based system that can navigate around Cambridge. Assume that you have a representation of a map in which every intersection is named, e.g. `Harvard-square` or `place-946`, and in which the existence of street segments that connect intersections is represented by assertions of the form

```
(connect-from place-946 Harvard-square)
```

Consider a small version of the Cambridge map, that consists of just the following places and connections:

```
(connect-from tech-square central-square)
(connect-from tech-square inman-square)
(connect-from tech-square cambridge-port)
(connect-from inman-square porter-square)
(connect-from inman-square central-square)
```

```
(connect-from harvard-square fresh-pond)
(connect-from cambridge-port harvard-square)
(connect-from central-square harvard-square)
```

where the assertions are ordered as shown in the rule-chainer's assertion database.

Part 1: Reaching TS

We can define two rules that will let us know where we can get from Tech Square:

```
(start-from-tech-square
  IF (connect-from tech-square ?x)
  THEN (reachable-from-tech-square ?x (tech-square ?x)))
(reach-from-tech-square
  IF (reachable-from-tech-square ?x ?path)
     (connect-from ?x ?y)
  THEN (reachable-from-tech-square ?y (?path ?y)))
```

Note: The last element of the `(reachable-from-tech-square ...)` assertions is meant to be the path of how the second element can be reached from Tech Square. Note that it is not a list of places because ?path is a list, so this will produce paths such as ((A B) C) instead of (A B C).

Fill in the following assertions in the order that they will be added to the database by our Scheme forward rule chainer (note that new assertions are added at the end of the list of assertions) . The rules are ordered as shown above. Please use the following abbreviations in your answer:

- **TS** = tech-square
- **CS** = central-square
- **HS** = harvard-square
- **IS** = inman-square
- **PS** = porter-square
- **CP** = cambridge-port
- **FP** = fresh-pond

1. `(reachable-from-tech-square)`
2. `(reachable-from-tech-square)`
3. `(reachable-from-tech-square)`
4. `(reachable-from-tech-square)`
5. `(reachable-from-tech-square)`
6. `(reachable-from-tech-square)`
7. `(reachable-from-tech-square)`
8. The total number of new assertions added to the database is _____
9. The total number of assertions added to the database that match the following pattern `(reachable-from-tech-square harvard-square ?p)` is _____

Problem PS.1.1.3: Finding the way

Impressed with the argument that all intelligent behavior may arise from rule-based processing, you decide to implement a rule-based system that can navigate around Cambridge. Assume that you have a representation of a map in which every intersection is named, e.g. `Harvard-square` or `place-946`, and in which the existence of street segments that connect intersections is represented by assertions of the form

```
(connect-from place-946 Harvard-square)
```

Consider a small version of the Cambridge map, that consists of just the following places and connections:

```
(connect-from tech-square central-square)
(connect-from tech-square inman-square)
(connect-from tech-square cambridge-port)
(connect-from inman-square porter-square)
(connect-from inman-square central-square)
(connect-from harvard-square fresh-pond)
(connect-from cambridge-port harvard-square)
(connect-from central-square harvard-square)
```

where the assertions are ordered as shown in the rule-chainer's assertion database.

Part 2: Reaching anywhere

Navigating a map of Cambridge only from Tech Square seems a bit too egocentric, so you define two, more general, rules **in place of** the two above.

We can define two rules that will let us know where we can get from Tech Square:

```
(reachable-by-connect
  IF (connect-from ?start ?end)
  THEN (reachable ?start ?end (?start ?end)))
(reachable-by-reach
  IF (reachable ?x ?y ?path1)
     (reachable ?y ?z ?path2)
  THEN (reachable ?x ?z (?path1 ?path2)))
```

Note: The third element of `(reachable ...)` again collects an indication of the path, but the format of the path is a bit convoluted. We really want to `append` the paths not just put them into a list. Also, in the rule `reachable-by-reach`, the place that matches `?y` will be part of both paths `?path1` and `?path2`, so it will show up twice in the combined path. So, the rule will construct paths such as ((A B) (B D)) instead of (A B D). We ignore these difficulties so as to simplify the problem a bit.

1. What are the number of physically distinct paths (in our little database) between `tech-square` and `harvard-square`?
2. What is the total number of assertions added to the database that match the pattern: `(reachable tech-square harvard-square ?p)`?
3. If we could append `?path1` and `?path2` in the second rule, how many assertions matching the pattern above would be added?

Part 3: Reaching anywhere again

Write a variant of the reachable-by-reach rule that generates the expected number of paths between two places

Part 4: Reaching symmetry

Thus far, our Cambridge has the odd characteristic that all paths can be walked in only one direction. Because real streets are very rarely like this, and because we don't want to double the size of the database of `(connect-from ...)` assertions, we add the following rule:

```
(symmetric-connect
  IF (connect-from ?x ?y)
  THEN (connect-from ?y ?x))
```

With this rule, we start worrying about the potential of infinite loops; let's consider the effect of several conflict-resolution strategies.

1. If this rule were the only one in the system:
    a. Our forward-chaining system, using rule-ordering, would generally go into an infinite loop
    b. Our forward-chaining system, modified so that a triggered rule-instance involving the least-recently added assertion is fired, would generally go into an infinite loop
    c. Our forward-chaining system, modified so that a triggered rule-instance involving the most-recently added assertion is fired, would generally go into an infinite loop
    d. Our forward-chaining system, modified so that a randomly chosen triggered rule-instance is fired, would generally go into an infinite loop
2. If we had the `symmetric-connect` rule followed by the two rules in the first part of this problem:
    a. Our forward-chaining system, using rule-ordering, would generally go into an infinite loop
    b. Our forward-chaining system, modified so that a triggered rule-instance involving the least-recently added assertion is fired, would generally go into an infinite loop
    c. Our forward-chaining system, modified so that a triggered rule-instance involving the most-recently added assertion is fired, would generally go into an infinite loop
    d. Our forward-chaining system, modified so that a randomly chosen triggered rule-instance is fired, would generally go into an infinite loop
    e. Assume we add an `AND-IF` clause to the rule `reach-from-tech-square` that makes sure that the place that matches `?y` is not part of

the path that matches `?path`. Then, our forward-chaining system, using rule-ordering, would generally go into an infinite loop

3. If we had the `symmetric-connect` rule followed by the two rules in the first part of this problem, but we added a mechanism to stop chaining once we reached our desired destination (e.g. HS) [assume the rule (stop IF (reachable-from-tech-square harvard-square ?x) THEN (STOP)) is added at the front of the list of rules]:
    a. A forward-chaining system, using rule-ordering and which adds new assertions to the front of the list of assertions, would generally go into an infinite loop before reaching the destination
    b. Our forward-chaining system, modified so that a triggered rule-instance involving the least-recently added assertion is fired, would generally go into an infinite loop before reaching the destination.
    c. Our forward-chaining system, modified so that a triggered rule-instance involving the most-recently added assertion is fired, would generally go into an infinite loop before reaching the destination.
    d. Our forward-chaining system, modified so that a randomly chosen triggered rule-instance is fired, would generally go into an infinite loop before reaching the destination.
    e. Assume we add an `AND-IF` clause to the rule `reach-from-tech-square` that makes sure that the place that matches `?y` is not part of the path that matches `?path`. Then, our forward-chaining system, using rule-ordering, would generally go into an infinite loop before reaching the destination.

True/False problems do not have a Check button.

Problem PS.1.1.4: Family Chaining

You are given a large genealogical database made up of assertions like the following:

```
(A is-parent-of B)
(A is-parent-of C)
(C is-parent-of D)
(A is-of-gender Female)
(B is-of-gender Male)
(C is-of-gender Female)
(D is-of-gender Female)
```

There are no other types of assertions in the database, only `is-parent-of` and `is-of-gender` assertions.

Part 1: Grampy

Write an IF-THEN rule which, given this type of database, adds assertions of the form `(A is-grandparent-of D)`. Gender is irrelevant.

> Please pick your variable names from the following: ?person, ?parent, grandparent.

Part 2: Crying Uncle

Write an IF-THEN rule which, given this type of database, adds assertions of the form `(B is-uncle-of D)`. Do not make distinctions between siblings who share two parents or just one parent.

> Please pick your variable names from the following: ?person, ?parent, ?uncle, ?grandparent.
>
> Note that to force two variables `?x` and `?y` to match different people you can add a clause to the rule like this:
>
> ```
> AND-IF (not (eq? '?x '?y))
> ```
> In general, the AND-IF clauses will be evaluated as Scheme expressions with the current bindings of the rule variables substituted in. Note that you need to use `'?x` to avoid trying to evaluate constants such as A and B. **Syntax errors in AND-IF expressions produce a scary and very uninformative Scheme error message - sorry.**

Problem PS.1.1.5: Anti-Tic-Tac-Toe

You are assigned to develop a program (Light Blue) that plays **Anti-Tic-Tac-Toe (ATTT)**. In ATTT, you lose by putting three of your marks (X or O) in a row on a 3X3 board. You win when the other player loses.

We will represent the locations on the board by symbols such as A1, B2, C3, etc. where the row is indicated by a letter (A, B or C) and the column by a number (1, 2 or 3):

```
  1   2   3
A   |   |
 -----------
B   |   |
 -----------
C   |   |
```

Part 1: Diagonals

You decide to start with the following assertions, noting that this is all you need to define the board layout:

```
(A1 top-of B1), (B1 top-of C1)
(A2 top-of B2), (B2 top-of C2)
(A3 top-of B3), (B3 top-of C3)
(A3 right-of A2), (A2 right-of A1)
(B3 right-of B2), (B2 right-of B1)
(C3 right-of C2), (C2 right-of C1)
```

However, you think that it would be nice to have 'diagonal' assertions of the form:

```
(A3 top-right-of B2), (B2 top-left-of C3)
```

Write IF-THEN rules to add all of the appropriate `top-right-of` and `top-left-of` assertions.

Part 2: Losing

You now encode what is in each location on the board by assertions such as:

`(A1 is X), (B3 is O), (B2 is blank), etc.`

You figure it would also be nice to have a rule that will add an assertion when a player has lost, e.g. `(X lost)` or `(O lost)`. Write one or more rules that add assertions of this type. You can assume that the `top-righth-of` and `top-left-of` assertions are already available (as produced by correct rules from Part 1).

Note that to test that a variable is NOT equal to a value you can do, for example:

```
AND-IF (not (eq? '?x 'blank))
```

In general, the AND-IF clauses will be evaluated as Scheme expressions with the current bindings of the rule variables substituted in. Note that you need to use `'?x` to avoid trying to evaluate constants such as A1. **Syntax errors in AND-IF expressions produce a scary and very uninformative Scheme error message - sorry.**

Part 3: Playing

Now, to encode some move-making logic into Light Blue. As a start, you decide that it is a good strategy to move between a mark of yours and a mark of the opponents, if possible. Thus, B2 is a good move here:

```
   1   2   3
A    |   |
 -----------
B  X |   | O
 -----------
C    |   |
```

Write a rule that will perform this move, regardless of the direction. You may assume that you are X.

Problem PS.1.2.1: A simple backward-chaining rule set (no rule variables)

You are given the following set of rules:

```
(r1
IF
(the patient is FATIGUED)
(the patient has a FEVER)
THEN
(the diagnosis is (patient has a COLD)))

(r2
IF
(the patient has MUSCLE aches)
(the patient has NORMAL temperature)
THEN
(the diagnosis is (patient has the FLU)))

(r3
IF
(the patient SLEEPS a lot)
(the patient avoids EXERCISE)
THEN
(the patient is FATIGUED))

(r5
IF
(the temperature of the patient is GREATER than 101)
THEN
(the patient has a FEVER))
```

Assume that there are initially no assertions in the database. Write the questions that would be asked of the user by a backward chaining system (as illustrated in the recitation material). The initial goal is:

```
(the diagnosis is ?x)
```

Assume that the user will answer the question

```
(the temperature of the patient is GREATER than 101)
```

as No but all other questions as Yes. To specify the question simply copy the assertion being queried into the space below (including parens). The correct order is part of the answer; assume the order is as in the recitation examples.

1. Questions asked (in order)
2. What is the diagnosis? Enter the complete assertion (including parents)

Problem PS.1.2.2: Unification

Fill in the variable assignments required to unify the expressions. Your answers should be a list of statements like this `(?x = A)` or `(?x = ?y)` `Your answer should include the parens around each assignment. There needs to be spaces around the =. If there is no unification possible, enter None.`

1. `(?x (B ?y)) and (A (?z ?w))`
2. `(?x (B ?y)) and (A (?y ?w))`
3. `(?x (B ?y)) and (A (?y C))`

Problem PS.1.2.3: A simple backward-chaining rule set (with rule variables)

Consider the following set of rules, which are intended to be a tiny fraction of a grammar of English.

```
 (r1
 IF                                    ; no antecedent, means always
true
 THEN (article (the . ?s1) ?s1))
 (r2
 IF                                    ; no antecedent, means always
true
 THEN (noun (ball . ?s2) ?s2))
 (r3
 IF   (article ?s3 ?s4)
      (noun    ?s4 ?s5)
 THEN (NP      ?s3 ?s5)
```

Each clause is of the form `(type start rest)`, where `type` is the type of a word or phrase, `start` is a pattern describing the start of the word or phrase in a list of words and `rest` is the rest of the list of words (past the end of the word or phrase). So, the goal `(article (the ball) (ball))` matches the consequent of `r1`. The goal `(NP (the ball) ())` can be shown to be true using these rules. Give the variable bindings for each rule as it is used in the backchaining process.

1. For rule `r3: ?s3 =  ?s4 =  ?s5 =`
2. For rule `r2: ?s2 =`
3. For rule `r1: ?s1 =`

Problem PS.1.2.4: Derivative Rules

In this problem we will explore the use of backward chaining to implement a simple differentiation system. Some sample rules would be:

```
 (plus
   IF
```

```
   (deriv ?v ?dv)
   (deriv ?w ?dw)
   THEN (deriv (plus ?v ?w) (plus ?dv ?dw)))
 (times
  IF
  (deriv ?v ?dv)
  (deriv ?w ?dw)
  THEN (deriv (times ?v ?w) (plus (times ?v ?dw) (times ?dv ?w))))
```

Each (deriv *w dw/dx*) assertion in these rules expresses that the second entry is the derivative of the first entry with respect to x.

We will use an auxiliary rule that can tell us if an expression can be treated as a constant, that is, when it does not contain x.

```
(aux
  IF
  AND-IF
  (not (contains? 'x '?x))
  THEN
  (constant ?x))
```

Your job is to finish up this system of rules so that we can differentiate expression involving the sum and product of terms (some of which involve the variable x). For example, we would do

```
(backchain '(deriv (plus x (times x x)) ?ans))
```

and expect to find a binding for ?ans. To keep things simple, we will not attempt to simplify any expressions for now. We are looking for a couple of very simple rules.

Problem PS.1.2.5: Production Rules

Imagine that we have a robot that moves around within a room in which there are a number of other objects. Each of these objects, and the robot, has some location characterized by a pair of numbers (x and y on some grid). The robot is also able to climb on objects. We describe the state of the world by a set of assertions of the following types:

```
    (at location)
    (on object)
    (object at location)
    (loc location)
```

The (at *location*>) and (on *object*) assertions describe the state of the robot. The (*object* at *location*) assertions describe each object's location. There is a special object, floor, with no specified location. The existence of a location is indicated by (loc *location*). There are two actions that the robot can perform:

- WALK - The robot must be on the floor as a precondition for walking and it will be at the target location afterwards.

- CLIMB - There are two cases. Climbing down from an object: the robot can do this at any time that it is not already on the floor. After the climb-down, the robot is on the floor. Climbing onto an object: the robot must be on the floor at the same location as that object. After the climb-up, the robot is on the relevant object.

Here are some production style rules that we could imagine using to model this sitation.

```
  (end
   IF
   (goal ?g)
   ?g
   SAYING "Stop"
   ADD (stop))                        ;-- will terminate chaining

  (climb-down
   IF
   (on ?support)
   AND-IF (not (equal? '?support 'floor)) ;-- note vars have to be
quoted
   SAYING "CLIMB down to floor"
   ADD    (on floor)
   DELETE (on ?support))

  (walk
   IF
   (on floor)
   (at ?old-loc)
   (loc ?new-loc)
   AND-IF (not (equal? '?old-loc '?new-loc)) ;-- note vars have to be
quoted
   SAYING "WALK to " ?new-loc
   ADD    (at ?new-loc)
   DELETE (at ?old-loc))
```

and here are some assertions for a simple situation:

```
  (loc (9 5))
  (loc (3 4))
  (loc (5 6))
  (at (9 5))
  (on floor)
  (table at (5 6))
  (stool at (3 4))
  (goal (on table))
```

Part 1: Forward ho!

Now we want to write a climb-up rule of the following form:

```
        (climb-up
         IF
         SAYING "CLIMB onto " ?object
         ADD
         DELETE )
```

In filling the questions below, use the variables `?loc` for locations and `?object` for objects

1. What should the IF clauses of the climb-up rule be?
2. What should the ADD clause of the climb-up rule be?
3. What should the DELETE clause of the climb-up rule be?

Part 2: Produce!

Assume we add the climb-up rule at the end of the rules given earlier. Now, we start the forward-chaining system going (using rule-ordering as a conflict resolution strategy).

1. The first rule-instance fired deletes the assertion and adds the assertion.
2. The second rule-instance fired deletes the assertion and adds the assertion.

Problem PS.1.3.1: Hello World

Write a simple Scheme function, hello-world, that takes no parameters, and returns the list (hello-world 5).

Problem PS.1.4.1: Hours

We want to understand how much time it took students to answer the questions on the problem sets.

> Approximately how many hours did you spend really working on this problem set?