CS 560: Homework 8

Eric Stevens

December 2, 2018

Question 1

Everyone is either a male or a female but not both.
$\forall X \ (female(X) \vee male(X)) \wedge \neg(female(X) \wedge male(X))$

All female vegetarians are butchers.
$\forall X \ female(X) \wedge vegetarian(X) \longrightarrow butcher(X)$

Only female vegetarians are butchers.
$\forall X \ \neg female(X) \vee \neg vegetarian(X) \longrightarrow \neg butcher(X)$

All butchers are female vegetarians.
$\forall X \ butcher(X) \longrightarrow female(X) \wedge vegetarian(X)$

No woman likes a butcher who is male.
$\neg \exists X \ woman(X) \longrightarrow \forall Y \ butcher(Y) \wedge likes(X, Y)$

Question 2

```
pickup(Obj, X)
    Preconditions:  empty, on(Obj,X), block(Obj), block(X), clear(O
bj)
    Delete List:    empty, on(Obj,X), clear(Obj)

    Add List:       holding(Obj), clear(X)

pickup(Obj, X)
    Preconditions:  empty, on(Obj,X), block(Obj), table(X), clear(O
bj)
    Delete List:    empty, on(Obj,X), clear(Obj)

    Add List:       holding(Obj)

putdown(Obj, X)
    Preconditions:  holding(Obj), block(X), clear(X)
    Delete List:    holding(Obj), clear(X)
    Add List:       empty, on(Obj,X), clear(Obj)
```

```
putdown(Obj, X)
    Preconditions:  holding(Obj), table(X)
    Delete List:    holding(Obj)
    Add List:       empty, on(Obj,X), clear(Obj)
```

Question 3

```
block(a), block(b), block(c), table(t)
on(a,t), on(c,a), clear(c)
on(b,t), clear(b)
empty
```

Question 4

```
block(a), block(b), block(c), table(t)
on(a,t), on(c,a), clear(c)
on(b,t), clear(b)
empty

pickup(c,a)

block(a), block(b), block(c), table(t)
on(a,t), clear(a)
on(b,t), clear(b)
holding(c)

putdown(c,t)

block(a), block(b), block(c), table(t)
on(a,t), clear(a)
on(b,t), clear(b)
on(c,t), clear(c)
empty

pickup(b,t)

block(a), block(b), block(c), table(t)
on(a,t), clear(a)
on(c,t), clear(c)
holding(b)




putdown(b,c)
```

```
block(a), block(b), block(c), table(t)
on(a,t), clear(a)
on(c,t), on(b,c), clear(b)
empty

pickup(a,t)

block(a), block(b), block(c), table(t)
on(c,t), on(b,c), clear(b)
holding(a)

putdown(a,b)

block(a), block(b), block(c), table(t)
on(c,t), on(b,c), on(a,b), clear(a)
empty
```

Question 5

```
peg(a), peg(b), peg(c)
disc(grade), disc(large), disc(medium), disc(small)
bigger(grande, large), bigger(grade, medium), bigger(grande, small)
bigger(large, medium) bigger(large, small)
bigger(medium, small)
```

Question 6

```
on(grande, pega), on(large, grande), on(medium, large), on(small, m
edium)
clear(small), clear(pegb), clear(pegc)
```

Question 7

```
on(grande, pegc) ^ on(large, grande) ^ on(medium, large) ^ on(smal
l, medium)
```

Question 8

```
move(X,Y,Z)
    Preconditions:  disc(X), clear(X), on(X,Y), disc(Z), clear(Z),
 bigger(Z,X)
    Delete List:    on(X,Y), clear(Z)
    Add List:       on(X,Z), clear(Y)

move(X,Y,Z)
    Preconditions:  disc(X), clear(X), on(X,Y), peg(Z), clear(Z)
    Delete List:    on(X,Y), clear(Z)
    Add List:       on(X,Z), clear(Y)
```

Question 8 Critique

In the first version of move I added the predicate `disc` to ensure that the predecate acted on the object it was intended to act on. Due to smart knowledge engineering, this is taken care of by the `bigger` predicate, making my `disc` predicate unneeded.

Question 9

To represent the primitive predicates in the initial state of the world you need to add 'init' as a term in each of the preimitive predicates in the initial state. This represents the fact these predicates are true in the initial state and could be false in others.

This does not need to be done to static primatives because the static primatives are defined to be true in all states of the world.

Question 10

$$
\begin{aligned}
&poss(move(X,Y,Z),S) \longleftarrow \\
&\quad disc(X) \land \\
&\quad clear(X,S) \land \\
&\quad on(X,Y,S) \land \\
&\quad disc(Z) \land \\
&\quad clear(Z) \land \\
&\quad bigger(Z,X)
\end{aligned}
$$

$$
\begin{aligned}
&poss(move(X,Y,Z),S) \longleftarrow \\
&\quad disc(X) \land \\
&\quad clear(X,S) \land \\
&\quad on(X,Y,S) \land \\
&\quad peg(Z) \land \\
&\quad clear(Z)
\end{aligned}
$$

Question 10 Critique

In the first version of `move` I added the predicate `disc` to ensure that the predecate acted on the object it was intended to act on. Due to smart knowledge engineering, this is taken care of by the `bigger` predicate, making my `disc` predicate unneeded.

Question 11

Question 12

In [68]:
```
from hw4standard import *

NumActions = 0
Action = {}

class Action:
    all = []
    def __init__(self,head,pre,add,dele):
        self.head = head
        self.pre = pre
        self.add = add
        self.dele = dele
        Action.all.append(self)

        vars = findvariables(head,[])
        vars = findvariables(pre,vars)
        self.vars = vars

    def __str__(self):
        s = "Action %s\n" % prettyexpr(self.head)
        s += "  Preconditions:"
        for i in self.pre:
            s += " %s" % prettyexpr(i)
        s += "\n"
        s += "  Add list:      "
        for i in self.add:
            s += " %s" % prettyexpr(i)
        s += "\n"
        s += "  Delete list:  "
        for i in self.dele:
            s += " %s" % prettyexpr(i)
        s += "  Variables are"
        for i in self.vars:
            s += " " + i
        return s

#pickup block on block
Action(['pickup', 'Obj', 'X'],
       [['on','Obj','X'],['empty'],['block','Obj'],['block','X'],['clear','
       [['holding','Obj'],['clear','X']],
       [['empty'],['on','Obj','X'],['clear','Obj']])

#pickup block on table
Action(['pickup', 'Obj', 'X'],
       [['on','Obj','X'],['empty'],['block','Obj'],['table','X'],['clear','
       [['holding','Obj']],
       [['empty'],['on','Obj','X'],['clear','Obj']])

#putdown block on block
Action(['putdown', 'Obj', 'X'],
       [['holding','Obj'],['block','X'],['clear','X']],
       [['empty'],['on','Obj','X'],['clear','Obj']],
       [['holding','Obj'],['clear','X']])

#putdown block on table
Action(['putdown', 'Obj', 'X'],
       [['holding','Obj'],['table','X']],
```

```
    [['empty'],['on','Obj','X'],['clear','Obj']],
    [['holding','Obj']])

for a in Action.all:
    print(a.__str__())
    print(' ')
```

```
Action pickup(Obj,X)
  Preconditions: on(Obj,X) empty block(Obj) block(X) clear(Obj)
  Add list:      holding(Obj) clear(X)
  Delete list:   empty on(Obj,X) clear(Obj)  Variables are Obj X

Action pickup(Obj,X)
  Preconditions: on(Obj,X) empty block(Obj) table(X) clear(Obj)
  Add list:      holding(Obj)
  Delete list:   empty on(Obj,X) clear(Obj)  Variables are Obj X

Action putdown(Obj,X)
  Preconditions: holding(Obj) block(X) clear(X)
  Add list:      empty on(Obj,X) clear(Obj)
  Delete list:   holding(Obj) clear(X)  Variables are Obj X

Action putdown(Obj,X)
  Preconditions: holding(Obj) table(X)
  Add list:      empty on(Obj,X) clear(Obj)
  Delete list:   holding(Obj)  Variables are Obj X
```

The `Action` class is a very interesting class that I do not fully understand. It appears that when the `Action` constructor is called it generates the action variable, then accesses a list, `all` , which was initialized at the calling of class but does not appear to be a member variable. Yet at the end of the constructor `all` is accesed as a member of `Action` in order to append `self` to it. It seems some sort of temporary class is being created called `self` that stores the action that the constructor is initializing, and then that is stored in the only `Action` class member variable `all` . I want to learn more about this design.

Question 13

In [69]:

```

Constants = ['a','b','c','t']
Primitives = ['on','clear','empty','holding']

Initial = [['block','a'],['block','b'], ['block','c'], ['table','t'],
           ['on','a','b'],['on','b','c'],['on','c','t'],
           ['clear','a'],['empty']]

Goal = [['on','c','b'],['on','b','a'],['on','a','t'] ]


```

Question 14

The only thing added to the prove function is checking to see if the Goal is in topWorld.

MY CODE DOES NOT WORK

In [71]:

```


def subset(sub,set):
    for i in sub:
        if not i in set:
            return False
    return True

def remove(sub,set):
    newset = []
    for i in set:
        if not i in sub:
            newset.append(i)
    return newset

def printWorld(world,indent):
    str = indent
    for p in world:
        if p[0] in Primitives:
            str += "  " + prettyexpr(p)
    print(str)

def printPlan(plan,indent):
    str = indent
    for p in plan:
        str += "  " + prettyexpr(p)
    print(str)

def prove():
    worldcnt = 0
    # each item on frontier is a tuple of plan + world that results fr
    frontier = [[[],Initial]]

    while frontier:
        topPlan,topWorld = frontier[0]

        print("Current plan:")
        printPlan(topPlan,"    ")

        print("Current world (primitives):")
        printWorld(topWorld,"    ")

        # check if the topWolrd has goal true in it
        # if so, say that the goal was found and return true

        # YOUR CODE HERE
        if subset(Goal, topWorld): return True

        # Your code should be able to find the plan after exploring al
        # I put in this stop code in case your code has a bug, so that
        print("")
        if len(topPlan) > 6:
            print("Could not find plan in 8 steps")
            return False

        neighbors = []

```

```
        for action in Action.all:

            # We could use a theorem prover to find variable instantia
            # the preconditions true.
            # Instead, let's enumerae over all variable instantiations

            #---------------------
            # The code between the two sets of dashes iterates all var
            # One of the questions asks you to explain how it works

            vars = action.vars
            numvars = len(vars)
            numconstants = len(Constants)
            cnt = int(pow(numconstants,numvars))
            for i in range(cnt):
                subs = {}
                j = i
                for v in vars:
                    c = j % numconstants
                    subs[v] = Constants[c]
                    j = (j - c)//numconstants
                #------------------

                head = substitute(action.head,subs)
                pre  = substitute(action.pre,subs)
                add  = substitute(action.add,subs)
                dele = substitute(action.dele,subs)

                if subset(pre,topWorld):
                    worldcnt += 1
                    print("%d: Found applicable action" % worldcnt)
                    print(action)

                    # Create the new world and the new plan
                    # And add them to the neighbors

                    # YOUR CODE HERE

                    print("    New world:")
                    printWorld(newWorld,"      ")
                    print("    New plan :")
                    printPlan(newPlan,"      ")
                    print("")

        frontier = frontier[1:]+ neighbors

    print("No world with goal true was found")

```

Question 15

Question 16

Question 17

A depth first search would result in cyclical action patterns that would loop infinitely and fail to resolve. Patterns such and puton(a,b) -> takeoff(a,b) -> puton(a,b) -> takeoff(a,b) would occour.

Question 18

```
kb([has_tree(T,T)]).
kb([has_tree(T,n(N,LT,RT)), has_tree(T,LT)]).
kb([has_tree(T,n(N,LT,RT)), has_tree(T,RT)]).
```

Question 19

```
depthFirstProve([yes(X,Y),has_tree(n(X,l(l4),Y),n(n1,n(n2,l(l1),l(l
2)),n(n3,l(l3),n(n4,l(l4),l(l5)))))])
```

Question 20

```
 neighbor(AnswerClause,NewAnswerClause) :-
        [Yes,Head|Body] = AnswerClause,
        kb([Head|NewBody]),
        NewAnswerClause = [Yes,Head,NewBody].
```

Results:

```
?- neighbor([yes(X,Y),has_tree(n(X,l(l4),Y),n(n1,n(n2,l(l1),l(l2)),
n(n3,l(l3),n(n4,l(l4),l(l5)))))],X).

X = [yes(X, Y), has_tree(n(X, l(l4), Y), n(n1, n(n2, l(l1), l(l2)),
 n(n3, l(l3), n(n4, l(l4), l(l5))))),
[has_tree(n(X, l(l4), Y), n(n2, l(l1), l(l2)))]] ;

X = [yes(X, Y), has_tree(n(X, l(l4), Y), n(n1, n(n2, l(l1), l(l2)),
 n(n3, l(l3), n(n4, l(l4), l(l5))))),
[has_tree(n(X, l(l4), Y), n(n3, l(l3), n(n4, l(l4), l(l5))))]].
```

Question 20 Critique

Here, I had a missunderstanding in terms of what should be returned. I believe that I would have the correct solution with only a slight alteration to my origional predicate.

```
neighbor(AnswerClause,NewAnswerClause) :-
       [Yes,Head|Body] = AnswerClause,
       kb([Head|NewBody]),
       NewAnswerClause = [Yes,Body,NewBody].
```