

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(Obj)`

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(Obj)`

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 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 primitive 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 primitives because the static primitives are defined to be true in all states of the world.

Question 10

```

poss(move(X,Y,Z), S) ←
  disc(X) ∧
  clear(X, S) ∧
  on(X,Y,S) ∧
  disc(Z) ∧
  clear(Z) ∧
  bigger(Z,X)

poss(move(X,Y,Z), S) ←
  disc(X) ∧
  clear(X, S) ∧
  on(X,Y,S) ∧
  peg(Z) ∧
  clear(Z)

```

Question 11

Question 12

```

In [68]: 1 from hw4standard import *
2
3 NumActions = 0
4 Action = {}
5
6 class Action:
7     all = []
8     def __init__(self, head, pre, add, dele):
9         self.head = head
10        self.pre = pre
11        self.add = add
12        self.dele = dele
13        Action.all.append(self)
14
15        vars = findvariables(head, [])
16        vars = findvariables(pre, vars)
17        self.vars = vars
18
19    def __str__(self):
20        s = "Action %s\n" % prettyexpr(self.head)
21        s += "  Preconditions:"
22        for i in self.pre:
23            s += " %s" % prettyexpr(i)
24        s += "\n"
25        s += "  Add list:      "
26        for i in self.add:
27            s += " %s" % prettyexpr(i)
28        s += "\n"
29        s += "  Delete list:  "
30        for i in self.dele:
31            s += " %s" % prettyexpr(i)
32        s += "  Variables are"
33        for i in self.vars:
34            s += " " + i
35        return s
36
37    # pickup block on block
38    Action(['pickup', 'Obj', 'X'],
39           [['on', 'Obj', 'X'], ['empty'], ['block', 'Obj'], ['block', 'X'], ['clear', 'Obj']],
40           [['holding', 'Obj'], ['clear', 'X']],
41           [['empty'], ['on', 'Obj', 'X'], ['clear', 'Obj']])
42
43    # pickup block on table
44    Action(['pickup', 'Obj', 'X'],
45           [['on', 'Obj', 'X'], ['empty'], ['block', 'Obj'], ['table', 'X'], ['clear', 'Obj']],
46           [['holding', 'Obj']],
47           [['empty'], ['on', 'Obj', 'X'], ['clear', 'Obj']])
48
49    # putdown block on block
50    Action(['putdown', 'Obj', 'X'],
51           [['holding', 'Obj'], ['block', 'X'], ['clear', 'X']],
52           [['empty'], ['on', 'Obj', 'X'], ['clear', 'Obj']],
53           [['holding', 'Obj'], ['clear', 'X']])
54
55    # putdown block on table
56    Action(['putdown', 'Obj', 'X'],
57           [['holding', 'Obj'], ['table', 'X']],

```

```

58         [['empty'], ['on', 'Obj', 'X'], ['clear', 'Obj']],
59         [['holding', 'Obj']])
60
61     for a in Action.all:
62         print(a.__str__())
63         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 accessed 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]: 1
2 Constants = ['a', 'b', 'c', 't']
3 Primitives = ['on', 'clear', 'empty', 'holding']
4
5 Initial = [['block', 'a'], ['block', 'b'], ['block', 'c'], ['table', 't'],
6           ['on', 'a', 'b'], ['on', 'b', 'c'], ['on', 'c', 't'],
7           ['clear', 'a'], ['empty']]
8
9 Goal = [['on', 'c', 'b'], ['on', 'b', 'a'], ['on', 'a', 't']]
10
11

```

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]:

```

1
2
3 def subset(sub,set):
4     for i in sub:
5         if not i in set:
6             return False
7     return True
8
9 def remove(sub,set):
10    newset = []
11    for i in set:
12        if not i in sub:
13            newset.append(i)
14    return newset
15
16 def printWorld(world,indent):
17     str = indent
18     for p in world:
19         if p[0] in Primitives:
20             str += " " + prettyexpr(p)
21     print(str)
22
23 def printPlan(plan,indent):
24     str = indent
25     for p in plan:
26         str += " " + prettyexpr(p)
27     print(str)
28
29 def prove():
30     worldcnt = 0
31     # each item on frontier is a tuple of plan + world that results fr
32     frontier = [[[],Initial]]
33
34     while frontier:
35         topPlan,topWorld = frontier[0]
36
37         print("Current plan:")
38         printPlan(topPlan," ")
39
40         print("Current world (primitives):")
41         printWorld(topWorld," ")
42
43         # check if the topWolrd has goal true in it
44         # if so, say that the goal was found and return true
45
46         # YOUR CODE HERE
47         if subset(Goal, topWorld): return True
48
49         # Your code should be able to find the plan after exploring al
50         # I put in this stop code in case your code has a bug, so that
51         print("")
52         if len(topPlan) > 6:
53             print("Could not find plan in 8 steps")
54             return False
55
56         neighbors = []
57

```



```

58     for action in Action.all:
59
60         # We could use a theorem prover to find variable instantia
61         # the preconditions true.
62         # Instead, let's enumerae over all variable instantiations
63
64         #-----
65         # The code between the two sets of dashes iterates all var
66         # One of the questions asks you to explain how it works
67
68         vars = action.vars
69         numvars = len(vars)
70         numconstants = len(Constants)
71         cnt = int(pow(numconstants,numvars))
72         for i in range(cnt):
73             subs = {}
74             j = i
75             for v in vars:
76                 c = j % numconstants
77                 subs[v] = Constants[c]
78                 j = (j - c)//numconstants
79             #-----
80
81             head = substitute(action.head,subs)
82             pre = substitute(action.pre,subs)
83             add = substitute(action.add,subs)
84             dele = substitute(action.dele,subs)
85
86             if subset(pre,topWorld):
87                 worldcnt += 1
88                 print("%d: Found applicable action" % worldcnt)
89                 print(action)
90
91                 # Create the new world and the new plan
92                 # And add them to the neighbors
93
94                 # YOUR CODE HERE
95
96                 print("    New world:")
97                 printWorld(newWorld,"    ")
98                 print("    New plan :")
99                 printPlan(newPlan,"    ")
100                 print("")
101
102             frontier = frontier[1:] + neighbors
103
104         print("No world with goal true was found")
105
106

```

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 as `puton(a,b) -> takeoff(a,b) -> puton(a,b) -> takeoff(a,b)` would occur.

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(l2)),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)))))].
```