CS560: Homework 4

Positive and Negative Integers

Question 1: Representation

In the previous homework, positive integers were represented using the constant 0 and the function symbol 's'.

Extend the scheme represent negative numbers as well. There should be exactly one way to represent any number.

Question 2: Successor

Define the predicate *succ(X,Y)*, which should be true exactly when $Y = X + 1$. It should work for all integers, including your representation for negative integers.

Make sure that anytime this predicate is true, there is only a single derivation for it. For example, if you check your definition in Prolog with 'succ(s(s(s(0))),X)', pressing ';' should not give you another derivation.

Question 3: Plus

Define *plus(X,Y,Z)* so that it works when the first two arguments are bound. It should work for all integers. You can use *succ* in your definition.

Towers

The following questions are based on Exercise 3.15 from the textbook.

Assume that you have the following blocks:

```
block(red1)
block(red2)
block(red3)
block(red4)
block(red5)
block(red6)
block(red7)
block(gre1)
block(gre2)
block(gre3)
block(blu1)
block(blu2)
block(yel1)
block(bla1)
```

And that you know their colors:

```
color(red1,red)
color(red2,red)
color(red3,red)
```

```
color(red4,red)
color(red5,red)
color(red6,red)
color(red7,red)
color(gre1,green)
color(gre2,green)
color(gre3,green)
color(blu1,blue)
color(blu2,blue)
color(yel1,yellow)
color(bla1,black)
```

You will create a predicate called `multicolortower(Height,List)` that is true if `List` is of length `Height`, `List` is a list of blocks, no block appears more than once, and no two adjacent elements have the same color.

Question 4: Duplicate Blocks

Say the base case is a list of length 1. (A list of length 1 turns out to be more convenient here than an empty list.) Define the base case.

Now define the recursive step. Assume you want a list of length N, but know how to make a list of length N - 1.

For the first version, assume that blocks can be reused and color doesn't matter.

You can either use the 's' function to represent numbers (e.g., where s(s(0)) is 2), or you can use Prolog's numbers (e.g., where you use arabic numbers, and use the 'is' predicate for subtraction). If you are using Prolog's numbers, make sure that the recursive definitive clause for 'mct' is not executed for the base case, by including 'N ¿ 1' as part of its body. This way, Prolog will not consider negative length lists, and go into an infinite loop.

Hand in the clause definitions.

Question 5: No Duplicate Blocks

Now add in the constraint that blocks cannot be reused. To do this, define a predicate called `differentFromList(Block,List` which is true if Block is different from every block in List. (You can use Prolog's **not** in your definition, but be careful where you place it.) Alter your definition of multicolortower to use `differentFromList`. Hand in your predicate definitions.

Question 6: Adjacent Blocks with Different Colors

Now add in the constraint that adjacent blocks must be of different colors. Define `differentcolor(X,Y)` if X and Y are different colors (see 3.11 part c). Since we are doing this recursively, we just need to make sure that the block we are adding to the top is different from the next block down. Hand in your revised predicate definitions.

Reasoning Procedures

Question 7: Probablistic Top Down Reasoning Procedure

In homework 2, you created a probablistic top-down ground clause reasoning procedure. In this homework, you will create create a probablistic top-down reasoning prcoedure that works with variables and function symbols.

You must use the routines in `hw4standard`, which has correct versions of the functions that you have built up in the last two homework assignments. The version of `unify` that is included looks a lot different from your homework answer as it does the unification in a more efficient manner. Make sure you look over `hw4standard` to understand the code and how they should be called.

Below is code to start you off. You must use that code, and fill in the missing parts. Make sure you include comments to say what each line of code is doing and why.

Your code should work for arbitrary queries, which might have more than one atom in it. In your code, you need to turn the query into an answer clause, by inserting a 'yes' predicate with arguments consisting of all of the free variables in the query. This way, when you finish deriving your proof, you will have the answer to the question.

```
import random
from hw4standard import *

def prove(query,kb):
    # YOUR CODE HERE to convert query TO answer cluase

    print("Initial answer clause is %s" % prettyclause(answer))

    while len(answer) > 1:
        #give answer clause fresh variables
        answer = freshvariables(answer)

        matches = []
        for r in kb:
            #YOUR CODE HERE

        if matches == []:
            print("No rules match %s" % prettyexpr(answer[0:0]))
            return False

        # YOUR CODE HERE

    return True

def multiprove(query,kb):
    proved = False
    for n in range(100):
        if prove(query,kb):
            print("Proved on iteration %d" % n)
            return True
    print("Could not prove")
    return False


kb = [[['animal','X'],['dog','X']],
      [['gets_pleasure','X','feeding'],['animal','X']],
      [['likes','X','Y'],['animal','Y'],['owns','X','Y']],
      [['does','X','Y','Z'],['likes','X','Y'],['gets_pleasure','Y','Z']],
      [['dog','fido']],
      [['owns','mary','fido']]]

multiprove([['does','mary','X','Y']],kb)
```

To test your theorem prover, have it do the proof of Exercise 3.1. For this question, the query just happens to have a single atom.

Hand in your code for prove.

Question 8: Another KB

Encode the following as a knowledge base and a query, and make sure your probablistic theorem prover works on it. Turn in the KB and query.

```
has_tree(T,T)
has_tree(T,n(N,LT,RT)) <- has_tree(T,LT)
has_tree(T,n(N,LT,RT)) <- has_tree(T,RT)

? 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 9: Depth-First Top Down Reasoning Procedure

Now, modify your proof procedure so that it does a depth first search. In the probabilistic procedure, you took the top atom in the answer clause and found all rules that matched it, and randomly chose one. Here, you will create a **frontier** from the initial answer clause. Take the top element from the frontier, which is an answer clause. Find all rules that match, and for each one, apply it to a copy of the answer clause (so that you don't destroy the original one), and gather all of the new answer clauses into a **neighbors** list, and then put that list at the front of your frontier.

Make sure you document your code.