# CS 560: Homework 4

### Eric Stevens

### October 27, 2018

## Question 1:

My initial thought was to add a `n(0)` predicate but then we would have the problem of being able to have expressions like `s(0) == s(n(s(0)))`, resulting in the same number being represented in two different ways. Therefore, it seems that they best way to extend the scheme is to add another constant. We will add the constant `-0`, which will represent a negative zero. The `s(X)` in increment the numerical value of `X` in the direction of the sign of `X`. Examples of the way this would work follows:

```
s('0') = 1
s(s('0')) = 2
s(s(s('0'))) = 3

s('-0') = -1
s(s('-0')) = -2
s(s(s('-0'))) = -3
```

If we ensure that we dont map `-0` to `0` and instead map `s(-0)` to `-1` we will have a number system where there will be exactly one representation for each numerical value.

## Question 1 Critique:

I am very disheartened by the provided solution. I spent a long time trying to solve this problem in a way that adheared to the requirements of the quesrion, which stated explicitly state: **"there should be exactly one way to represent any number."** Neither of the provided solutions held to this, allowing any number to be produced in an infinite number of ways and then using the hand waving statement that **this will be considered not "well-formed"**.

```
p(s(0)) = p(p(s(s(0)))) = p(s(p(s(0)))) = 0
n(s(0)) = n(n(n(s(0)))) = n(n(n(n(s(0))))) = -1
etc..
```

We have not learned anything about functions that says that this can be prevented and I dont see how any implementation of these functions could pervent this. Nothing to prevent this is provided in the solutions. My solution actually does acomplish the abbility to represent each number in only one way.

This is disapointing to me because the provided solutions are the ones that were obvious to. I would have undoubtedly used one of them had it not been for the statement restricting us from schemes that have multiple representations for a given number. Furthermore, this confusion cascaded into the following problems. I hope this misunderstanding will be considered during the grading of this assignment.

# Question 2:

The language of this question confused me. There seems to be a mixing of Prolog and Datalog in the instructions: "check your definition in Prolog with ‘succ(s(s(s(0))),X)’". It is my understanding that functions are meaningless in Prolog and are treated as constants. Therefore there is no way to parse the `s()` predicates and I believe that they would have to be hard coded. As a result I am turnning in both a Datalog definition and a working simple Prolog line to cover my bases.

***Datalog***

```
% if X is greater than 0
succ(X,s(X)) :- lt(0,X)

% if X is less than 0
succ(s(X),X) :- lt(X,0)
```

***Prolog***

Here I am also adding a `predecessor` function to aid in the next question.

```
% simple addition of 1
succ(X,Y) :- Y is X+1.

% predecessor function
pred(X,Y) :- Y is X-1.
```

# Question 2 Critique:

Again, I had confusion about this problem as a result of the previous question. Also there is confusion about whether you are expecting Prolog solutions or Datalog solutions, which is also true in the next problem. I also forgot to include a base case for the problems.

# Question 3:

```
% base: X+0=X
plus(X,0,X).
% decrement Y and put onto Z
plus(X,Y,Z) :-
```

```
        succ(X,NewX),
        pred(Y,NewY), % see previous code sample
        plus(NewX,NewY,Z).

% Alternative plus function
alt_plus(X,Y,Z) :- Z is X+Y.
```

## Question 3 Critique:

As in previous cases I was confused again about what was expected. I wrote a function that works in Prolog. My mistake from the first question has cascaded down to this point. This function works in Prolog.

## Question 4:

```
% Base Case: if tower height is 1 then a list of a single block shou
ld be returned.
multicolortower4(1,[X]) :- block(X).

% Recursive: Builds on the way out.
multicolortower4(Height,[Top|Rest]) :-
        Height > 1,
        block(Top),
        NewHeight is Height-1,
        multicolortower4(NewHeight,Rest).
```

The following is a query and its ouput using the above predicates:

```
?- multicolortower4(9,List).
List = [red1, red1, red1, red1, red1, red1, red1, red1, red1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red2] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red3] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red4] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red5] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red6] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, red7] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, gre1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, gre2] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, gre3] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, blu1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, blu2] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, yel1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red1, bla1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red2, red1] ;
List = [red1, red1, red1, red1, red1, red1, red1, red2, red2] ;
...
...
...
```

## Question 4 Critique:

It appears that I am building the list at the time of the function call where the solution builds on the way in. As can be seen from the output, my function works.

## Question 5:

```
% Base Case: any block will be different in empty list
differentFromList(Block,[]) :- block(Block).
% recurse through list and ensure Block is not equal to Top
differentFromList(Block,[Top|Rest]) :-
        block(Block),
        not(Block = Top),
        differentFromList(Block,Rest).

% mct5: This function adds Height blocks to list and
% ensures that the blocks that are added are not
% already in the list. This builds the list on the
% way in to minimize the search space.

% Base Case: 0 hight will return Out with List
mct5(0,List,Out) :-
        Out = List.
% Build on way in in at the recursive call
```

```
mct5(Height,List,Out) :-
        Height > 0,
        Height < 14, % only 13 blocks
        block(Top),
        differentFromList(Top,List), % distinct
        NewHeight is Height-1,
        mct5(NewHeight,[Top|List],Out). % build


% multicolortower5(Height,List): This function formats the operation
% in the way described in the assignment. It allows user to ignore
% extra input variable and ensures the desired result, as mct5 can
% have results that are not what is required if used directly.
multicolortower5(Height,List) :-
        mct5(Height,_,List).
```

The following is a query and its ouput using the above predicates:

```
?- multicolortower5(9,List).
List = [gre2, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre3, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu1, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu2, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [yel1, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [bla1, gre1, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre3, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu2, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [yel1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [bla1, gre2, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre1, gre3, red7, red6, red5, red4, red3, red2, red1] ;
List = [gre2, gre3, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu1, gre3, red7, red6, red5, red4, red3, red2, red1] ;
List = [blu2, gre3, red7, red6, red5, red4, red3, red2, red1] ;
...
...
...
```

## Question 5 Critique:

When looking at the notes I saw that building lists on the way in involved a storage variable. I wrote the function after looking at notes that set an extra variable after the recursive call. In the case of this function the setting of the ouput variable is done in the base case. Otherwise, the function seems to be correct as the output is correct.

# Question 6:

```
% differentcolor(): Simply checks to ensure that two blocks
% are not the same color.
differentcolor(X,Y) :-
        block(X),
        block(Y),
        color(X,CX),
        color(Y,CY),
        not(CX=CY).


% performs way in list building while checking to ensure that
% any two touching blocks do not have the same color.
mct6(1,List,Out) :-
        Out = List.
mct6(Height,[Tl|List],Out) :- % needs top of list to check for color
        Height > 1,
        Height < 14,
        block(Top),
        differentFromList(Top,List),
        differentcolor(Top,Tl), % ensure color difference
        NewHeight is Height-1,
        mct6(NewHeight,[Top|[Tl|List]],Out).

% same purpose as in question 5
multicolortower6(Height,List) :-
        mct6(Height,_,List).
```

The following is a query and its ouput using the above predicates:

```
?- multicolortower6(9,List).
List = [blu2, red4, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red4, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red4, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [blu2, red5, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red5, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red5, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [blu2, red6, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red6, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red6, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [blu2, red7, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [yel1, red7, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [bla1, red7, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [red4, yel1, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [red5, yel1, blu1, red3, gre3, red2, gre2, red1, gre1] ;
List = [red6, yel1, blu1, red3, gre3, red2, gre2, red1, gre1] ;
...
...
...
```

## Question 6 Critique:

When looking at the notes I saw that building lists on the way in involved a storage variable. I wrote the function after looking at notes that set an extra variable after the recursive call. In the case of this function the setting of the ouput variable is done in the base case. Otherwise, the function seems to be correct as the output is correct.

## Question 7:

In [8]:
```
# Question 7

import random
from hw4standard import *

# time seed random variable
from datetime import datetime
random.seed(datetime.now())

def prove(query,kb):

    #*************************************************************#
    answer = query.copy() # Make a copy of query so query is not altered.
    answer.insert(0,findvariables(query,[])) # add variables from query
    answer[0].insert(0,'yes') # add 'yes'
    #*************************************************************#

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

    while len(answer) > 1:
        #give answer clause fresh variables
        answer = freshvariables(answer)
        matches = []
        for r in kb:
            #unify right most atom for scripting ease
            if unify(answer[-1],r[0],{}):
                matches.append(r)


        if matches == []:
            print("No rules match %s" % prettyexpr(answer[-1]))
            return False

        #*************************************************************;
        print(prettyclause(answer)) # output formatting

        # Unification
        a = answer[-1] # right-most atom
        b = matches[random.randint(0,len(matches)-1)] # random match
        sub = {} # get subs
        unify(a,b[0],sub)

        # Substitution
        resolution = substitute(b,sub) # sub for match to be used in resolu
        answer = substitute(answer,sub) # sub current answer clause

        # output formatting
        print("Resolve with: ",prettyclause(b))
        print("Substitution: " ,sub, "\n\n")

        # Replace
        answer = answer[:-1] # cut right-most atom
        for a in resolution[1:]: answer.append(a) # add resolution body
        #*************************************************************;


    print(prettyclause(answer))
```

```
    return True

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

In [ ]:
```
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']]]

query=[['does','mary','X','Y']]

multiprove(query,kb)
```

## Question 8:

In [ ]:
```
kb2 = [
    [['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']]
    ]


query2 = [['has_tree',['n','X',['e', 'e4'],'Y'],['n','n1',['n','n2',['e', 'e
        ['n','n4',['e', 'e4'],['e', 'e5']]]]]]

# call
multiprove(query2,kb2)

# selecting submission output of smaller number of iterations.
# somtimes it took 80 iterations to resolve, other times it
# happened on the first iteration.
```

## Question 9:

**This code does not work.** Below are two different attempts that I made to solve this problem. The first tries to call a function recursivly and hold the frontier in the recursion. The second tries to maintain the frontier in a list in the the for loop of the program. I have not had success. These functions are both capable of solving the first query as the output demonstrates, but are unable to solve the second query. Something is wrong with the way they move from child back up to parent nodes. They either get stuck in an infinite loop or terminate without completing the proof. I ran out of time to solve the problem.

In [54]:
```
def depth_first(answerclause,kb):

    answer = answerclause.copy()


    # fresh
    answer = freshvariables(answer)


    # get list of matches for right most pred
    matches = []
    for r in kb:
        if unify(answer[-1],r[0],{}):
            matches.append(r)

    if matches == []: return False
    for m in matches:
        print(prettyclause(answer))
        a = answer[-1]
        b = m
        sub = {}
        unify(a,b[0],sub)

        resolution = substitute(b,sub)
        answer = substitute(answer,sub)
        #print("matches: ", matches[0])
        print("Resolve with: ",prettyclause(b))
        print("Substitution: " ,sub, "\n\n")


        answer = answer[:-1]

        if len(resolution) > 0:
            for z in resolution[1:]: answer.append(z)
        else:
            answer.append(resolution)

        if len(answer)==1:

            print(prettyclause(answer))
            return True
        if depth_first(answer,kb): return True

    return False

def dfr(query,kb):
    print("QUERY: ", prettyclause(query),"\n\n")
    # add yes to answer clause
    answer = query.copy()
    answer.insert(0,findvariables(query,[])) # add variables from query
    answer[0].insert(0,'yes') # add 'yes'
    return depth_first(answer,kb)
```

In [ ]:
```
dfr(query,kb)
```

In [56]:
```
# Question 7

import random
from hw4standard import *

# time seed random variable
from datetime import datetime
random.seed(datetime.now())

def deep(query,kb,front):

    #*************************************************************#
    answer = query.copy() # Make a copy of query so query is not altered.
    answer.insert(0,findvariables(query,[])) # add variables from query
    answer[0].insert(0,'yes') # add 'yes'
    #*************************************************************#

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

    while len(answer) > 1:
        #give answer clause fresh variables
        answer = freshvariables(answer)
        matches = []
        for r in kb:
            #unify right most atom for scripting ease
            if unify(answer[-1],r[0],{}):
                matches.append(r)
                front.append(r)


        if matches == []:
            if front == []:
                print("No rules match %s" % prettyexpr(answer[0]))
                return False
            else:
                front.pop()
                deep(front[-1],kb,front)
        #*************************************************************;
        print(prettyclause(answer)) # output formatting

        # Unification
        a = answer[-1] # right-most atom
        b = front.pop() # random match
        sub = {} # get subs
        unify(a,b[0],sub)

        # Substitution
        resolution = substitute(b,sub) # sub for match to be used in resolu
        answer = substitute(answer,sub) # sub current answer clause

        # output formatting
        print("Resolve with: ",prettyclause(b))
        print("Substitution: " ,sub, "\n\n")

        # Replace
        answer = answer[:-1] # cut right-most atom
        for a in resolution[1:]: answer.append(a) # add resolution body
```

```
        #**************************************************************;


    print(prettyclause(answer))
    return True
```

In [ ]:
```
deep(query,kb,[])
```

## Question 9 Critique:

My second attempt seems to be closer to what was intended so I will discuss that attempt. The idea was to append the frontier with any match that is found in the process of colecting matches. After each matching round, the final match is popped of the list and unified. In other words, I am treating frontier as a stack in which matches are pushed onto and poped off of. The idea was that treating the matches as a stack and poping the top of the stack to resolve would result in a depth first search. For some reason I could not get it to resolve clauses that faild on some braches. The case above results in an infinite loop with a building frontier. Other options I tried resulted in termination at the bottom of a failed branch.