

# CSE560: Homework 4

## Question 1

**Part A:** 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 only one way to represent any number.

**Part B:** Define the predicate *succ*(*X*,*Y*), which should be true exactly when  $Y = X + 1$ . 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.

**Part C:** Define *plus*(*X*,*Y*,*Z*) so that it works when the first two arguments are bound. It should work for both positive and negative numbers. You can use *succ* in your definition.

## Question 2: 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 are asked to make a predicate, let's call it `multicolortower(Height,List)` which 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.

**Part A:** 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. Give the clause definitions.

**Part B:** 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.

**Part C:** 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.

### Question 3: Probabilistic Top Down Reasoning Procedure

In `hw4standard.tcl`, there are correct versions of the procedures 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 much more efficient manner. Now, take your code from homework two, which was for a probabilistic ground clause reasoning procedure, and make it work for the case with variables.

Make sure you include comments to say what each line of code is doing and why.

Note that you will need to use the `freshvariables`, `unify`, and `substitute` routines.

Use it to do the proof of Exercise 3.1.

Here is some code to start you off.

Note that your code should work for arbitrary queries, which might have more than one atom in it. For this question, the query just happens to have a single atom.

```
source hw4standard.tcl

set 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}}}

proc prove {query} {
    # YOUR CODE HERE to convert query TO answer clause

    puts "Initial answer clause is [prettyclause $answer]"

    while {[llength $answer] > 1} {
        #give answer clause fresh variables
    }
}
```

```

    set answer [freshvariables $answer]

    set rules {}
    foreach r $::kb {
        #YOUR CODE HERE
    }

    if {$rules == {}} {
        puts "No rules matched [prettyexpr [lindex $answer 0]]"
        return 0
    }

    # YOUR CODE HERE
}
return 1
}

proc multiprove {query} {
    set n 0
    while {$n < 100} {
        set ret [prove $query]
        if {$ret == 1} {
            puts "Proved on iteration $n"
            return 1
        }
        incr n
    }
    puts "Could not prove"
    return 0
}

multiprove {{does mary X Y}}

```

Note that to turn the query into an answer clause, you need to insert a ‘yes’ predicate with arguments consisting of all of the free variables in the query. This way, when you get finish deriving your proof, you will have the answer to the question.

In the class notes and in homework questions, we have been giving the clause from the KB the fresh variables. Here we are giving the *answer clause* fresh variables. Is this sufficient?

## Question 4: 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 find all rules that match, for each one, apply it to a different instance of the answer clause, and put all of the new answer clauses at the front of your frontier.

Make sure you document your code.

```

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(14),Y),n(n1,n(n2,l(11),l(12)),n(n3,l(13),n(n4,l(14),l(15))))))

```