

CSE560: Homework 3

Question 1: Exercise 3.1 from the textbook

To get you on track, think of the sentence “dogs are animals” as being “if something is a dog, then it is an animal.”

Question 2: Exercise 3.5 a & b from textbook

Question 3: Exercise 3.6 from textbook

For 3.6 c), the subsequence does not have to be consecutive. So `subsequence([a,c],[a,b,c])` should be true.

Just use the simple list notation of head and tail.

For those of you interested in Prolog, you can use it to verify your answers.

Question 4

For this question, we will be using the number representation from class, in which the constant symbol `0` maps to the number 0, `s(0)` maps to 1, `s(s(0))` maps to 2, etc. In class, we defined `subtract` and `lt` to use this representation. For your convenience, this is repeated here.

```
lt(X,s(X)).  
lt(X,s(Y)) <-- lt(X,Y).
```

```
subtract(X,0,X).  
subtract(X,s(Y),Z) <-- s(X,Y,s(Z)).
```

`lt(X,Y)` is true if $X < Y$ and `subtract(X,Y,Z)` is true if $X - Y = Z$.

Part A: Define `plus`. `plus(X,Y,Z)` should be true if $X + Y = Z$.

Note that your predicate should only return a single solution. In Prolog, if you press “;” after an answer, it should not give another one. (You can avoid this by not defining redundant base cases.)

Part B: Define `times` and `neq`. `times(X,Y,Z)` is true if $X \times Y = Z$ and `neq(X,Y)` is true if $X \neq Y$. Note that you can use `plus`, `subtract` and `lt` in defining them.

Part C: Define ‘`is(X,Expr)`’. Assume that expression is in prefix format, and is some combination of `plus(_,_)`, `times(_,_)`, and `subtract(_,_)`.

Note that these 2-ary `plus`, `times` and `subtract` are different from the 3-ary ones before. The 3-ary ones are predicate symbols. The 2-ary ones are function symbols. In datalog, you do not directly define functions. Instead you define them in terms of the predicates that use them, which in this case is the `is` predicate.

So, `is(X,plus(s(s(0)),times(subtract(s(s(s(0))),s(0)),s(s(1)))))` is valid.

Question 5: Exercise 3.11 from textbook

For part a and b, I am looking for nice short answers.

For part c, I want the clause definitions. Do not use ‘not’. *Hint:* Use the answer from the previous question for representing `neq`.

Question 6: Occurs Check

Make a procedure called `occurscheck` that takes a variable and a term as input. It should succeed (return 1) if the variable does not appear in the expression, and should fail otherwise (return 0).

Note that your procedure needs to work whether the term is a variable, constant or an arbitrarily deep function. Here is some code to start you off. Do this without recursion (without having `occurscheck` call itself).

```
proc occurscheck {var term} {
    while {$term != {}} {
        # note that Tcl really doesn't distinguish between
        # lists of length 1 and simple tokens
        set first [lindex $term 0]
        set rest [lrange $term 1 end]

        #YOUR CODE HERE
        # note that as soon as you find the var in the term,
        # you can return the value 0 right away
    }

    # we haven't seen the var in the term, so return 1
    return 1
}
```

Hand in the code and show that it works properly on a few examples.

Question 7: Programming: Unification for Datalog with Functions

In class, we discussed the unification algorithm. `Unify` should take two atoms and return 0 if they cannot be unified, or return the most general unifier if they can. In this question, you will program the unification algorithm.

As with the previous homework, we will represent the substitution set as a Tcl list of the form $V_1T_1V_2T_2\dots$. The `unify` procedure will return a list, as Tcl cannot return an associative array as the value of a procedure. The `unify` procedure will work for any pair of atoms, and arbitrary nesting of functions inside of the atoms.

In your code, the most important issue is that the code is simple and easy to understand. Do not worry too much about the efficiency of the code.

In the class webpage, right after the link for this homework, there is a link for some code for you to use in this homework. It includes code for the `substitute` procedure, which takes as input an atom or a term. Note that it will allow illegal Datalog sentences (variables used as the predicate or function symbol), but as long as you give it legal Datalog sentences, it will work correctly. There are also a few other procedures to make your life easier. `test` will call your `unify` and nicely format the result of the unification.

In your Tcl program, add the following line at the beginning of your code to automatically include the code from `hw3standard.tcl` using the `tcl source` command. You do not need to hand in `hw3standard.tcl`. You will also use your `occurscheck` procedure from the previous question.

To help you out further, I am giving you the following piece of code. This code takes two atoms and looks for the *first* difference between them. If there is no difference, it returns a 1. Otherwise, if the first difference can be resolved by a substitution, it returns the substitution (as a list with the variable followed by its value). Otherwise, the procedure returns 0. Note that it does not do the occurs check.

```
proc diff {a b} {
    if {$a == $b} {return 1}
    if {[isVar $a]} {return [list $a $b]}
```

```

if {[isVar $b]} {return [list $b $a]}
if {[llength $a] == 1 || [llength $b] == 1} {return 0}
if {[llength $a] != [llength $b]} {return 0}

foreach parta $a partb $b {
    set result [diff $parta $partb]
    if {$result == 0} {
        return 0
    }
    if {$result != 1} {
        return $result
    }
}
return 1
}

```

With the above `diff` procedure, the code for `unify` is drastically simplified. The `unify` will use `diff` to determine all of the substitutions necessary in order to unify the two atoms. `unify` will work as follows. It will repeatedly call `diff`. For each difference that is found, `unify` will make sure that the new substitution does not violate the occurs check. It will then apply the substitution to both atoms, and to the substitutions already found.

Hand in your code and show that it works for the Exercise 2.10. Make sure you use `test` to format your results.

Also, comment on why this is a very inefficient way of writing the `unify` procedure.

Question 8: Renaming

To use the `unify` procedure as part of a top-down proof procedure, we need a way to rename the variables in an expression, so we don't get any variable clashes when unifying.

Let's allow a variable to also start with “_”. We will reserve these for system use. When we rename a variable, we will start it with “_” followed by a unique number. Let the global variable `::nextvariable` keep track of the next unique number to use. Each time you need a new variable, you will use `_${::nextvariable}` as the variable, and then you will increment `::nextvariable`.

Make a procedure `freshvariables` that first uses `findvariables` (in `hw3standard.tcl`) to find all of the variables in the expression. It then creates a substitution list that will map the found variables to new variables that begin with “_” followed by a unique number. It then uses `substitute` to apply the substitutions. It should take as input the original expression and return a new expression.

Hand in a copy of your code, and show its results on several examples.