Spring, 1992 Name: _____

CS 342 — Principles of Programming Languages

Prolog Test (Problem Solving, Version 3)

Before you begin, please take a moment to look over the entire test so that you can budget your time.
The usual conventions from the Course Specification for problem solving tests apply.
A special rule is that *your programs only have to be able to answer queries of the form given in the examples for each problem.* If there are no queries that tell you that you have to be able to run the program "backwards," then you don't have to be able to run it backwards.
In your solutions you may use the following facts and rules from the textbook without further mention. Any other facts or rules that you wish to use must be written in your solution. If you wish to use facts or rules from your solutions in other parts of the exam, you must clearly state where we can find those. Remember that you can define "helping" predicates if you wish. You may also use simplify to convert from difference lists and back again unless the problem specifically prohibits that.

```
(infer (member X (cons X L)))
(infer (member X (cons Y M)) from (member X M))
(infer (addtoend nil X (cons X nil)))
(infer (addtoend (cons Y L) X (cons Y M)) from (addtoend L X M))
(infer (reverse nil nil))
(infer (reverse (cons X L) M) from (reverse L N) (addtoend N X M))
(infer (append nil L L))
(infer (append (cons X L) M (cons X N)) from (append L M N))
(infer (member X L) from (append L1 (cons X L2) L)).
(infer (lookup K A L) from (member (pair K A) L))
(infer (mult 0 Y 0))
(infer (mult X Y Z) from (minus X 1 V) (mult V Y W) (plus W Y Z))
(infer (fac 0 1))
(infer (fac N R) from (minus N 1 N1) (fac N1 R1) (mult R1 N R))
(infer (naive-sort L M) from (permutation L M) (ordered M))
(infer (<= X X))
(infer (<= X Y) from (less X Y))
(infer (ordered nil))
(infer (ordered (cons A nil)))
(infer (ordered (cons A (cons B L))) from (<= A B) (ordered (cons B L)))
(infer (permutation nil nil))
(infer (permutation L (cons H T))
      from (append V (cons H U) L) (append V U W) (permutation W T))
(infer (partition H (cons A X) (cons A Y) Z) from (<= A H) (partition H X Y Z))
(infer (partition H (cons A X) Y (cons A Z)) from (less H A) (partition H X Y Z))
(infer (partition H nil nil nil))
(infer (quicksort nil nil))
(infer (quicksort (cons H T) S)
      from
      (partition H T A B)
             (quicksort A A1)
             (quicksort B B1)
             (append A1 (cons H B1) S))
(infer (simplify (diff X X) nil))
(infer (simplify (diff (cons X Y) Z) (cons X W)) from (simplify (diff Y Z) W))
(infer (diffappend (diff L X) (diff X Y) (diff L Y))).
```

1. (5 points) Write a predicate **equal** that is satisfied if and only if its first argument can be unified with its second. For example,

```
-> (infer? (equal 3 4))
Not satisfied

-> (infer? (equal (study hard) (study hard)))
Satisfied

-> (infer? (equal (foo X) (foo 2)))
Satisfied

-> (infer? (equal (foo a) (foo b)))
Not satisfied
```

2. (10 points) Write a predicate `remove-every-other`, such that `(remove-every-other L1 L2)`, holds if the list `L2` has second, fourth, sixth, etc. elements of the list `L1` in that order. For example:

```
-> (infer? (remove-every-other (cons 1 (cons 2 (cons 3 (cons 4 nil)))) Lst)
           (print Lst))
(cons 2 (cons 4 nil))
Satisfied

-> (infer? (remove-every-other (cons 1 (cons 2 (cons 3 nil))) X) (print X))
(cons 2 nil)
Satisfied

-> (infer? (remove-every-other nil nil))
Satisfied
```

3. (15 points) Write a predicate `adding` such that `(adding X L2 Y)` holds if there is some element of the list `L2`, say z, such that `X` + z equals `Y`. For this predicate, you may assume that *all* arguments are defined. For example,

```
-> (infer? (adding 3 (cons 1 (cons 2 nil)) 5))
Satisfied

-> (infer? (adding 3 (cons 1 (cons 2 nil)) 6))
Not Satisfied

-> (infer? (adding 3 nil 6))
Not Satisfied
```