Name: ____________________________________ **50 pts. possible**

CS 3500 – Programming Languages & Translators
Spring 2019
Exam #3

Due by noon on Tuesday, May 14, 2019

You must turn in a paper copy of your exam to the CS Office (325 CS)

Do ***NOT put it under Dr. Leopold's office door (or any place other than the CS Office) or it will ***NOT*** be graded!!!***

This exam is open-book. You may use your notes from the course, any textbook, and/or the internet. You may not communicate with any person about any aspect of the exam until after the due date. If there is any question about what resources may or may not be used during the exam, or if you have any questions about any of the problems, please contact Dr. Leopold (leopoldj@mst.edu), NOT the graders. Note that you can only ask for clarification on what a problem is asking for; no hints or help solving the problems will be given!

Name: ___________________________________ 50 pts. possible

For problems 1 through 9 circle the **single best** answer. Each of these questions is worth **1 point**.

1. Given below is some code written in a programming language that we did not study this semester. It evaluates to 10. Semantically, this code would be most equivalent to ____________.

   (\fluffy -> 1 + fluffy) 9

   a. If (!x) x + 1 else 9
   b. f(Fluffy, Result) :- \+ Fluffy, Result is 1 + Fluffy. f(_, 9).
   [✓ c. funcall #'(lambda (x) (1 + x)) 9)]
   d. All of the above

2. In the Prolog relation given below, the first argument is a list and the second argument is intended to be the last element of the list (e.g., atTheEnd([1,2,3], 3)). In order for this to work, the **underlined** portion in the second rule should be:

   atTheEnd([X], X).
   atTheEnd(_______________) :- atTheEnd(L, X).

   a. [ ], X
   b. [L], X
   c. L + 1, X
   [✓ d. [_ | L], X]
   e. None of the above is correct

3. The Prolog expression ***f(X, Y) :- g(X, Y).*** is semantically equivalent to saying:

   a. there exists some *X* and *Y* (where *X* and *Y* must be different values) such that *f(X, Y)* and *g(X, Y)* are both true
   [✓ b. if *g(X, Y)* is true, then *f(X, Y)* is true]
   c. if *f(X, Y)* is true, then *g(X, Y)* is true
   d. all of the above

4. Which of these Prolog list expressions is **NOT** equivalent to **[A, B, C, D]**?

   a. [A, B, C | [D]]
   b. [A | [B, C, D]]
   [✓ c. [[A, B, C, D] | [ ]]]
   d. [A, B | [C, D]]
   e. None of the above (i.e., all of the above are the same list)

5. In the Prolog program shown below, what kind of **cut** is being used?

```
d(3).
d(1).
b(1).
b(2).
b(3).
c(1).
c(2).
a(X) :- b(X), c(X), !, d(X).
```

[✓ a. green cut]
b. red cut

6. The Python programming language supports *lambda* expressions. Even though you may not know Python, you should be able to figure out that the Python function given below is semantically equivalent to which of the **Lisp** expressions listed below:

```
def foo(n) : return lambda a : a + n    # This is a python function
```

a. (defun foo (a n) (+ a n))
[✓ b. (defun foo (n) #'(lambda (a) (+ a n)))]
c. (defun foo ( ) #'(lambda (a n) (+ a n)))
d. (funcall #'(lambda (n) (+ a n)) a)
e. None of the above

7. The way that you implemented the **Mini R** interpreter in HW #5 utilized _____ evaluation.

a. lazy
[✓ b. eager]

8. Which of the **Lisp** expressions given below will accomplish the same thing as the following **Prolog** code:

```
printList([]).
printList([H|T]) :- print(H), nl, printList(T).
printList([1, 2, 3]).
```

a. (apply #'print ‘(1 2 3))
b. (funcall #'print ‘(1 2 3))
[✓ c. (mapcar #'print ‘(1 2 3))]
d. (print ‘(1 2 3))
e. All of the above

Name: ____ 50 pts. possible

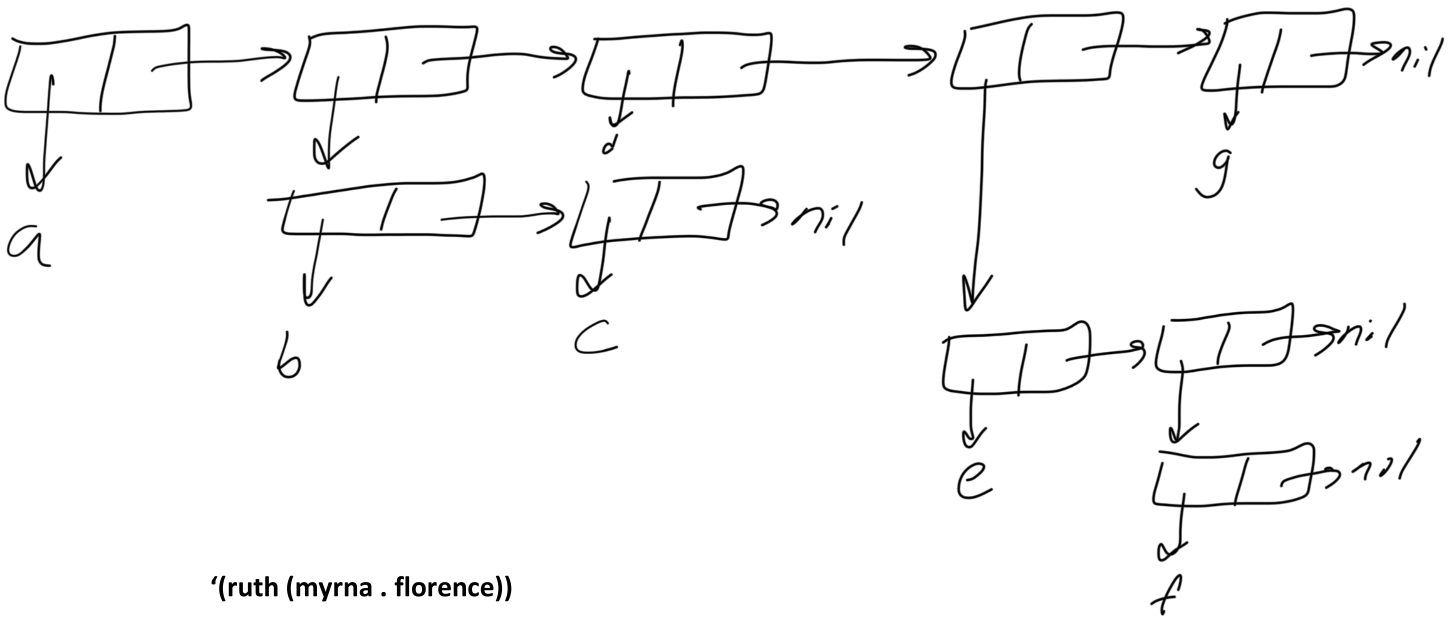
9. Suppose that the code listed below is loaded into **Prolog**, and the following commands are executed (in this order): **goat. wolf. cabbage.** If we then execute the query ***passenger(X).***,then ***X*** will be ___ .

```
:- dynamic passenger/1.

passenger(w).

goat :- passenger(g), retract(passenger(g)), assert(passenger(c)).

wolf :- passenger(w), retract(passenger(w)), assert(passenger(g)).

cabbage :- passenger(c), retract(passenger(c)), assert(passenger(w)).
```

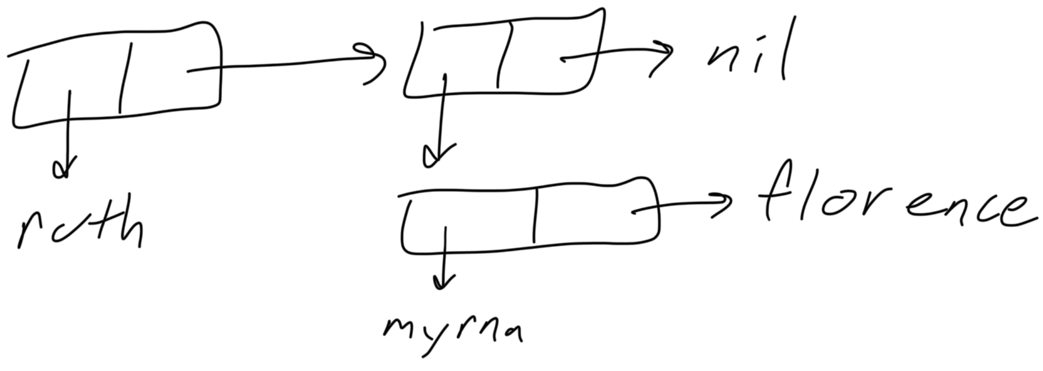
[✓ a. w]
b. c
c. g
d. none of the above

10. Draw a **Lisp box-and-pointer diagram** for each expression given below. **(7 pts.)**

'(a (b c) d (e (f)) g)



'(ruth (myrna . florence))



Name: ________________________________ **50 pts. possible**

11. Consider the **Prolog** program given below.

```
% Dave has 5200 frequent flier miles, etc.
frequentFlierMiles(dave,5200).
frequentFlierMiles(susan,2500).

% Susan works for the airline, etc.
worksForAirline(susan).
worksForAirline(bob).

% There is a 260 mile flight from St. Louis to Chicago, etc.
flight(stLouis,chicago,260).
flight(chicago,losAngeles,1750).
flight(stLouis,denver,786).
flight(denver,losAngeles,843).
flight(losAngeles,honolulu,2553).

% Chicago is an airport hub, etc.
isHub(chicago).
isHub(losAngeles).

% You can fly from X to Y and it will take Miles miles if…
canFly(X,Y,Miles) :- flight(X,Y,Miles).
canFly(X,Y,Miles) :- flight(X,Z,Miles1), canFly(Z,Y,Miles2), Miles is Miles1 + Miles2.

% Who can fly for free from From to To if…
canFlyForFree(From,To,Who) :-
  canFly(From,To,Miles),
  isHub(From),
  frequentFlierMiles(Who,Credits),
  Credits >= Miles.
```

Explain **IN DETAIL!!!** how Prolog would prove ***canFlyForFree(chicago, honolulu, dave).*** using backward chaining. **(4 pts.)**

Name: ________________________________ 50 pts. possible

12. Write a **Lisp** function named ***completelyConnectedGraph*** which will be given **2 parameters:** (1) a positive integer n representing **the number of vertices** in an undirected graph, and (2) an **undirected graph as a list of edges** where each edge is itself a list of two integers representing two vertices where each vertex has a value 1..n inclusive (e.g., (2 3) represents an undirected edge from vertex 2 to vertex 3)). Assume there are no edges where the two vertices are the same value (e.g., (2 2)). You cannot make any assumptions about the order of the vertices listed for an edge; for example, an edge between vertices 2 and 3 may be represented as either (2 3) or (3 2). Your function should return **t** if the graph is **completely connected** (i.e., each vertex is connected to every other vertex); otherwise, your function should return **nil**.

    Your solution can include other user-defined functions besides one named ***completelyConnectedGraph***.

    You can **NOT use loops** anywhere in your solution for this problem; you must solve the problem with **recursive** functions! **(10 pts.)**

    (setf g1 '((1 2) (2 4) (4 3) (3 1) (1 4) (3 2)))
    ; This graph is completely connected. You'd test this by calling:
    ; (completelyConnectedGraph 4 g1)
    ; and should get back t

    (setf g2 '((1 2) (3 4) (4 1) (2 3)))
    ; This graph is NOT completely connected You'd test this by calling:
    ; (completelyConnectedGraph 4 g2)
    ; and should get back nil

    Include a listing of your source code AS WELL AS a screenshot showing your code being loaded into gcl and being run for each of the two graphs given above.

Name: ________________________________ **50 pts. possible**

13. Write a **Prolog** relation named ***completelyConnectedGraph*** which will solve exactly the same situation as was described in the problem 12. You can write additional relations (besides one named ***completelyConnectedGraph***) as necessary. **(10 pts.)**

```
% Assume each undirected graph is specified as a list of edges
% Assume each edge is specified as a list of 2 vertices: [vertex, vertex]

% This graph is completely connected. Your relation should return true (or something
% other than false).
g1([[1,2], [2,4], [4,3], [3,1], [1,4], [3,2]]).

% This graph is NOT completely connected. Your relation should return false.
g2([[1,2], [3,4], [4,1], [2,3]]).
```

Include a listing of your source code <u>AS WELL AS</u> a screenshot showing your code being loaded into prolog and being run for each of the two graphs given above.

Name: ___________________________________ **50 pts. possible**

14. Write a **bash** script that will do **all** of the following:

    (1) For every file in the current working directory (i.e., the same directory from which this bash script is being executed) that has the extension **.cpp**:

    a) Add ***#include <stdlib.h>*** to the list of *#include*'s in the file; you can assume that each cpp file already has at least one *#include* in it. Do this step even if this particular *#include* is already in the file.

    b) Change ***int main()*** to ***int main(int argc, char** argv)***.

    c) Find the first occurrence of ***cin >> n;*** after ***int main*** and comment out that line. You can assume there is such a statement and that it is on a line by itself. However, you don't know how many lines after ***int main*** it is in the file. Do NOT change any other occurrences of ***cin >> n;*** that may be in the file!

    d) Right before (or after) the ***cin >> n;*** line that you commented out, add the following line: **n = atoi(argv[1]);**

    (2) For every file modified in step (1) do the following:

    a) Compile it using g++.

    b) If there are any compilation errors (or warnings), output just the name of that file to a file named **problems.txt**, which you will create (also in the current working directory). This file should only contain the names of files that had problems, one file name per line. If a file compiled with no errors, you should do nothing with it.

    Give a listing of your **bash** code here. **(10 pts.)**