Name and Section: Nick Palacio, CSCI-8000

Instructor: Dr. Gelfond

1. Below are my answers to question 1, parts A and B:
    a. The 3 fundamental programming paradigms are **declarative**, **functional** and **imperative**.
        i. The **declarative** programming paradigm can be oversimplified by saying that it has you talk about *what* something is. This paradigm can be viewed in the same light as **Post Grammars**. Post Grammars (or context free grammars) are a computational model based around string manipulation and substitution where you have terminal symbols, non-terminal symbols (variables), an initial start symbol and a set of production rules. A production rule is where you say a symbol can be substituted using another symbol, either terminal or non-terminal. A production rule must be of the form X --> Y where X is 1 or many symbols (terminal or non-terminal) and Y is 0 or many symbols (terminal or non-terminal). An algorithm in the Post Grammar computational model would be seen as the sets of terminal symbols, non-terminal symbols, production rules and the initial start symbol together to compute something. It does not look like a set of steps to perform in a particular order like you might see in an imperative algorithm but instead a set of declarations where you define each set described above in order to create an algorithm for whatever you are trying to compute. Both the declarative paradigm and Post Grammars concern themselves with talking about *what* something is, not necessarily *how* to compute anything. The production rules of a Post Grammar can be viewed as declaring what a symbol is, or can be substituted with.
        ii. The **functional** programming paradigm is based on defining functions that can be composed together to compute something. This paradigm can be viewed in the same light as **Lambda Calculus**. Lambda calculus is another computational model based on the idea of defining functions that are abstracted to use variables that can be substituted to make the function very reusable. An algorithm in Lambda calculus would be a set of functions that are composed together in order to compute something. Recursion is heavily used when implementing functions in this computational model. Typically, the output of one function would be used as the input of another in order to compose them together. Lambda calculus is the idea that the functional programming paradigm is based on. In both you define functions with inputs and outputs that are composed together in order to compute something. The variables are not bounded until you execute an algorithm with some sort of starting input.
        iii. The **imperative** programming paradigm can be oversimplified by saying that it has you describe *how* to compute something. This paradigm can be viewed in the same light as the **Turing Machine**. The Turing machine is another computational model that can be seen as a finite state machine where variables are written to and read from and computations are performed and results stored. You move from one state to another in this model by executing an instruction. An instruction might be writing a new value to a variable or

performing a math operation. This model is based heavily on executing a sequence of instructions in a particular order, as defined by the algorithm. An algorithm in this model would look like a lot of assignment statements and mathematical computations that are executed in a specific order. Both the imperative paradigm and the idea of the Turing Machine would have you implement an algorithm as a set of variable assignments and computations that are executed in an order. Both deal very heavily with HOW to compute something. This can be contrasted with the declarative paradigm which deals with describing *what* something is.

b. Below I will describe the ideas behind the 3 languages we studied in class:
    i. The **Prolog** programming language was based on the declarative programming paradigm. When you are defining facts and relations in prolog you are declaring *what* something is. You are not talking about *how* to compute anything, Prolog abstracts this away from you, for the most part. One of the key ideas behind prolog is a Horn clause. A horn clause is a disjunction of literals with at most one positive literal. Horn clauses can be rewritten as implications which is the part of Horn clauses that is evident in Prolog. Prolog facts and relations are implications that resemble Horn clauses. Prolog relations can be interpreted in a simplified form as 'if these things are true then this thing is true' which is exactly what an implication in logic is.
    ii. **Lisp** and a dialect called **Scheme** are based on the functional programming paradigm which is based on Lambda Calculus. It is clear to see this when you look at the fact that (almost) everything in Scheme is a pure function definition or invocation. You define functions with inputs and outputs and compose them together to compute whatever you are trying to compute.
    iii. **Go** is based on the imperative programming paradigm which is based on the Turing machine. In Go you write sequences of statements that are executed in a particular order that you also define. The assignment statement is one of the fundamental ideas of this language/programming paradigm where you read from and write to memory locations (variables) that can be viewed as the tape used when talking about a Turing machine.

2. Below is my Prolog code for question 2:

```
equal(X, X).
different(X,Y):- \+ equal(X,Y).

slice(_, M, N, []):-
    equal(M, N).

slice([_|T], M, N, S):-
    M > 0,
    slice(T, M - 1, N - 1, S).

slice(_, _, N, []):-
    N < 0.

slice([H|T], M, N, [H|T1]):-
    M =< 0,
    N >= 0,
    slice(T, M - 1, N - 1, T1).
```

3. Below is my Prolog code with comments for question 3:

```
% Helper relations
equal(X, X).
different(X,Y):- \+ equal(X,Y).
member(X, [X|_]).
member(X, [Y|Tail]) :-
    different(X,Y),
    member(X, Tail).

% Not sure if I even need these, the edge relation pretty much handles
everything
node(x).
node(y).
node(z).
node(a).
node(b).
node(c).
node(d).
node(e).

edge(x,y).
edge(y,z).
edge(z,a).
edge(a,b).
edge(e,x).

% Cycle x -> c -> d -> x
edge(x,c).
edge(c,d).
edge(d,x).

path(X, Y):-
    edge(X, Y).

path(X, Y):-
    edge(Z, Y),
    path(X, Z).

% I got some help online with this one so I am going to talk through what it
% is doing and why it works. In my initial attempts at solving this problem I
% started out with:
% cyclic(X):-
%   path(X,X).
% This made sense to me, there is a cycle from X to X if there exists a path
% from X to X in the directed graph. The issue that this ran into was that it
% would get caught in an infinite % % loop looking for a path through nodes
% that it has already actually visited because there were other cycles in the
% graph. This made me think that I was going to have to keep track of nodes
% that I visited so that I would not visit the same one twice. My first
% relation here is cyclic(X) which is from the problem definition. From here
%   I will dispatch the work to another relation cycle(X,VisitedNodes) which
% keeps track of what nodes we have gone through so far so as to enforce that
% we do not visit the same node twice. If we ever find that our current node
% already exists in the list of visited nodes then we know that we have found
```

```
% a cycle. HOWEVER, there is a small bug in this implementation that I will
% point out. This relation will return true if you give it a starting node
% where there is no cycle to/from that node but that starting node has an
% edge that connects to a cycle. This bug can be demonstrated by querying
% with 'cycle(e).'. This will return true even though there is no cycle
% to/from e because e has an edge to the cycle containing x (x -> c -> d ->
% x).

cyclic(X) :-
  cycle(X, []).

cycle(CurrentNode, VisitedNodes):-
  member(CurrentNode, VisitedNodes).

cycle(CurrentNode, VisitedNodes):-
  edge(CurrentNode, NextNode),
  cycle(NextNode, [CurrentNode|VisitedNodes]).
```

4. The list operations **car** and **cdr** were initially assembly language instructions that would give you the first half of a word of memory (car) or the second half of a word of memory (cdr). It was discovered that if you structured your data in a word of memory such that you placed a datum in the first half of a word and the address to the next node in the second half of the word you could create what we know today as a linked list. In assembly you could load the memory address of a node and use car to get the datum at that node and then cdr to get the address of the next node you. Repeating this pattern you could easily traverse the linked list. Lists in Lisp are implemented as a linked list which is why you can use car to get the head of a list and cdr to get the tail of a list which is really the memory address of the next node in the list.
5. Below is my Scheme code for question 5:

```
#lang racket
(define (iota-helper N) (cond
                  [(zero? N) null]
                  [else (cons (- N 1) (iota-helper (- N 1)))]))
(define (iota N) (reverse (iota-helper N)))
```

6. Below is my Scheme code for question 6:

```
#lang racket

(define (factorial N) (cond
                        [(< N 2) 1]
                        [else (* N (factorial (- N 1)))]))
(define (sum-digits N acc) (cond
                             [(< N 10) (+ N acc)]
                             [else (sum-digits (floor (/ N 10)) (+ (modulo N
10) acc))]))
(define (f N) (sum-digits (factorial N) 0))
```

7. See below
    a. Below is my Go code for question 7a:

```
func sum(ch <-chan int) int {
    s := 0
    for e := range ch {
        s += e
    }
    return s
}
```

    b. Below is my Go code for question 7b which makes use of the answer in 7a:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println(f(9))
}

func f(n int) int {
    c := make(chan int)
    go sendDigits(factorial(n), c)
    return sum(c)
}

func sendDigits(n int, ch chan<- int) {
    currN := n
    for currN >= 10 {
        ch <- currN % 10
        currN = currN / 10
    }
    ch <-currN
    close(ch)
}

func sum(ch <-chan int) int {
    s := 0
    for e := range ch {
        s += e
    }
    return s
}

func factorial(n int) int {
    acc := n
    for n > 1 {
        n -= 1
        acc = acc * n
    }
    return acc
}
```

8. The primary distinction between the procedural/imperative programming paradigm and the declarative/functional paradigm is that the procedural/imperative paradigm talks about HOW to calculate something. It has you write a series of steps that must be executed in an order that will eventually take you to a final state where you have calculated something useful (hopefully). The declarative/functional paradigm has you talk about WHAT something is. It has you describe what things are as opposed to how to calculate them. The declarative/functional paradigm abstracts you from lower level implementation concerns that do not typically have anything to do with the problem you are solving. It allows you to not have to worry about lower level concerns which can be very freeing while you are solving a problem in the real world because it frees up that brain space and computational energy which can be put towards the real problem at hand.
9. Tail recursion is a particular application of recursion where the last instruction executed inside the recursive function is a recursive function call to itself. This can be contrasted with 'standard' recursion where the last instruction executed would be some other computation. Below I have defined two functions that sum the elements of a list. One is tail recursive (tail-recursive-sum) where we can see that the last function call that will execute is a recursive one. In the non tail recursive function we can see that the last function call is actually doing addition. The main idea to note here which makes the tail recursive implementation much faster is that it does not need to reverse back up the call stack in order to finish its calculation because it utilizes an accumulator to keep track of the current sum value as it moves along.

```
#lang racket
(require racket/trace)

(trace-define (recursive-sum l) (cond
                            [(null? (cdr l)) (car l)]
                            [else (+ (car l) (recursive-sum (cdr l)))]))

(trace-define (tail-recursive-sum-helper l acc) (cond
                                            [(null? l) acc]
                                            [else (tail-recursive-sum-helper
(cdr l) (+ (car l) acc))]))
(trace-define (tail-recursive-sum l) (tail-recursive-sum-helper l 0))
```

   a. We can see the difference in these functions by looking at the invocations we get from using trace-define below. Specifically we can see that the non-tail recursive function has to make the recursive calls all the way down to the base case and then propagate those values back up the call stack. Where as the tail recursive function is actually finished once it hits its base case because it has been keeping the accumulator up to date with the current sum of the list so far.

```
> (recursive-sum '(1 2 3))
>(recursive-sum '(1 2 3))
> (recursive-sum '(2 3))
> >(recursive-sum '(3))
< <3
< 5
<6
6
> (tail-recursive-sum '(1 2 3))
>(tail-recursive-sum '(1 2 3))
>(tail-recursive-sum-helper '(1 2 3) 0)
>(tail-recursive-sum-helper '(2 3) 1)
>(tail-recursive-sum-helper '(3) 3)
>(tail-recursive-sum-helper '() 6)
<6
6
```