LISP Programming

1. **Name:** Jose Franco Baquera

2. **Class:** CS471 – Programming Languages 1 at NMSU

3. **Name of Assignment:** LISP Programming

4. **Date:** December 1, 2019

5. **Aggie ID:** N/A

6. **Problem Description and Purpose**

The main problem being addressed in this programming assignment is the following: *How do we use LISP programs to accomplish specific tasks?* In essence, the assignment instructs us to write a LISP program that includes several function definitions. More specifically, students are instructed to write one or more functions that do the following:

- A function which takes two lists as input. We note that the first list (L1) is a list of indexes into the second list (L2). The output is a list of items from L2 which are selected using L1. It is critical to mention that the second list is 0 indexed. The following two examples illustrate potential statements and their corresponding output:

Example Statement	Corresponding Output List
(extract '(0 1 0) '(a b c))	'(a b a)
(extract '(1 1 1) '(a b c))	'(b b b)

- A function that applies a given list of lambda functions from right to left on a second integer element. For example, the statement (apply-function-list '((lambda (x) (+x 1)) (lambda (x) (*x 2)) 1) would return the number 3. The function implementation should also allow us to do the following:

```
(define  A '((lambda (x) (+ x 1))
  (lambda (x) (- x 1))
  (lambda (x) (* x 2))
  (lambda (x) (* x 3))
 ))

(apply-function-list  (extract '(0 0 2 2 2 3 2 0) A) 0)
```

  The previous apply-function-list input string should return 50.

The assignment also instructs us to create an apply-function-list input string to compute our last 2 digits of our Aggie ID. We note that the purpose of this assignment is to allow students to learn how to program in LISP and how to use this programming language to manipulate lists. Furthermore, the assignment also allows students to experience how functional languages

operate, especially to those who have little to no experience with such language category. Learning how lambda functions work in LISP was also an important aspect of this assignment.

7. extract/extractOneElement Functions with Proper Documentation

```
;;  -----------------------------------------------------------------------------
;;  Name: Jose Franco Baquera
;;  Course: CS 471 - Programming Language Structures 1 at NMSU
;;  Instructor: Shaun Cooper
;;  Assignment: LISP Programming
;;  Program/File Name: Lisp_Programming_Assignment.lsp
;;  Date: December 1, 2019
;;
;;  Assumptions - There are several assumptions for each function in this program. For
;;  function 1, N must be an integer that is greater than or equal to 0 and the length of the list must be
;;  greater than or equal to N + 1. For function 2, both lists MUST be the same length. Furthermore,
;;  function 2 will use the extractOneElement function, meaning that more assumptions must be met. For
;;  example, list L1 can only have integers that are greater than or equal to 0. Let G equal the
;;  largest integer number in list L1. The length of list L2 must then be greater than or equal to G + 1.
;;  Lastly, for function 3, all lambda functions passed must be syntactically correct. In addition, the
;;  second argument must be an integer.
;;
;;  Description / Explanation of Code - Function 1 will take one integer (N) and one list (L) as input.
;;  The integer will represent the index of the element we want to extract from the inputted list. We
;;  note that indexing starts at 0. If N = 0, then we are in the base case. Simply return the first element
;;  in the list. Else, recursively call the function with N - 1 and the tail of the list. Function 2
;;  will take two lists as input. The first list (L1) is a list of indexes into the second list (L2). The
;;  output is a list of items from L2 which are selected using L1. The second list is 0 indexed. If L1 is null,
;;  then we are in the base case. Simply return L1, a null/empty list. Else, construct a new list by recursively
;;  calling functions 1 and 2. That is, function 2 will traverse through list L1 while function 1 will be used
;;  to actually retrieve the element from the list. Lastly, function 3 will take one list (L) and one integer (N)
;;  as input. The list is a list of lambda functions. This function will apply from right to left a given list
;;  of lambda functions on a second integer element. If L is null, then we are in the base case. Simply return N.
;;  Else, make a head of the list a list and recursively call function 3 with N and the tail of the list. The
;;  function will evaluate the lambda functions from right to left.
;;
;;  Purpose - the purpose of this assignment is to allow students to learn how to program in LISP and how to use
;;  this programming language to manipulate lists. Furthermore, the assignment also allows students to experience
;;  how functional languages operate, especially to those who have little to no experience with such language
;;  category. Learning how lambda functions work in LISP was also an important aspect of this assignment.
;;
;;  Program Input / Precondition - One/two lists and one integer value. Please refer to the preconditions
;;  of each function for more information.
;;
;;  Program Output / Post-condition - A list or an integer value. Please refer to the postconditions of each function
;;  for more information.
;;-----------------------------------------------------------------------------

;; Turning on trace mode for debugging purposes. Tracing must be done before defining a function.
;; (require trace)

;; ********** Solution to Problem 1 **********

;; Function 1: This function takes one integer (N) and one list (L) as input. The integer will represent the
;;      index of the element we want to extract from the inputted list. We note that indexing starts at 0.
;; Precondition: N must be an integer that is greater than or equal to 0 and the length of the list must be
;;      greater than or equal to N + 1.
;; Postcondition: The selected index element from the list.
(define (extractOneElement N L )

   (if (eq? N 0)
      (car L) ;; Base case. Return the first element in the list if N equals 0 since indexing starts at 0.
      (extractOneElement (- N 1) (cdr L) ) ;; Recursive/else step. Send N - 1 and the
                                           ;; tail of the list as arguments. This step
                                           ;; will simulate traversing trough the list
                                           ;; until we reach the base case of N = 0.

   ) ;; end if
) ;; end define

;; Function 2: This function takes two lists as input. The first list (L1) is a list of indexes into the second
;;      list (L2). The output is a list of items from L2 which are selected using L1. The second list is 0 indexed.
;; Precondition: Both lists MUST be the same length. Furthermore, we will use the extractOneElement function, meaning
;;      that more assumptions must be met. For example, list L1 can only have integers that are greater than or equal to 0.
;;      Let G equal the largest integer number in list L1. The length of list L2 must then be greater than
;;      or equal to G + 1.
;; Postcondition: A list of items from L2 which are selected using the integers in list L1.
(define (extract L1 L2)

   (if (null? L1)
      L1 ; Base case. Return L1 (a null list) if we have traversed through the entire list L1.
```

```
      (cons (extractOneElement(car L1) L2 ) ; Recursive/else step. Construct a new list. The first argument is the actual
            (extract (cdr L1) L2)           ; indexed item while the second argument is the recursive call to the reminder
                                            ; of list L1.
      ) ; end cons

   ) ; end if
) ; end define
```

8. *apply-function-list* Function with Proper Documentation

```
;;  ------------------------------------------------------------------------------
;;  Name: Jose Franco Baquera
;;  Course: CS 471 - Programming Language Structures 1 at NMSU
;;  Instructor: Shaun Cooper
;;  Assignment: LISP Programming
;;  Program/File Name: Lisp_Programming_Assignment.lsp
;;  Date: December 1, 2019
;;
;;  Assumptions - There are several assumptions for each function in this program. For
;;  function 1, N must be an integer that is greater than or equal to 0 and the length of the list must be
;;  greater than or equal to N + 1. For function 2, both lists MUST be the same length. Furthermore,
;;  function 2 will use the extractOneElement function, meaning that more assumptions must be met. For
;;  example, list L1 can only have integers that are greater than or equal to 0. Let G equal the
;;  largest integer number in list L1. The length of list L2 must then be greater than or equal to G + 1.
;;  Lastly, for function 3, all lambda functions passed must be syntactically correct. In addition, the
;;  second argument must be an integer.
;;
;;  Description / Explanation of Code - Function 1 will take one integer (N) and one list (L) as input.
;;  The integer will represent the index of the element we want to extract from the inputted list. We
;;  note that indexing starts at 0. If N = 0, then we are in the base case. Simply return the first element
;;  in the list. Else, recursively call the function with N - 1 and the tail of the list. Function 2
;;  will take two lists as input. The first list (L1) is a list of indexes into the second list (L2). The
;;  output is a list of items from L2 which are selected using L1. The second list is 0 indexed. If L1 is null,
;;  then we are in the base case. Simply return L1, a null/empty list. Else, construct a new list by recursively
;;  calling functions 1 and 2. That is, function 2 will traverse through list L1 while function 1 will be used
;;  to actually retrieve the element from the list. Lastly, function 3 will take one list (L) and one integer (N)
;;  as input. The list is a list of lambda functions. This function will apply from right to left a given list
;;  of lambda functions on a second integer element. If L is null, then we are in the base case. Simply return N.
;;  Else, make a head of the list a list and recursively call function 3 with N and the tail of the list. The
;;  function will evaluate the lambda functions from right to left.
;;
;;  Purpose - the purpose of this assignment is to allow students to learn how to program in LISP and how to use
;;  this programming language to manipulate lists. Furthermore, the assignment also allows students to experience
;;  how functional languages operate, especially to those who have little to no experience with such language
;;  category. Learning how lambda functions work in LISP was also an important aspect of this assignment.
;;
;;  Program Input / Precondition - One/two lists and one integer value. Please refer to the preconditions
;;  of each function for more information.
;;
;;  Program Output / Post-condition - A list or an integer value. Please refer to the postconditions of each function
;;  for more information.
;;------------------------------------------------------------------------------

;; ********** Solution to Problem 2 **********

;; Function 3: This function takes one list (L) and one integer (N) as input. The list is a list of lambda functions.
;; This function will apply from right to left a given list of lambda functions on a second integer element.
;; Precondition: All lambda functions passed must be syntactically correct. Furthermore, the second argument must be
;;      an integer.
;; Postcondition: An integer value that is computed after the function applies from right to left the given list of lambda
;;      functions on the second integer element.
(define (apply-function-list L N )

   (if (null? L)
      N ;; Base case. Return N if L is null since there are no more lambda functions to evaluate.
      (eval ( list (car L) ;; Make the head of the list a standalone list by itself.
            (apply-function-list (cdr L)  N ) ;; Recursively call the function with N and the tail
                                              ;; of the list. The evaluation of the lambda functions
                                              ;; will be from right to left since the recursive step is done
                                              ;; AFTER the head is converted into a standalone list.

            ) ;; end list

      ) ;; end eval

   ) ;; end if
) ;; end define
```

9. Demonstration that *extract* Works Properly

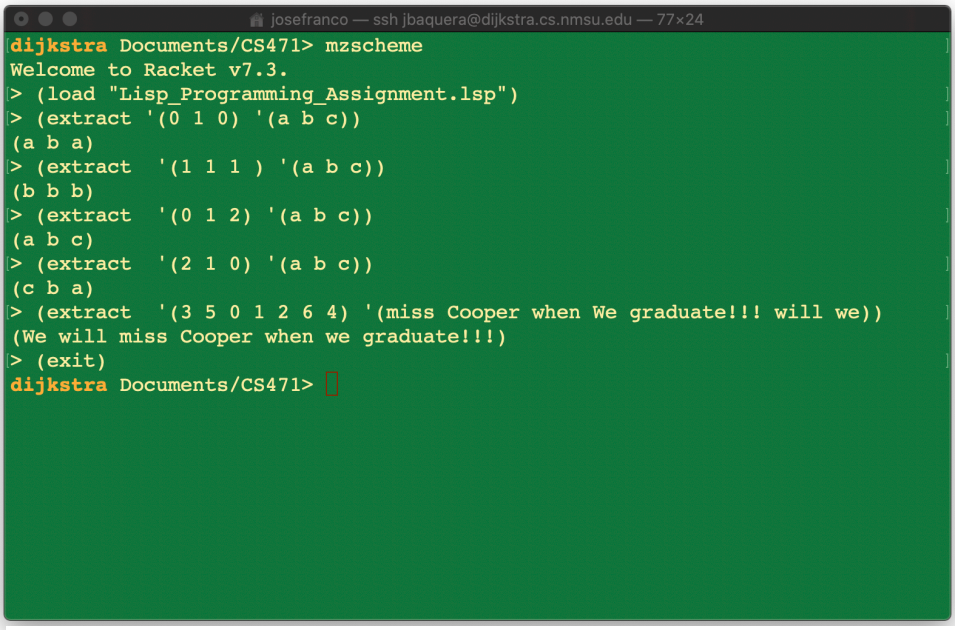

Figure 1 – Demonstration that the *extract* function works correctly.

10. Demonstration that *apply-function-list* Works Properly

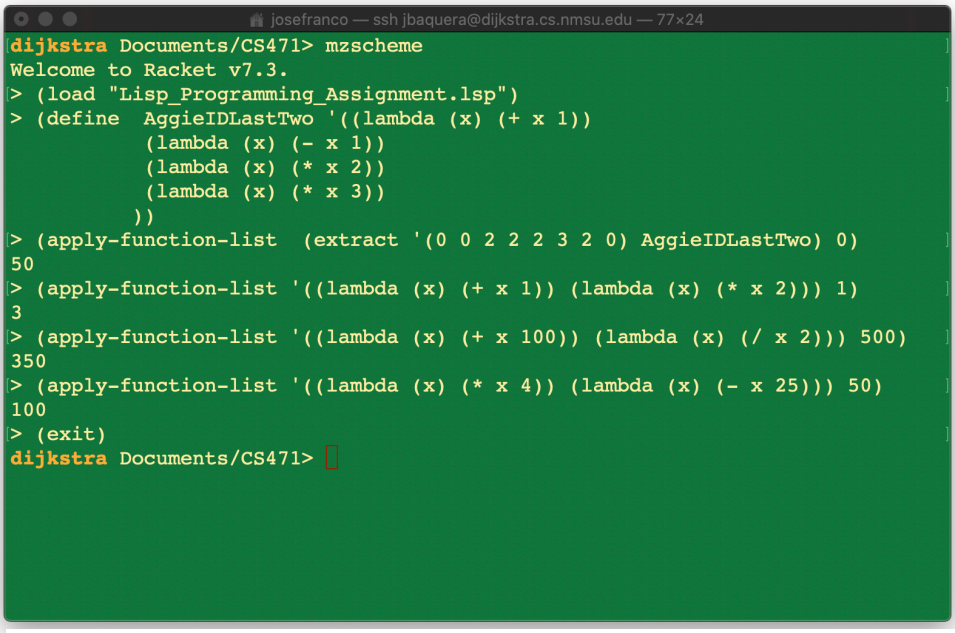

Figure 2 – Demonstration that the *apply-function-list* function works correctly.

11. *apply-function-list* Input String Used to Compute Last 2 Digits of Aggie ID

We use the following list definition provided by the assignment:

```
;; Definition of trivial lambda functions provided by the assignment.
(define  AggieIDLastTwo '((lambda (x) (+ x 1))
         (lambda (x) (- x 1))
         (lambda (x) (* x 2))
         (lambda (x) (* x 3))
        ) ;; end list
) ;; end define
```

By using the previous list definition, the following *apply-function-list* input string can be used to compute the last two digits of my Aggie ID:

```
(apply-function-list  (extract '(0 3 2 3 0) AggieIDLastTwo) 1)
```

12. Demonstration of the String that Generates Last Two Digits of Aggie ID

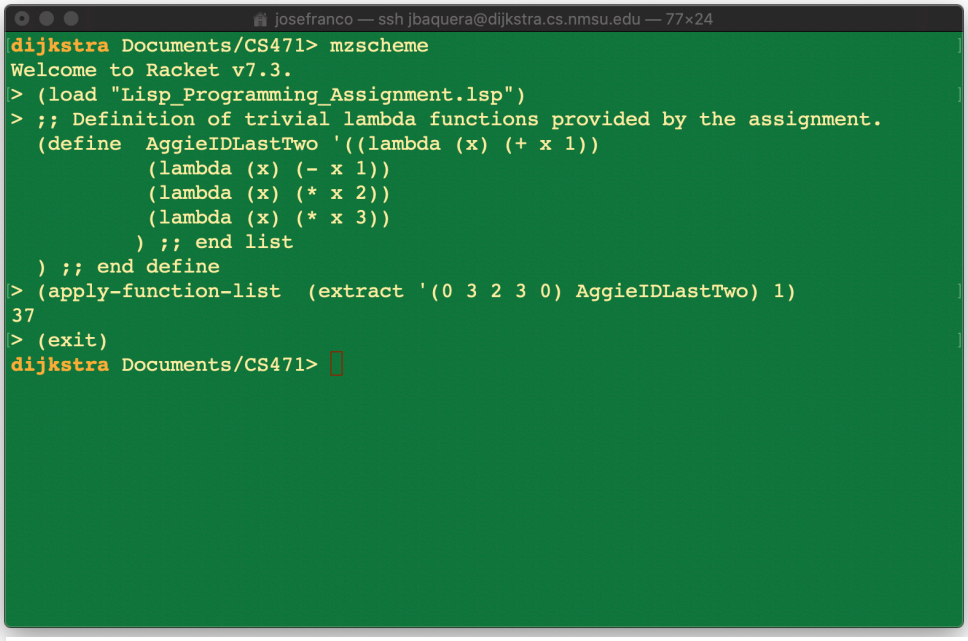

Figure 3 – Demonstration of the string that generates the last two digits of my Aggie ID.

13. Conclusion

At times, LISP is extremely difficult to read and write, especially since this language does Polish notation (i.e. + x 1 instead of x + 1). Nevertheless, the assignment made me realize how powerful LISP can be and how it can be used to quickly manipulate/evaluate lists, as well as lambda functions.