**Lab 7**

Problem 1

* Fetch

(defun **fetch** (key assoc\_list)

    (if (assoc key assoc\_list)

        (cadr (assoc key assoc\_list))

        '?

    )

)

* Test Cases:

(defvar bob '(

    (temperature 100)

    (pressure (120 60))

    (pulse 72)

))

(print

    (fetch 'temperature bob)

)

*; 100*

(print

    (fetch 'pressure bob)

)

*; (120 60)*

(print

    (fetch 'complaints bob)

)

*; ?*

Problem 2

* List Keys:

(defun **list\_keys** (assoc\_list)

    (mapcar

        (lambda (kv\_pair) (car kv\_pair))

        assoc\_list

    )

)

* Test Cases:

(defvar bob '(

    (temperature 100)

    (pressure (120))

    (pulse 72)

))

(print

    (list\_keys bob)

)

*; (TEMPERATURE PRESSURE PULSE)*

(defvar dan nil)

(print

    (list\_keys dan)

)

*; NIL*

[for the next 3 problems]

Helper Function: Make Person

(defun **make\_person** (name father mother)

    (setf (get name 'father) father)

    (setf (get name 'mother) mother)

)

Variables

*; GGG Grandparent*

(make\_person 'markus nil nil)

*; Great Great Grandparents*

(make\_person 'john 'markus nil)

(make\_person 'emi   nil nil)

*; Great Grandparents*

(make\_person 'john\_II 'john  nil)

(make\_person 'sara     nil   nil)

(make\_person 'rich     nil   nil)

(make\_person 'cara     nil  'emi)

*; Grand Parents*

(make\_person 'john\_III 'john\_II 'sara)

(make\_person 'ali      'rich    'cara)

*;Parents*

(make\_person 'dan 'john\_III 'ali)

*; Person*

(make\_person 'bob 'dan nil)

Problem 3

* Grandfather

(defun **grandfather** (person)

    (let\* (

        (father (get person 'father))

        (grandfather (get father 'father))

    )

        grandfather

    )

)

* Test Cases:

(print

    (grandfather 'bob)

)

*; JOHN\_III*

(print

    (grandfather 'rich)

)

*; NIL*

(print

    (grandfather 'john\_II)

)

*; MARKUS*

Problem 4

* Adam

(defun **adam** (person)

    (let (

        (father (get person 'father))

    )

        (if (null father)

            person

            (adam father)

        )

    )

)

* Test Cases:

(print

    (adam 'bob)

)

*; MARKUS*

(print

    (adam 'dan)

)

*; MARKUS*

(print

    (adam 'ali)

)

*; RICH*

(print

    (adam 'markus)

)

*; MARKUS*

Problem 5

* Ancestors:

(defun **ancestors** (person)

    (if (null person)

        nil

        (let (

            (father (get person 'father))

            (mother (get person 'mother))

        )

            (append (list person) (ancestors father) (ancestors mother))

        )

    )

)

* Test Cases:

(print

    (ancestors 'bob)

)

*; (BOB DAN JOHN\_III JOHN\_II JOHN MARKUS SARA ALI RICH CARA EMI)*

(print

    (ancestors 'ali)

)

*; (ALI RICH CARA EMI)*

(print

    (ancestors 'markus)

)

*; (MARKUS)*

Problem 6 & 7

* Print Matrix:

(defun **print\_matrix** (matrix)

    (let (

        (rows (array-dimension matrix 0))

        (cols (array-dimension matrix 1))

    )

        (do ((i 0 (1+ i))) ((= i rows))

            (do ((j 0 (1+ j))) ((= j cols))

                (princ (aref matrix i j))

                (princ " ")

            )

            (terpri)

        )

    )

)

* List to Array:

1. Version 1:

(defun **to\_matrix** (lst2d)

    (let\* (

        (rows (length lst2d))

        (cols (length (car lst2d)))

        (matrix (make-array (list rows cols)))

        (row 0)

        (col 0)

    )

        (mapcar (lambda (lst)

                (mapcar (lambda (elem)

                        (setf (aref matrix row col) elem)

                        (setq col (1+ col))

                    )

                    lst

                )

                (setq row (1+ row))

                (setq col 0)

            )

            lst2d

        )

        matrix

    )

)

1. Version 2:

(defun **to\_matrix** (lst2d)

    (defun **elem\_to\_matrix** (lst row col matrix)

        (cond

            ((null lst) matrix)

            (t

                (setf (aref matrix row col) (car lst))

                (elem\_to\_matrix (cdr lst) row (1+ col) matrix)

            )

        )

    )

    (defun **row\_to\_matrix** (lst2d row matrix)

        (cond

            ((null lst2d) matrix)

            (t

                (elem\_to\_matrix (car lst2d) row 0 matrix)

                (row\_to\_matrix (cdr lst2d) (1+ row) matrix)

            )

        )

    )

    (let\* (

        (rows (length lst2d))

        (cols (length (car lst2d)))

        (matrix (make-array (list rows cols)))

    )

        (row\_to\_matrix lst2d 0 matrix)

    )

)

* Test Cases (to\_matrix):

(print (to\_matrix '(

    (1 2)

    (3 4)

    (5 6)

)))

*; #2A((1 2) (3 4) (5 6))*

(print (to\_matrix '(

    (1)

    (2)

    (3)

)))

*; #2A((1) (2) (3))*

(print (to\_matrix '(

    (1)

)))

*; #2A((1))*

(print (to\_matrix '(

    ()

)))

*; #2A(())*

(print (to\_matrix '(

)))

*; #2A()*

* Test Cases (print\_matrix):

(print\_matrix (to\_matrix\_2 '(

    (1 2)

    (3 4)

    (5 6)

)))

*; 1 2*

*; 3 4*

*; 5 6*

(print\_matrix (to\_matrix\_2 '(

    (1)

    (2)

    (3)

)))

*; 1*

*; 2*

*; 3*

(print\_matrix (to\_matrix\_2 '(

    (1)

)))

*; 1*

(print\_matrix (to\_matrix\_2 '(

    ()

)))

*;*

(print\_matrix (to\_matrix\_2 '(

)))