תרגיל בית 1

מגישים : אסף לובטון תז 209844414 עדן דמינסקי תז 212227888

Part 1 lisp:

1.

```
(defun cat (first second)
  (cond
          ((null first) second)
      ((atom first) (cons first second))
      (t (cat (car first) (cat (cdr first) second)))
  )
)
```

2. A.

ההבדל בין המימושים הוא בערך ההחזרה במקרה הריק עבור המימוש המשתמש ב**eq** הערך המוחזר הוא **nil** ואילו במימוש השני הוא **xs** (הרשימה ריקה ששקול ל**nil**). אין שינוי בסמנטיקה מאחר ועבור המימוש הראשון יתקבל שיוויון בהשוואה ואף במימוש השני תוחזר הרשימה הריקה שערכה שקול ל**nil** כלומר קיבלנו כי ערך ההחזרה זהה.

B.

```
(defun member(x xs) (
cond
      ((null xs) xs)
      ((eq x (car xs))    (cons x (cdr xs)))
      (t (member x (cdr xs)))
))
```

ניתן להחליף כל קריאה ל-**exists** בקריאה ל-**member** משום שאם האטום לא קיים ברשימה אז יתקבל **nil** עבור שתי הפונקציות ואם לא שווה ל**nil** אזי האטום בהכרח קיים. לכן אם הערך שהוחזר מהקריאה ל**member** אינו **nil** בהכרח האטום קיים ברשימה.

3.

```
(defun andalso (b1 b2) (

cond

((null b1) nil)

((null b2) nil)

(t t)

))
```

```
(defun equal(first second)(
cond
((andalso (null first) (null second)) t);both are empty return t
((null first) nil);one is empty return nil
((null second) nil);second empty return nil
((andalso (atom first) (atom second)) (eq first second));compare if both atoms, return ans of eq
((equal(car first) (car second)) (equal (cdr first) (cdr second)));comp if both lists call func on cdr of them
(t nil)
))

;lines for checking:
(print (equal ‘(a b c) '(a b c)))     ;returned t
(print (equal  '(a b c (b)) '(a b c (b f))))     ;returned nil
```

4.

```
;we took badd form tutorial slides, our code is in bold
(defun list−contains−only−zeros (l) (
cond
((null l) t)
((eq (car l) (quote Z))
(list−contains−only−zeros (cdr l)))
((eq (car l) (quote O)) nil)
))
```

```
(defun bnormalize (l) (
cond
((list-contains-only-zeros l) nil)
(t (cons (car l )  (bnormalize( cdr l))))
))
(defun andalso (b1 b2)(
cond
((null b1) nil)
((null b2) nil)
(t t)
))
(defun badd (l1 l2)
cond
((null (bnormalize l1))(bnormalize l2))
((null (bnormalize l2))(bnormalize l1))
((andalso (eq (car l1) (quote O)) (eq (car l2) (quote O)))
 (cons (quote Z) (badd (badd (quote (O)) (cdr l1)) (cdr l2))))
((andalso (eq (car l1) (quote Z)) (eq (car l2) (quote Z)))
 (cons (quote Z) (badd (cdr l1) (cdr l2))))
(t (cons (quote O) (badd (cdr l1) (cdr l2))))
))
```

;gets two binary numbers and return the mult of them, iterates over l2 bits calculate the sum of the multiplications ;with l1, while increasing l1 by doing left shift (adding zero as an lsb) in each iteration

```
(defun bmul (l1 l2) (
cond
((null (bnormalize l1)) nil);if l1 is empty returns nil
((null (bnormalize l2)) nil);if l2 is empty returns nil
```

((eq (car l2) (quote Z)) cons (quote Z) (bmul l1 (cdr l2)))

;l2 ‘s lsb is zero, push zero to l1’s lsb and call the func with the next l2’s lsb this iteration does not

;add anything to the result

((eq (car l2) (quote O)) (badd l1 (bmul (cons (quote Z) l1) (cdr l2))))

;l2’s lsb is one return the sum of l1 and l1 with zero pushed in lsb call the func with next l2’s lsb

;this iteration adds l1 multiplication with the l2’s lsb bit value to the result.

))

5.

```
(defun is−primitive(name); determine whether name denotes a primitive function
  (exists name ’(atom car cdr cond cons eq error eval set)))
;
return t if name exists in the list above otherwise false
;
(defun exists (x xs) ; determine whether atom x is in list xs
 (cond ; Three cases to consider:
            ((null xs) xs) ; (i) list of xs is exhausted
             ((eq x (car xs)) t) ; (ii) item x is first in xs
             (t (exists x (cdr xs))))) ; (iii) otherwise, recurse on rest of xs
;
gets an atom x and a list xs
if the list is empty returns false
iterates over xs’s components if the atom exists return true otherwise false
;
(defun lookup (id a−list) ; lookup id in an a−list
(cond ; Three cases to consider:
            ((null a−list) ; (i) a−list was exhausted.
            (error ’unbound−variable id))
            ((eq id (car (car a−list))) ; (ii) found in first dotted−pair
            (car (cdr (car a−list)))) ; return value part of dotted pair
            (t (lookup id (cdr a−list))))) ; (iii) otherwise, recursive call on remainder of a−list


;
gets an identifier to look for in the a list
if a list is empty calls error (the value we were looking for did not exists in the a list )
then itarates on a list recursively comparing each of the dotted pairs if the first of the dotted pair equals to
the identifier we are looking for if it does returns the second of the pair if not found moves on to the next
pair
;
(defun bind (names values a−list) ; bind names to values, and append to a−list
 (cond ((null names) ; no more names left
            (cond ((null values) a−list) ; no more values left, binding done−> return a−list
                        (t (error ’missing−names)))) ; more values than names
```

```
            ((null values) ; names is not nil but values is, i.e., more names than values
             (error 'missing−values))
             (t ; both names and values are not empty
             (cons ; create new binding and prepend it to result of recursive call
             (cons (car names) (car values)) ; new dotted−pair defines single binding
 (bind (cdr names) (cdr values) a−list))))) ; recursive vall
;
itartates on the name's lis and value's list  and add them as dotted pairs to the a-list recursively.
if names is empty return
;
(defun evaluate(S−expression a−list) ; evaluate S−expression in the environment defined by a−list
(cond ((atom S−expression) ; recursion base: lookup of atom in a−list
              (lookup S−expression a−list))
            ((is−primitive (car S−expression)) ; special case handling of primitive functions
                (evaluate−primitive S−expression a−list))
             (t ; recursive step−−−lambda applied to parameters
                        (apply (evaluate (car S−expression) a−list) ; find lambda expression
                        (cdr S−expression) ; find actual parameters
                         a−list))))
;
```

1.if S-exp is an atom look it up in the a list (bring it's meaning)
2.if the first atom of the S-exp is an atom representing a primitive function call evaluate-primitive func
3.call evaluate for the first atom of the S-exp and call apply with the evaluated first atom of S-exp and the rest of the S-exp

```
;
(defun apply(lambda−expression actuals a−list)
(apply−decomposed−lambda
            (car lambda−expression) ; tag=lambda or nlambda
            (car (cdr lambda−expression)); list of formal parameters
             (car (cdr (cdr lambda−expression))); body
             actuals
             a−list))
;
```

call apply-decomposed-lambda
first atom of lambda-exp (tag which is lambda or nlambda)
list of formal parameters (the arguments of the func)
body (the body of the function)
actuals(passed from before)
a-list(passed from before)

```
;
(defun apply−decompsed−lambda
 (tag formals body actuals a−list)
             (evaluate body
                        (cond
                                    (eq tag 'nlambda) (bind formals actuals a−list)
                                    (eq tag 'lambda) (bind formals (evaluate−list actuals a−list) a−list)
                                    (t (error 'unkown−lambda tag)))))


;
```

if tag== nalambda
call bind of formals and actuals on a list which means creating dotted pairs of formals and actuals and adding them to the a list
if tag==lambda
before calling bind we need to evaluate the actuals list so call evaluate list passing actuals and a list
do anyway if got here
the tag was not valid

```
;
 (defun evaluate−list(S−expressions a−list)
             (cond ((null S−expressions) nil) ; no more S−expressions to evaluate
                        (t (cons
                                     (evaluate (car S−expressions)) ; evaluate first S−Expression
                                      (evaluate−list (cdr S−expressions)))))) ; recursive call on remainder
```

;
if S-exp is empty return nil means there is no more S-exp to evaluate
do anyway if got here
make a list of the evaluated S-exp by iterating recursively on the S-exp list by making a dotted pair with the returned value of the func evaluate on the first component of the S-exp list and the returned value of the func evaluate list on the reset of the components of S-exp list.
;

```
(defun evaluate−primitive (S−expression a−list)
 (apply−primitive (car S−expression) (cdr S−expression) a−list))
```

;
call the func apply-primitve the first component of S-exp is the primitive func name the rest is the func parameters not evaluated yet and passing a list
;

```
(defun apply−primitive (primitive actuals a−list)
        (cond ((eq 'cond primitive) ; special case for cond that has normal semantics
                (evaluate−cond actuals a−list)) ; don't evaluate actuals
               (t (apply−eager−primitive ; all other primitives have eager semantics
               primitive (evaluate−list actuals a−list) a−list)))|)
```

;
if primitive == cond then call evaluate cond func passing actuals and a list
do anyway if got here
primitive must represent an eager primitive function which means that the values passed to it need to be evaluated before calling it therefore calls the primitive func after the actuals list will be returned from the func evaluated list.
;

```
(defun evaluate−cond(test−forms a−list)
        (cond ((null test−forms) nil) ; if no more test−forms, return nil
         ((evaluate (car (car test−forms)) a−list) ; evaluate test−condition of first test−form
        (evaluate (car (cdr (car test−forms))) a−list)) ; if true, evaluate test−value of first test−form
        (t (evaluate−cond (cdr test−forms) a−list)))) ; otherwise, recurse on remainder list of test−forms
```

;
if the test forms list is empty return nil
starts to evaluate the test forms recursively
calls evaluate func on the first test condition of the first test form if the value was true goes to evaluate the test value of the first test form, if it was false calls the recursion on the test forms list removing the current test form.
;

```
(defun apply−eager−primitive (primitive actuals a−list)
        (cond ((eq primitive 'error) (error actuals))
        ((eq primitive 'eval) (evaluate (car actuals) a−list))
        (t (apply−trivial−primitive
                primitive ; one of atom, car, cdr, cons, eq, or set
         (car actuals) ; first actual parameter
         (car cdr actuals))))) ; second actual parameter, could be nil
```

;
if primitive==error call error passing it actuals list
if primitive==eval call evaluate on the on the first component of actuals
do anyway if got here
call apply trivia primitive func passing primitive and 2 parameters not evaluated from actuals (first and second)
;

```
(defun apply−trivial−primitive (primitive first second)
 (cond ((eq primitive 'atom) (atom first))
 ((eq primitive 'car) (car first))
 ((eq primitive 'cdr) (cdr first))
 ((eq primitive 'cons) (cons first second))
 ((eq primitive 'eq) (eq first second))
((eq primitive 'set) (set first second))
 (t (error 'something−went−wrong primitive))))
```

;
the first arg is the primitive func we want to apply goes over the optional values of the primitive func

if it was none of the options call error
;

final explanation

evaluate gets an S-expression to evaluate and a-list which is the “database” of the symbols both arguments are lists.
there are 3 options:
1.if the S-exp is an atom call the lookup function in a list and return its evaluation (meaning of the atom).
2.the S-exp is not an atom therefore check if it begins with a primitive function calls is-primitive that checks if the passed name is one of the “saved names” for primitive functions.
if true calls evaluate-primitive.
evaluate primitive: calls apply-primitive when the first component of the S-expression is the name of the primitive the rest is the actuals.
Apply-primitive distincts between two cases, "cond" or one of the other primitive functions. (because cond has normal semantics, we won’t be evaluating the actuals). for cond call evaluate-cond. for the other primitive functions call apply-eager-primitive after calling evaluate-list on actuals (we evaluate the actuals list before calling the function because it’s eager typed).
evaluate-cond:
goes over the test-forms recursively return nil if there are no more test forms. call evaluate on the first test condition of the first test form, if the value was true goes to evaluate the test value of the first test form, if it was false call the recursion on the test forms list removing the first form.
apply -eager-primitive:
distinct between error, eval, and the rest. error gets the actuals list without evaluation. eval gets the first component of actuals after it was evaluated by calling the evaluate function. all the other primitive functions are handled by calling the apply-trivia-primitive function by passing primitive, the first component of actuals as first and the rest of actuals as second (can be nil).
apply-trivia-primitive:
goes over the optional values of the primitive functions (atom, car, cdr, cons, eq, and, set)
if it was none of the above call error.
3.calls apply after calling evaluate on the first component of the S-exp (to find the lambda expression) the reset of the S-exp is the actual parameters.
apply:
calls apply-decompose-lambda, the first argument (tag) can be lambda or lambda(differs the semantics, eager or normal) the first component after the tag is the list of formal parameters and the component afterwards is the body of the function.
apply−decomposed−lambda:
distinct between two cases first one is tag equals nlambda and the second is lambda otherwise calls error (error with the tag). In case of nlambda calls bind on the formals and actuals, in case of lambda first call evaluate-list on actuals and then calls bind on formals and evaluated actuals.
bind:
adds the actuals and formals to the a-list as dotted pairs.