Kuncheng Feng
CSC 416

Programming Challenge: Recursive List Processing and HOFs

Task 1: singleton-p - rac - rdc - snoc - palindrome-p

Singleton-p

```
[]> (singleton-p '(a))
T
[]> (singleton-p '(a b))
NIL
[]> (singleton-p '(1 2 3 4 5 6 7))
NIL
```

Rac

```
[]> (trace rac)
;; Tracing function RAC.
(RAC)
[]> (rac '(a))
1. Trace: (RAC '(A))
1. Trace: RAC ==> A
A
[]> (rac '(a b c d))
1. Trace: (RAC '(A B C D))
2. Trace: (RAC '(B C D))
3. Trace: (RAC '(C D))
4. Trace: (RAC '(D))
4. Trace: RAC ==> D
3. Trace: RAC ==> D
2. Trace: RAC ==> D
1. Trace: RAC ==> D
D
[]> (untrace rac)
(RAC)
```

Rdc

```
[]> (trace rdc)
;; Tracing function RDC.
(RDC)
[]> (rdc '(a))
1. Trace: (RDC '(A))
1. Trace: RDC ==> NIL
NIL
[]> (rdc '(a b c d e))
1. Trace: (RDC '(A B C D E))
2. Trace: (RDC '(B C D E))
3. Trace: (RDC '(C D E))
4. Trace: (RDC '(D E))
5. Trace: (RDC '(E))
5. Trace: RDC ==> NIL
4. Trace: RDC ==> (D)
3. Trace: RDC ==> (C D)
2. Trace: RDC ==> (B C D)
1. Trace: RDC ==> (A B C D)
(A B C D)
[]> (untrace rdc)
(RDC)
```

Snoc

```
[]> (trace snoc)
;; Tracing function SNOC.
(SNOC)
[]> (snoc 'blue '())
1. Trace: (SNOC 'BLUE 'NIL)
1. Trace: SNOC ==> (BLUE)
(BLUE)
[]> (snoc 'blue '(red))
1. Trace: (SNOC 'BLUE '(RED))
2. Trace: (SNOC 'BLUE 'NIL)
2. Trace: SNOC ==> (BLUE)
1. Trace: SNOC ==> (RED BLUE)
(RED BLUE)
```

```
[]> (snoc 'blue '(cornflower-blue prussian-blue royal-blue
navy-blue))
1. Trace: (SNOC 'BLUE '(CORNFLOWER-BLUE PRUSSIAN-BLUE ROYAL-BLUE
NAVY-BLUE))
2. Trace: (SNOC 'BLUE '(PRUSSIAN-BLUE ROYAL-BLUE NAVY-BLUE))
3. Trace: (SNOC 'BLUE '(ROYAL-BLUE NAVY-BLUE))
4. Trace: (SNOC 'BLUE '(NAVY-BLUE))
5. Trace: (SNOC 'BLUE 'NIL)
5. Trace: SNOC ==> (BLUE)
4. Trace: SNOC ==> (NAVY-BLUE BLUE)
3. Trace: SNOC ==> (ROYAL-BLUE NAVY-BLUE BLUE)
2. Trace: SNOC ==> (PRUSSIAN-BLUE ROYAL-BLUE NAVY-BLUE BLUE)
1. Trace: SNOC ==> (CORNFLOWER-BLUE PRUSSIAN-BLUE ROYAL-BLUE
NAVY-BLUE BLUE)
(CORNFLOWER-BLUE PRUSSIAN-BLUE ROYAL-BLUE NAVY-BLUE BLUE)
[]> (untrace snoc)
(SNOC)
```

Palindrome-p

```
[]> (trace palindrome-p)
;; Tracing function PALINDROME-P.
(PALINDROME-P)
[]> (palindrome-p '())
1. Trace: (PALINDROME-P 'NIL)
1. Trace: PALINDROME-P ==> T
T
[]> (palindrome-p '(palindrome))
1. Trace: (PALINDROME-P '(PALINDROME))
1. Trace: PALINDROME-P ==> T
T
[]> (palindrome-p '(clos sloc))
1. Trace: (PALINDROME-P '(CLOS SLOC))
1. Trace: PALINDROME-P ==> NIL
NIL
[]> (palindrome-p '(food drink food))
1. Trace: (PALINDROME-P '(FOOD DRINK FOOD))
2. Trace: (PALINDROME-P '(DRINK))
2. Trace: PALINDROME-P ==> T
1. Trace: PALINDROME-P ==> T
T
[]> (palindrome-p '(1 2 3 4 5 4 2 3 1))
```

```
1. Trace: (PALINDROME-P '(1 2 3 4 5 4 2 3 1))
2. Trace: (PALINDROME-P '(2 3 4 5 4 2 3))
2. Trace: PALINDROME-P ==> NIL
1. Trace: PALINDROME-P ==> NIL
NIL
[]> (palindrome-p '(hey hey my my my my hey hey))
1. Trace: (PALINDROME-P '(HEY HEY MY MY MY MY HEY HEY))
2. Trace: (PALINDROME-P '(HEY MY MY MY MY HEY))
3. Trace: (PALINDROME-P '(MY MY MY MY))
4. Trace: (PALINDROME-P '(MY MY))
5. Trace: (PALINDROME-P 'NIL)
5. Trace: PALINDROME-P ==> T
4. Trace: PALINDROME-P ==> T
3. Trace: PALINDROME-P ==> T
2. Trace: PALINDROME-P ==> T
1. Trace: PALINDROME-P ==> T
T
[]> (untrace palindrome-p)
(PALINDROME-P)
```

Task 2: select - pick

Code for Select

```
(defun select (number *list*)
     (cond
          ((> 0 number)
               nil
          )
          ((equal nil *list*)
               nil
          )
          ((equal 0 number)
               (car *list*)
          )
          (t
               (select (- number 1) (cdr *list*))
          )
     )
)
```

Demo for select

```
[]> (trace select)
;; Tracing function SELECT.
(SELECT)
[]> (select 1 '(0 1 2 3))
1. Trace: (SELECT '1 '(0 1 2 3))
2. Trace: (SELECT '0 '(1 2 3))
2. Trace: SELECT ==> 1
1. Trace: SELECT ==> 1
1
[]> (select 4 '())
1. Trace: (SELECT '4 'NIL)
1. Trace: SELECT ==> NIL
NIL
[]> (select 8 '(0 1 2 3 4 5 6 7))
1. Trace: (SELECT '8 '(0 1 2 3 4 5 6 7))
2. Trace: (SELECT '7 '(1 2 3 4 5 6 7))
3. Trace: (SELECT '6 '(2 3 4 5 6 7))
4. Trace: (SELECT '5 '(3 4 5 6 7))
```

```
5. Trace: (SELECT '4 '(4 5 6 7))
6. Trace: (SELECT '3 '(5 6 7))
7. Trace: (SELECT '2 '(6 7))
8. Trace: (SELECT '1 '(7))
9. Trace: (SELECT '0 'NIL)
9. Trace: SELECT ==> NIL
8. Trace: SELECT ==> NIL
7. Trace: SELECT ==> NIL
6. Trace: SELECT ==> NIL
5. Trace: SELECT ==> NIL
4. Trace: SELECT ==> NIL
3. Trace: SELECT ==> NIL
2. Trace: SELECT ==> NIL
1. Trace: SELECT ==> NIL
NIL
[]> (untrace select)
(SELECT)
```

Code for pick

```
(defun pick (*list* &aux number)
     (setf number (random (length *list*)))
     (select number *list*)
)
```

Demo for pick

```
[]> (pick '(a b c d))
B
[]> (pick '(0 1 2 3 4 5 6 7))
0
[]> (pick '())
NIL
```

Task 3: sum - product

Code for sum (from class)

```
( defun sum ( l )
     ( cond
          ( ( null l )
               0
          )
          ( t
               ( + ( car l ) ( sum ( cdr l ) ) )
          )
     )
)
```

Demo for sum

```
[]> (trace sum)
;; Tracing function SUM.
(SUM)
[]> (sum '())
1. Trace: (SUM 'NIL)
1. Trace: SUM ==> 0
0
[]> (sum '(486))
1. Trace: (SUM '(486))
2. Trace: (SUM 'NIL)
2. Trace: SUM ==> 0
1. Trace: SUM ==> 486
486
[]> (sum '(1 11 111))
1. Trace: (SUM '(1 11 111))
2. Trace: (SUM '(11 111))
3. Trace: (SUM '(111))
4. Trace: (SUM 'NIL)
4. Trace: SUM ==> 0
3. Trace: SUM ==> 111
2. Trace: SUM ==> 122
1. Trace: SUM ==> 123
123
[]> (sum '(1 2 3 4 5 6 7 8 9 10))
```

```
1. Trace: (SUM '(1 2 3 4 5 6 7 8 9 10))
2. Trace: (SUM '(2 3 4 5 6 7 8 9 10))
3. Trace: (SUM '(3 4 5 6 7 8 9 10))
4. Trace: (SUM '(4 5 6 7 8 9 10))
5. Trace: (SUM '(5 6 7 8 9 10))
6. Trace: (SUM '(6 7 8 9 10))
7. Trace: (SUM '(7 8 9 10))
8. Trace: (SUM '(8 9 10))
9. Trace: (SUM '(9 10))
10. Trace: (SUM '(10))
11. Trace: (SUM 'NIL)
11. Trace: SUM ==> 0
10. Trace: SUM ==> 10
9. Trace: SUM ==> 19
8. Trace: SUM ==> 27
7. Trace: SUM ==> 34
6. Trace: SUM ==> 40
5. Trace: SUM ==> 45
4. Trace: SUM ==> 49
3. Trace: SUM ==> 52
2. Trace: SUM ==> 54
1. Trace: SUM ==> 55
55
[84]> (untrace sum)
(SUM)
```

Code for product

```
(defun product (*list*)
     (cond
          ((null *list*)
               1
          )
          (t
               (* (car *list*) (product (cdr *list*)))
          )
     )
)
```

Demo for product

```
[]> (trace product)
;; Tracing function PRODUCT.
(PRODUCT)
[]> (product '())
1. Trace: (PRODUCT 'NIL)
1. Trace: PRODUCT ==> 1
1
[]> (product '(496))
1. Trace: (PRODUCT '(496))
2. Trace: (PRODUCT 'NIL)
2. Trace: PRODUCT ==> 1
1. Trace: PRODUCT ==> 496
496
[]> (product '(1 11 111))
1. Trace: (PRODUCT '(1 11 111))
2. Trace: (PRODUCT '(11 111))
3. Trace: (PRODUCT '(111))
4. Trace: (PRODUCT 'NIL)
4. Trace: PRODUCT ==> 1
3. Trace: PRODUCT ==> 111
2. Trace: PRODUCT ==> 1221
1. Trace: PRODUCT ==> 1221
1221
[]> (product '(1 2 3 4 5 6 7 8 9 10))
1. Trace: (PRODUCT '(1 2 3 4 5 6 7 8 9 10))
2. Trace: (PRODUCT '(2 3 4 5 6 7 8 9 10))
3. Trace: (PRODUCT '(3 4 5 6 7 8 9 10))
4. Trace: (PRODUCT '(4 5 6 7 8 9 10))
5. Trace: (PRODUCT '(5 6 7 8 9 10))
6. Trace: (PRODUCT '(6 7 8 9 10))
7. Trace: (PRODUCT '(7 8 9 10))
8. Trace: (PRODUCT '(8 9 10))
9. Trace: (PRODUCT '(9 10))
10. Trace: (PRODUCT '(10))
11. Trace: (PRODUCT 'NIL)
11. Trace: PRODUCT ==> 1
10. Trace: PRODUCT ==> 10
9. Trace: PRODUCT ==> 90
8. Trace: PRODUCT ==> 720
7. Trace: PRODUCT ==> 5040
```

```
6. Trace: PRODUCT ==> 30240
5. Trace: PRODUCT ==> 151200
4. Trace: PRODUCT ==> 604800
3. Trace: PRODUCT ==> 1814400
2. Trace: PRODUCT ==> 3628800
1. Trace: PRODUCT ==> 3628800
3628800
[]> (untrace product)
(PRODUCT)
```

Task 4: iota - duplicate

Code for iota (from class)

```
( defun iota ( n )
     ( cond
          ((= n 0)
               ()
          )
          ( t
               ( snoc n ( iota ( - n 1 ) ) )
          )
     )
)
```

Demo for iota

```
[]> (trace iota)
;; Tracing function IOTA.
(IOTA)
[]> (iota 1)
1. Trace: (IOTA '1)
2. Trace: (IOTA '0)
2. Trace: IOTA ==> NIL
1. Trace: IOTA ==> (1)
(1)
[]> (iota 10)
1. Trace: (IOTA '10)
2. Trace: (IOTA '9)
3. Trace: (IOTA '8)
4. Trace: (IOTA '7)
5. Trace: (IOTA '6)
6. Trace: (IOTA '5)
7. Trace: (IOTA '4)
8. Trace: (IOTA '3)
9. Trace: (IOTA '2)
10. Trace: (IOTA '1)
11. Trace: (IOTA '0)
11. Trace: IOTA ==> NIL
10. Trace: IOTA ==> (1)
9. Trace: IOTA ==> (1 2)
8. Trace: IOTA ==> (1 2 3)
```

```
7. Trace: IOTA ==> (1 2 3 4)
6. Trace: IOTA ==> (1 2 3 4 5)
5. Trace: IOTA ==> (1 2 3 4 5 6)
4. Trace: IOTA ==> (1 2 3 4 5 6 7)
3. Trace: IOTA ==> (1 2 3 4 5 6 7 8)
2. Trace: IOTA ==> (1 2 3 4 5 6 7 8 9)
1. Trace: IOTA ==> (1 2 3 4 5 6 7 8 9 10)
(1 2 3 4 5 6 7 8 9 10)
[]> (untrace iota)
(IOTA)
```

Code for duplicate

```
(defun duplicate (n lo)
     (cond
          ((> 0 n)
               nil
          )
          ((= 0 n)
               ()
          )
          (t
               (cons lo (duplicate (- n 1) lo))
          )
     )
)
```

Demo for duplicate

```
[]> (trace duplicate)
;; Tracing function DUPLICATE.
(DUPLICATE)
[]> (duplicate 3 'boring)
1. Trace: (DUPLICATE '3 'BORING)
2. Trace: (DUPLICATE '2 'BORING)
3. Trace: (DUPLICATE '1 'BORING)
4. Trace: (DUPLICATE '0 'BORING)
4. Trace: DUPLICATE ==> NIL
3. Trace: DUPLICATE ==> (BORING)
2. Trace: DUPLICATE ==> (BORING BORING)
1. Trace: DUPLICATE ==> (BORING BORING BORING)
(BORING BORING BORING)
```

```
[]> (duplicate 9 9)
1. Trace: (DUPLICATE '9 '9)
2. Trace: (DUPLICATE '8 '9)
3. Trace: (DUPLICATE '7 '9)
4. Trace: (DUPLICATE '6 '9)
5. Trace: (DUPLICATE '5 '9)
6. Trace: (DUPLICATE '4 '9)
7. Trace: (DUPLICATE '3 '9)
8. Trace: (DUPLICATE '2 '9)
9. Trace: (DUPLICATE '1 '9)
10. Trace: (DUPLICATE '0 '9)
10. Trace: DUPLICATE ==> NIL
9. Trace: DUPLICATE ==> (9)
8. Trace: DUPLICATE ==> (9 9)
7. Trace: DUPLICATE ==> (9 9 9)
6. Trace: DUPLICATE ==> (9 9 9 9)
5. Trace: DUPLICATE ==> (9 9 9 9 9)
4. Trace: DUPLICATE ==> (9 9 9 9 9 9)
3. Trace: DUPLICATE ==> (9 9 9 9 9 9 9)
2. Trace: DUPLICATE ==> (9 9 9 9 9 9 9 9)
1. Trace: DUPLICATE ==> (9 9 9 9 9 9 9 9 9)
(9 9 9 9 9 9 9 9 9)
[]> (untrace duplicate)
(DUPLICATE)
```

Task 5: factorial - power

Code for factorial

```
(defun factorial (number &aux *list*)
     (setf *list* (iota number))
     (product *list*)
)
```

Demo for factorial

```
[]> (factorial 5)
120
[]> (factorial 10)
3628800
```

Code for power

```
(defun power (number exponent &aux *list*)
     (setf *list* (duplicate exponent number))
     (product *list*)
)
```

Demo for power

```
[]> (power 2 16)
65536
[]> (power 5 6)
15625
```

Task 6: filter-in – filter-out

Code for filter-in

```
(defun filter-in (predicate *list* &aux this function)
     (setf this (car *list*))
     (setf function (append predicate (list this)))
     (cond
          ((equal this nil)
               nil
          )
          ((eval function)
               (cons this (filter-in predicate (cdr *list*)))
          )
          (t
               (filter-in predicate (cdr *list*))
          )
     )
)
```

Demo for filter-in

```
[]> (filter-in '(< 0) '(0 1 2 3))
(1 2 3)
[]> (filter-in '(> 2) '(0 1 2 3 4))
(0 1)
[]> (filter-in '(= 5) '(1 2 3 4 5))
(5)
```

Code for filter-out

```
(defun filter-out (predicate *list* &aux this function)
     (setf this (car *list*))
     (setf function (append predicate (list this)))
     (cond
          ((equal this nil)
               nil
          )
          ((not (eval function))
               (cons this (filter-out predicate (cdr *list*)))
          )
          (t
```

```
                (filter-out predicate (cdr *list*))
            )
        )
)
```

Demo for filter-out

```
[]> (filter-out '(< 0) '(0 1 2 3))
(0)
[]> (filter-out '(> 8) '(6 7 8 9))
(8 9)
[]> (filter-out '(= 3) '(1 2 3 4 5))
(1 2 4 5)
```

Task 7: take-from

Code for take-from

```
(defun take-from (object *list* &aux this)
     (setf this (car *list*))
     (cond
          ((equal this nil)
               nil
          )
          ((equal this object)
               (take-from object (cdr *list*))
          )
          (t
               (cons this (take-from object (cdr *list*)))
          )
     )
)
```

Demo for take-from

```
[]> (take-from 'a '(a b c))
(B C)
[]> (take-from (car '(a)) '(a a b b c c))
(B B C C)
[]> (take-from '(a b c) '((a b c) (a c c) (a b b)))
((A C C) (A B B))
```

Task 8: random-permutation

Code for random-permutation

```
(defun random-permutation (*list* &aux element *rest*)
     (if (equal *list* nil)
          (return-from random-permutation nil)
     )
     (setf element (pick *list*))
     (setf *rest* (remove element *list* :count 1))
     (cons element (random-permutation *rest*))
)
```

Demo for random-permutation

```
[]> (random-permutation '(a b c d e))
(B C D A E)
[]> (random-permutation '(1 2 3 4 5))
(4 1 5 3 2)
[]> (random-permutation '())
NIL
[]> (random-permutation '((a b c) (d e f) (g h i)))
((D E F) (A B C) (G H I))
[]> (random-permutation '(1 1 1 2 2 2 3 3 3))
(1 3 3 2 1 1 2 2 3)
[]> (trace random-permutation)
;; Tracing function RANDOM-PERMUTATION.
(RANDOM-PERMUTATION)
[]> (random-permutation '(~.~ >_< ^_^ XD -_-))
1. Trace: (RANDOM-PERMUTATION '(~.~ >_< ^_^ XD -_-))
2. Trace: (RANDOM-PERMUTATION '(~.~ >_< XD -_-))
3. Trace: (RANDOM-PERMUTATION '(~.~ >_< -_-))
4. Trace: (RANDOM-PERMUTATION '(>_< -_-))
5. Trace: (RANDOM-PERMUTATION '(>_<))
6. Trace: (RANDOM-PERMUTATION 'NIL)
6. Trace: RANDOM-PERMUTATION ==> NIL
5. Trace: RANDOM-PERMUTATION ==> (>_<)
4. Trace: RANDOM-PERMUTATION ==> (-_- >_<)
3. Trace: RANDOM-PERMUTATION ==> (~.~ -_- >_<)
2. Trace: RANDOM-PERMUTATION ==> (XD ~.~ -_- >_<)
1. Trace: RANDOM-PERMUTATION ==> (^_^ XD ~.~ -_- >_<)
(^_^ XD ~.~ -_- >_<)
```

Task 9: Mapping Examples

Mapcar

```
[]> (mapcar #'car '((a b c) (d e) (f g h i)))
(A D F)
[]> (mapcar #'cons '(a b c) '(x y z))
((A . X) (B . Y) (C . Z))
[]> (mapcar #'* '(1 2 3 4) '(4 3 2 1) '(1 10 100 1000))
(4 60 600 4000)
[]> (mapcar #'cons '(a b c) '((one) (two) (three)))
((A ONE) (B TWO) (C THREE))
```

Mapcan

```
[]> (mapcan #'cons '(a b c) '((one) (two) (three)))
(A ONE B TWO C THREE)
```

Task 10: Mapping Exercises

Mapping Exercise 1

1). What is the value of the following form?
(mapcar #'expt '(2 2 2 2 2) '(0 1 2 3 4))
(1 2 4 8 16)

2). What is the value of the following form?
(mapcar #'cadr '((a b c) (d e f) (g h i) (k j l)))
(B E H J)

Mapping Exercise 2

1). What might be the value of the following form?
(mapcar #'pick '((big small) (red yellow blue green) (machine moon book)))
(SMALL YELLOW MOON)

2). What is the value of the following form?
(mapcar #'cons (iota 4) (duplicate 4 'and))
((1 . AND) (2 . AND) (3 . AND) (4 . AND))

3. What is the value of the following form?
(mapcar #'iota (iota 4))
((1) (1 2) (1 2 3) (1 2 3 4))

4. What is the value of the following form?
(mapcan #'iota (iota 4))
(1 1 2 1 2 3 1 2 3 4)

Task 11: Lisp Exercises

Code for Lisp Exercise 1

```
(defun replace-lcr (location element *list*)
     (cond
          ((equal location 'left)
               (list element (second *list*) (third *list*))
          )
          ((equal location 'center)
               (list (first *list*) element (third *list*))
          )
          ((equal location 'right)
               (list (first *list*) (second *list*) element)
          )
          (t
               nil
          )
     )
)
```

Demo for Lisp Exercise 1

```
[]> (replace-lcr 'left 'black '(red yellow blue))
(BLACK YELLOW BLUE)
[]> (replace-lcr 'right '(1 2 3) '((a b c) (d e f) (g h i)))
((A B C) (D E F) (1 2 3))
```

Code for Lisp Exercise 2

```
(defun uniform-p (*list*)
     (cond
          ((equal *list* nil)
               t
          )
          ((equal (length *list*) 1)
               t
          )
          ((equal (first *list*) (second *list*))
               (uniform-p (cdr *list*))
          )
          (t
```

```
                nil
            )
        )
)
```

Demo for Lisp Exercise 2

```
[]> (uniform-p '(red red red))
T
[]> (uniform-p '(blue green green blue))
NIL
```

Code for Lisp Exercise 3

```
(defun flush-p (*list*)
      (uniform-p (mapcar #'cdr *list*))
)
```

Demo for Lisp Exercise 3

```
[]> (flush-p '((3 . club) (queen . club) (10 . club) (king .
club) (2 . club)))
T
[]> (flush-p '((ace . clubs) (2 . diamonds) (3 . hearts) (4 .
spades)))
NIL
```

Task 12: Post solutions

If you can see this, then it's probability posted.