Q1:

```
;;return t if the n is a decimal number, otherwise return nil
(defun decimalp (n)
    (and (numberp n) (not (integerp n)))
)

;;return t if the n is a positive integer, otherwise return nil
(defun valid (n)
   (and (integerp n) (plusp n))
)

;;print the result
(defun print-triangle(n)
    (dotimes (i n)
        (dotimes (j (+ i 1))
            (format t "~A " (+ j 1))
        )
        (write-char #\linefeed)
        (write-char #\linefeed)
    )
)

;;print the message for invalid input
(defun triangle(n)
    (cond   ((decimalp n)   (print "decimal numbers are not valid input, please enter an integer"))
            ((stringp n)    (print "String are not valid input, please enter an integer"))
            ((valid n)      (print-triangle n))
            (t              (print "Please enter an positive integer."))
    )
)
```

;;return t if the n is a decimal number, otherwise return nil

(defun decimalp (n)

(and (numberp n) (not (integerp n)))

)

;;return t if the n is a positive integer, otherwise return nil

(defun valid (n)

(and (integerp n) (plusp n))

)

;;print the result

(defun print-triangle(n)

(dotimes (i n)

(dotimes (j (+ i 1))

```
        (format t "~A " (+ j 1))
      )
      (write-char #\linefeed)
      (write-char #\linefeed)
   )
)

;;print the message for invalid input
(defun triangle(n)
   (cond   ((decimalp n)   (print "decimal numbers are not valid input, please enter an integer"))
         ((stringp n)    (print "String are not valid input, please enter an integer"))
         ((valid n)      (print-triangle n))
         (t            (print "Please enter an positive integer."))
   )
)

;;test
(triangle 5)
```

Q2:

1- Using anonymous function (lambda):

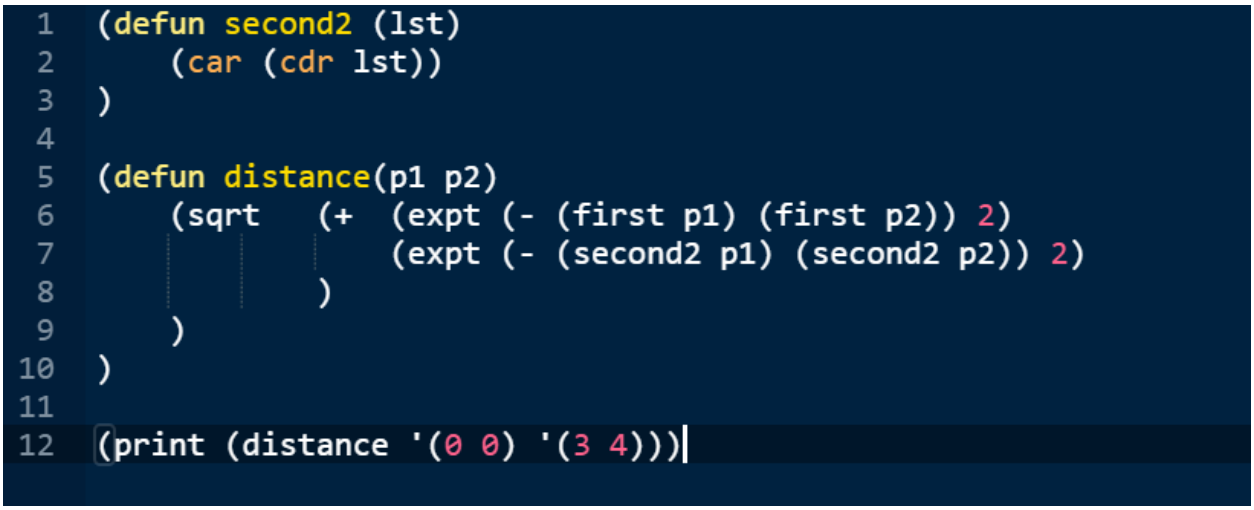
```
(defun distance(p1 p2)
    (sqrt   (+  (expt (- (first p1) (first p2)) 2)
                (expt (- ((lambda (lst) (car (cdr lst))) p1) ((lambda (lst) (car (cdr lst))) p2)) 2)
            )
    )
)

(print (distance '(0 0) '(2 2)))
```

```
(defun distance(p1 p2)
   (sqrt   (+  (expt (- (first p1) (first p2)) 2)
            (expt (- ((lambda (lst) (car (cdr lst))) p1) ((lambda (lst) (car (cdr lst))) p2)) 2)
```

```
        )
    )
)
(print (distance '(0 0) '(2 2)))
```

2- Without applying the anonymous function.



```
(defun second2 (lst)
    (car (cdr lst))
)

(defun distance(p1 p2)
    (sqrt   (+  (expt (- (first p1) (first p2)) 2)
                (expt (- (second2 p1) (second2 p2)) 2)
            )
    )
)

(print (distance '(0 0) '(3 4)))
```

Anonymous function (lambda) is more efficient, because it is not stored in memory.

Q3:

```
(defun take-out-element(lst)
    (cond   ((null lst) nil)
            ((atom (car lst)) (cons (car lst) (take-out-element (cdr lst))))
            (t (append (take-out-element(car lst)) (take-out-element(cdr lst))))
    )
)
(print (take-out-element '(a ((b (c)) d) 3.5)))

(defun remove-number(lst)
    (cond   ((null lst) nil)
            ((numberp (car lst))  (remove-number (cdr lst)))
            (t  (cons (car lst) (remove-number (cdr lst))))
    )
)
(print (remove-number '(Z F B A 5 3.5 6 7 A C)))

(defun member2(ele lst)
    (cond   ((null lst) nil)
            ((equal ele (car lst)) T)
            (t (member ele (cdr lst)))
    )
)
(print (member2 4 '(5 2 3)))

(defun remove-duplicate (lst)
    (cond   ((null lst) nil)
            ((member (car lst) (cdr lst)) (remove-duplicate (cdr lst)))
            (t (cons (car lst) (remove-duplicate (cdr lst))))
    )
)
(print (remove-duplicate '(a a s b b c d d d d 1 a c 2 1)))

(defun atom-set(lst)
    (remove-duplicate (remove-number (take-out-element lst)))
)
(print (atom-set '((z f) (b a 5 3.5) 6 (7) (a) c)))
(print (atom-set '( (n) 2 (6 h 7.8) (w f) (n) (c) n)))
```

(defun take-out-element(lst)
    (cond   ((null lst) nil)
            ((atom (car lst)) (cons (car lst) (take-out-element (cdr lst))))
            (t (append (take-out-element(car lst)) (take-out-element(cdr lst))))
)
)
(print (take-out-element '(a ((b (c)) d) 3.5)))
(print (take-out-element '((z f) (b a 5 3.5) 6 (7) (a) c)))

```
(defun remove-number(lst)
   (cond   ((null lst) nil)
           ((numberp (car lst))  (remove-number (cdr lst)))
           (t  (cons (car lst) (remove-number (cdr lst))))
   )
)
(print (remove-number '(Z F B A 5 3.5 6 7 A C)))

(defun member2(ele lst)
   (cond   ((null lst) nil)
           ((equal ele (car lst)) T)
           (t (member ele (cdr lst)))
   )
)
(print (member2 4 '(5 2 3)))

(defun remove-duplicate (lst)
   (cond   ((null lst) nil)
           ((member (car lst) (cdr lst)) (remove-duplicate (cdr lst)))
           (t (cons (car lst) (remove-duplicate (cdr lst))))
   )
)
(print (remove-duplicate '(a a s b b c d d d d 1 a c 2 1)))

(defun atom-set(lst)
   (remove-duplicate (remove-number (take-out-element lst)))
)
(print (atom-set '((z f) (b a 5 3.5) 6 (7) (a) c)))
(print (atom-set '( (n) 2 (6 h 7.8) (w f) (n) (c) n)))
```

Q4:

```
(defun last2 (lst)
    (cond   ((null lst) nil)
            ((null (cdr lst)) (car lst))
            (t (last2 (cdr lst)))
    )
)

(defun remove-last(lst)
    (cond   ((null lst) nil)
            ((null (cdr lst)) nil)
            (t (cons (car lst) (remove-last (cdr lst))))
    )
)
(print (remove-last '(a b c d)))

(defun f-l-swap(lst)
    (cond   ((null lst) nil)
            ((null (cdr lst)) (car lst))
            (t  (append (cons (last2 lst) (cdr (remove-last lst))) (list (car lst))))
    )
)
(print (f-l-swap '(a b c d)))
(print (f-l-swap '((a d) f 10 w h)))
(print (f-l-swap '(g 6 p 10 m)))

```

(defun last2 (lst)
    (cond   ((null lst) nil)
            ((null (cdr lst)) (car lst))
            (t (last2 (cdr lst)))
)
)

(defun remove-last(lst)
    (cond   ((null lst) nil)
            ((null (cdr lst)) nil)
            (t (cons (car lst) (remove-last (cdr lst))))
)
)
(print (remove-last '(a b c d)))

```
(defun f-l-swap(lst)
   (cond   ((null lst) nil)
           ((null (cdr lst)) (car lst))
           (t  (append (cons (last2 lst) (cdr (remove-last lst))) (list (car lst))))
   )
)
(print (f-l-swap '(a b c d)))
(print (f-l-swap '((a d) f 10 w h)))
(print (f-l-swap '(g 6 p 10 m)))
```

Q5:

NOTE: This program assumes the tree format in the textbook:

The list representing a tree must has EXACTLY three elements: 1. The value of this node; 2. The left subtree; 3. The right subtree. If a node has no subtree, an empty list, “()”, should be written as a placeholder.

```
(defun left-child(tree)
    (car (cdr tree))
)

(defun right-child(tree)
    (car (cdr (cdr tree)))
)

(defun isLeaf(tree)
    (and    (not (null tree))
            (null (left-child tree))
            (null (right-child tree))
    )
)

(defun less-than-right(tree)
    (cond   ((null (right-child tree)) t)
            (t (< (car tree) (car (right-child tree))))
    )
)

(defun greater-than-left(tree)
    (cond   ((null (left-child tree)) t)
            (t (>= (car tree) (car (left-child tree))))
    )
)

(defun bstp(tree)
    (cond   ((null tree) t)
            ((isLeaf tree) t)
```

```
27
28  (defun bstp(tree)
29      (cond   ((null tree) t)
30              ((isLeaf tree) t)
31              ((null (left-child tree)) (and (bstp (right-child tree)) (less-than-right tree)))
32              ((null (right-child tree)) (and (bstp (left-child tree)) (greater-than-left tree)))
33              (t      (and (less-than-right tree)
34                           (bstp (right-child tree))
35                           (greater-than-left tree)
36                           (bstp (left-child tree))))
37      )
38  )
39
40  (print (bstp '(8 (3 (1 () ()) (6 (4 () ()) (7 () ()))) (10 () (14 (13 () ()) ())))))
```

```
(defun left-child(tree)
   (car (cdr tree))
)

(defun right-child(tree)
   (car (cdr (cdr tree)))
)

(defun isLeaf(tree)
   (and    (not (null tree))
        (null (left-child tree))
        (null (right-child tree))
   )
)

(defun less-than-right(tree)
   (cond   ((null (right-child tree)) t)
        (t (< (car tree) (car (right-child tree))))
   )
)

(defun greater-than-left(tree)
```

```
   (cond   ((null (left-child tree)) t)
         (t (>= (car tree) (car (left-child tree))))
   )
)

(defun bstp(tree)
   (cond   ((null tree) t)
         ((isLeaf tree) t)
         ((null (left-child tree)) (and (bstp (right-child tree)) (less-than-right tree)))
         ((null (right-child tree)) (and (bstp (left-child tree)) (greater-than-left tree)))
         (t     (and (less-than-right tree)
                  (bstp (right-child tree))
                  (greater-than-left tree)
                  (bstp (left-child tree))))
   )
)
(print (bstp '(8 (3 (1 () ()) (6 (4 () ()) (7 () ()))) (10 () (14 (13 () ()) ()))))
```

Q6:

```
(defun decimalp (n)
    (and (numberp n) (not (integerp n)))
)

(defun sin-cos-comp (x n)
    (cond   ((decimalp n)    (print "decimal numbers are not valid input, please enter an integer"))
            ((stringp n)     (print "String are not valid input, please enter an integer"))
            ((not (plusp n)) (print "Please enter a positive integer for n"))
            ((and (oddp n) (> x -10) (< x 10)) (sin2 x n))
            ((evenp n) (cos2 x n))
            (t               (print "Please enter an positive integer for n."))
    )
)

(defun power (a n)
    (if (zerop n)
        1
        (* a (power a (- n 1)))
    )
)

(defun factorial (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))
    )
)
```

```

(defun sin2(x n)
    (let ((i 1) (result 0) (c 0))
        (loop
            (setf result (+ result (* (power -1 c) (/ (power x i) (factorial i)))))
            (when (equal i n) (return result))
            (incf i 2)
            (incf c)
        )
    )
)

(defun cos2(x n)
    (let ((i 0) (result 0) (c 0))
        (loop
            (setf result (+ result (* (power -1 c) (/ (power x i) (factorial i)))))
            (when (equal i n) (return result))
            (incf i 2)
            (incf c)
        )
    )
)

(print (sin-cos-comp 0 21))
(print (sin 0))
(print (sin-cos-comp (/ pi 2) 31))
(print (sin (/ pi 2)))
(print (sin-cos-comp pi 30))
(print (cos pi))

```

(defun decimalp (n)

(and (numberp n) (not (integerp n)))

)

```
(defun sin-cos-comp (x n)
   (cond   ((decimalp n)    (print "decimal numbers are not valid input, please enter an integer"))
         ((stringp n)     (print "String are not valid input, please enter an integer"))
         ((not (plusp n)) (print "Please enter a positive integer for n"))
         ((and (oddp n) (> x -10) (< x 10)) (sin2 x n))
         ((evenp n) (cos2 x n))
         (t              (print "Please enter an positive integer for n."))
   )
)

(defun power (a n)
   (if (zerop n)
      1
      (* a (power a (- n 1)))
   )
)

(defun factorial (n)
   (if (= n 0)
      1
      (* n (factorial (- n 1)))
   )
)

(defun sin2(x n)
   (let ((i 1) (result 0) (c 0))
      (loop
         (setf result (+ result (* (power -1 c) (/ (power x i) (factorial i)))))
```

```
        (when (equal i n) (return result))
        (incf i 2)
        (incf c)
      )
   )
)

(defun cos2(x n)
   (let ((i 0) (result 0) (c 0))
      (loop
        (setf result (+ result (* (power -1 c) (/ (power x i) (factorial i)))))
        (when (equal i n) (return result))
        (incf i 2)
        (incf c)
      )
   )
)

(print (sin-cos-comp 0 21))
(print (sin 0))
(print (sin-cos-comp (/ pi 2) 31))
(print (sin (/ pi 2)))
(print (sin-cos-comp pi 30))
(print (cos pi))
```

Q7:

a) Iterative approach

```
(defun pellnumbers (n)
    (let ((lst '()) (i 0))
        (loop
            (cond   ((= i 0) (progn (setf p2 0) (setf lst (add-at-last 0 lst))))
                    ((= i 1) (progn (setf p1 1) (setf lst (add-at-last 1 lst))))
                    (t       (progn (setf p (+ (* 2 p1) p2))
                                    (setf p2 p1)
                                    (setf p1 p)
                                    (setf lst (add-at-last p lst))))
            )
            (when (= i n) (return lst))
            (incf i)
        )
    )
)

(defun add-at-last (ele lst)
    (cond   ((null lst) (cons ele lst))
            (t (cons (car lst) (add-at-last ele (cdr lst))))
    )
)

(print (pellnumbers 50))
```

(defun pellnumbers (n)

(let ((lst '()) (i 0))

(loop

(cond ((= i 0) (progn (setf p2 0) (setf lst (add-at-last 0 lst))))

((= i 1) (progn (setf p1 1) (setf lst (add-at-last 1 lst))))

(t (progn (setf p (+ (* 2 p1) p2))

(setf p2 p1)

(setf p1 p)

(setf lst (add-at-last p lst))))

)

(when (= i n) (return lst))

(incf i)

)

```
    )
)

(defun add-at-last (ele lst)
    (cond   ((null lst) (cons ele lst))
            (t (cons (car lst) (add-at-last ele (cdr lst))))
    )
)

(print (pellnumbers 50))
```

b) Recursive approach

```
(defun pellnumbers (n)
    (reverse2 (pellnumbers2 n))
)

(defun pellnumbers2 (n)
    (cond   ((= n 0) '(0))
            ((= n 1) '(1 0))
            (t  (let    ((lst (pellnumbers2 (- n 1))))
                        (cons (+ (* 2 (car lst)) (car (cdr lst))) lst)
                )
            )
    )
)

(defun reverse2 (lst)
    (cond   ((null lst) '())
            (t (append (reverse2 (cdr lst)) (list (car lst))))
    )
)

(print (pellnumbers 50))
```

```
(defun pellnumbers (n)
    (reverse2 (pellnumbers2 n))
)
```

```
(defun pellnumbers2 (n)
   (cond   ((= n 0) '(0))
           ((= n 1) '(1 0))
           (t  (let    ((lst (pellnumbers2 (- n 1))))
                       (cons (+ (* 2 (car lst)) (car (cdr lst))) lst)
               )
           )
   )
)

(defun reverse2 (lst)
   (cond   ((null lst) '())
           (t (append (reverse2 (cdr lst)) (list (car lst))))
   )
)

(print (pellnumbers 50))
```