2009 November 30

# Lisp Basic
# Example Test Questions

**1.**

Assume the following forms have been typed into the interpreter and evaluated in the given sequence.

```
( defun a ( y ) ( reverse y ) )
( setq a ‘(1 2 3) )
( setq c ‘d )
( setq b ‘e )
( set c ‘b )
( setq f  ‘( ( a ) b ) )
( defun b ( x ) ( mapcar #’list x ) )
( setq e ( ( lambda ( x ) x ) ‘a ) )
```

What will the following forms evaluate to? State **error** if the form does not evaluate correctly and explain why?

1. ( a  a )
2. ( funcall  ‘a  (list c  b  d))
3. ( apply  ‘a   ‘(a b c))
4. ( eval  ‘a  ‘(a b c) )
5. ( eval  e )
6. ( b  f )
7. (mapcar  ‘b  ‘( (1 2 3) (2 3) (3) ) )

**2.**

Write a recursive function **interleave (list-1 list-2)** that returns a list in which the elements of list-1 are interleaved with list-2. The following are examples. Use only the Lisp functions from the list: cond, list, append, cons, car, cdr, +, atom, numberp, null.

```
(interleave ‘(a (b b) c) ‘(1 2 3 4 5)) → (a 1 (b b) 2 c 3 4 5)
(interleave ‘(1 2 3 4 5) ‘(a (b b) c)) → (1 a 2 (b b) 3 c 4 5)
```

**3.**

Assume the following forms have been typed into the interpreter and evaluated in the given order.

```
( defun a ( y ) ( reverse y ) )
( setq a ‘(x y z) )
( defun b ( x ) ( mapcar #‘list x ) )
( setq b ‘e )
( setq c ‘d )
( set c ‘b )
( setq e  #‘(lambda ( x ) x ) )
```

What will the following forms evaluate to? If the form does not evaluate correctly, state **error** and give a reason why?

1. ( a  a )

2. ( funcall ‘a (list a c b))
3. ( apply ‘a ‘(a b c))
4. ( eval c)
5. ( funcall (eval b) a )
6. (mapcar ‘b (maplist e a ) )

**4.**

Write and document a recursive Lisp definition for the mapcar function that can have an arbitrary number of input lists. The mapcar functions stops as soon as any one of the input lists becomes empty. Do not use global variables. Do not use let, prog and similar features to introduce local variables; use only parameters to functions as local variables. You may use support functions.

**5.**

Define a recursive Lisp function, **simple**, which takes one argument, a list of numbers and atoms and returns a list containing two items. The first is the sum of the numbers and the second is the number of non-number atoms. For example:
(simple ‘(a 2 3 b d 1 c 4) ) → (10 4)
(simple ‘(a b () ) ) → (0 3)
For full marks, use only one function,; do not use a helper function. Use only the Lisp functions cond, list, append, cons, car, cdr, +, atom, numberp, null.

**6.**

Assume the following forms have been typed into the interpreter and evaluated in the given sequence.

( defun a ( y ) ( list y ‘y ) )

( setq a ‘c )

( defun b ( x ) ( maplist #’list x ) )

( setq c ‘d )

( setq e ( ( lambda ( x ) x ) ‘a ) )

( setq b ‘e )

( set c ‘b )

( setq f ‘( ( a ) b ) )

What will the following forms evaluate to?

1. ( a a )
2. ( funcall e ‘c)
3. ( cons a b )
4. ( eval d )
5. ( eval ( eval (eval (eval (quote e ) ) ) ) )
6.
7. (mapcar ‘b ‘( ( 1 ) ( 2 ) ( 3 ) ) )

**7.**

Write a recursive function, `(defun index (???) ???)` such that a call of the form `(index array indexList)` returns the array element `array[I1,I2,...,In]`, where indexlist = (I1,I2,...,In). Assume no errors; i.e. no out of bound indices. There is no fixed size for the number of dimensions. Use the function `pth` from part B. Do not use length, last, butlast, etc., use first (or car) and rest (or cdr).

**8.**

Write a your own recursive version `myMaplist` of the maplist function. If possible, do not define additional functions but it is better to have them with a correct and commented function than have an incorrect function. Hints: Recall the functions some and every. Maplist terminates when one of the input lists becomes nil.

**9.**

**B** Write a recursive function, `(defun nth (pos list) ???)`, that returns the n'th item from a list. Assume the list has at least n items. `(nth 1 aList)` is to return the first item in `aList`.

**10.**

Write a recursive function, `(defun diagOf(theMatrix) ...)` to return the diagonal of a square matrix. Assume the input is error free. You may write support functions. Do not use global variables. Do not use let, prog and similar features to introduce local variables; use only parameters to functions as local variables.

**11.**

**A** What is the result of evaluating the following Lisp expression? Explain your answer

```
 (setq a '(cons a a))
```

**B** Given the following function definition.

```
                (defun oh (condition thenPhrase elsePhrase)
                  (cond (condition thenPhrase)
                        (t elsePhrase)))
```

Explain the following

```
>(setq a 4)
4
>(oh (< 4 5) (setq a (1+ a)) (setq a (1- a)))
5
>a
4
>(oh (> 4 5) (setq a (1+ a)) (setq a (1- a)))
4
>a
4
```

**12.**

**C** Suppose the following function is defined `(defun test (x) (reverse x))`. Will the expression `(apply 'test '(a b c))` execute correctly. If yes, explain why. If not, explain why and modify the expression so it works.

**C** Suppose the following function is defined `(defun test (x) (reverse x))`. Will the expression `(funcall 'test (list a b c))` execute correctly. If yes, explain why. If not, explain why and modify the expression so it works.
`(eval '(test (a b c)))`

**13.**

The following Lisp function is defined where the function reverse reverses the list `x`.

```
(defun test (x) (reverse x))
```

Will the following expressions execute correctly. If yes, explain why. If not, explain why not and modify the expression so it works.

```
(apply 'test '(a b c))

(funcall 'test (list 'a 'b 'c))

(eval 'test '(a b c))
```

```
(eval '(test (a b c)))
```

**14.**

Write a recursive Lisp function **flatten(alist)** to return all the atoms, except nil, at all levels in **alist** as a single level list while retaining their order. The following are examples.

```
(flatten '(A (B (C D) E) (F G))) ⇒ (A B C D E F G)
(flatten '(1 () 2)) ⇒ (1 2)
(flatten '(1 () (()) ( 2 () 3))) ⇒ (1 2 3)
```

For this question, both parts A and B, you may use only the following Lisp functions:
append, atom, car, cdr, cond, cons, defun, list, null, +, – and =.
You may use combinations of car and cdr such as caddr.

**15.**

Lower-atoms was called deeper-atoms in F01 and F04

**A** Write a recursive program, **lower-atoms**, that embeds each atom, except nil, in a list containing only the atom. Use only the basic Lisp operators: car, cdr, cons, cond, null and atom.

Examples: `(a) → ((a))` `(()) → (())` `(a () b) → ((a) () (b))`
`(a b (c (b d)) (e)) → ((a) (b) ((c) ((b) (d))) ((e)))`

**B** Write a recursive program, **lift-atoms**, that lifts each atom, except nil, that is in a list containing only the atom up one level. This is the inverse of part A. Use only the basic Lisp operators: car, cdr, cons, cond, null and atom.

Examples: `((a)) → (a)` `(()) → (())` `((a) () (b)) → (a () b)`
`((a) (b) ((c) ((b) (d))) ((e))) → (a b (c (b d)) (e))`

**16.**

You may write support functions, although you get lower evaluation if you do. Do not use global variables. Do not use let, prog and similar features to introduce local variables; use only parameters to functions as local variables.

**A** Write a recursive function, `(defun myinter (set1 set2) ... )`, that computes the set intersection of set1 and set2. Use the member function – `(member item list)`, it returns the sublist beginning at the item, if item is in the list and returns nil otherwise.

**17.**

Write a recursive function, `remove-nth` that removes the `n`'th item from every `list` at all levels. Counting begins at 1. You cannot use any implicitly recursive function, such as `mapcar`, `length`, etc., except for `append`. You must use the following definitions of `prefix` and `suffix` in your definition.

(prefix 3 (10 20 30 40 50)) → (10 20 30) ;; First 3 items.
(suffix 3 (10 20 30 40 50)) → (30 40 50) ;; Last 3 items.

Use `car`, `cdr` and `cons` for the basic list operations, and `cond` for conditional statements. You are permitted to and will need to write recursive support functions.

Precondition: n ≥ 1.

`(defun remove-nth (n list) ;;` You supply the rest

**18.**

Write a recursive function, `remove-nth` that removes the `n`'th element from every `list` at all levels. Counting begins at 1. You cannot use any implicitly recursive function, such as `mapcar`, `length`, etc., except for `append`, and the following functions `prefix` and `suffix`. You must use `prefix` and `suffix` in your definition.

(prefix 3 (10 20 30 40 50)) → (10 20 30)
(suffix 3 (10 20 30 40 50)) → (30 40 50) Next time define 3 as the position not length.

Use `car`, `cdr` and `cons` for the basic list operations, and `cond` for conditional statements. You are permitted to and will need to write recursive support functions.

Precondition: n ≥ 1

```
(defun remove-nth (n list)  ;; You supply the rest
```

**19.**

Write a recursive function, `insert-nth` that inserts `item` as the `n`'th element into every `list` at all levels. Counting begins at 1. You cannot use any implicitly recursive function, such as **mapcar**, **length**, etc. Use **car**, **cdr** and **cons** for the basic list operations. You are permitted to and will need to write recursive support functions.

Precondition: n ≥ 1 and n ≤ 1+length(shortest list at any level in `list`)

```
(defun insert-nth (item n list)  ;; You supply the rest
```

**20.**

Write a recursive function, `removeContig(theList)` that reduces all instances of a contiguous sequence of identical items to one instance at all levels of `theList`. You cannot use any implicitly recursive function, such as `mapcar`, `length`, etc. Use `cond,` `car`, `cdr` and `cons` for the basic list operations. You are permitted to write recursive support functions.

Precondition: `theList` is a list

```
(defun removeContig (theList)  ;; You supply the rest
```

**21.**

Assume the following forms have been typed into the Lisp interpreter and evaluated.

```
( defun a ( x ) ( values (list x) x ) )

( setq a '( a b ) )

( defun b ( x ) `( x ,x ) )

(setq b ( cdr a ) )

( setq c ( car a ) )

( setq d c )

( setq e ( (lambda ( x ) ( list x ) ) d ) )
```

What will the following forms evaluate to?

1. ( cons c ( car a ) )
2. ( cons e b )
3. ( eval a )
4. ( let (( a b ) ( y a )) ( append a y ) )
5. ( multiple-value-list ( a a ) )
6. ( b c )
7. ( set ( car a ) (cdr a) )
8. ( setf ( car a ) (cdr a) )     a =

**22.**

**B.** The following forms are entered into the Lisp interpreter and evaluated.

```
(defun f1 ( v1 ) ( f2 v1 ) )
```

```
(defun f2 ( v2 ) ( 1+ v1 ) )
```

Under an environment with static scoping what do the following forms evaluate to?

1. ( f1 6 )
2. ( setq v1 10)
   ( f1 6)
3. For 2 above what does the environment look like in f2?

Under an environment with dynamic scoping what do the following forms evaluate to?

1. ( f1 6 )

2. ( setq v1 10)
   ( f1 6)

For 2 above what does the environment look like in f2?

**23.**

**A.** Write a recursive Lisp function **nodups(alist)**, which takes one argument, a list, and returns a list with any consecutive identical top-level items removed.

For example:

| | | |
|---|---|---|
| (nodups '( a a a b c c d ) ) | should return | ( a b c d) |
| (nodups '( a a b a c c ) ) | should return | ( a b a c) |

Note: for this question, you may use only the Lisp functions defun, cond, null, car, cdr, cons, and equal. You may use combinations of car and cdr such as caddr.

**B.** Write a recursive Lisp function **dups(alist, N),** which takes two arguments, a list and an integer, and returns a list with every top-level item in **alist** duplicated N times. Assume N ≥ 1.

For example:

| | | |
|---|---|---|
| (dups '( a b c ) 3 ) | should return | ( a a a b b b c c c) |
| (dups '( a ( ) ( a ) ) 2 ) | should return | ( a a nil nil ( a ) ( a ) ) |

Note: for this question, you may use only the Lisp functions defun, cond, null, car, cdr, cons, append, list, +, - and =. You may use combinations of car and cdr such as caddr.

**24.**

You may write support functions, although you get lower evaluation if you do. The only functions you may use are the Lisp functions car and cdr and longer abbreviations, cons, cond, equal, atom, null and specific functions mentioned in each part.

**B** Define a recursive Lisp function, `dup`, which checks whether its argument is a list containing two successive elements at the top level that are equal.

```
(dup '(A B B C) ⇒ t
(dup '(A (B) B C) ⇒ nil
```

**25.**

]Consider the following Lisp function.

```
(defun mystery (s)
 (cond ((null s) 1)
       ((atom s) 0)
       (t (max (+ mystery (first s)) 1)
              (mystery (rest s)))))
```

```
    ) )
```

Explain in general what the function mystery returns.