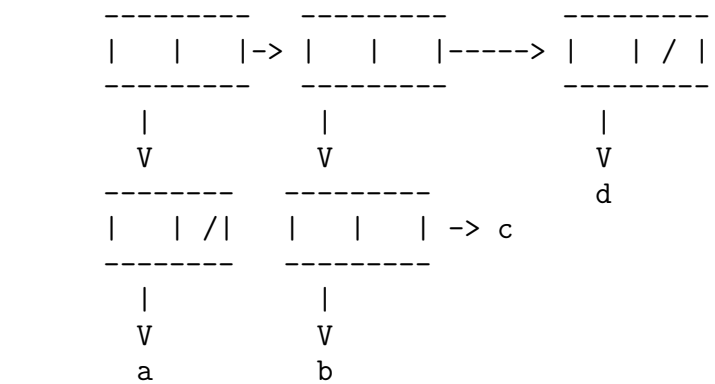
Midterm Examination, Selected Solutions, Cmput 325

** Show an S-expression that is stored internally by the following structure.

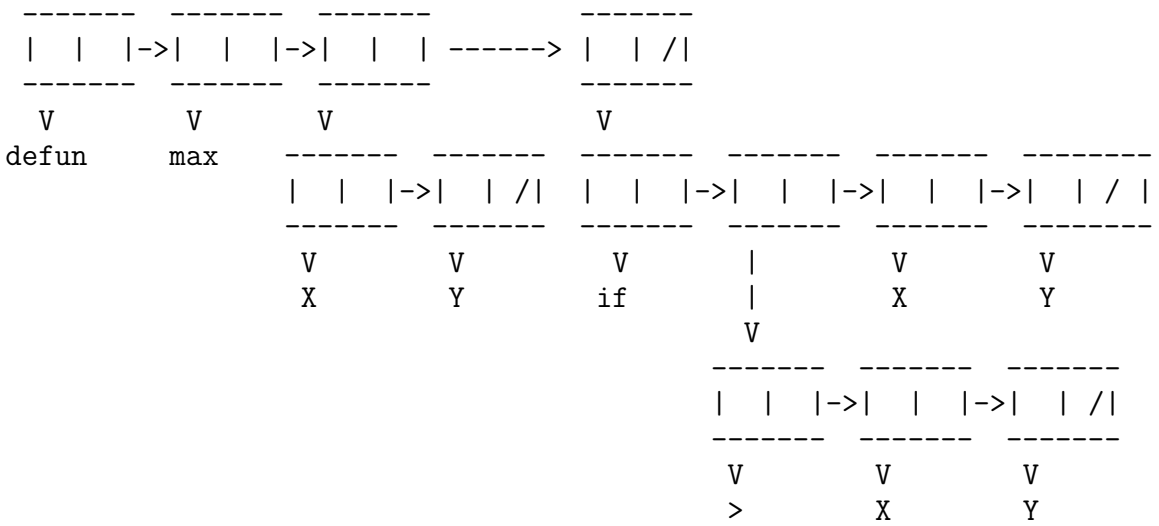


Answer: ((a) (b . c) d)

** When we define a function in Lisp using defun, the definition has to be stored before it can be processed. Draw the machine level representation of the following expression.

```
(defun max (X Y) (if (> X Y) X Y))
```

Answer: (We may just use V for down arrow.)



** Recall that boolean constants $true$, $false$, and operator NOT are defined as: $T = (\lambda xy \mid x)$, $F = (\lambda xy \mid y)$, and $NOT = (\lambda x \mid xFT)$, respectively. Simplify the following expression (show all the steps).

$$
\begin{aligned}
& F\ T\ F\ a\ (NOT\ F\ b\ c) \\
& \rightarrow F\ a\ (NOT\ F\ b\ c) \\
& \rightarrow (NOT\ F\ b\ c) \\
& \rightarrow T\ b\ c \\
& \rightarrow b
\end{aligned}
$$

** Consider a boolean operator, denoted by OP, which has the following truth table.

X	Y	OP	X Y
T	T		T
T	F		F
F	T		F
F	F		T

Define a lambda expression for OP, and simplify it if possible. And verify that your definition works by applying it to at least two of the four cases below. Hint: OP is the negation of *exclusive or*.

The table says, if x is true then the truth value of OP of x and y is y; otherwise it is the negation of y. Thus,

$$OP = (\lambda xy \mid xy(NOTy))$$

For example,

$$OP\ F\ T \rightarrow F\ T\ (NOT\ T) \rightarrow NOT\ T \rightarrow F$$

Similarly, one can show $(OP\ T\ F) \rightarrow ... \rightarrow F$.

[5 marks] Compile the following expression to SECD code.

```
(- (* 5 4) (+ 2 4))
```

answer: `(LDC 4 LDC 2 + LDC 4 LDC 5 * -)`

** For the lambda expression below (we draw some underlines to help you identify its components), evaluate it by the interpreter based on context and closure by answering the corresponding questions. The initial context is assumed to be CT0 = $\{z \rightarrow 4\}$. Clearly indicate your answer. You don't need to show how you get your answer. Assume the evaluation starts from

```
eval
    ((lambda (f x y) (f (f x y) z)) (lambda (w v) (+ w v)) 3 5)
     ----------------------------- ---------------------- - -
in CT0
```

When `(f (f x y) z)` is evaluated, we get

```
eval (f (f x y) z)
     in {f -> [(lambda (w v) (+ w v)), CT0], x -> 3, y ->5} U CT0   (a)
```

Let CT1 denote the above context. When `(f x y)` is evaluated, we get

```
     eval f in CT1  ==> the closure bound to f                      (b)
     eval the argument (f x y) in CT1                               (c)
     eval the argument z in CT1.                                    (d)
```

The expression `(+ w v)` will be evaluated twice, i.e., the interpreter will make recursive calls, twice with `(+ w v)` as the expression to be evaluated. Before we apply function f above in (b), (c) and (d) turn out to be

```
For (d) above, we have
    eval z in CT1 ==> 4

For (c) above
    eval (f x y) in CT1 goes as follows.  We get bindings of variables
    f, x, and y, and then eval the body of the function in the extended context;
    the body is obtained from the closure bound to f, so is the existing context,
    i.e.,
              eval (+ w x) in {w -> 3, v -> 5} U CT0

   This is the first call in the question, which leads to 8.  Note that this
   is the result of (c)

Now both arguments in (a) are fully eval'd, and the next step is

     eval (+ w v) in {w -> 8, v -> 4} U CT0

This is the second call to (+ w v) in the question.
```

** Consider the following lisp program.

```
(defun f (L1 L2)
    (if (null L1)
        (let ((s (g L2))) (- 0 s))
        (+ (car L1) (f (cdr L1) L2))))

(defun g (L)
    (cond
        ((null L) 0)
        ((null (cdr L)) (car L))
        (t (+ (cadr L) (g (cddr L))))))
```

Show the result of evaluating each expression below.

```
(g '(1 2 3))                        answer:  5
(g '(1 2 3 4))                      answer:  6
(f '(1 2 3) '(2 4))                 answer:  2
(f '(5 3 6) '(1 4 2))               answer:  8
```

[6 marks] Consider the following lisp program:

```
(defun h (L)
   (cond
       ((null L) T)
       ((atom L) (not L))
       (t (h (h (cdr L))))
   )
)
```

Show the result of evaluating each expression below.

```
(h '(a b c d))                  answer: T
(h '((a b) (b (e)) d))          answer: NIL
```

**Suppose we have the following definitions in Lisp.

```
(defun filter (p L)
         (if (null L)
             nil
             (if (funcall p (car L))
                 (cons (car L) (filter p (cdr L)))
                 (filter p (cdr L)))))
(defun g5 (x) (> x 5))
(defun c3 (x) (if (> x 3) (+ x 1) (- x 1)))
```

Show the result of evaluating each expression below.

```
        (filter 'g5 '(7 2 8 4 5 6))

  your answer:  (7 8 6)

        (mapcar 'c3  (filter 'numberp '(a a b 3 8 d 6)))

  your answer:  (2 9 7)
```

** Write a Lisp function `(defun complement (S1 S2) ...)`, where `S1` and `S2` are lists of atoms such that `S2` is a subset of `S1`, and the function returns the list of those atoms that are in `S1` but not in `S2`. The order of the resulting list is unimportant. E.g.,

```
      (complement '(a b c d e) '(a c)) ==> (b d e)
```

As `S1` and `S2` are supposed to represent sets, assume that neither may contain duplicate elements.

```
(defun complement (S1 S2)
    (cond  ((null S1) nil)
           (t  (if (memberp (car S1) S2)
                   (complement (cdr S1) S2)
                   (cons (car S1) (complement (cdr S1) S2))))))
```

** Define a Lisp function

```
(defun rotate (L) ...)
```

The function moves the first element of L to the end of the list, and every other element to its left. Note that although an element of a list may be an atom or a sublist, no sublist should be rotated. In the case where L is an empty list, NIL should be returned. E.g.

```
(rotate '(a b (1 2) (c d)))  ==> (b (1 2) (c d) a)
```

```
(defun rotate (L)
  (if (null L)
      nil
      (append (cdr L) (cons (car L) nil))) )
```

** Extend the above function so that every sublist is also rotated. Call the resulting function rotateAll. For example,

```
(rotateAll '(a b (1 2 3 4) (c d)))  ==> (b (2 3 4 1) (d c) a)
(rotateAll '((a b) (1 2 3 4) c d))  ==> ((2 3 4 1) c d (b a))
(rotateAll '(a b (1 2 (3 4))))      ==> (b (2 (4 3) 1) a)
```

Hint: You may need to decompose the function in order to make your solution clean and understandable.

```
(defun rotateAll (L)
   (cond ((null L) NIL)
         ((atom L) L)
         (t (append (rotateRest (cdr L)) (cons (rotateAll (car L)) nil)))))

(defun rotateRest (L)
   (cond ((null L) nil)
         (t (cons (rotateAll (car L)) (rotateRest (cdr L))))))

; if you use mapcar,  the code can be extremely compact.

(defun rotateAll0 (L)
     (if (atom L)
          L
         (totate (mapcar 'rotateAll0 L))))


; acknowledgement: This excellent encoding is taken from a student's answer given
   in the midterm.
```