1. Understanding Lisp code (20 marks total)

1.1 Consider the following Lisp function:

```
(defun g (L)
        (if (null L)
                1
                (+ (car L) (g (cdr L)))
        )
)
```

Show the result of evaluating each of the following expressions (2 marks each). Assume that the function + is defined only for arguments that evaluate to numbers, and gives an error otherwise.

1.1.1 (g '(1 2 3))

7

1.1.2 (g nil)

1

1.1.3 (g '(nil nil))

error, nil not a number

1.1.4 (g (list (g '(1 2 3)) (g nil)))

9

1.2 Consider the following Lisp function:

```
(defun f (L)
        (if (null L)
                nil
                (cons  (cons (f (cdr L)) nil)
                        (cons (car L) nil)
                )
        )
)
```

Show the result of evaluating each of the following expressions (2 marks each).
Hint: be careful near the end of the recursion, and make sure to do all the cons calls.

1.2.1 (f '(a))

((nil) a)

1.2.2 (f '(a b))

((((nil) b)) a)

1.2.3 (f '(a b c))

((((((nil) c)) b)) a)

1.3 What is the result of evaluating the following expressions? (3 marks each)

1.3.1

Let function h be defined by (defun h (x) (list x x)).

Expression: (mapcar 'h '((a b) c))

(((a b) (a b)) (c c))

1.3.2

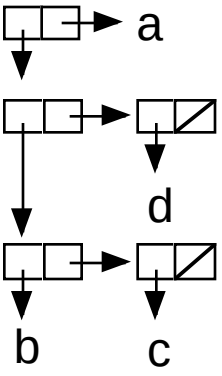
Let function l12 be defined by (defun l12 (L) (< (car L) (cadr L)))

Expression: (filter 'l12 '((1 2 3) (3 2 1) (1 3 2) (3 1 2)))

((1 2 3) (1 3 2)))

2. Machine representation of Lisp (10 marks total)
2.1 (2x3 marks) For the given diagram showing a machine representation, write the corresponding S-expression both in full dotted-pair form and in the simplest form.

Remark: in the diagram, nil is represented by a crossed-out box ☒.



2.1.1 full dotted-pair form:

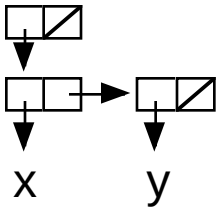
(((b.(c.nil)).(d.nil)).a)

2.1.2 simplest form:

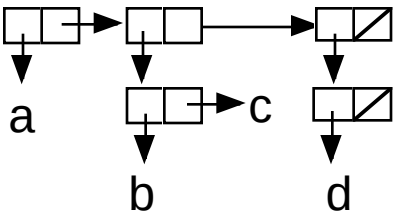
(((b c) d).a)

2.2 (2x2 marks)
Draw the diagram showing the machine representation of the following S-expressions:

2.2.1 ((x y))



2.2.2 (a (b.c) (d))



3. Writing Lisp code (20 marks total)
For 3.1 and 3.2, use only the functions *car, cdr, cons*. Do not use the *list* function.

3.1 Write the lisp code that constructs the following symbolic expressions. (2x2 marks)

3.1.1 (a (b) c)

(cons 'a (cons (cons 'b nil) (cons 'c nil)))

3.1.2 ((1 2) (nil))

(cons (cons '1 (cons '2 nil)) (cons (cons nil nil) nil))

3.2 Write the Lisp code that returns a given element from the expression. (2x3 marks)

3.2.1 Code that returns **x** from ((a b **x**) c (y))

(caddar '((a b x) c (y)))

3.2.2 Code that returns **y** from ((a b x) c (**y**))

(caaddr '((a b x) c (y)))

3.3 Writing Lisp functions

3.3.1 (5 marks) Write a Lisp function **even** that tests whether a given list has even length. Your function should return **T** for even length lists and **nil** for lists of odd length. An empty list (with length 0) has even length.

Examples: (even '(5 8 11 17)) --> T
(even '(1 2 3 5 4)) --> nil

```
(defun even (L)
        (if (null L) T
                (not (even (cdr L)))
        )
)
```

3.3.2 (5 marks) Write another function, **alleven**, that takes a list of lists as an input, and tests whether all these lists have even length.

Examples: (alleven '((a b) nil (e f g h))) --> T
(alleven '((1 2) (4) (c d) (3 5))) --> nil. because the length of (4) is odd.

```
(defun alleven (L)
        (if (null L) T
                (and (even (car L) (alleven (cdr L)))
        )
)
```

An elegant solution using reduce and mapcar:

```
(reduce 'and (mapcar 'even L))
```

Note: this works even if L is empty. See the definitions of *reduce* and *and* in the Liusp manual. Another way is to supply the identity element of *and*, which is *T*.

4. Lambda Expressions (10 marks total)

Consider the lambda expression:

```
((lambda (x y) (x y)) (lambda (y) (* y 3)) 7)
```

Assume that the expression is evaluated in initial context CT0.

4.1 (4 marks) What is the context at the time when (x y) is evaluated?
{x→[(lambda (y) (* y 3)),CT0],y→7} u CT0

4.2 (4 marks) What is the context at the time when (* y 3) is evaluated?
{y→7} u CT0

4.3 (2 marks) What is the final result of reducing the expression?
21

Derivation:
Expression is an application.
fun = (lambda (x y) (x y))
arg1 = (lambda (y) (* y 3))
arg2 = 7

eval fun in CT0 = closure [fun,CT0]
eval arg1 in CT0 = closure [arg1,CT0]
eval arg2 = 7

bind x→[arg1,CT0]
bind y→7
extend context: CT1 = {x→[arg1,CT0],y→7} u CT0
eval body of fun in CT1:
eval (x y) in CT1. (This gives answer to 4.1)

(x y) is application.
eval fun: eval x in CT1 = [arg1,CT0]
eval arg: eval y in CT1 = 7
bind y (from arg1) y→7
extend context CT0 found in closure x: CT2 = {y→7} u CT0
eval body in CT2: eval (* y 3) in CT2: (* 7 3) = 21

5. Reductions and Church-Rosser Theorem (10 marks total)

As in the lecture notes, we use $\rightarrow$ to denote a sequence of zero or more reduction steps. Which of the following statements are true, and which are not? Why?
1 mark for each right answer plus 1 mark for each *clear* explanation of a correct answer.

5.1 if $A \rightarrow B$ and $A \rightarrow C$, then there is a normal order reduction $B \rightarrow C$.
no. For example if B is already in normal form and C is not, then normal order reduction of B cannot change B.

5.2 if $A \rightarrow B$ and $B \rightarrow C$, then there is a reduction $C \rightarrow A$.
no. Reductions are not reversible. e.g. If A is a complex expression and C is a constant such as 7, there is no way back from 7 to A.

5.3 if $A \rightarrow B$ and $A \rightarrow C$, then there is a D such that $B \rightarrow D$ and $C \rightarrow D$.
Yes. Guaranteed by Church-Rosser theorem part 1.

5.4 If there is an applicative order reduction $A \rightarrow B$, and B is in normal form, then there is also a normal order reduction $A \rightarrow B$.
Yes. Guaranteed by Church-Rosser theorem part 2.

5.5 A sequence of normal order reductions finds a normal form for any expression.
No. see counterexample in lecture notes.

6. Lisp Interpreter (10 marks total)

You are in the middle of a Lisp evaluation. The current context is given by the name list `n = ((x y) (x) (z y))` and the value list `v = ((1 2) (3) (4 5)).`
Show how the Lisp interpreter **eval** evaluates the following expressions. Show *each* recursive call to **eval**, as well as the final result.

6.1 (1 mark) eval[(+ x 1), n, v]
eval[x, n,v] = assoc(x,n,v) = 1
eval [(+1 1)] = 2

6.2 (1 mark) eval[(quote y), n, v]
y

6.3 (1 mark) eval[(quote (quote z)), n, v]
(quote z)

6.4 (2 marks) eval[(if (atom x) y z), n, v]
eval[(atom x), n,v]
eval[x,n,v] -> 1
eval[(atom x), n,v]-> T
eval[y,n,v] -> 2

result = 2

6.5 (2 marks) eval[(car (quote (1 2 3))), n, v]
(car eval[(quote (1 2 3)), n, v])
eval[(quote (1 2 3)), n, v] = (1 2 3)
(car (1 2 3)) = 1

6.6 (3 marks) eval[((lambda (x) (+ x 1)) 3), n, v]
application: eval[body(c), cons(parms(c), names(c)), cons(z, values(c))]
where c = eval[e, n, v] and z = evlis[(e1 ... ek), n, v]
eval fun:
c = eval[(lambda (x) (+ x 1)), n, v] = cons(cons((x),(+x 1), cons(n,v))
= ((x).(+ x 1)).(n.v)) [or: ((x) + x 1).(n.v), or: same with n, v written out]
evlis args:
z = evlis((3),n,v) = (3)
-> evlis calls eval[3, n, v] = 3

n1 = ((x).n) , v1 = ((3).v)
eval[(+x 1),n1,v1]
eval[x,n1,v1] = assoc(x,n1,v1) = 3
(+ 3 1) = 4