PROBLEMS SHEET 1

1. Write the sequences of CAR & CDR expressions that retrieve the X from the following lists. Do not attempt to do this by trial & error, work through it on paper first. Don't forget the quote character in front of the lists when you try your solutions out.

1.1: (a X b c)
1.2: (a b X c)
1.3: ((X))
1.4: (a (b (X (c))))
1.5: (a (b (X)) c)

2. Using only the following symbols: cons 'a 'b 'c () Write the expressions which build the following lists:

2.1: (a b c)
2.2: (a (b c))
2.3: ((a) b c)

3. LENGTH is a CL function that takes a list as its only argument and returns as its value the length of that list. So LENGTH applied to '(a b c d) gives a value of 4. Look at the lists in problem-2 and 1.4 & 1.5 in problem-1, noting what you think their lengths should be, then check using LENGTH. If you find the answer is not what you had expected make sure you know why.

4. Write the following functions

4.1: INC-NUM
takes 1 arg which is a number
returns the number incremented by 1

```
eg:
    (inc-num 5)  =>  6
```

4.2: INC-1ST
takes 1 arg which is a list of numbers
returns the list with the first number incremented

```
eg:
    (inc-1st '(1 2 3 4))  =>  (2 2 3 4)
```

4.3: BRACKET-HEAD
takes a list as its arg
returns the list with extra brackets round the first item

```
eg:
    (bracket-head '(a b c))  =>  ( (A) B C )
```

PROBLEM SHEET 2.

1. Write a function that takes two lists as its args & returns the longest as its result. (hint: nothing too elaborate, see Fn bigger in the notes)

2. Write a function which takes one integer as its argument and calculates its factorial. (hint: Q2 & Q3 require repetitive solutions)

3. Write a function that takes a list of numbers as its argument and returns the sum of that list of numbers, eg:
(sum-list '(1 3 7 5)) => 16

PROBLEM SHEET 3.

3.1 In last week's tutorial exercises Q.2 & Q.3 (problem sheet 2) asked you to develop two repetitive Fn.s

Q2. Write a function which takes one integer as its argument and calculates its factorial.

Q3. Write a function that takes a list of numbers as its argument and returns the sum of that list of numbers, eg:
(sum-list '(1 3 7 5)) => 16

You probably used a tail recursive approach with an &optional argument. Rewrite these functions using a head recursive approach and compare trace information for the two.

3.2. Develop a function which takes a list of integers as its only argument and returns a list of the factorials of those numbers. (hint: look at inc-list in the notes).
eg: (fac-lis '(2 4 3)) => (2 24 6)

Keep the numbers small and try tracing both this function and the factorial function.

3.3. Write a function that takes a list as its only argument & returns that list after enclosing every item in it in an extra set of brackets.
ie: (brack-lis '(b l o b)) => ((b) (l) (o) (b))

PROBLEM SHEET 4.

There are some awkward problems on this sheet, you should tackle them in order as I have tried to get one to lead on to the next. You should not expect to finish this sheet in 1 hour tutorial.

4.1 Write a Fn, replace, which globally replaces some specified item in a list. Replace should take 3 args:

(a) the item to be replaced
(b) the new item
(c) the list

eg:

```
(replace 'the 'a '(the cat sat on the mat))
==>  (a cat sat on a mat)
```

4.2 If you have managed 4.1 then you have almost certainly developed a head recursive Fn. Why is tail recursion inappropriate for this type of problem. Try making the Fn tail recursive & look at its output.

4.3 Write a tail recursive reverse Fn to reverse the order of items in a list. Note CL has a Fn called "reverse" so you will have to call it something else.

4.4 APPEND is a Fn which takes 2 lists and concatenates them, ie:

```
(append '(a b c) '(d e f))  ==>  (a b c d e f)
(append '(a b c) '(x) )     ==>  (a b c x)
```

Write a Fn back-cons which takes 2 args, a list and an item & produces a new list formed by putting its 2nd arg on the back of its 1st.

ie:

```
(back-cons '(a b c) 'x)   ==>   (a b c x)
```

Do not use reverse to develop this Fn, do use append.

4.5 Write a head recursive version of reverse.

4.6 Test your reverse Fn (either head or tail version) with the following:

```
((a b c) d (e f g))
```

Alter the Fn so that with the above argument it returns:

```
((g f e) d (c b a))
```

PROBLEM SHEET 5.

This tutorial involves playing around with the data structures that underpin the CL system. Start by drawing structures on paper, then try out your ideas in CL and remember, CNTRL C is what you do to interrupt the CL system when you are in a perminent loop or have created circular data structures.

5.1.
Set a variable up with the list '(A B C D)

Work out the setf expression(s) that removes the second item from the list in such a way that the list is permanently altered.

Work out the setf expression(s) that splice in 'X as the new second item in the list. Check that this alters the list referenced by the variable you set up.

5.2.
Write a fn which splices out the Nth element of a list in such a way that it
alters the actual structure of the list.

```
==
== (setf list1 '(a b c d e))
(A B C D E)
== (delete-Nth list1 2)
(A B D E)
== list1
(A B D E)
==
```

This will be a recursive function which carries out setf expressions as it progresses. Think about what you want to do before reaching for the keyboard. Draw it out on paper first.

Remember, you can have more than one expression in the action part of a COND statement, the value returned is that produced by the last expression.
eg:

```
(cond   :
        :
      ( (test expression)
        (action expr 1)
        (action expr 2)   <-- this value is returned
      )                         if test is true
        :
```

5.3.
Write a function INSERT-N which takes a list of numbers in ascending order as one argument, and a number as its other argument. INSERT-N should splice the number into the list maintaining the order of the list but changing the underlying structure.

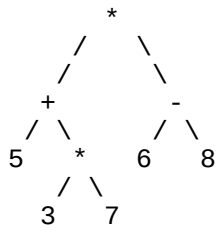
```
==
== (setf nums '(1 3 5 9 11))
(1 3 5 9 11)
== (insert-n nums 7)
(1 3 5 7 9 11)
== nums
(1 3 5 7 9 11)
==
```

5.4.
Write a fn SPLICE-OUT which takes a list, a low index & a high index as its arguments & returns the list with that portion between the two indexes spliced out. SPLICE-OUT should alter the underlying structure of its list argument.

```
==
== (setf list1 '(a b c d e f g))
(A B C D E F G)
== (splice-out list1 2 4)
(A B F G)
== list1
(A B F G)
==
```

PROBLEM SHEET 6.

A tree is a data structure made up of a hierarchy of things called nodes. The example below is of a tree which holds an arithmetic expression:



ie: (5 + 3 * 7) * (6 - 8)

Each node has a piece of data attached to it as well as a number of branches. Data in the above tree is either an arithmetic operator or a number. Generally the node at the top of a tree is known as the 'root node', nodes at the bottom of a tree have no branches and are called 'terminal nodes' or 'leaf nodes' (get it? root, branch, leaf & tree). In an expression tree like the one above, all terminal nodes are numbers.

In list form the above tree would be represented as:

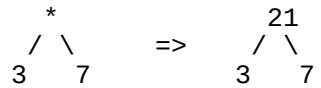
```
'(* (+ 5 (* 3 7)) (- 6 8))
```

where each node is of the form (data left-tree right-tree).

PROBLEM 6.1

Produce a function which takes an expression tree as its argument and returns that tree with the relevant calculated values in place of the operators.

Eg:



So:
```
(evaltree '(* (+ 5 (* 3 7)) (- 6 8)) )
==> (-52 (26 5 (21 3 7)) (-2 6 8))
```

HINT : apply is a CL function which applies other functions, eg:

```
(apply '* '(3 4))  ==>  12
```

PROBLEM 6.2

Produce a function which takes an expression tree as its argument and returns the expression which it contains in infix form containing brackets only where necessary.

So:
```
(rewrite '(* (+ 5 (* 3 7)) (- 6 8)) )
==> ((5 + 3 * 7) * (6 - 8))
```

For both presentations you may assume that each arithmetic operator will always have two operands and that the data you are given is error-free.

EXTRA 1

Write a function, splice-out, which takes 3 arguments: a list and two indexes. The indexes mark start and end points within the list (lists, by convention, are indexed from 0 by CL programmers). The function should return those elements in the list that fall between the two indexes.

eg:

```
(splice-out '(a b c d e) 1 3)  ==>  (b c d)
```

You should be clear about the way your function will behave in situations where the indexes are outside the bounds of the list.

EXTRA 2

Write a function which takes two lists of numbers (both assumed to be in ascending order) and merges them into a single list (also in ascending order).

eg:

```
(merge2 '(2 5 6 12) '(1 4 5 11 15))
==> (1 2 4 5 5 6 11 12 15)
```

PROBLEM SHEET 7 (towards a pattern matcher).

Q1. Write a function called compare which takes two (possibly nested) lists as arguments and returns T if the two lists are **equal** in the sense that...

(compare x y) → T when (equal x y) → T
and (compare x y) → nil when (equal x y) → nil

The compare function should be written using **eq** not **equal** (or **equalp**).

Q2. introduce the **=** symbol as a wild card which may appear in the first argument to compare so...

(compare '(the cat = the =) '(the cat ate the mouse)) → T
(compare '(the cat = the =) '(the cat chased the (small green frog))) → T
(compare '((a b c) = (d e f)) '((a b c) blah (d e f))) → T

Q3. introduce the **?** symbol as a wild card which may appear in the first argument to compare and saves the matching symbol/sublist so...

(compare '(the cat ? the ?) '(the cat ate the mouse)) → (ate mouse)
(compare '(the cat ? the ?) '(the cat chased the (small green frog)))
→ (chased (small green frog))

NB: compare should allow both **?** and **=** to appear in its first argument and both should be ok nested to any extent, so...

(compare '((a ? c) = (d ? f)) '((a b c) blah (d e f))) → (b e)

PROBLEM SHEET 8 (mapping functions)

Q1.1
A data-structure ***phone*** holds details of people & their monthly phone costs, its structure is
((Simon 350) (Andrew 75) ....)

Write a **mapcar** expression to retrieve the costs only (as a list) from ***phone***

Q1.2
use **reduce** with your answer to Q1.1 to calculate the total monthly costs of all phone calls.

Q1.3
use **remove-if** or **remove-if-not** to filter ***phones*** so only those entries with charges greater than 200 remain.

Q1.4
use **mapcar** with your answer from Q1.3 to list only the names of the people whose phone charges is more than 200.

Q1.5
using the ideas above write a function which takes 2 args: (i) a structure like ***phones*** (ii) a number. The function should return the names of all people in the first arg who have charges greater than the second arg.

Q2.1
A structure ***numbers*** is as follows...
((3 three) (7 seven) (0 zero) (1 one) ....)

write a predicate called **compare** which compares a number with an element-pair from ***numbers*** & returns as follows...
(compare 5 '(5 five)) ==> t
(compare 5 '(7 seven)) ==> nil

Q2.2
use compare as the :test arg for member to see if a number is represented in ***numbers***

Q2.3
write a function called **int-to-sym** which uses member to convert a number to its symbolic equivalent, ie:
(int-to-sym 7 *numbers*) ==> seven

Q2.4
rewrite int-to-sym so it uses find-if instead of member - you will need an anonymous function call within the call to find-if.