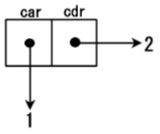
AS you work through this walk through (much of this will be review but some will be new) write the up answers for the highlighted "**Exercise** #" (questions that are **highlighted**). Have your writeup be executable Scheme. That is plain test, with a .scm ending, and all non executable scheme preceded by a ';'. Make sure your name, course, project info, and all your comments and question text etc. are comments. The answers that require working code should all be valid scheme code executable under Kawa. If your answers depend on function that are not define in Kawa you will need to define them as well. This starts off slow (easy-review) but does get a harder.

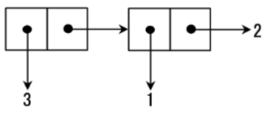
Making Lists
As Scheme belongs to Lisp linguistic family, it is good at list operations. Understanding lists and list operations will help your understanding of Scheme. Lists play important roles in recursive and higher order functions.
In this section, we will look at basic list operators, such as cons, car, cdr, list and quote.

Function cons allocates a memory space for two addresses and stores the address to 1 in one part and to 2 in the other part. The part storing the address to 1 is called **car** part and that storing the address to 2 is called **cdr** part.. The name **cons** is an abbreviation of an English term 'construction'.

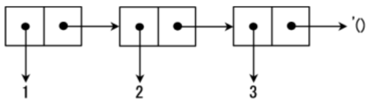


```
(cons 3 (cons 1 2))
 ⇒ (3 1 . 2)
```

(3 1 . 2) is a convenient notation for (3 . (1 . 2)). The memory space of this situation is shown in Figure 2.



Lists
Lists are (beaded) cons cells with the cdr part of the last cons cell being **'()**. **'()** is called the empty list, which is included in every lists. To be a valid list the last cell must be nill/null. Even if the data consists of only one cons cell, it is a list if the cdr part is '(). The figure shows the memory structure of a list (1 2 3).



Actuary, list can be defined recursively like as follows:

1. '() is a list.
2. If **ls** is a list and **obj** is a kind of data, (**cons obj ls**) is a list.

As lists are data structure defined recursively, it is reasonable to be used in recursive functions.

atoms
Data structures, which do not use cons cells, are called atom. Numbers, Characters, Strings, Vectors, and '() are atom. '() is an atom and a list as well.

quote
All tokens are ready to be evaluated due to the Scheme's rule of the evaluation that tokens in parentheses are evaluated from inner to outer and that the value comes out from the outermost parentheses is the value of the S-expression. A special form named **quote** is used to protect tokens from evaluation. It is for giving symbols or lists to a program, which became something else by evaluation.
For instance, while (+ 2 3) is evaluated to be 5, (quote (+ 2 3)) gives a list (+ 2 3) itself to the program. As quote is frequently used, it is abbreviated as **'**.

For example:
'(+ 2 3) represents a list (+ 2 3) itself.
'+ represents a symbol + itself.
Actually, '() is a quoted empty list, which means that you should write '() to represent an empty list while the interpreter responds () for an empty list.

Special forms
Scheme has two kinds of operators: One, functions. Functions evaluate all the arguments to return value. The other is special forms. Special forms do not evaluate all the arguments. Some example of special forms: **quote**, **lambda**, **define**, **if**, **set**!.

Functions car and cdr
The functions that returns the car part and the cdr part of a cons cell is called **car** and **cdr**, respectively. If the value of cdr is beaded cons cells, the interpreter prints whole values of the car parts. If the cdr part of the last cons cell is not '(), the value is also shown after ..

```
(car '(1 2 3 4))
⇒  1

(cdr '(1 2 3 4))
⇒ (2 3 4)
```

Function list
Function **list** is available to make a list consisting of several elements. Function list takes arbitrary numbers of arguments and returns a list.

```
(list)
⇒ ()

(list 1)
 ⇒ (1)

(list '(1 2) '(3 4))
⇒ ((1 2) (3 4))

(list 0)
 ⇒ (0)

(list 1 2)
 ⇒ (1 2)
```

Exercise 1 What is the result of:

a. `(cdr (car s))`, where `s` is `((banana tree) bark)`?

b. `(cons (car l) (cdr (cons s (cdr l))))`, where `l` is `(big hairy nose)` and `s` is `(victorian (handkerchief))`
note: `caddr`; is is short hand for `(car (cdr (cdr alist)))`

Defining Functions
How to define simple functions and to load them

You use **define** to bind a symbol to a value. You can define any kind of global parameters, such as numbers, characters, lists, etc. and functions by this operator.

```
Create a File “hello.scm”
; Hello world as a variable
(define vhello "Hello world")

; Hello world as a function
(define fhello (lambda ()
                 "Hello world"))
```

Next, give the following command to the Scheme interpreter.

```
(load "hello.scm")
 fhello
```

By this, hello.scm is loaded to the scheme interpreter. If the current directory of the interpreter is the directory where the script is, you don't need the first line. Then give the following command to the interpreter.

```
vhello
"Hello world"

(fhello)
"Hello world"
```

The **define** is an operator to declare variables and it takes two arguments. The operator declares to use the first argument as a global parameter and binds it to the second argument.

lambda is a special form of define procedures. The lambda takes more than one arguments and the first argument is the list of parameters that the procedure takes as arguments. As fhello takes no argument, the list of the parameters is empty.

Give vhello to the interpreter then it returns the value, "Hello world" which means that the Scheme interpreter treats procedures and conventional data type in a same way. The Scheme interpreter manipulates all the data by their addresses in the memory space and any kinds of object existing in the memory can be treated in the same way.

To call fhello as a procedure, you should bracket off the symbol like (fhello). Then the interpreter evaluates it and returns "Hello world"

Exercise 2

1. Write a function (`CtoF`)that converts Centigrade temperatures to Fahrenheit.
   To convert a temperature from Centigrade to Fahrenheit, multiply the temperature by `9/5`, and then add `32` to the result.

2. Write a function (`pb`) which takes no arguments and returns `peanut-butter`.

and and **or**

The **and** and **or** are special forms to be used to combine conditions. Scheme's and and or are different from those of conventional languages such as C. They do not return boolean (#t or #f) but returns one of given arguments. The and and or can makes your code shorter.

and

The and takes arbitrary number of arguments and evaluates them from left to right. It returns #f if one of the arguments is #f and the rest of arguments are not evaluated. On the other hand, if all arguments are not #f, it returns the value of the last argument.

```
(and #f 0)
;Value: ()

(and 1 2 3)
;Value: 3

(and 1 2 3 #f)
;Value: ()
```

or

The or takes arbitrary number of arguments and evaluates them from left to right. It returns the value of the first argument which is not #f and the rest of arguments are not evaluated. It returns the value of the last argument if it is evaluated.

```
(or #f 0)
;Value: 0

(or 1 2 3)
;Value: 1

(or #f 1 2 3)
;Value: 1

(or #f #f #f)
;Value: ()
```

Exercise 3 Make following functions (these are tricky):

a. A function that takes three real numbers as arguments. It returns the product of these three numbers if all them is positive.

b. A function that takes three real numbers as arguments. It returns the product of these three numbers if at least one of them is negative.

cond expression

Even though all branchings can be expressed by if expression, you should nest it if the branching has a wide variety, which makes code complicated. The cond expression is available for such cases. The format of the cond expression is like as follows:

```
(cond
  (predicate_1 clauses_1)
  (predicate_2 clauses_2)
    ......
  (predicate_n clauses_n)
  (else        clauses_else))
```

In the cond expression, predicates_i are evaluated from top to bottom, and clauses_i is evaluated and returns from the cond expression if the predicates_i is true. Other clauses and predicates after i are not evaluated. If all of predicates_i are false it returns the value of clauses_else. You can write more than one S-expression in one clauses and the value of the last S-expression is the value of the clauses.

Example: Fee of a city-run swimming pool

The fee of a city-run swimming pool of Foo city depends on the age of users (age):

- free if age ≤ 3 or age ≥ 65.
- 0.5 dollars for 4 ≤ age ≤ 6.
- 1.0 dollars for 7 ≤ age ≤ 12.
- 1.5 dollars for 13 ≤ age ≤ 15.
- 1.8 dollars for 16 ≤ age ≤ 18.
- 2.0 dollars for others.

The function that returns the fee of the city-run swimming pool is as follows:

```
(define (fee age)
  (cond
   ((or (<= age 3) (>= age 65)) 0)
   ((<= 4 age 6) 0.5)
   ((<= 7 age 12) 1.0)
   ((<= 13 age 15) 1.5)
   ((<= 16 age 18) 1.8)
   (else 2.0)))
```

Functions that makes predicates

functions with '?' at the end of their names.

eq?, eqv?, and equal?

The eq?, eqv?, and equal? take exactly two arguments and are basic functions to check if the arguments are 'same'. These three functions are slightly different from each other.

eq?
It compares addresses of two objects and returns #t if they are same. In the following example, the function returns #t as str has the same address as itself. On the contrary, as "hello" and "hello" are stored in the different address, the function returns #f. Don't use eq? to compare numbers because it is not specified in Scheme standard even it works in MIT-Scheme. Use eqv? or = instead.

```
(define str "hello")
⇒ str

(eq? str str)
⇒ #t

(eq? "hello" "hello")
⇒ ()          ← It should be #f

;;; comparing numbers depends on implementations
(eq? 1 1)
⇒ #t

(eq? 1.0 1.0)
⇒ ()
```

eqv?
It compares types and values of two object stored in the memory space. If both types and values are same, it returns #t. Comparing procedure (lambda expression) depends on implementations. This function cannot be used for sequences such as lists or string, because the value stored in the first address are different even the sequences looks same.

```
(eqv? 1.0 1.0)
⇒ #t

(eqv? 1 1.0)
⇒ ()

;;; don't use it to compare sequences
(eqv? (list 1 2 3) (list 1 2 3))
⇒ ()

(eqv? "hello" "hello")
⇒ ()
;;; the following depends on implementations
(eqv? (lambda(x) x) (lambda (x) x))
⇒ ()
```

equal?
It is used to compare sequences such as list or string.

```
(equal? (list 1 2 3) (list 1 2 3))
⇒ #t

(equal? "hello" "hello")
⇒ #t
```

Functions that check data type
Followings are type checking functions. All of them take exactly one argument.

pair?	returns #t if the object consists of cons cells (or a cons cell).
list?	returns #t if the object is a list. Be careful in that '() is a list but not a pair.
null?	returns #t if the object is '().
symbol?	returns #t if the object is a symbol.
char?	returns #t if the object is a character.
string?	returns #t if the object is a string.
number?	returns #t if the object is a number.
complex?	returns #t if the object is a complex number.
real?	returns #t if the object is a real number
rational?	returns #t if the object is a rational number.
integer?	returns #t if the object is an integral
exact?	returns #t if the object is not a floating point number.
inexact?	returns #t if the object is a floating point number.

Functions that compare numbers
=, <, >, <=, >=

These functions takes arbitrary number of arguments. If arguments are ordered properly indicated by the function name, functions return #t.

```
(= 1 1 1.0)
⇒ #t

(< 1 2 3)
⇒ #t
(< 1)
⇒ #t
(<)
⇒ #t

(= 2 2 2)
⇒ #t

(< 2 3 3.1)
⇒ #t

(> 4 1 -0.2)
⇒ #t

(<= 1 1 1.1)
⇒ #t
(>= 2 1 1.0)
⇒ #t

(< 3 4 3.9)
⇒ ()
```

odd?, even?, positive?, negative?, zero?
These functions take exactly one argument and return #t if the argument satisfies the property indicated by the function name.

Functions that compare characters
Functions char=?, char<?, char>?, char<=?, and char>=? are available to compare characters. See R5RS for detailed information.

Functions that compare strings
Functions string=? and string-ci=? etc are available. See R5RS for detailed information.

Local Variables
let expression
Local variables can be defined using the let expression. The format is like as follows:
(let binds body)

Variables are declared and assigned to initial values in the binds form. The body consists of arbitrary numbers of S-expressions.
The format of the binds form is like as follows:
[binds] → ((p1 v1) (p2 v2) ...)

Variables p1, p2 ... are declared and bind them to the initial values, v1, v2 ... The scope of variables is the body, which means that variables are valid only in the body.
Example 1: Declaring local variables i and j, binding them to 1 and 2 and then making a sum of them.

```
(let ((i 1) (j 2))
            (+ i j))
⇒ 3
```

The let expressions can be nested.
Example 2: Making local variables i and j, binding them to 1 and i+2, and multiplying them.

```
(let ((i 1))
  (let ((j (+ i 2)))
    (* i j)))
⇒ 3
```

As scope of the variables is the body, the following code causes an error, because the variable i is not defined in the scope of the variable j.

```
(let ( (i 1) (j (+ i 2) ) )
   (* i j))
;Error
```

The let* expression is available to refer variables which is defined in the same binding form. Actually, the let* expression is a syntax sugar of nested let expressions.

```
(let* ((i 1) (j (+ i 2)))
  (* i j))
⇒ 3
```

Example 3: A function quadric-equation that calculates the answers of quadratic equations.
It takes three coefficient a, b, c (a x2 + b x + c = 0) as arguments and returns the list of answers in real numbers. The answers can be calculated without useless calculations by using let expressions step by step.

```
;;; scopes of variables d, e, and f are the regions starting after their let expression
(define (quadric-equation a b c)
  (if (zero? a)
      'error                                      ; 1
      (let ((d (- (* b b) (* 4 a c))))            ; 2     d
        (if (negative? d)                         ;       d
         '()                                      ; 3     d
         (let ((e (/ b a -2)))                    ; 4     d       e
           (if (zero? d)                          ;       d       e
               (list e)                           ;       d       e
               (let ((f (/ (sqrt d) a 2)))        ; 5     d       e       f
                 (list (+ e f) (- e f)))))))))    ;       d       e       f

 (quadric-equation 3 5 2)  ; solution of 3x2+5x+2=0
 ⇒ (-2/3 -1)
```

The function behaves like as follows:
If the 2nd order coefficient (a) is 0, it returns 'error.
If a ≠ 0, it binds the value of the discriminant (b2 - 4ac) to a variable d.
If d is negative, it returns '().
If not, it binds -b/2a to a variable e.
If d is zero, it returns a list consisting e.
If d is positive, it binds √(d/2a) to a variable f and returns a list of (+ e f) and (- e f).
Actually let expression is a syntax sugar of lambda expression:

```
(let ((p1 v1) (p2 v2) ...) exp1 exp2 ...)
⇒
((lambda (p1 p2 ...)
exp1 exp2 ...) v1 v2)
```

As the lambda expressions is function definition, it makes a scope of variables.

Repetition: Recursion
A recursive function is a function that calls itself in its definition. Calculating factorial is often used to explain recursion.
[code 1] Function fact that calculates factorials.

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

(fact 5) is calculated like as follows:

```
(fact 5)
⇒ 5 * (fact 4)
⇒ 5 * 4 * (fact 3)
⇒ 5 * 4 * 3 * (fact 2)
⇒ 5 * 4 * 3 * 2 * (fact 1)
⇒ 5 * 4 * 3 * 2 * 1
⇒ 5 * 4 * 3 * 2
⇒ 5 * 4 * 6
⇒ 5 * 24
⇒ 120
```

(fact 5) calls (fact 4), (fact 4) calls (fact 3), then finally (fact 1) is called. (fact 5), (fact 4) ,.., and (fact 1) are allocated at different memory spaces and (fact i) stays there until (fact (- i 1)) returns a value, which wastes the memory space and takes more calculation time because of the overhead of function call.
However, recursive functions can express repetition in a simple manner. Further as lists are defined recursively, lists and recursive functions fit together. For instance, a function that makes all list items twice is defined as follows. The function should return an empty list if the argument is an empty list to terminate the calculation.

```
(define (list*2 ls)
  (if (null? ls)
      '()
      (cons (* 2 (car ls))
            (list*2 (cdr ls)))))
```

Exercise 4 : Write following functions using recursion.

a. A function that counts the number of list items, my-length. (Function **`length`** is pre-defined and uses is not allowed.)
```
        (my-len '(1 2 3 (4 5)))     => 4
```

b. A function that counts the number of items in the list, my-length, but this time includes items in sub-lists
```
        (my-length '(1 2 3 (4 5)))  => 5
```

c. A function that summarizes (sums up all) numbers in a list (called my-sum).
```
        (my-sum '(1 2 3)  => 6
```

d. my-summarize that summarizing items of a list consisting of numbers and sub-lists of numbers.
```
        (my-summarize '(1 (2 3) 4)) => 10
```

Tail Recursive

Ordinary a recursive function is not efficient because of wasting memory and function call overhead. On the contrary, tail recursive functions include the result as argument and returns it directory when the calculation finishes. Especially, as Scheme specification requires conversion of a tail recursive to a loop, there is no function call overhead.

[code 2] shows a tail recursive version of function fact shown in [code 1].

[code 2] fact-tail, tail recursive version of fact
```
(define (fact-tail n)
  (fact-rec n n))

(define (fact-rec n p)
  (if (= n 1)
      p
      (let ((m (- n 1)))
        (fact-rec m (* p m)))))
```
fact-tail calculates factorial like as follows:
```
(fact-tail 5)
⇒ (fact-rec 5 5)
⇒ (fact-rec 4 20)
⇒ (fact-rec 3 60)
⇒ (fact-rec 2 120)
⇒ (fact-rec 1 120)
⇒ 120
```
As fact-rec does not wait for the result of other functions, it disappears from the memory space when it finishes. The calculation proceeds by changing argument of fact-rec, which is basically the same as loop. As mentioned previously, as Scheme convert a tail recursive to a loop, Scheme can do repetition without syntax for looping.

Exercise 5: Write the following functions using tail recursive.

e. my-reverse that reverse the order of list items. (Function reverse is pre-defined and should not be used.)

(my-reverse-wrapper '(1 2 3 4))

(4 3 2 1)

Named let
The named let is available to express loop. [code 3] shows a function fact-let that calculates factorials using named let. The fact-let uses a named let expression (loop), instead of the fact-rec shown in [code 2].
First it initializes parameters (n1, p) with n at the line marked with ; 1.
These parameters are updated at the line marked with ; 2 after each cycle:
Subtracting n1 by one and multiplying p by (n1-1)
A named let is a conventional way to express loops in Scheme
[code 3]

```
(define (fact-let n)
  (let loop((n1 n) (p n))           ; 1
    (if (= n1 1)
        p
        (let ((m (- n1 1)))
          (loop m (* p m))))))      ; 2
```

Higher Order Functions
Higher order functions are functions that takes functions as arguments. They are used for mapping, filtering, folding, and sorting of lists. The higher order functions promote modularity of programs. Writing higher order functions that are applicable in many cases makes program readable rather than writing recursive functions for individual cases.
For instance, using a higher order function for sort allows sorting by variety of conditions, which separates the sorting condition and the sorting procedure sharply. The function sort takes two arguments, one is a list to be sorted and the other is an ordering function. The following shows the sort of a list of integral numbers in ascending order by their size. The function < is the ordering function of two numbers.

```
(sort '(7883 9099 6729 2828 7754 4179 5340 2644 2958 2239) <)
⇒ (2239 2644 2828 2958 4179 5340 6729 7754 7883 9099)
```

On the other hand, sort by the size of last two digits can be done like as follows:

```
(sort '(7883 9099 6729 2828 7754 4179 5340 2644 2958 2239)
   (lambda (x y) (< (modulo x 100) (modulo y 100))))
⇒ (2828 6729 2239 5340 2644 7754 2958 4179 7883 9099)
```

As shown here, sorting procedures such as quick sort, merge sort, etc. and ordering functions are separated completely, which promote reuse of codes. Scheme does not distinguish procedures and other data structures, thus we can make our own higher order functions quite easily by just giving procedures as arguments.

Actually, substantial parts of pre-defined functions in Scheme is higher order functions, because Scheme does not have a syntax that defines block structure, and because lambda expression is used as a block.

Mapping
Mapping is a procedure that treats all list items in a same manner. Two mapping functions are defined in the Scheme standard. One of them is map which returns the converted list and the other is for-each which is used for side effects.
map
The format is like as follows:

```
(map procedure list1 list2 ...)
```

The procedure is a symbol bound to a procedure or a lambda expression. Number of lists as arguments depend on the number of arguments of the procedure.
Example:

```
; Adding each item of '(1 2 3) and '(4 5 6).
(map + '(1 2 3) '(4 5 6))
⇒ (5 7 9)
; Squaring each item of '(1 2 3)
(map (lambda (x) (* x x)) '(1 2 3))
⇒ (1 4 9)
```

for-each
The format is the same as that of map. The function does not return a specified value and is used for side effects.
Example:

```
(define sum 0)
(for-each (lambda (x) (set! sum (+ sum x))) '(1 2 3 4))
sum
```

```
        ⇒ 10
```

Exercise 6: Write followings using map.
A function that triples each item of a list of numbers.

Filtering
Even filtering functions are not defined in the Scheme standard but keep-matching-items and delete-matching-items are available in the MIT-Scheme. Other implementations should have similar functions.

```
        (keep-matching-items '(1 2 -3 -4 5) positive?)
        ⇒ (1 2 5)
```

Folding

```
        (reduce + 0 '(1 2 3 4))                 ⇒  10
        (reduce + 0 '(1 2))                     ⇒  3
        (reduce + 0 '(1))                       ⇒  1
        (reduce + 0 '())                        ⇒  0
        (reduce + 0 '(foo))                     ⇒  foo
        (reduce list '() '(1 2 3 4))            ⇒  (((1 2) 3) 4)
```

Function apply
This function is to apply a procedure to a list. Even the function takes arbitrary numbers of arguments, the first and last arguments should be a procedure and a list. The function is really convenient even it does not seem to be.

```
(apply max '(1 3 2))       ⇒   3
(apply + 1 2 '(3 4 5))     ⇒  15
(apply - 100 '(5 12 17))   ⇒  66
```

Making Higher Order Functions
Writing higher order functions is quite easy. Functions member-if and member are defined here as examples. member-if and member

A function member-if takes a predicate and a list as arguments and returns a sub-list of the original whose car is the first item that satisfies the predicate. The function member-if can be defined like as follows:

```
(define (member-if proc ls)
  (cond
   ((null? ls) #f)
   ((proc (car ls)) ls)
   (else (member-if proc (cdr ls)))))
   (member-if positive? '(0 -1 -2 3 5 -7))
⇒  (3 5 -7)
```

Further, the function member that checks if the specified item is in the list can be defined like as follows. The function takes three arguments, a function to compare, the specified item, and a list:

```
(define (member proc obj ls)
  (cond
   ((null? ls) #f)
   ((proc obj (car ls)) ls)
   (else (member proc obj (cdr ls)))))
   (member string=? "hello" '("hi" "guys" "bye" "hello" "see you"))
⇒  ("hello" "see you")
```

Exercise 5: Write a **single** function

- that squares each item of a list, t
- then sums all the items in the list, and
- then lastly, takes square root of the sun.

```
(my-function '(1 2 3))
 3.7416573867739413
```