Evolution of Programming Languages

Evolution has been influenced by a number of factors, among which are:

- computer architecture: the Von Neuman model where the CPU is separate from the memory. Both program and data co-exist in memory. Languages based on this model are called imperative languages. Their central features are: variables, which model memory cells, assignments which allow values to be placed in memory cells, and iteration which allows repetition.
- programming methodology: over time the major cost of computing has shifted from hardware to software. New software development methodologies (top-down design, stepwise refinement) highlighted deficiencies of programming languages (incompleteness of type checking and inadequacy of control statements).
- abstraction: provides better readability for humans. Shift from procedural abstraction to data abstraction (object-oriented).

process abstraction	basic (combine a few machine instructions into an abstract statement, e.g. assignment), structured (divide a program into groups of instructions, e.g. if, case, switch statements; looping; subprogram/procedure)
data abstraction	basic (abstract the internal representation of built-in types) introduces the concept of information hiding structured (abstract collection of values that are related, e.g. records & arrays)
abstract data types,	object, combination of data and operations to manipulate them, representation is hidden

Programming Paradigms

Have evolved over the years. It started with paralleling and abstracting the operations of a computer. As indicated before it was based on the Von Neumann model (computers not hard-wired but series of code stored as data determine the actions taken by the CPU). Those original languages are called imperatives

1. **Imperative**: As indicated before, programs consist of a list of statements executed sequentially with variables representing memory locations and using assignments to change the value of variables. Supports procedural and structured abstraction. The drawback is this restricts the ability of the language to indicate parallel computations (von Neumann bottleneck) and nondeterministic computations (those that do not depend on order). Examples of languages in this category: C, FORTRAN, Pascal, Algol, Perl, PHP, most versions of BASIC.
2. **Object-Oriented:** it represents an extension of the imperative paradigm but allows programmer to write reusable and extensible code. It is data driven instead of operation driven and supports data abstraction. Examples: Smalltalk, C++, C#, Java, Eiffel, Ada 95, Python, Ruby, Visual BASIC.
3. **Functional:** comes from mathematics and is based on the abstraction of a function as studied in the lambda calculus. The basic mechanism is the evaluation of functions with returned values. There no notion of variable or assignment, only binding of values to names. Repetitive operations are not expressed by loops but by recursion. Examples: LISP (Scheme), ML, Miranda, Haskell, Erlang.
4. **Logic:** based on symbolic logic. A program consists of a set of statements that describe what is true about a desired result, not a sequence of statements to be executed in a fixed order to produce the result. In its pure form no need for control abstraction as the control is provided by the underlying system (the engine). Sometimes called declarative programming. Properties are declared but no execution sequence is specified. Example: Prolog, Datalog, ASP, Florid, Logtalk.

The last two paradigms have the advantage that since they are founded on mathematics the program behavior can be described abstractly and precisely which makes it much easier to judge that a program will execute correctly. It also permit very concise code to be written even for complex tasks.

We will now focus on functional languages.

Functional Languages

Different view from that provided by imperative or object-oriented languages. They provide several advantages:

- uniform view of programs as functions
- treatment of function as data
- limitation of side effects
- automatic memory management

They are flexible, have concise notation, and simple semantics.

Their main drawback has been the inefficiency of their execution. Historically those languages have been interpreted rather than compiled. Little by little compilers have become available and improvements in their compilation in the last twenty years have made them much more attractive especially in situations where execution efficiency is not crucial.

However, they may appear as more formidable to the average programmer. We will describe "pure functional programming." In practice many functional languages have incorporated some features of imperative languages. Those will not be used in this class.

Functions

A function is a rule that associate to each x from some set X of values a unique y from another set of values Y. It is noted

$$y = f(x) \qquad \text{or} \qquad F: X \rightarrow Y$$

X is called the **domain** of f, Y is called the **range**. x is the **independent variable** and y is the **dependent variable**. The function is **partial** if it is not defined for all x in X, **total** otherwise.

$$\text{in} \rightarrow \boxed{f} \rightarrow \text{out}$$

There are significant differences between functional languages and imperative languages:

- no concept of location, no variables (considered a name for a memory location), no assignment. Only constants, parameters, and values, a name is viewed as something to refer to a value.
- Repeated operations are performed via recursion, there are no loops.
- No internal state of a function, the value of a function depends only on the values of the arguments (with possibly non local constants)
- The value of a function cannot depend on the order of evaluation of its arguments

The property of a function that its value depends only on the value of its arguments is called **referential transparency**. That implies that a function without argument always returns the same value (i.e. is like a constant).

The run-time environment associates names to value and a name can never change value (this is called **value semantics).**

Functions are manipulated in arbitrary ways as general objects. Functions can be viewed as values, computed by other functions and can be parameters to functions. Specifically:

Function as value

Example: let us define the following two Boolean functions:
A is true if its parameter is even
B is true if its parameter is odd
And another function C that takes an array X & a Boolean function F and returns an array containing only the elements of X satisfying F.
Hence
C (B, [1, 2, 3, 4]) returns [1, 3]
C (A, [1, 2, 3, 4]) returns [2, 4]

Function as return value

Example: D is a function that takes a Boolean function F and returns a function which does C above with the Boolean function built-in.

E = D (B)	E ([1, 2, 3, 4])	returns	[1, 3]
F = D (A)	F ([1, 2, 3, 4])	returns	[2, 4]

SCHEME

Scheme is a dialect of LISP. LISP was the first functional language developed by an MIT team led by John Mc Carthy. The earliest version was specified in 1958. You'll find a history written by McCarthy at
http://www-formal.stanford.edu/jmc/history/lisp/lisp.html.
Scheme is a version of LISP developed at MIT in the mid 70's and later revised. We will refer to the version published by Abelson in 1998. We will now look at some of the underpinning

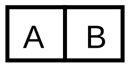
S-expressions (for symbolic expressions)

An S-expression is either an atom (written as a symbol), or a pair written in the form (S_1 • S_2) [called a dotted pair], where S_1 and S_2 stand for arbitrary S-expressions.

Examples:

atoms	**pairs**
A	(A • B)
APPLE	(A •(B • C))
PART2	((U • V) • (X • (Y • Z)))

The pair (A • B) can be represented graphically as:



As we will see later, A is called the CAR and B is called the CDR. That terminology comes from the first implementation on a machine which had words containing an address (where the CAR was stored) and a decrement (where the CDR was stored).

Lists are a subset of S-expressions defined as:

1. an atom is a list iff it is the atom NIL [also denoted by ( )]

2. a pair (X • Y) is a list iff Y is a list

S-expressions	List Notation
NIL	( )
(APPLE • NIL)	(APPLE)
((A • NIL) • (B • (C • NIL)))	((A) B C)

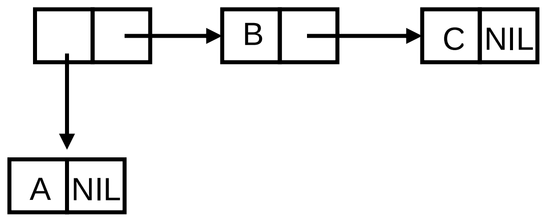
Note that the CDR lose its parenthesis. Another way to look at it is that, in the list representation the CAR slips in the first position inside the CDR and the outside parenthesis disappear, as in:

(A • NIL) which is also (A • ( )) becomes (A)

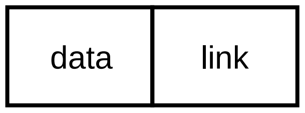
or

(A • (C)) becomes (A C)

The last list above would be represented graphically as:



Notice the similarity with the representation of linked list you have seen before:



Note that, as we have seen above, the pair (A • B) and the list (A B) are two completely different things, as are A and (A). The list (A B) is equivalent to (A • (B • NIL)) and the list (A) is equivalent to (A • NIL). Note also that (A • B) is an S-expression which has no list equivalent.

Your turn:

Draw box diagrams for the following scheme lists:

((((a))))

(1 (2 (3 4)) (5))

((a ( )) ((c) (d) b) e)

Use those box diagrams in question 1 to compute the following for each Scheme list for which they make sense.

(car (car L))

(car (cdr L))

(car (cdr (cdr (cdr L))))

(cdr (car (cdr (cdr L))))

S-expressions (primarily in the form of lists) are the values manipulated by LISP/Scheme but programs are also S-expressions

Hence, the syntax of Scheme is simple:

```
<expression>    ::=    <atom>     |  <list>
<atom>          ::=    <number>|  <string>  |    <identifier>   |
                                 <character>    |    <Boolean>
```

Predicates can determine the nature of a Scheme elements as indicated below.

Data Types

Type	**Predicate to Test**	**Examples**
boolean	boolean?	#t, #f
number	number?	1.3, 2
string	string?	“thing”
pair	pair?	(a • b)
empty list	null?	( ) or NIL
character	char?	#\a, #\x
symbol	symbol?	y, a+b
vector	vector?	#(a b 1 2)
list	list?	(1, 4, a, f)

Examples of Scheme expressions:

Expression	**Description**	**Value (see below the rules)**
56	a number	56
“welcome”	a string	“welcome”
#F or #f	the Boolean value “false”	false
#\b	the character “b”	“b”
(2.1 2.2 3.1)	a list of numbers	cannot be evaluated
abc	an identifier (i.e. a name)	whatever value it is associated with
welcome	another identifier	same
(+ 2 3)	a list consisting of the identifier “+” and two numbers	5 [+ evaluate to addition 2 & 3 evaluate to themselves]
(* (+ 2 3) (/ 6 2))	a list consisting of the identifier “*” followed by two lists	15

Rules of Evaluation

The rules of evaluation are as follows:

1. Constant atoms evaluate to themselves (as 56 and “welcome” above)
2. Identifiers are evaluated to their value in the current environment (maintained in a symbol table)
3. Lists are evaluated by recursive evaluation of each element of the list as an expression. ***The first expression in the list must evaluate to a function***. That function is in turn applied to the ***evaluated values*** of the elements of the list.

Hence an expression is of the form:

$$(E_0\ E_1\ E_2\ \ldots\ E_n) \quad , n >= 0$$

is an invocation where E_0 is the function and E_1 E_2 ... E_n are the arguments. *Those will be evaluated before the function is applied*.

To prevent evaluation the built-in function QUOTE is provided (i.e. it evaluates to a literal)

(QUOTE S) also denoted as ‘S , where S is an S-expression.

The value of (QUOTE APPLE), or ‘APPLE is APPLE, the value of (QUOTE (A B)), or ‘(A B) is (A B). In Scheme an unquoted atom used as an expression is an identifier (except for NIL and T which denote themselves).

We will now review a number of functions available in Scheme

Functions for the Control of Execution

In Scheme the control of execution is controlled by two basic functions: IF and COND.

1. IF

   (IF p e_1 e_2) where p is a predicate (i.e., evaluates to true or false), and e_1 and e_2 are expressions. It evaluates to the value of e_1 if p evaluate to true, that of e_2 otherwise

   Note: e_2 may be omitted. In such case if p evaluate to false, the value of the expression is undefined.

   Example:

   ```
   ( IF ( =  a  3 )  ( *  a  5 )  ( /  a  3 ) )
   ```

   Evaluates to 15 if a is equal to 3, and a/3 otherwise

   To specify the value of a, we would put it in parenthesis after the expression as in

   (IF (= a 3) (* a 5) (/ a 3)) (3) means a is equal to 3, so the expression evaluates to 15, but
   (IF (= a 3) (* a 5) (/ a 3)) (9) means a is equal to 9, so the expression evaluates to 3

   Evaluate the following:

   (if (= a 5)10 15) (5)

```
( if ( =  a  5 )10    15 )     ( 8 )
```

Evaluate the following:

```
( if ( =  a  3)  ( *  a  5)    ( /  a  3) )  ( 6 )

( if ( =  a  3)  ( *  a  5)    ( /  a  3) )  ( 3 )
```

2. COND

```
(COND   (p1  e1)
        (p2  e2)
        .
        .
        (pn  en)
        (else    e) )     (this last one is used often but not always)
```

where p_1, p_2, etc. are predicates and e_1, e_2, etc. are expressions. It evaluates to the first e for which the corresponding p is true (i. e. the value of the expression is the value of the e_i associated with the first p_i which is true).

Note: as indicated previously, else may be omitted. In such case if none of the p's evaluate to true, the value of the expression is undefined

Example:

```
( COND  ( ( =  a  3)   5)
        ( ( =  a  5)   8)
        ( else ( *  a  5) ) )
```

Evaluates to 5 if a equals 3, to 8 if a equals 5, and to 5*a otherwise.
Evaluate the following:

```
( cond  ( ( =  a  3)   5)
        ( ( =  a  5)   8)
        ( else  ( *  a  5) ) )  ( 3 )
( cond  ( ( =  a  3)   5)
        ( ( =  a  5)   8)
        ( else  ( *  a  5) ) )   ( 5 )

( cond  ( ( =  a  3)   5)
        ( ( =  a  5)   8)
        ( else  ( *  a  5) ) )   ( 2 )
```
Note that in the cases of both IF and COND the usual rules of evaluation do not apply (if they did all-arguments would be evaluated first before applying the function). What happens with IF is that the predicate p will be evaluated first, then either e_1 or e_2 , but not both will be evaluated, depending on the value of p. This is called **delayed arguments**. Functions that used delayed evaluation are called **special forms**. The QUOTE function is another example as is the LET function which we will see later.

Functions & Lambda Calculus

Unnamed Functions

Early work by a mathematician called Church separated the task of defining a function from that of naming it. This is done using the lambda notation. A lambda expression specifies the parameters and the mapping of a function but does not give a name to the function. The lambda expression is the function itself. For example:

```
(lambda (x) x * x * x) ( 2 )          evaluates to 8
```

Note that lambda expressions can have more than one parameter. For example:

```
(lambda (x, y) x * y) ( 2, 3)         evaluates to 6
```

Your turn:

```
( ( lambda  ( x )   ( +     x       2 ) )   4 )          returns
( ( lambda  ( x )   ( *     x       x ) )   4 )          returns
```

Lambda expressions are particularly useful for defining functions that are return values of other functions. For example (not in Scheme notation):
Function to return a cube function if its parameter is even, the squatre function otherwise.

```
function foo ( A )
    if A is even        return λ (x) x * x * x
    else [A is odd]     return λ (x) x * x
```

In Scheme to define a function with a given name we use

3. DEFINE

```
( DEFINE    ( <function name>      <formal parameters> )   <expression> )
```

This allows the definition of recursive and non-recursive functions. This is again a special form. It can also be used to associate a name directly with a value.

Examples:

```
( define     a              2)            associates the value 2 with the name a
( define     emptylist    '( ) )          associates the name emptylist with ()
( define     ( square    x )    ( *  x  x ) )          creates a square function
```

This can also be used in conjunction with anonymous functions:

```
( LAMBDA  ( <formal-parameters> )      <expression>  )
```

Examples:

```
( lambda  ( x )  ( *  x  x ) )
( define  square  ( lambda  ( x )  ( *  x  x ) ) )
```

This implies that (square 5) and ((lambda (x) (* x x)) 5) evaluates to the same value, which is

Functions to Manipulate Data Structures

Additional functions allow the manipulation of data structures (i.e. S-expressions or lists):

4. CONS to construct a dotted pair from its two arguments.

   (CONS e_1 e_2) returns the dotted pair the value of e_1 and the value of e_2. Remember that if e_2 is a list the result will also be a list

   Examples:

(cons 'A 'B)	is (A • B) [note the quote on arguments]	
(cons ‘A NIL)	is (A)	
(cons 'A (cons 'B NIL))	is (A B)	[could be (cons ‘A ‘(B))]
(cons 'A (cons ‘A '(B C D)))	is (A A B C D)	

5. CAR & CDR require a pair (S_1 • S_2) as argument and return S_1 and S_2 respectively (remember that a list is a doted pair as well).

   Examples:

(car '(A • B))	is A
(cdr '(A • B))	is B
(car '(A B))	is A
(cdr '(A B))	is (B)[remember that (A B) is (cons ‘A ‘(B))

   Hence the CAR will return the first element of the list (note that could be a list as well) while ***CDR of a list will always return a list***.

   Notations for shortcuts in case of sequential use of CAR & CDR have been devised

(car (cdr L))	becomes	(cadr L)
(cdr (cdr L))	becomes	(cddr L)

6. LET allows values to be given temporary names as in:

(LET ((x_1 e_1) (x_2 e_2) (x_3 e_3) (x_k e_k)) F)

The expressions e_i are all evaluated and e_i become the value of the corresponding x_i, then F is evaluated using the values of the x's. The value of the expression is the value of F
Examples:

```
( let   ( ( three-sq  ( square  3) )
          ( four-sq   ( square  4) ) )
          ( +  three-sq     four-sq )  )           evaluates to
```

This can be used to avoid recomputation. For example:

```
( +  (square    3 )  (square    3) )
```

can be rewritten as

```
( let   ( ( three-sq  (square    3) ) )
        ( +  three-sq  three-sq )  )               evaluates to
```

Examples of use of the functions previously defined:

a. function to return the sign of an integer

```
( define  ( sign  x )  ( cond   ( ( = x  0 )  0 )
                                ( ( < x  0 )  -1 )
                                ( ( > x  0 )  1 ) ) )
( sign  6 )              is     1
(sign  -13)              is     -1
(sign  0)                is     0
```

b. factorial function

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

c. function to append a new element to the end of a list

```
        ( define ( apnd  L  a )      [Note: L is list, a is an element to be added at
                                     the end of L]
             ( if  ( null?  L )  ( list  a )   ( cons  (car  L )  ( apnd  ( cdr  L )  a ) ) ) )
```

d. function to select the last element of a list

```
        ( define  ( last  L )  ( if  ( null?  ( cdr  L ) )  ( car  L )  ( last  ( cdr  L ) ) ) )
```

What are the results for the following?

```
    ( define  (foo  x  y)
        (cond  ( ( <  x  y ) ( foo ( +  6  x )  y ) )
               ( ( >  x  y ) ( foo ( -  x  y ) ( * ( - 3  x  )  ( + 3  y) ) )  )
               ( else ( - ( *  4  x )  y )  )  )  )

        (foo  6  12)        evaluates to
        (foo  10  6)        evaluates to
        (foo  15  15)       evaluates to
```

more functions:

e. function to concatenate two lists (L & M are lists):

```
    ( define  ( appendlist  L  M )  ( if  ( null?  L )  M
                                          ( cons ( car  L )
                                          ( appendlist  ( cdr  L )  M ) ) ) )

        or another way:

    ( define  ( appendlist L  M )  ( cond  ( ( null?  L )  M )
                                           ( ( null?  M)      L )
                                           ( #t  ( cons  ( car  L )
                                                      ( appendlist  ( cdr  L )  M ) ) ) ) )
```

f. function to reverse the top elements of a list L

```
( define  ( reverse  L )  ( if  ( null?  L )  (  )
                                        ( append  ( reverse  ( cdr  L ) )  ( list  ( car  L ) ) ) ) ) )
```

For example, if the list is ((a b) c (d e)) the function returns ((d e) c (a b))

Let us now look at some predicate functions related to list structure (remember: a predicate is a function that returns a Boolean value):

7. EQ? is a predicate to check if two objects are identical (i.e. are represented internally by the same pointer value).

   (EQ? e_1 e_2) is #T if e_1 and e_2 are the same object, #F otherwise

   Examples:

(eq? A A)	returns	#T
(eq? A B)	returns	#F or ( )
(eq? ‘(A B) ‘(A B))	returns	#T
(eq? ‘(A B) (A B))	returns	#F [Note one is quoted, the other one is not]

8. EQUAL? is a predicate to check for structural equality (i.e. they have the same structure and the atoms are the same. Don’t confuse with EQ? which returns true if its arguments are the same object.

   (EQUAL? e_1 e_2) is #T if e_1 and e_2 evaluate to the same structure with the same atoms in the same position

   Examples

(equal? ‘((1 2) 3) ‘((1 2) 3))	returns #T
(equal? ‘((1 2) 3) ‘((3 2) 1)	returns #F
(eq? ‘((1 2) 3) ‘((1 2) 3))	returns #T
(define foo ‘((1 2) 3))	[used to associate a name & a value]
(equal? ‘((1 2) 3) foo)	returns #T
(eq? ‘((1 2) 3) foo)	returns #F

```
( equal?  foo  foo )                                returns #T
( eq?  foo  foo )                                   returns #T
```

9. PAIR? is a predicate that returns #t if it is a dotted pair, #f otherwise

   (PAIR? e_1) is #T if e_1 is a dotted pair

   Examples:

```
( pair?  ‘x)                                        returns #F
( pair?  ‘( x ) )                                   returns #T
```

Note that PAIR? does not exists in the version of Scheme we will use as it does not allow S-expressions which are not lists.

10. NULL? is a predicate to check if its argument is the empty list

    (NULL? e_1) returns #T if e_1 evaluates to the empty list

```
( null?  ‘( ) )                                     returns #T
( null?  ‘( (  ) ) )                                returns #F
```

11. LIST? is a predicate to check if argument is a list.

    (LIST? e_1) returns #T if e_1 evaluates to a list

    Examples

```
( list?  ‘(X  Y ) )                                 returns      #T
( list?  ‘X )                                       returns      #F
( list?  ‘( ) )                                     returns      #T
( list?  ‘(X • Y)                                   returns      #F
```

12. ATOM? is a predicate to check if its argument is an atom.   [Note: again not standard in the version of Scheme we will use (not (LIST? X) instead]. You may want to define your own ATOM? first.

    (ATOM? e_1)   returns #T if e_1 evaluates to an atom

(atom? 'A)	returns	#T
(atom? '(A))	returns	#F

Your turn:

Write a Scheme function predicate function that tests for the structural equality of two given lists. Two lists are structurally equal if they have the same list structure, although their atoms may be different, For example

```
        ( (a b ) c ( ( d ) e f ( g ) ) )
    and
        ( (x y ) z ( ( u ) v t ( w ) ) )
```

are structurally equal. Name your function (cSL   L   M)

Write a Scheme function that computes the depth of a list. The depth of a list is defined as follows:

NULL	has depth	0
(a)	has depth	1
((a) b (c) d)	has depth	2
((a) (b (c)))	has depth	3

There are also a number of built in arithmetic functions:

Function	Meaning	Comment
+	addition	Can have zero or more parameters. If it has no parameter, it returns 0
-	subtraction	Can have zero or more parameters. If it has more than one parameter all but the first parameter are subtracted from the first
*	multiplication	Can have zero or more parameters. If it has no parameter it returns 1
/	division	Can have zero or more parameters. If it has more than one parameter the first parameter is divided by all the other parameters in turn modulo remainder

In addition Scheme provides a collection of predicate functions for numeric data, among which are:

Function	Meaning
=	Equal
<>	Not Equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?

Other examples:

a. function to square all the elements of a list of numbers and return a list of these squares
   Comment: uses cdr to go down the recursion, cons to go up

```
( define  ( square-list  L )
          ( if ( null?  L )  ‘( )
               ( cons       ( *  ( car  L )  ( car  L ) )
                            ( square-list  ( cdr  L ) )  )  )   ))
```

b. Find the maximum value in a (possibly nested) list of integers.

```
(define ( maxall  L )
          ( cond    ( ( not  ( list?  L )   L )
                      ( ( null?  ( cdr  L ) )  ( maxall  ( car  L ) ) )
                      ( else  ( let ( ( a  ( maxall  ( car L ) ) )
                                      ( b  ( maxall  ( cdr L ) ) ) )
                                      ( if ( > a  b )  a   b  )   ) ) )  )
```

Comment: the following recursion control are not based on list.

c. Produce a list of integer from n to 1:

```
(define  (countdown  n )
          ( if ( = n 0 ) ‘( 0 )  ( cons  n  ( countdown ( - n  1 ) ) ) )    )
```

d. function to print the squares of the integers from 1 to n

```
( define  ( print-squares  low  high )
          ( cond      ( ( >  low  high )  ‘( ) )
                       ( else  ( display  ( * low  low ) )  ( newline )
                       ( print-squares..( +  1  low )  high )   )   )   )   )
```

Your turn:

1. Write a Scheme function with two parameters, an atom and a list, that returns the list with all occurrences, no matter how deep, of the given atom deleted. The returned list cannot contain anything in place of the deleted atom. For example:

```
( removeatom    a       ‘( a ( b c ( a ) a ) ( a b ) ) )
returns      ( ( b c (  )  ) (  b ) )
```

2. The Scheme function reverse described earlier in these notes reverses only the “top level” of a list: if L = ((2 3) 4 (5 6)) then (reverse ‘L) = ((5 6) 4 (2 3)). Write a Scheme function deep-reverse that also reverses all sublists (deep-reverse L) - ((6 5) 4 (3 2)).

3. Write a Scheme function that removes the last element from a given list. For example

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

will return

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

4. Write a Scheme function with three parameters, two atoms and a list, that returns the list with all occurrences, no matter how deep, of the first atom replaced by the second atom. For example:

```
( replaceatom    ‘a     ‘x     ‘( a ( b c ( a ) a ) ( a b ) ) )
```

returns

```
( x ( b c ( x ) x ) ( x  b ) ) )
```

Tail Recursion

- Problem: One of the problem associated with recursion is the cost both in time and especially in space. Since functional languages replace all loops by recursion it makes them slower and more space consuming.
- Consider: One form of recursion can easily be converted internally into a loop structure, it is **tail recursion** where the last operation to be performed in a function is a call to itself with different arguments. In such case, it is unnecessary to preserve the return address and all the information typically kept in the activation record and the function can simply return to the top of the function.
- Solution: To transform a non-tail recursive function into a tail recursive one by using one or more accumulating parameter(s) that is(are) used to pre-compute operations performed after the recursive call and to pass the result to the recursive call.
- Example: If we apply this idea to the *square-list* function we can define a helping function *square-list-aux* with an extra parameter to accumulate the intermediate result as shown below.

Remember: the function square-list takes a list of integers as a parameter and returns a list of the squares of the integers in the input list.

A non tail-recursive version is:

```
        ( define  ( square-list  L )
            ( if  ( null?   L )  '( )
                                   ( cons  ( *  ( car  L )  ( car  L ) )
                                        ( square-list  ( cdr  L )  )  )    )   )
```

A tail recursive version involves defining an **auxiliary function** *square-list-aux* which is tail recursive:

```
    ( define  (square-list-aux  L  list-so-far )         [list-so-far is the accumulator]
        ( if  ( null?  L )  list-so-far
                            (square-list-aux  ( cdr L )
                              ( append  list-so-far  ( list  ( *  ( car  L )  ( car  L )  ) ) ) )  ) )
```

We can then call square-list-aux from square-list

```
        ( define  ( square-list  L )   ( square-list-aux  L  ‘(  ) ) )
```

We can do the same for reverse by creating a reverse-aux function

```
        ( define  ( reverse-aux  L  list-so-far )
                ( if  ( null?  L )  list-so-far )
                          ( reverse-aux  ( cdr  L )
                                ( cons  ( car  L )   list-so-far ) )   ) )
```

and reverse would be re-written as:

```
        ( define  ( reverse       L )  ( reverse-aux  L     ‘(  )    ) )
```

Your turn:

5. Write a tail-recursive procedure to compute the length of an arbitrary list.

6. Consider the following (countdown   n) function that returns a list of the integers from n to 1. For example, (countdown 8) returns (8 7 6 5 4 3 2 1).:

```
        ( define ( countdown  n )
            ( if  ( =  n  0 )  ‘(  )  ( cons n ( countdown  ( - n  1 ) ) ) ) )
```

Write a tail recursive version of "countdown". Name it "tcountdown". It must be tail recursive!

High Order Functions

In Scheme (and other functional languages) functions can have functions as parameters (NOTE: we are not talking about the value of a function, we are talking about passing to a function f a parameter which is a function which can then be called within f), or as a returned value. A function which does either or both is called a high-order function.

As an example of a function that takes a function as a parameter let us look at the **map** function that has two parameters: a function and a list. It applies the function to each element of the list and returns a list of the results.

```
( define ( map  f  L )
    ( if  ( null?  L)  ‘( )  ( cons  ( f  ( car  L )  ( map  f  ( cdr  L ) ) )   ) )
```

Note: map is actually a built-in function that cannot be redefined but this definition illustrates how it works.

```
( map  car  ‘( ( a  b ) ( c  d ) ( e  f ) ) )              returns ( a  c  e )
( map  cdr  ‘( ( a  b ) ( c  d ) ( e  f ) ) )              returns ( ( b ) ( d ) ( f ) )
```

if we call map with a function square defined as:

```
( define  ( square  x )  ( *  x  x ) )
```

we have another definition for square-list:

```
( define  ( square-list  L )  ( map  square  L ) )
```

Examples of the use of remove-if (see #9 in Handout 8):

```
(remove-if pair? '( ( a b ) c ( d e f ) g ( h i j ) ) )        returns      `(c g)

(remove-if ( lambda ( x ) (not ( pair? x) ) ) '( ( a b ) c ( d e f ) g ( h ) ) )
                                                   returns      ((a b) (d e f) (h))
```

Another situation is that of a function that returns a function as its value

For example this returns a function that increments its argument by n

```
        (define  (inc-n  n)  (lambda  (x)  (+  x  n ) )   )

        (define  5inc  (inc-n  5 ) )                            returns (lambda (x) ( + x 5) )
            (5inc    12)                                        returns              17
```

Let us now look at a function duplicate-param that, has an argument which is a function f and creates a function that is called with one parameter and duplicates that parameter in a call to f.
This can be done by returning an unnamed function (i.e. a lambda expression) or returning the function value doublefn that is created in a local define and then written at the end to be returned as the value of duplicate-param. Note that x is not a parameter of duplicate-param but a parameter of the function created by duplicate-param.

```
                ( define  ( duplicate-param  f )
                                    ( lambda  ( x )  ( f  x  x ) )  )
```

Examples of use of duplicate-param:

```
                ( define  square  ( duplicate-param  * ) )
                ( define  double  ( duplicate-param  + ) )
```

Note that this uses the simple form of define that assigns a computed value to a name. In the expressions above the functions returned by double-param are assigned to the names square and double respectively.

We can use the same idea to create a compose function from two other function:
A compose function is denoted as $h = f \circ g$ or $h(x) = f(g(x))$

```
        ( define  ( compose  f  g )
        ( lambda  ( x )  ( f  ( g  x ) ) )   )
```

for example we can use compose to define a new function newF:

```
    ( define  newF  ( compose    ( lambda ( x )  ( -  0  x ) )
                                 ( lambda ( x )  ( *  x  x ) )   ) )
```

creates a function newF which is the composite of f (x) = -x and g(x) = x^2. Hence

```
        ( newF       10 )                        returns      -100
```

Yet another example is the function makepair which takes two functions f and g as parameters and produces a function which takes a list of two elements and applies the first function to the first element of the list and the second function to the second element of the list and returns the results as a list.

```
    ( define ( makepair f  g )
        ( lambda  ( x  y )  ( cons ( f  x )  ( cons ( g  y ) '(  )  ) ) )   )
```

Examples of its use:

```
    ( define FOne  ( makepair  negate  square ) )         [negate returns –x]
        (FOne 10 20 ) will return ( -10  400 )

    (define fTwo  ( makepair car  cadr ) )
        ( fTwo  '( a b c)  '( 1  2  3 4) )  will return ( a  2 )
```

A final example: construct a function removeBuilder which will return a function which will remove the elements of a list which satisfy a predicate f.

```
    ( define ( remBuilder   f )
        ( lambda ( L ) ( remove-if  f  L ) )
```

where remove-if is the function previously defined.

Examples of use:

```
( define remNegs        ( remBuilder ( lambda ( x ) ( < x  0 ) ) ) )
( define remPos         ( remBuilder ( lambda ( x ) ( > x  0 ) ) ) )

( remNegs ‘( 1 2 -3 -4 5 -6 )                          returns     ( 1 2 5 )
( remPos ‘( 1 2 -3 -4 5 -6 ) )                         returns     ( -3  -4  -6 )
```

Your turn:

Write a tail-recursive procedure to compute the length of an arbitrary list.

Create a function "buildSeries" which takes as input one list, and outputs a function. The function that is produced (by buildSeries) takes as intput an integer, and runs that integer through a series of computations as specified by the list. The computations can be any combination of addition, multiplication, and squaring a number, and are specified as follows: a positive number means addition, a negative number means multiplication, and a zero means squaring. For example, if buildSeries were called as follows:

```
(buildSeries '(5 0 -4 -2  8) )
```

a function would be produced that takes as input an integer, adds 5, squares it, multiples by 4, multiplies by 2, and adds 8. Thus, if the original call had been made as follows:

```
(define S ( buildSeries '(5 0 -4 -2  8) ) )
```

then the the produced function S would behave as follows:

```
(S 4)   *** would return 656, because 4+ 5 = 9 squared = 81 * 4 = 324 * 2 = 648 + 8 = 656
(S -3)  *** would return 40, because ( (-3 + 5 ) 2 * 4 * 2 ) + 8 = 40
```

Your task is just to write buildSeries. You may find it useful to write one or more utility functions. Of course, buildSeries should work for ANY input list of integers, not just the one shown above.

Using remBuilder define a function remZero which will remove the zero from a list of integers.