In pure functional programming a program is viewed as a function which can be called with a variety of actual parameters such a call constitutes the input for this run of the program. It is built by defining it in terms of other functions which are themselves defined in terms of functions etc. until, at bottom, function definitions are based on a few Data-Structures and built in functions. In LISP and Scheme, the main the data-structure is the List- ex. ‘( a b c 23 5). Basic built in commands can pick out the first member of a List (car), the List that remains when the first member is removed (cdr), and a List constructor which can extend a List by a new firstmember (cons).

In pure functional programming the functions are all call by value; there are no side-effects, every function returns a value. Throughout one uses prefix notation within parenthesis ex. (f a b c). There are no assignment statements. The only “variables” are formal parameters. In the early versions, of such languages, Lisp and Scheme, there is no provision for data type declarations, and a variable may take any type as long as the operations with which it is associated make sense for that type. (Elegant data structure definition facilities are found in functional languages developed later, ex. ML),

The management of storage is implicit-done by the interpreter. Also, typically, functions are first class values, that is: a function can be the argument of a function, and a function can be returned by a function. Furthermore function definitions within a function definition is allowed, so calling a function results in defining a new function. This brings in a question of scoping - in Scheme scoping is static (lexical) (depends on the initial program nesting) (depends on the call), while in older Lisp’s it is dynamic (depends on the call).

Conceptually such a language is simple, uniform, elegant and of course has the full expressive power of any language. Without assignments however the same function on the same arguments will have to be repeated when needed at more than one place in the same definition (SCHEME has a let instruction which serves for limited form of assignments.. Without typing the compiler (if there is one) has no opportunity to catch many errors which will only become apparent at run time. A relatively minor problem is the large number of parenthesis require in Lisp and Scheme, for example, to define a function to return the reverse of the List given as a parameter requires about 20 parenthesis as shown.

\[ \text{Example} \]
\[
\begin{cases}
\text{define} (\text{reverse} \ L) (\text{cond} \ [\ (\text{null?} \ L) \ () \ ] \ [\ \text{else} \ (\text{cons} \ (\text{lastmember} \ L) \ (\text{reverse} \ (\text{headlist} \ L))) \ ] \ ) \ )
\end{cases}
\]

Different types of parenthesis are used here to help differentiate what is delineated.

We only present a limited subset of the commands in Scheme since we we are focussing on its pure functional aspects..

Although compilers exist for this language, an interpreter handles our version. The execution model to keep in mind is the (print (eval (read))) loop in the interpreter; it reads an S-expression, evaluates it, and then prints the result. In evaluating an S-expression, \((e_1 \ e_2 \ e_3 \ ... \ e_k)\), a sequence of events occur:

1. (1) evaluation of \(e_1\) to get the name of a function (often evaluation is trivial because the name of the function is given explicitly.)
2. (2) evaluation of \(e_2, e_3, \ ... \ e_k\) to get values of the arguments of this function.
3. (3) apply the function to these values.

An obstacle for beginning Scheme programmers is remembering that this evaluation takes place. For example, typing \((1 \ 2)\) at the Scheme prompt would result in an error because the system would not have a function named \(1\). We can inhibit this evaluation in Scheme by quoting the object. For example, typing \('(1 \ 2)\) which literally represents the list whose first member is 1 and second member 2 would have resulted in Scheme writing back to us the list \((1 \ 2)\).
1 Some **Primitive (Builtin) Operations**

**The List is SCHEMES Major Data Structure.**

A list is represented by `(l₁ l₂ ... lₙ)` where `lᵢ` can be list or an element (symbol, boolean, number, function, or itself a list) ex (it seems that) (it (seems that) you)

A symbol, or a list  may be interpreted literally or as variable or function names-quote ‘(...)’ makes the difference.

The functions car and cdr are used to get at elements or sublists of a list.

**car** is used to extract the first element of a list; that is, `(car '(1 2))` is (returns) 1 and `(car '((1) 2))` is (1). The difference between these two lists `(1 2)` and `((1) 2)` is that the first one has two elements, the atoms 1 and 2, whereas the second one has two elements, the list (1) and the atom 2.

**cdr** is used to get at the rest of the list, leftover after a car operation. Therefore, `(cdr '(1 2))` is the list (2) and `(cdr '((1) 2))` is the list (2) as well. Note that after we take car or cdr of a list that list remains unchanged; nothing is destroyed in it. This is sometimes called copy semantics. These functions are merely a way of delineating sub-lists within a list.

These two functions can be nested; for example, 

```
(car (cdr '(1 (2 3) 4))) = (2 3) because (cdr '(1 (2 3) 4)) = ((2 3) 4) whose car is (2 3). One can write this combined function in shorthand notation as (cadr '(1 (2 3) 4)), thereby using the ‘d” and ‘a” to indicate the order of application of the car’s and cdr’s. cons is used to construct lists from an element and a list (i.e., a car and a cdr). Thus, (cons 1 '(2 3)) is (1 2 3) and (cons '(1) '(2 3)) is ((1) 2 3).
```

**Booleans**

**Unary**

`null?` is a unary predicate `ex. (null? E)` which returns #t if its argument, `E`, is `NIL` (i.e., empty list () or the atom `NIL`) and `NIL` otherwise.

`zero?` is a unary predicate which returns #t if its argument has value 0 and `NIL` otherwise. `integer?` is a unary predicate which returns true if its argument is an integer and `NIL` otherwise.

`list?` is a unary predicate that returns true if its argument is a list and `NIL` otherwise.

`char?` `number?`

**Binary**

`, >, =` Arithmetic Less than, greater, =, (`>` x y `)

`eqv?` Non Arithmetic

**Conditionals**

`cond` is the conditional operator in Scheme. Its syntax is:

```
(cond (c₁ f₁) (c₂ f₂) ... (cₖ fₖ))
```

The `cᵢ` are S-expressions for conditons (ex. `(> a b)`). which evaluate to true #t or false #f (often `NIL` or `()`),

The `fᵢ` are S-expressions which return values ex. functions or constants.

First `c₁` is evaluated. If it is true then `f₁` is evaluated and returned as the value of the cond. Otherwise, `c₂` is evaluated and checked for truth value etc. We continue to evaluate `c₃`, `c₄`, `c₅`, ... until we find the first true value. Then the corresponding expression `fᵢ` is evaluated and its value returned as the value of the cond. Often `cₖ` is `else`, which always evaluates to true so that the last case "catches" all other cases. If no `cᵢ` evaluates to true, then the cond expression returns `NIL`.

```
(if (C₁) (E₁) (E₂))
If condition C₁ is satisfied do E₁ else do E₂
```

**Arithmetic**

`+,-,*,/`
First evaluate each $E_j$ independently as though the others did not exist. $F$ which is an S expression involving $x_k$ is then evaluated with the value of $E_k$ replacing $x_k$. The value of $F$ is returned by the $let$ S expression.

Evaluate each $E_j$ in order valuing any earlier computed $x_{j-k}$ appearing therein with that of $E_{j-k}$ and finally $F$ valuing each $x_k$ appearing therein with that of $E_k$. The value of $F$ is returned by the $let^*$ expression.

**First Class Functions: Built-ins:**

- $map (map f L)$ applies $f$ to every element of $L$.
- $(remove\_if cond L)$ removes every element in $L$ that satisfies $cond$.

The function $(map f x)$ can be defined by the user because an actual parameter can be a function. Its definition for any function $f$ is:

$$(define\ (map\ f\ x)\ (cond\ ((null?\ x)\ nil)\ ((f\ (car\ x))\ (map\ f\ (cdr\ x))))$$

$$(define\ (remove\_if\ fc\ x)\ (cond\ ((null?\ x)\ nil)\ ((fc\ (car\ x))\ (remove\_if\ fc\ (cdr\ x)))\ (else\ (cons\ (car\ x)\ (remove\_if\ fc\ (cdr\ x)))))$$

Use of let:

$$(define\ (remove\_if\ fc\ x)\ (let\ ((L\ (cdr\ x))\ (a\ (car\ x)))\ ;L=cdr\ x\ a=car\ x\ (cond\ ((null?\ x)\ nil)\ ((fc\ a)\ (remove\_if\ fc\ L))\ (else\ (cons\ a\ (remove\_if\ fc\ L)))))$$

The formal parameter $fc$ is a unary condition like equal? or null? in a call of remove\_if $fc$ would be replaced by an actual parameter.

Definition

$$\begin{align*}
(\text{define } x\ (+\ 1\ x))\ & \text{defines a list} \\
(\text{define } (add1 x)\ (+\ 1\ x))\ & \text{defines a function This function can be used; for example, (add1 2) would return 3.}
\end{align*}$$

We can define a function where its call is needed without actually naming using the anonymous lambda.

$$(\text{define } (\text{third} \ l)\ (\text{car} (\lambda (z) (\text{cddr} z)\ L)))\ z\ \text{is a formal argument of function, cddr is applied to that argument and this function is applied to L.}$$

Here the lambda ($\lambda$) expression* defines a function which finds the sublist of the original list headed by the third element and continuing through the rest of the list.

**Comments:** (semicolon) is the comment character. If it appears anywhere on a line, the characters to the right of it are ignored as they are a comment.

Eliminating Common Subexpressions

Nested Calls and common subexpressions

$$(let\ (x_1\ E_1)\ (x_2\ E_2)\ ...\ (x_k\ E_k)\ F)$$

First evaluate each $E_j$ independently as though the others did not exist. $F$, which is an S expression involving $x_k$, is then evaluated with the value of $E_k$ replacing $x_k$. The value of $F$ is returned by the $let$ S expression.

$$(let^*\ (x_1\ E_1)\ (x_2\ E_2)\ ...\ (x_k\ E_k)\ F)$$

Evaluate each $E_j$ in order valuing any earlier computed $x_{j-k}$ appearing therein with that of $E_{j-k}$ and finally $F$ valuing each $x_k$ appearing therein with that of $E_k$. The value of $F$ is returned by the $let^*$ expression.

First Class Functions: Built-ins: map (map f L) applies f to every element of L. (remove_if cond L) removes every element in L that satisfies cond.

```
(define (sq x) (* x x))
(define (y '(1 2 3)) )
(map (lambda (x) (*x x)) y )
```

```
(define (eq0? x) (equal? 0 x) )
(define (y '(1 2 0 3)) )
(remove_if ( lambda (x) (equal? 0 x)) y )
```

The function $(map f x)$ can be defined by the user because an actual parameter can be a function. Its definition for any function $f$ is:

- **define and lambda** allow you to define Scheme functions, named and unnamed, respectively. For example, we can define the function that increments integer values by 1:

```
(define (third l) (car ( lambda (z) (cddr z) L)) z is a formal argument of function, cddr is applied to that argument and this function is applied to L. Here the lambda (\lambda) expression* defines a function which finds the sublist of the original list headed by the third element and continuing through the rest of the list.
```

```
(* (lambda (x) (car (lambda (z) (cddr z) L))) x) z is a formal argument of function, cddr is applied to that argument and this function is applied to L. Here the lambda (\lambda) expression* defines a function which finds the sublist of the original list headed by the third element and continuing through the rest of the list.
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

```
(define (third l) (car ( lambda (z) (cddr z) L)))
```

\* From the "\lambda calculus" was originally used in demonstrating the universatility of pure functional language by F. Church (@ Princeton). Lisp was based on this notation.

© 2006 M C Paull
Defining Functions In Scheme:
The List is the basic data-structure and there are three fundamental built-in operations which operate on a list.

\[ car = \text{first member of } x \quad \text{cdr} = \text{last member of } x \]

- **car**: first member of the list.
- **cdr**: last member of the list.

\[ \text{LIST } x = (\text{cons } (\text{car } x) (\text{cdr } x)) \]

**Built-In**

**Defined**

- **firstmember** is the first element of a string.
- **taillist** = list after first member is removed.

The \{define (name args) \ldots\} is used to define new functions. So we can simply rename built-ins.

\{define (firstmember x) (car x) \}

\{define (taillist x) (cdr x) \}

**Defining Functions**

(a) [lastmember of list x is the first member of x if the taillist (cdr) of x is null (empty) otherwise it is the last member of taillist of x]

(b) lastmember(X) = firstmember(X) : if taillist(X) = ()

lastmember(X) = lastmember(taillist(X)) : if taillist(X) != ()

(c) 1 \{define (lastmember x) (cond \((\text{null? } (\text{taillist } x)) \ (\text{firstmember } x) \) \(\text{else} \ \text{lastmember}(\text{taillist } x)) \)}

\{define (lastmember x) (cond \((\text{null? } (\text{taillist } x)) \ (\text{firstmember } x) \) \(\text{else} \ \text{lastmember}(\text{taillist } x)) \)}

**Tail Recursion**

1') \{define (lastmember x) \(\text{cond} \((\text{null? } (\text{taillist } x)) \ (\text{firstmember } x) \) \(\text{else} \ \text{(let} \ (y \ (\text{taillist } x)) \ (\text{lastmember } y) \ )) \))\}

**Use Of Let**

Because

1) \{lastmember ’(a,b,c)\} = \{taillist (a,b,c) = \{b,c\}\}

\{lastmember ’(b,c)\} = \{taillist (b,c) = \{c\}\}

\{lastmember ’(c)\} = \{firstmember c = \{c\}\}

**Example Of Definition (lastmember):** Three representatives (a) English, (b) Mathematical (c) Scheme

(a) [headlist of list x = x if x has 1 member( taillist =()) otherwise \{first member of x \|| headlist list x- 1st member of x\}]

(b) headlist(X) = X : if taillist(X) = ()

headlist(X) = cons (fistmember(X), headlist(taillist(X))): if taillist(X) != ()

(c) 2 \{define (headlist x) \(\text{cond} \((\text{null? } (\text{taillist } x)) \ (\text{(()}) \) \(\text{else} \ \text{(cons} \ (\text{firstmember } x) \ (\text{headlist } (\text{taillist } x))))\)}

2') \{define (headlist x) \(\text{let} \ ((y \ (\text{taillist } x)) \ (z \ (\text{firstmember } x)) \) \(\text{cond} \((\text{null? } y) \ (\text{(()})) \) \(\text{else} \ \text{(cons} \ z \ (\text{headlist } x) \ )) \))\}

**Use Of Let**

Because

2) \{headlist ’(a,b,c)\} = \{\text{taillist (a,b,c) = (b,c)}\}

\{\text{taillist (b,c) = (c)}\}

\{\text{headlist (c) = (a)}\}

\{\text{cons b ()} = (b)\}

\{\text{cons a (b)) = (a b)\}

**Examples Call**

**Example Of Definition (headlist):** Three representatives (a) English, (b) Mathematical (c) Scheme
There is a built-in `append` function, but this may also be defined using the basic list functions:

```scheme
(define (append X Y) (cond ((null? X) Y) (else (cons (firstmember X) (append (taillistl X) Y)))))
```

**PROLOG PARAPHRASE**

`append (X, Y, Y) :- null?(X)`

`append (X, Y, Z) :- firstmember(X,A), taillstl X,B, append (B, Y, C), cons(A, C, Z).`

```scheme
(define (revs L) (cond ((null? (taillist L)) L) (else (append (revs (tail L)) (list (first L))))))
```

**PROLOG PARAPHRASE**

```
(revs (a b c)) = (append (revs (b c)) (list a))
(append (revs (b c)) (a))
(append (append (revs (c)) (list b) (a))
(append (append (revs (c)) (b) (a))
(c b a)
```

```scheme
(define (revs L) (cond ((null? (taillist L)) L) (else (append ((list(lastmember L)) (revs (headlist L))) (list (first L))))))
```

**PROLOG PARAPHRASE**

```
(revs (a b c)) = (append ((c) (revs (a b) ) (a))
(append ((c) (append (b) (revs (a) ) )
(append ((c) (append (b) ( a) )))
(c b a)
```

**Conditionals:** Functions which return #T (True) or #F (false)

```scheme
(define (size? L p) (equal? (length L) p))
(define (size L) (cond ((size? L 0) 0) (else (+ (size (cdr L)) 1))))

(define (member? x L) (cond ((null? L) #F) (((equal? x (car L)) #T) (else (member? x (cdr L))))))

(define (rp? L) (cond ((null? (cdr L)) #F) (((member? (car L) (cdr L)) #T) (else (rp? (cdr L))))))
```

[Look for repeats in list]
More complex functions can be developed top-down. The final function F is defined assuming the existence of
simpler functions. Then those simpler functions are defined. This is done in a sequence of stages. In the
sorting examples that follow the final sorting function defined in terms of simpler functions is the last
one defined—the simpler functions are designed earlier in stages with the simpler functions designed
earliest.

A. To Build a Sorter: By finding largest member of unsorted list and putting in front of sorted version
1. Find Largest member of list L (MAX) \( \text{maxlst}(L) \) returns MAX
2. Remove a member (MAX) from list \( \text{rmv}(X,L) \) returns \( L - MAX \)
3. sort\( (L) = \text{MAX} || L - \text{MAX} \)

\[
\text{define} \ (\text{bigger} \ X \ Y) \ (\text{cond} \ (\ (> \ X \ Y) \ X) \ \text{[} > \text{built-in (type by operator)}\)\
(\ else \ Y)\)
\]

\[
\text{define} \ (\text{maxlst} \ L) \ (\text{cond} \ (\null? \ (\text{tail} \ L)) \ (\text{first} \ L) \ \text{[} \text{returns single element in list} \)\
(\ else \ (\text{bigger} \ (\text{first} \ L) \ (\text{maxlst} \ (\text{tail} \ L))))\)
\]

\[
\text{define} \ (\text{rmv} \ X \ L) \ (\text{cond} \ (\null? \ L) \ () \ \text{[} \text{if L empty returns}()\])\
(\ (= \ (\text{first} \ L) \ X) \ (\text{tail} \ L) \ \text{[} \text{if L has 1 member} \)\
(\ else \ (\text{cons} \ (\text{first} \ L) \ (\text{rmv} \ X \ (\text{tail} \ L))))\]) \ \text{[returns a list with (one) maximum removed]}\)
\]

\[
\text{define} \ (\text{sort} \ L) \ (\text{cond} \ (\null? \ (\text{tail} \ L)) \ L \ \text{[} \text{L empty}\)\
(\ else \ (\text{let} \ (\text{max} \ (\text{maxlst} \ L)) \ (\text{cons} \ \text{max} \ (\text{sort} \ (\text{rmv} \ (\text{maxlst} \ L) \ L))))\) \ \text{max first}\)
\]

Use (remove_if ?null L)

\[
\text{define} \ (\text{sort1} \ L) \ (\text{cond} \ (\null? \ (\text{tail} \ L)) \ L \ \text{[} \text{L empty}\)\
(\ else \ (\text{let} \ (\text{max} \ (\text{maxlst} \ L)) \ (\text{cons} \ \text{max} \ (\text{sort} \ (\text{rmv} \ (\text{maxlst} \ L) \ L))))\) \ \text{max first}\)
\]

Use Of Let

( sort \ [5 3 4])
(cons (maxlst [5 3 4]) (sort (rmv (maxlst [5 3 4]) [5 3 4] )) )
(cons 5 (sort (rmv 5 [5 3 4] )))
(cons 5 (sort [3 4] ))
(cons 5 (cons (maxlst [3 4]) (sort (rmv (maxlst [3 4]) [3 4] )) )
(cons 5 (cons 4 (sort (rmv 4 [3 4] )))
(cons 5 (cons 4 (sort [3] ))
(cons 5 (cons 4 (cons (maxlst [3]) (sort (rmv (maxlst [3]) [3] )) )
(cons 5 (cons 4 (cons [3] ))
(cons 5 (cons 4 [3] ))
(cons 5 [4 3] )

[5 4 3]
B. To Build Sorter: By Merging 2 already sorted lists

1. Given 2 already sorted lists, L1 and L2, comparing pairs of members putting greatest in front of result
2. Find first and 2nd half of a list (firsthalf L), lasthalf L), returns halfs
3. sort(L) = sortmerge two halfs, and merge(firsthalf L)

(merge L1 L2): if either L1 or L2 is null return other if first of L1 is bigger than first of L2 return first of L1 cons-ed with merge of tail(L1) and L2

(define (merge L1 L2) (cond ((null? L1) L2) ((null? L2) L1) ((> (first L1) (first L2)) (cons (first L1) (merge (tail L1) L2))) (else (cons (first L2) (merge L1 (tail L2)))))

Trace 1
(merge (3 1) (6 5)) = else (cons 6 (merge (3 1) (5)) (cons 6 (cons 5 (merge (3 1) ()))) (cons 6 (cons 5 (3 1))) (cons 6 (5 3 1)) (6 5 3 1)

Trace 2
(merge 6 5) = (cons 6 (merge () (5)) (cons 6 5) (6 5)

(dec n) (- n 1)
(hlfn L) (/ (length L) 2)
(hlfn0 L) (floor (hlfn L))
(hlfn1 L) (ceiling (hlfn L))

(define (firstof n L) (cond ((= n 0) ()) (else (cons (first L) (firstof (dec n) (tail L)))))) [returns list consisting of the first n elements of list L]

(define (firsthalf L) (firstof (hlfn0 L) L))

(define (lastof n L) (cond ((equal? n (length L)) L) (else (lastof n (tail L)))))

(define (lasthalf L) (lastof (hlfn1 L) bL))

(define (sortmerge L) (cond ((null? L) L) ((null? (tail L)) L) (else (merge (sortmerge (firsthalf L)) (sortmerge (lasthalf L)))))) [break list in half sortmerge each half and the merge those sortmerge d halfs.]

sortmerge (3,1,5,6) = (merge (sortmerge (3,1), sortmerge (5,6),) = (merge (merge(sortmerge(3), sortmerge(1) ) (merge(sortmerge(5), sortmerge(6) ) = (merge (merge (3), (1) ) (merge (5), (6) ) = (merge ( (cons 3 (1 ) ) (merge (cons 6 (5) ) = (merge ( (3 1 ) ) (merge ( 6 5) )

© 2006 M C Paull
HIGHER ORDER FUNCTIONS

Functions Can Be Arguments:
Functions can be defined which take functions as an argument. Here is a function, applst1 which is used to apply another function, f, to every member of a list to produce a new list.

```
(define (applst1 f L) ( cond ( (null? L) ( ) )
                           ( else (cons (f (firstmember L)) (applst1 f (tailistl L)))) ) )
```

Applying f to the ith member of 2 lists to get the ith member of a resultant list

```
(define (applst2 f L1 L2) ( cond ( (null? L1) ( ) )
                                   ( (null? L2) ( ) )
                                   (else (cons (f (firstmember L1) (firstmember L2)) (applst2 f (tailistl L1) (tailistl L2)))) ) )
```

So given the additional two definitions:

```
(define (double X) (* 2 X) ) and (define (sum X Y) (+ X Y) )
```

We can define:

```
(applst1 double L) return a list, L2, in which each entry is twice that of the entry in L
(applst1 sum L1 L2) return a list, L3, in which each entry is the sum of the corresponding members in L1 and L2.
```

The Builtin map Function and lambda system for Local Function Definition

There is, in fact, a built-in function, map which can be used to apply a function, f, to a list L. This is the map function.

```
(map f L) [returns result of appying f to each member of L]
(map double L) [multiplies each member of L by 2 using double as defined above]
```

```
(define (incr1 x) (+ x 1))
```

```
(define (incr L) (cond ( (null? L) '() )
                        (else cons (incr1 (car L)) (incr (cdr L)))) )
```

```
(define (Lincr L) (map incr1 L) )
```

There is also a way of defining a function locally:
```
lambda (x) ( f x ) in place of a definition of a function returns result of applying f to x
```

USES OF lambda and map

```
(define (mapconcat L) (map (lambda (x) (cons x '(a))) L) ) [ concatenatess a to every member of L ]
```

```
(define (Ldouble L) (map (lambda(x) (* 2 x)) L) ) [ multiplies each member of L by 2 like above, but a separate definition of double is not required ]
```

© 2006 M C Paull
Cumulating Functions

if \( L = (a_1 \ a_2 \ a_3) \) a cumulating function, say \((\text{cum } f \ L)\) returns \(f \ (a_1 \ f \ (a_2 \ a_3))\)

\[
\text{define (cum } f \ L \) \ ( \text{cond ( (null? (tail } L) \) (first } L) \) \)
\[
\quad \text{else ( (first } L \) (cum } f \ (\text{tail } L))) \)
\]

Or one can define a cumulating function for a particular function, f, ex. +

\[
\text{define (Lsum } L \) \ ( \text{cond ( (null? (tail } L) \) (first } L) \) \)
\[
\quad \text{else ( (first } L \) (Lsum (tail } L))) \)
\]

\[\text{Lsum if } L = (a_1 \ a_2 \ a_3) \text{ returns } (+ \ a_1 \ (+ \ a_2 \ a_3))\]

All Permutations of the first n digits

IDEA:

\[
\begin{align*}
( & (1) \\
( & (21) \\
( & (231) (213) (321) (312) (132) (123) \\
( & (4321) (3421) (3241) \\
\end{align*}
\]

\[
\text{define (first } n \ L) \ ( \text{cond ( (= n 0) '()} \\
\quad \text{else ( cons (car } L \) (first (- n 1) (cdr } L))) \))
\]

\[
\text{define (remain } n \ L) \ ( \text{cond ( (= n 0) } L) \\
\quad \text{else ( remain (- n 1) (cdr } L)) \))
\]

\[
\text{define (concat3 } L1 \ n \ L2) \ (\text{append } L1 \ (\text{cons } n \ L2))
\]

\[
\text{define (insert1 } n \ p \ L) \ (\text{concat3} \ (\text{first } p \ L) \ n \ (\text{remain } p \ L))
\]

\[
\text{define (insert } n \ p \ LL) \ ( \text{cond ( (null? } LL) '()) \\
\quad \text{else ( append (list } (\text{insert1 } n \ p \ (\text{car } LL)) \ (\text{insert } n \ p \ (\text{cdr } LL))) \))
\]

\[
\text{define (insertAll } n \ p \ LL) \ ( \text{cond ( (< p 0) '()) } \\
\quad \text{else ( append (insert } n \ p \ LL) \ (\text{insertAll } n \ (- p 1) \ LL))) \))
\]

\[
\text{define (inAll } n \ LL) \ (\text{insertAll } n \ (- n 1) \ LL) \\
\text{define (innest } n) \ ( \text{cond ( (= n 1) '()) } \\
\quad \text{else ( inAll } n \ (\text{innest (- n 1) }) \))
\]

© 2006 M C Paull
x = [ab z q www]

z = [12 653 y]

z = [12 653 [zzz end]]

y = [zzz end]

now x = [ab [12 653 [zzz end]] q www]

if functions are
defined with result(car (cdr x)) = chg

z = (cdr x)

Now there is no way to reach list *

if instead of (car y) = 'zzz, (car y) had been made = z then the reference count at * would still be 1
and Eager Garbage Collection would not have removed anything so only Lazy Garbage
collection would have found the unreachable cells
Fibonacci Original Recursive Definition Not Tail Recursion
f(n) = 1 : n = 0, 1
f(n) = f(n-1) + f(n-2) : n > 1

f(4) ← f(2) ← f(0) = 1 a

f(5) ← f(3) ← f(1) = 1 b

Fibonacci Recursive Definition Tail Recursion
f2(n) = g(0, 1, n)
g(a, b, n) = a : n = 0
g(a, b, n) = g(a+b, a, n-1) : n > 0

f2(5) = g(1, 0, 5) = g(1, 1, 4) = g(2, 1, 3) = g(3, 2, 2) = g(5, 3, 1) = g(8, 5, 0) = 8

Because its Tail Recursion It can be implemented by a simple while procedure f2(n)
{return g(1, 0, n); }
procedure g(int a, int b, int n)
while(n > 0)
{ temp = a + b;
  b = a;
  a = temp;
  n = n - 1
} return a;

Greatest Common Divisor Tail Recursion
a, b are integers
1 h(a, b) = a : a = b
2 h(a, b) = h(a-b, b) : a > b

Proof that it gives Greatest Common Divisor.
Claim
h(a, b) = gcd(a, b) : a > b that is : a/h(a, b) = integer &
b/h(a, b) = integer and h(a, b) is the largest such divisor)

Proof:
1 is obvious
2 gcd(a, b) =? gcd(a-b, b) : a > b
Need to prove that gcd(a, b) = gcd(a-b, a) if a > b
by definition: a/gcd(a, b) = int1, b/gcd(a, b) = int2
and gcd(a, b) is the largest such number)

therefore the same gcd(a, b)
(a - b)/gcd(a, b) = a /gcd(a, b) (a - b )/gcd(a, b) = int1-int 2
= int3 and since integer 3 divides (a-b) it is = gcd(a-b, b).

3 Like 2

Because its Tail Recursion It can be implemented by a simple while procedure h(int a, int b)
while(a != b)
{ if(a > b) a = a-b;
  if(a < b) b = b-a
} return a;