`cons`, `car`, `cdr`

Who came up with these names?

`cons` stands for *construct*.

From original Lisp implementation on IBM 704 where "address" or "decrement" part of memory could be accessed.

`car` stands for "Contents of Address part of Register"

`cdr` stands for "Contents of Decrement part of Register"

Why do we still use these names? Because of contractions:

```
(define (cadr x) (car (cdr x)))     ;; second element of list
(define (cdar x) (cdr (car x)))     ;; ??
(define (cadaaar x) (car (cdr (car (car (car x))))))
```


List Dissecting

```
(define (sum lst)
   (if (null? lst)
       0                             ; base case
       (+ (car lst) (sum (cdr lst)))      ; recursive call
   )
)
```

and

```
(define (double-list lst)
   (if (null? lst)
       '()
       (cons (double (car lst)) (double-list (cdr lst)))
   )
)
```


Linear Recursion

```
guile> (sum '(1 2 3))

(sum '(1 2 3))
=> (+ 1 (sum '(2 3)))
=> (+ 1 (+ 2 (sum '(3))))
=> (+ 1 (+ 2 (+ 3 (sum '()))))
=> (+ 1 (+ 2 (+ 3 0)))
=> (+ 1 (+ 2 3))
=> (+ 1 5)
=> 6
```


Tail Recursion

Linear Recursion builds up call stack to depth of recursion.

If last thing a function does is make a recursive call, that recursive call's stack frame replaces the current functions stack frame.

```
;; return last element in list
(define (last lst)
   (if (null? (cdr lst))
       (car lst)
       (last (cdr lst))
   )
)
```

Linear Recursion

```
guile> (last '(1 2 3))

(last '(1 2 3))
=> (last '(2 3))
=> (last '(3))
=> 3
```


Tail Recursive `sum`

Can we make `sum` tail recursive?

Trees

Lists can contain lists.

```
'(1 (2 3) ((4) 5))
```

Tree Recursion:

```
(define (sum-tree t)
   (cond ((null? t)   0)
         ((number? t) t)
 ((pair? t)   (+ (sum-tree (car t)) (sum-tree (cdr t))))
   )
)
```


Lists and Memory

Appending two lists:

```
(define (append first second)
   (if (null? first)
       second
       (cons (car first) (append (cdr first) second))
   )
)
```

What happens below?

```
(define x '(1 2 3))
(define y '(a b c))
(define z (cdr x))
(define w (append x y))
```

Numbers

Three types of numbers *integers*, *reals*, and *rationals*.

There are different internal representations for the same number.

Integers can be any size.

Ever wonder what `(factorial 1000)` is?

Equality

Testing for equality:

`eq?` tests pointer equality. Equal numbers may not be `eq?`

`eqv?` tests pointer and number equality.

`equal?` tests *deep* equality (lists, trees, numbers, not procedures).

`=` tests number equality only. Non numbers are errors.

Equality Examples

How about?

```
(eq? 5 5)
(eq? 'the-little-red-hen 'the-little-red-hen)
(eq? "hi there" "hi there")
(eq? (list 1 2) (list 1 2))
(eq? (factorial 200) (factorial 200))
(eqv? 5 'a)
(eqv? (factorial 200) (factorial 200))
(equal? (list 1 2) '(1 2))
(= 5 'a)
```


First Class Functions

Like numbers, Scheme functions are first class citizens.

They

- can put stored in variables or lists,
- can be passed to other functions, and
- can be returned from other functions.

1st Class Func Examples

Consider:

```
guile> (define x 10)
guile> +
#<primitive-procedure +>
guile> x
10
```

What is special about +?

```
guile> (define plus +)
guile> (plus 1 2 3 4 5)
15
```


Data Abstractions

Providing an interface to a data type. Actual implementation of data type can change.

Binary Tree abstract data type, provide *constructor* and *selectors*:

- `(make-tree <data> <left-child> <right-child>)` ⟶ `<tree>`
- `(tree-data <tree>)` ⟶ `<data>`
- `(tree-left-child <tree>)` ⟶ `<left-child>`
- `(tree-right-child <tree>)` ⟶ `<right-child>`

Someone can provide binary search tree facilities `insert`, `lookup`, and `delete` using this abstract binary tree.

Procedural Abstractions

Recall `double-list`. Other functions, e.g. `square-list`, have same structure.

```
(define (double-list lst)
   (if (null? lst)
       '()
       (cons (double (car lst)) (double-list (cdr lst)))
   )
)
```

Abstract that Structure:

```
(define (my-map func lst)
   (if (null? lst)
       '()
       (cons (func (car lst)) (my-map func lst))
   )
)
```


First-Order Funcs

First-Order Functions take other functions as arguments. E.g. `map`.

What do these do?

```
(map double '(1 2 3))
(map sqrt '(1 4 9))
(map symbol? '(3 a 6 b 9 c))
(map + '(1 2 3) '(4 5 6))
```

What if no existing function does exactly what you want?

Lambdas

Anonymous Functions: `lambda`

```
(map (lambda (x) (* x x)) '(1 2 3))
```

Compare

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

Rename this `mystery` function:

```
(define (mystery lst)
   (sum (map (lambda (x) 1) lst))
)
```


Abstracting Computations

Abstracting computational patterns gives, flexibility, power, and glue.

Can we abstract `sum` so that we can easily implement `product`?

```
(define (sum lst)
   (if (null? lst)
       0                                 ; base case
       (+ (car lst) (sum (cdr lst)))     ; recursive call
   )
)
```


Fold

Fold is useful for many things.

```
(define (append first second)
   (foldr cons second first)
)

(define (length lst)
   (foldr (lambda (x y) (+ 1 y)) 0 lst)
)

(define (product lst)
   (foldr + 1 lst)
)
```

We defined `foldr` (fold right) which starts at the rear and works forwards. `foldl` (fold left) does the opposite and is also useful.

Funcs as Return Values

Recall constant function `(lambda (x) 1)`.

In general we might want a constant function to return any constant.

```
(define (const-func val)
   (lambda (x) val)
)
```

Now, rewrite `mystery` as:

```
(define (length lst)
   (sum (map (const-func 1) lst))
)
```

Benefits . . .

Compare:

```
(define (length lst)
   (sum (map (const-func 1) lst))
)

(define (length lst)
   (if (null? lst)
       0
       (+ 1 (length (cdr lst)))
   )
)
```


Not Convinced?

How about:

```
(define (sum lst)
   (foldr + 0 lst)
)

(define (sum lst)
   (if (null? lst)
       0
       (+ (car lst) (sum (cdr lst)))
   )
)
```