Definición de funciones (cont.)

Tres constructos para flujo de control:

- `IF`
- ...

`(IF` predicate then_expression else_expression`)`

```
(DEFINE (factorial n)
  (IF (<= n 1)
      1
      (* n (factorial (- n 1)))
 ))
```

Definición de funciones (cont.)

Tres constructos para flujo de control:

- **IF**
- **COND**
- ...

(IF predicate then_expression else_expression)

```
(DEFINE (factorial n)
  (IF (<= n 1)
      1
      (* n (factorial (- n 1)))
))
```

Los predicados son evaluados de a uno, en orden a partir del primero, hasta que uno evalúe a **#T**; entonces, la expresión correspondiente es evaluada y su valor es devuelto como el valor de **COND**

Si ningún predicado evalúa a #T, entonces el valor de **COND** es el valor de la expresión asociada al **ELSE**

```
(COND
  (predicate_1 expression_1)
  (predicate_2 expression_2)
  ...
  (predicate_n expression_n)
  [(ELSE expression_n+1)]
)
```

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4)))
))
```

Definición de funciones (cont.)

Tres constructos para flujo de control:

- `IF`
- `COND`
- recursión

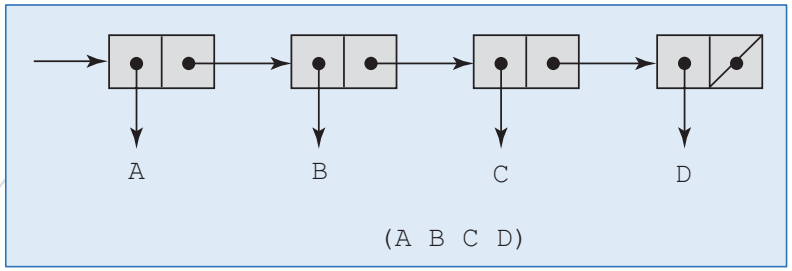
`(IF` predicate then_expression else_expression`)`

```
(DEFINE (factorial n)
  (IF (<= n 1)
      1
      (* n (factorial (- n 1)))
 ))
```

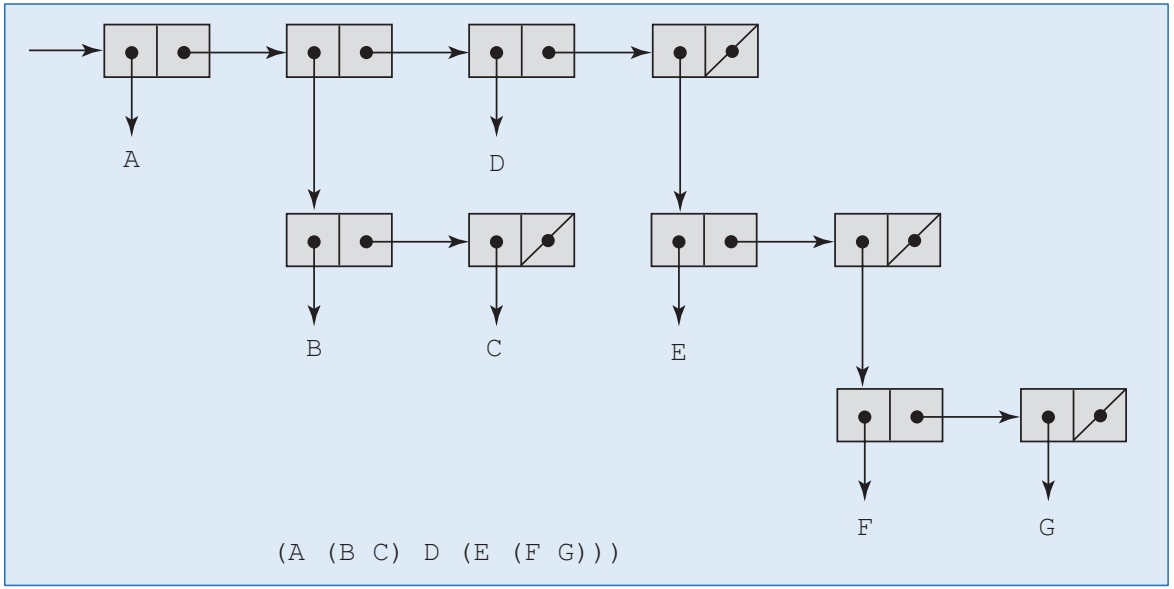
```
  (COND
    (predicate1  expression1)
    (predicate2  expression2)
    ...
    (predicaten  expressionn)
    [(ELSE  expressionn+1)]
  )
```

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4)))
))
```

Definición de funciones (cont.)



Listas



Definición de funciones (cont.)

Si un parámetro no tiene que ser evaluado (por **EVAL**) cuando se llama a la función que lo recibe, entonces se especifica como parámetro (a su vez) de la función primitiva **QUOTE**, la cual simplemente lo devueve sin modificarlo

`(QUOTE A)` returns `A`
`(QUOTE (A B C))` returns `(A B C)`

`(QUOTE (A B))` se puede escribir simplificadamente como `'(A B)`

Definición de funciones (cont.)

Las funciones primitivas **CAR** y **CDR** devuelven, respectivamente, el primer elemento y el resto de una lista (i.e., la lista sin su primer elemento), sin evaluar la lista ni sus elementos

```
(CAR '(A B C)) returns A
(CAR '((A B) C D)) returns (A B)
(CAR 'A) is an error because A is not a list
(CAR '(A)) returns A
(CAR '()) is an error
(CDR '(A B C)) returns (B C)
(CDR '((A B) C D)) returns (C D)
(CDR 'A) is an error
(CDR '(A)) returns ()
(CDR '()) is an error
```

```
(DEFINE (second a_list) (CAR (CDR a_list)))
```

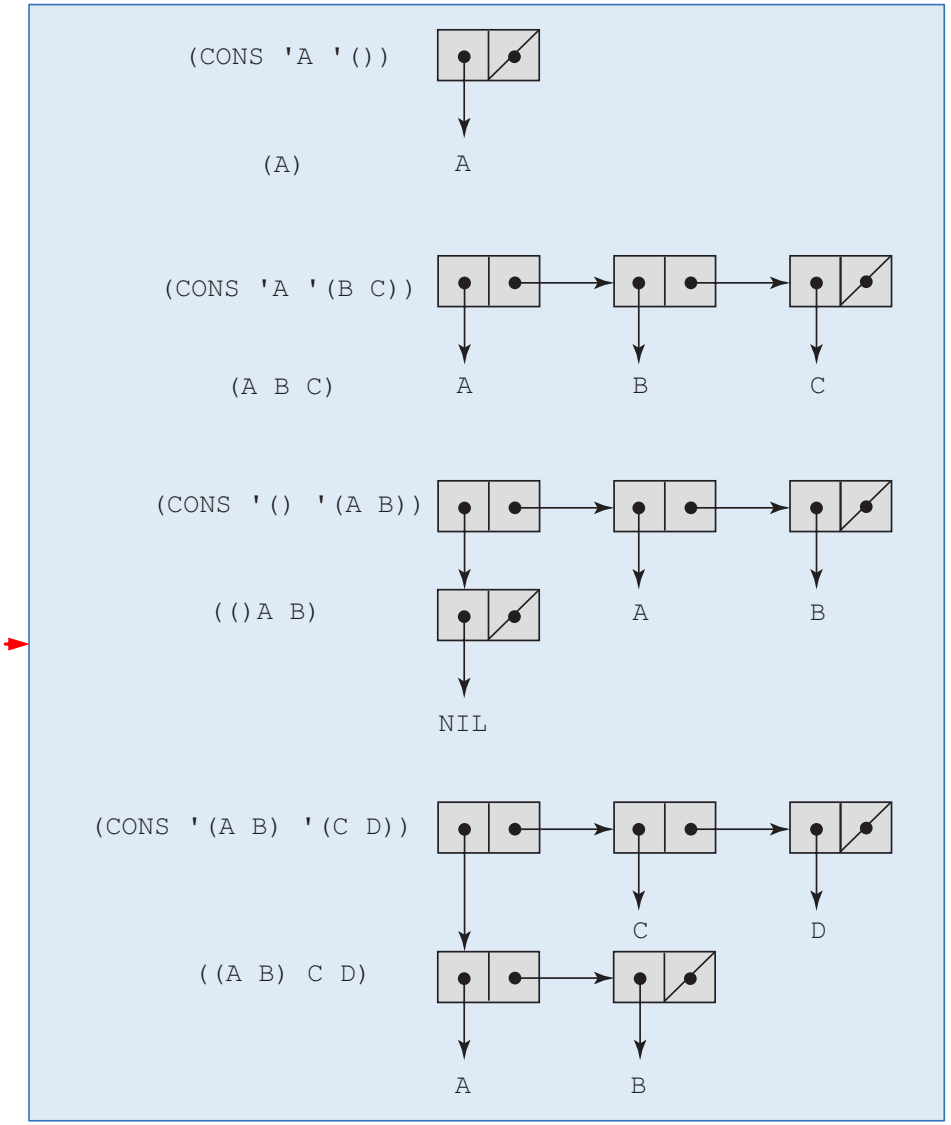
```
(second '(A B C))
```

... produce **B**

Definición de funciones (cont.)

La función primitiva **CONS** devuelve una lista, cuyo primer elemento es el primer parámetro y cuyos otros elementos son los elementos del segundo parámetro (una lista)

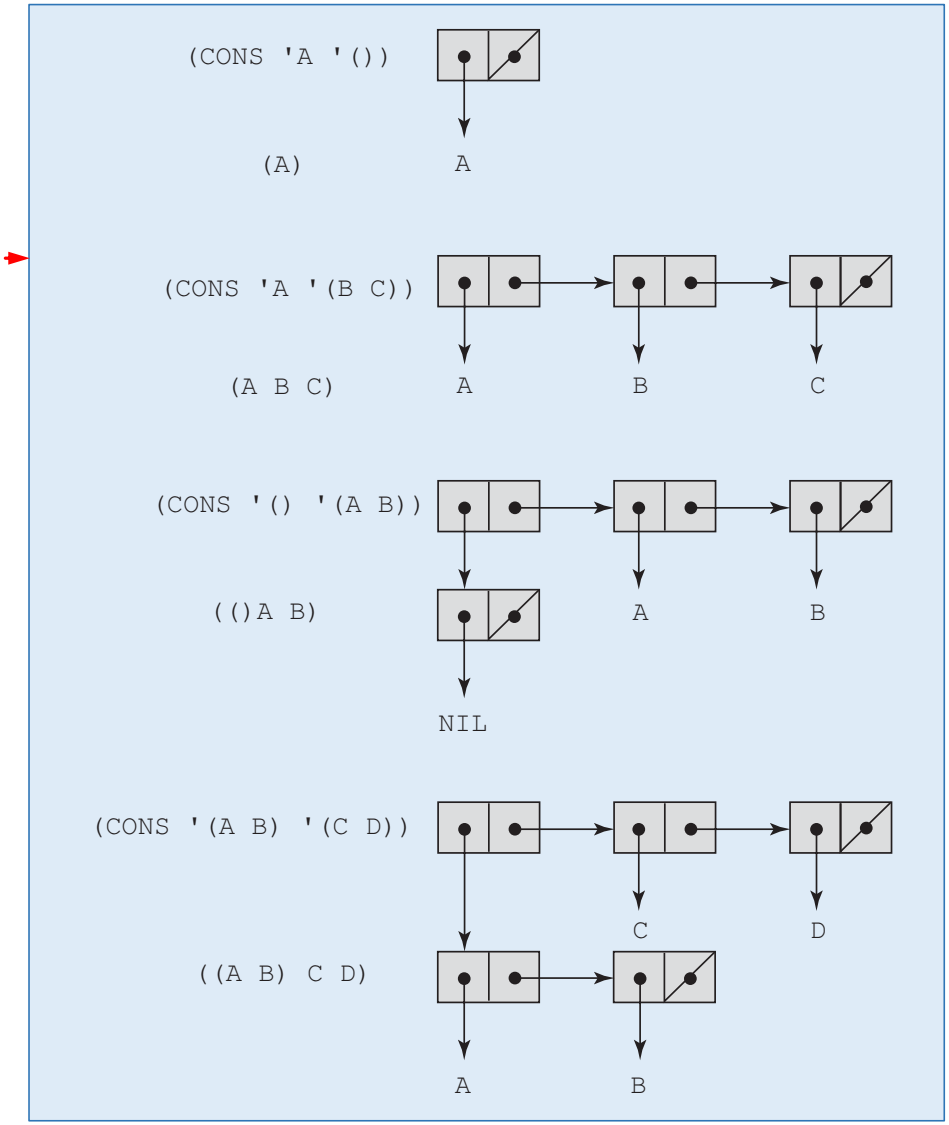
```
(CONS 'A '()) returns (A)
(CONS 'A '(B C)) returns (A B C)
(CONS '() '(A B)) returns (() A B)
(CONS '(A B) '(C D)) returns ((A B) C D)
```



Definición de funciones (cont.)

```
(CONS 'A '()) returns (A)
(CONS 'A '(B C)) returns (A B C)
(CONS '() '(A B)) returns (() A B)
(CONS '(A B) '(C D)) returns ((A B) C D)
```

(CONS ‘A ‘B) = (A . B)



Definición de funciones (cont.)

```
(CONS 'A '()) returns (A)
(CONS 'A '(B C)) returns (A B C)
(CONS '() '(A B)) returns (() A B)
(CONS '(A B) '(C D)) returns ((A B) C D)
```

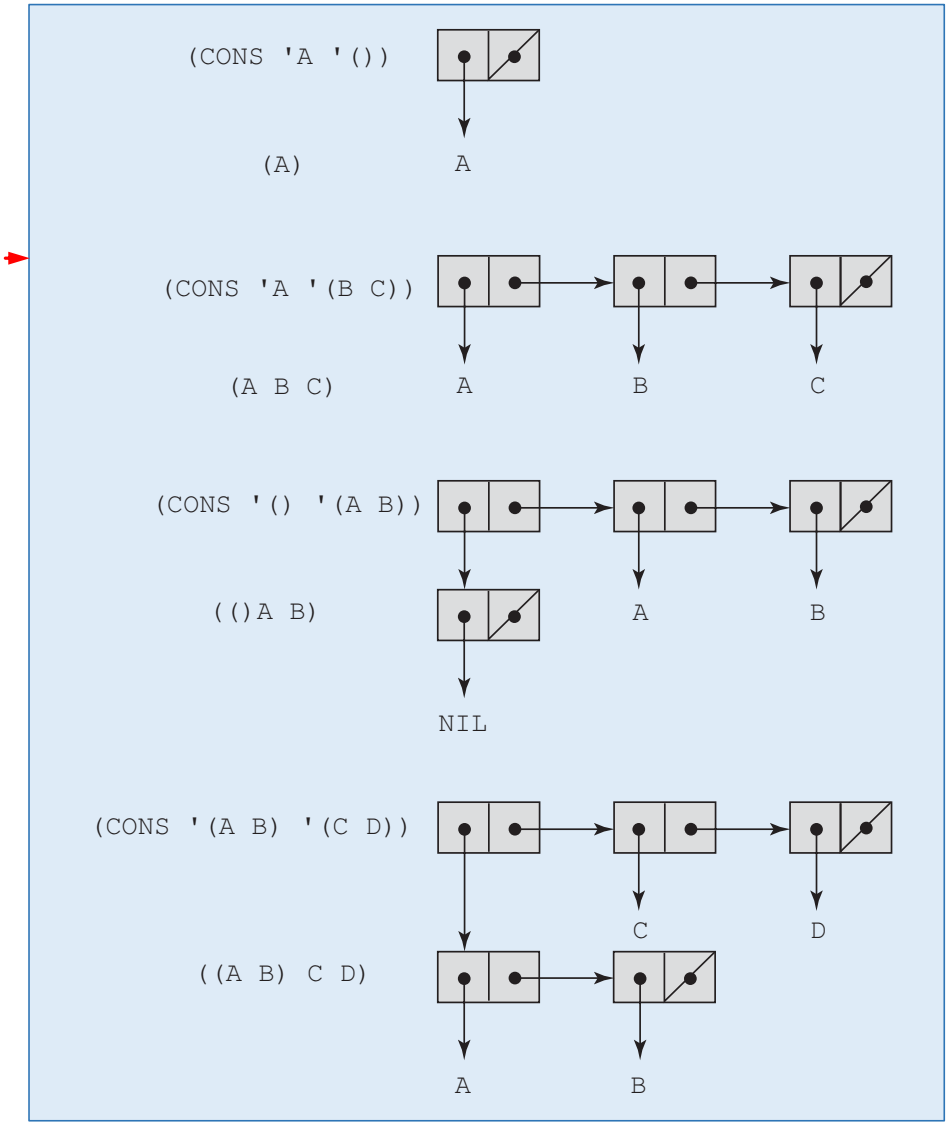
(CONS ‘A ‘B) = (A . B)

La función primitiva **LIST** es una versión abreviada de funciones **CONS** anidadas

```
(LIST 'apple 'orange 'grape)
```

```
(CONS 'apple (CONS 'orange (CONS 'grape '())))
```

```
(apple orange grape)
```

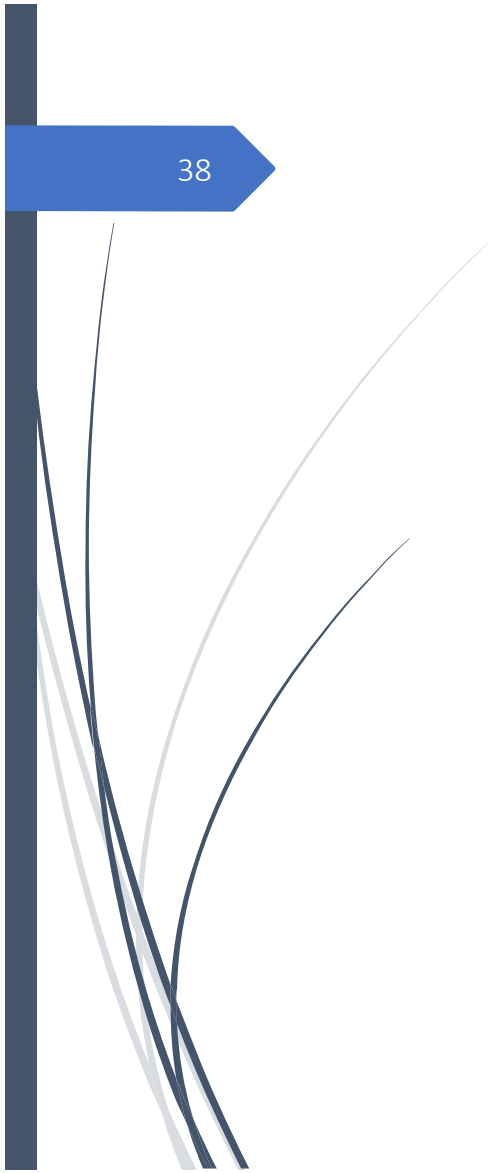


```
(EQ? 'A 'A) returns #T
(EQ? 'A 'B) returns #F
(EQ? 'A '(A B)) returns #F
(EQ? '(A B) '(A B)) returns #F or #T
(EQ? 3.4 (+ 3 0.4)) returns #F or #T
```

```
(LIST? '(X Y)) returns #T
(LIST? 'X) returns #F
(LIST? '()) returns #T
```

```
(EQV? 'A 'A) returns #T
(EQV? 'A 'B) returns #F
(EQV? 3 3) returns #T
(EQV? 'A 3) returns #F
(EQV? 3.4 (+ 3 0.4)) returns #T
(EQV? 3.0 3) returns #F
```

```
(NULL? '(A B)) returns #F
(NULL? '()) returns #T
(NULL? 'A) returns #F
(NULL? '(())) returns #F
```



```
(member 'B '(A B C)) returns #T
(member 'B '(A C D E)) returns #F
```

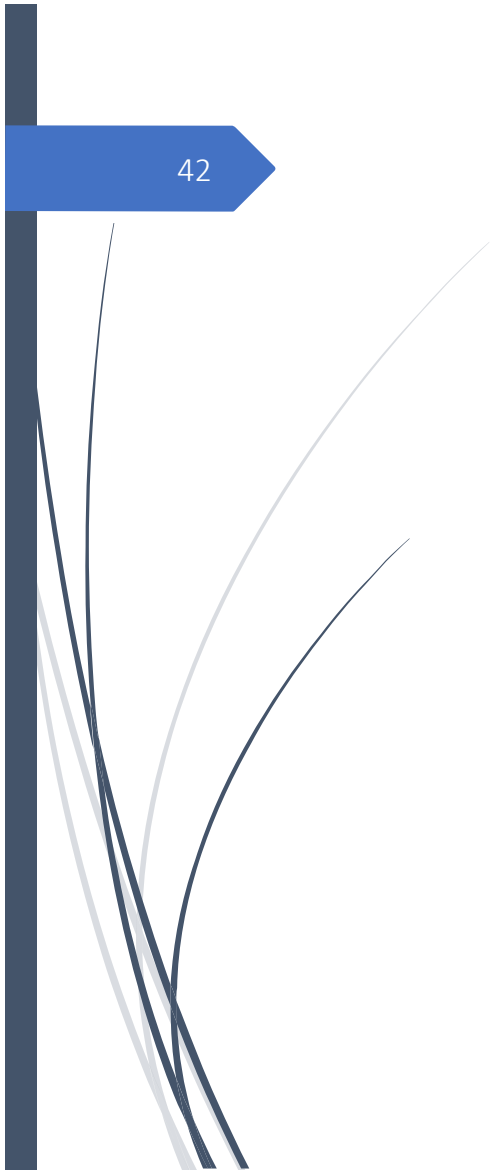
`(member 'B '(A B C))` returns `#T`
`(member 'B '(A C D E))` returns `#F`

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list)))
))
```

```
(DEFINE (equalsimp list1 list2)
  (COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

```
(DEFINE (equalsimp list1 list2)
  (COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1)) (EQ? list1 list2))
    ((NOT (LIST? list2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
            (equal (CDR list1) (CDR list2)))
    (ELSE #F)
))
```



`(append '(A B) '(C D R))` returns `(A B C D R)`
`(append '((A B) C) '(D (E F)))` returns `((A B) C D (E F))`

(append '(A B) '(C D R)) returns (A B C D R)
(append '((A B) C) '(D (E F))) returns ((A B) C D (E F))

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1) (append (CDR list1) list2)))
))
```

La función primitiva **LET** crea un *ámbito local*:

- los nombres $name_1$, ...$name_n$ son (temporalmente) asociados a valores de las expresiones correspondientes
- luego se evalúa la última expresión y se devuelve como valor de **LET**
- los nombres no pueden ser reasociados a nuevos valores en el mismo **LET** —no son variables locales
- los nombres podrían ser eliminados mediante la sustitución textual de las expresiones por los nombres en la última expresión del **LET**

```
(LET (
   (name1  expression1)
   . . .
   (namen  expressionn))
   expression
)
```

Cáulculo de las raíces de una ecuación cuadrática, suponiendo raíces reales

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
                (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
       )
  (LIST (+ minus_b_over_2a root_part_over_2a)
            (- minus_b_over_2a root_part_over_2a))
))
```

LET es una expresión **LAMBDA** aplicada a un parámetro:

- e.g., estas dos expresiones son equivalentes

```
(LET ((alpha 7))(* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
```

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
```

Composición funcional de **f** y **g**

```
(DEFINE (h x) (+ 2 (* 3 x)))
```

La función de composición funcional **compose**, en Scheme:

```
(DEFINE (compose f g) (LAMBDA (x)(f (g x))))
```

```
((compose CAR CDR) '((a b) c d))
```

CADR

```
((compose CDR CAR) '((a b) c d))
```

CDAR

CADDR

```
(DEFINE (third a_list)
  ((compose CAR (compose CDR CDR)) a_list))
```

Aplicación de **fun** a cada elemento de **a_list**

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list)) (map fun (CDR a_list))))
))
```

E.g., fun es elevar al cubo

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

(27 64 8 216)