Cálculo Lambda – sintaxis

<termino> :=: < variable> |

λ<variable> . <termino> | **Abstracciones funcionales**

(<termino> <termino>) **Aplicaciones funcionales**

Scheme – sintaxis

<expresion> :=: < variable> |

(lambda (<variable>) (<expresión>)) | **Abstracciones funcionales**

(<expresion> <expresion>) **Aplicaciones funcionales**

En scheme las funciones son datos de primera clase

Un tipo de dato se considera de primera clase si puede ser:

- El valor de un variable (esto es, puede ser nombrado)
- Un argumento de una función
- El valor que devuelve una función
- Componente de una estructura de datos mayor

DEFINICIÓN DE FUNCIONES

```
(lambda (x) (+ x 1))
➢#<procedure>
```

Definición de función sin nombre

```
(define suma_1 (lambda (x) (+ x 1)))
➢suma_1
(define resta_1 (lambda (x) (- x 1)))
➢resta_1
(suma_1 3)
➢4
(suma_1 (resta_1 3))
➢3
```

Si no tuviera nombre debo escribir

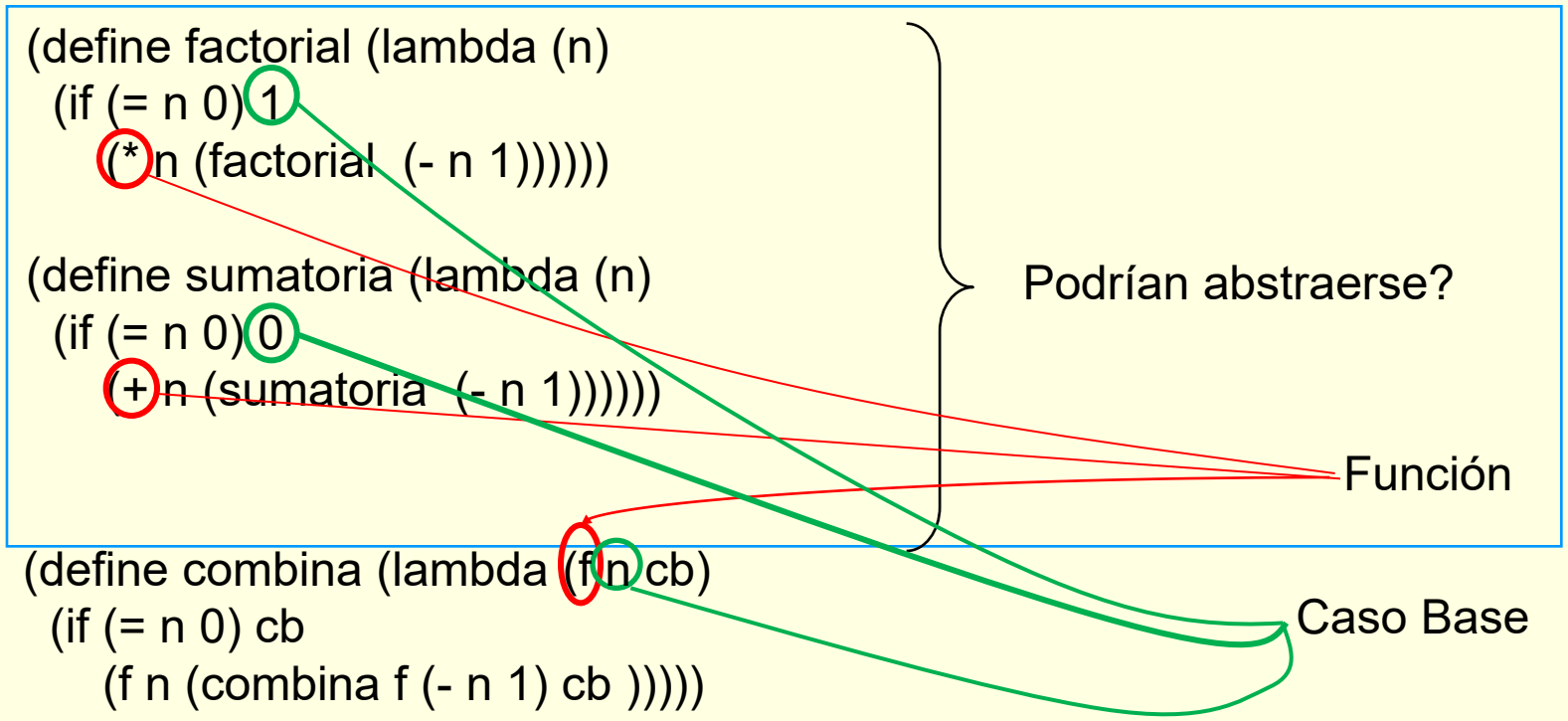
```
((lambda (x)(+ x 1)) ((lambda (x) (- x 1)) 3))
```

suma_1 resta_1

```
(define prefijo (lambda (p l)
        (if (null? p) #t (if (equal? (car p) (car l))
                             (prefijo (cdr p) (cdr l))
                             #f))))
```

FUNCIONES DE ORDEN SUPERIOR

Funciones como argumentos



FUNCIONES DE ORDEN SUPERIOR

Funciones como argumentos

Cómo se reescribirían factorial y sumatoria?

```
(define combina (lambda (f n cb)
  (if (= n 0) cb
      (f n (combina f (- n 1) cb )))))


(define sumatoria2 (lambda (n)
                     (combina + n 0)))


(define fact2 (lambda (n)
                (combina * n 1)))
```

FUNCIONES DE ORDEN SUPERIOR

Funciones que construyen funciones

Aplicación → Resultado

(*función*$_1$ arg$_1$ arg$_2$ …. arg$_n$) ⇒ *función*$_2$

```
(define funcion1
        (lambda (arg1 arg2 …. argn)
                (lambda (argi argj … argk)
                .....)
        )
)
```

función$_2$ ⇒ *Función retornada*

Funciones que construyen funciones

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

Función retornada

- (composicion suma_1 resta_1)

#<procedure>

- ((composicion suma_1 resta_1) 2)

2

Construcción de una Lista de Funciones

```
(define cuadrado (lambda (x)
                   (* x x)))

(define cubo (lambda (x)
                   (* x x x)))

> (define listaf '(cubo cuadrado))
> ((car listaf) 3)
. procedure application: expected procedure, given:
cubo; arguments were: 3

> (define listaf (list cubo cuadrado))
> ((car listaf) 3)
27
```

Función de orden superior *map*

map es una función de orden superior, primitiva de Scheme, que tiene como argumentos una:

- *función de un argumento*
- *una lista*

retornando la lista resultante de los resultados de la aplicación de la función a todos los elementos de la lista argumento.

➢(**map** suma1 ´(1 2 3))
´(2 3 4)

➢(**map** invertir ‘((1 2) (3 4))
‘((2 1) (4 3))

Ejercicios

Definir la función **mapeo** que recibe como argumento una función de 1 argumento y una lista. Mapeo devuelve una lista que resulta de aplicar la función recibida como argumento a cada elemento de la lista

Por ejemplo:

```
> (mapeo suma_1 '(1 2 3) )
(2 3 4)

> (mapeo car '((1 2 3) ('a 'b 'c) ("hola" "chau")))
1 'a "hola")

> (mapeo cdr '((1 2 3) ('a 'b 'c) ("hola" "chau")))
((2 3) ('b 'c) ("chau"))
```

```
(define mapeo (lambda (func lista)
                (if (null? lista) '()
                    (cons (func (car lista))
                          (mapeo func (cdr lista))))))
```

Función de orden superior *filtro*

filtro es una función de orden superior, que tiene como argumentos una:

- ***función de un argumento que devuelve #t o #f***
- ***una lista***

retornando una lista cuyos elementos son los elementos de la lista original que devolvieron #t al aplicarle la función recibida como arguemento

- **(filtro mayor2 '(1 2 3 4))**

(3 4)

- **(filtro list? '((1 2 3) 3 (12 3)))**

((1 2 3) (12 3))

```
(define filtro (lambda (test lista)
                 (if (null? lista) '()
                     (if (test (car lista))
                       (cons (car lista)
                             (filtro test (cdr lista)))
                       (filtro test (cdr lista))))))
```

Ejercicios

Defina la función **(aplicarMapeos LFunciones Lista)** que devuelve una lista que resulta de aplicar cada una de las funciones que son miembros de LFunciones a cada elemento de Lista.

Por ejemplo:

```
> (define suma_1 (lambda (x) (+ x 1)))
> (define resta_1 (lambda (x) (- x 1)))
>  (define por_2 (lambda (x) (* x 2)))

> (aplicarMapeos (list suma_1 por_2) '(1 2 3))
(4 6 8)

> (aplicarMapeos (list cdr car) '((1 2 3) ('a 'b 'c) ("hola"
"chau")))
(2 'b "chau")
```

```
(define aplicarMapeos (lambda (lf lista)
                (if (null?  lf)  lista
                 (aplicarMapeos
                                (cdr lf)
                             (mapeo (car lf) lista)
                        ))))
```

Ejercicios

Utilizando la función anterior defina una función **(aplicarFiltros listaFiltros)** que retorne una función cuyo argumento es una lista. En este caso, listaFiltros es una lista de funciones de un argumento que devuelven verdadero o falso. Al evaluar la función resultante con una lista específica se deben aplicar todos los filtros de la lista de funciones para obtener la lista resultante.

Por ejemplo:

> (**aplicarFiltros** (list mayor2 menor4))
#<procedure>

> ((**aplicarFiltros** (list mayor2 menor4)) '(4 2 3 5 1 6))
(3)

```
(define aplicarFiltros (lambda (lf)
   (lambda (lista)
     (if (null? lf)
        lista
        ((aplicarFiltros (cdr lf)) (filtro (car lf) lista))
))))
```

```
(define aplicarMapeos (lambda (lf lista)
        (if (null?  lf)  lista
                (aplicarMapeos (cdr lf)
                                (mapeo (car lf) lista))
        )))
```

```
(define aplicarFiltros (lambda (lf)
  (lambda (lista)
    (if (null? lf)
       lista
       ((aplicarFiltros (cdr lf)) (filtro (car lf) lista))))
))
```

Ejercicios

RECURSIÓN DE COLA (“Tail Recursion”)

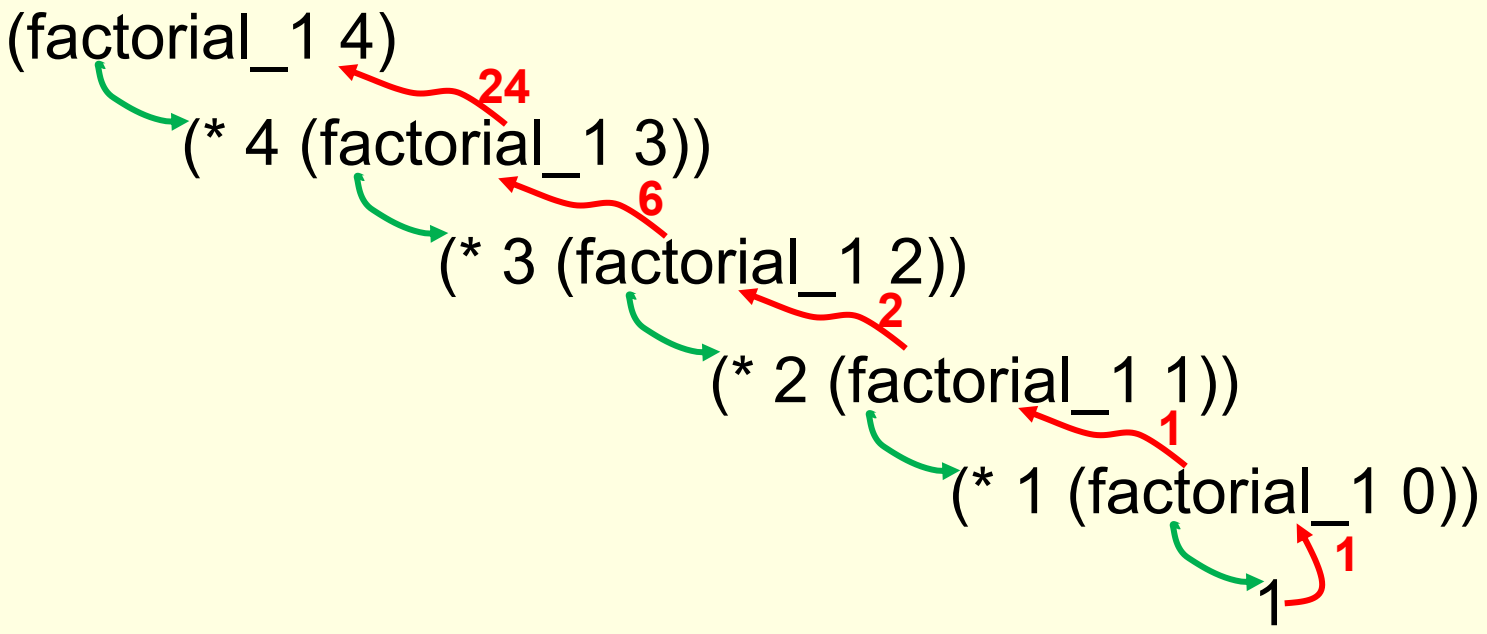
- Recursión de cola es un caso especial de la recursividad en el que la llamada recursiva es la última cosa que se hace en una función.
- Este tipo de recursión puede ser optimizado para reutilizar el medio ambiente que se une a los parámetros de la función recursiva para sus argumentos.
- Este es un caso especial de lo que se conoce como última optimización de llamada.

RECURSIÓN SIMPLE

Factorial

```
(define factorial_1 (lambda (n)
          (if  (= n 0)
               1
               (* n (factorial (- n 1))))))
```

Debe mantener en memoria la pila de invocaciones hasta llegar al caso base.



RECURSIÓN DE COLA (“Tail Recursion”)

Factorial

```
(define factorial_1 (lambda (n)
          (if  (= n 0)
               1
              (* n (factorial (- n 1))))))
```

Debe mantener en memoria la pila de invocaciones hasta llegar al caso base.

Factorial implementado con recursión de cola

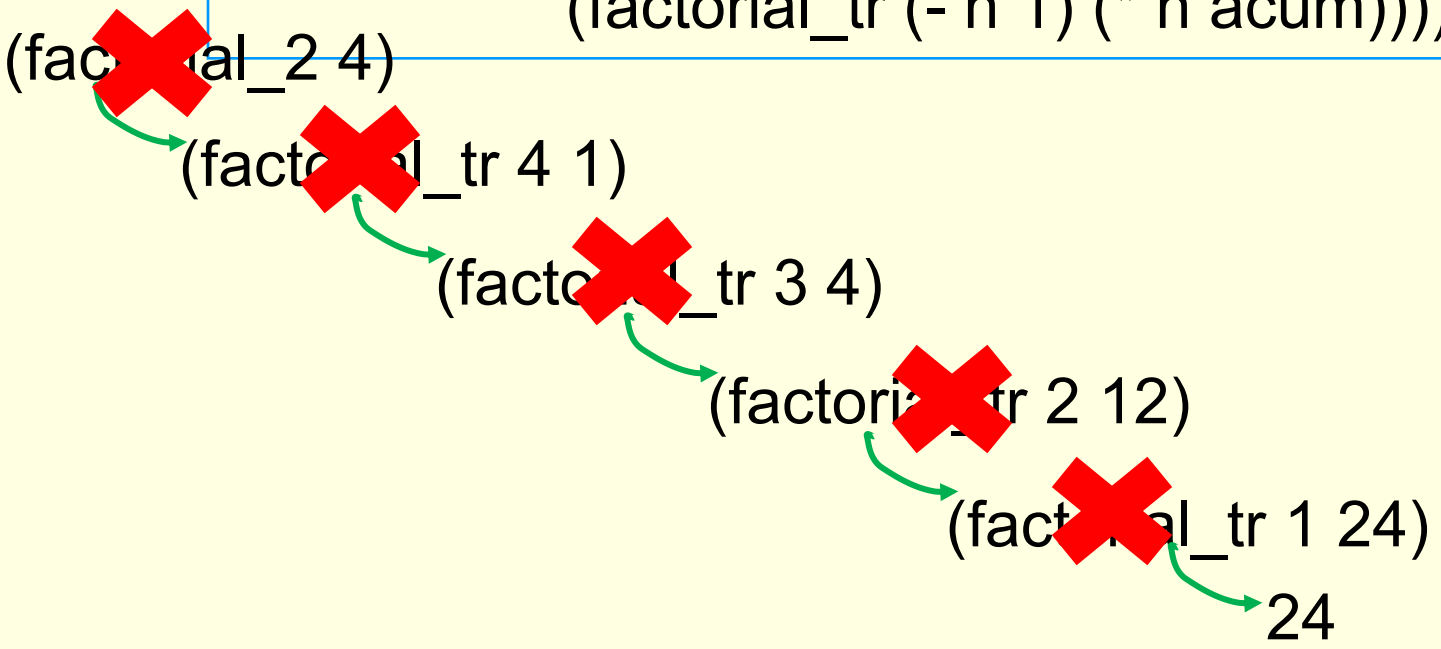
```
(define factorial_2 (lambda (n)
                (factorial_tr n 1)))

(define factorial_tr (lambda (n acum)
          (if  (= n 0)
               acum
              (factorial_tr (- n 1) (* n acum)))))
```

RECURSIÓN DE COLA (“Tail Recursion”)

Factorial implementado con recursión de cola

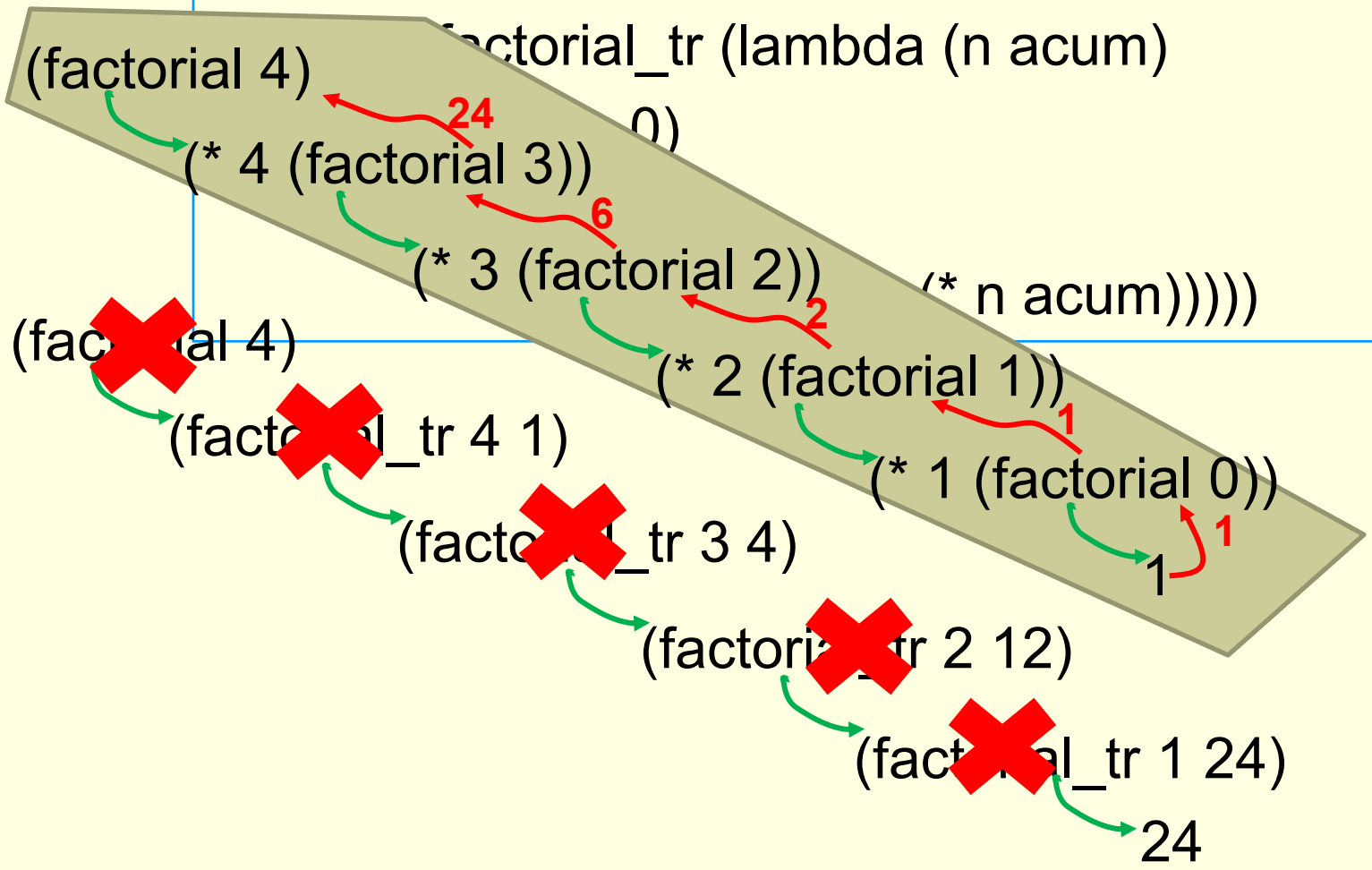
```
(define factorial_2 (lambda (n)
                      (factorial_tr n 1)))
(define factorial_tr (lambda (n acum)
            (if  (= n 0)
                 acum
                 (factorial_tr (- n 1) (* n acum)))))
```



RECURSIÓN DE COLA (“Tail Recursion”)

Factorial implementado con recursión de cola

```
(define factorial_2 (lambda (n)
                      (factorial_tr n 1)))
                    factorial_tr (lambda (n acum)
                                   0))
                                                  (* n acum)))))
```



RECURSIÓN DE COLA (“Tail Recursion”)

Definición de la función **primerosN**, que toma como argumentos una **lista** y un número entero N y retorna una lista con los primeros N elementos de lista

Definición recursiva

```
(define primerosN (lambda(L N)
  (if (= N 1) (list (car L))
      (cons (car L)
            (primerosN (cdr L) (- N 1))
      )
)))
```

RECURSIÓN DE COLA (“Tail Recursion”)

Definición de la función **primerosN**, que toma como argumentos una **lista,** un número entero N y retorna una lista con los primeros N elementos de lista

Definición con recursión de cola

```
(define primerosN2 (lambda (L N)
  (primerosNRC L N `())))

(define primerosNRC (lambda (L N LR)
   (if (= N 0) LR
     (primerosNRC (cdr L) (- N 1)
          (append LR (list (car L)))
     )
)))
```

RECURSIÓN DE COLA ("Tail Recursion")

Función de Fibonacci.

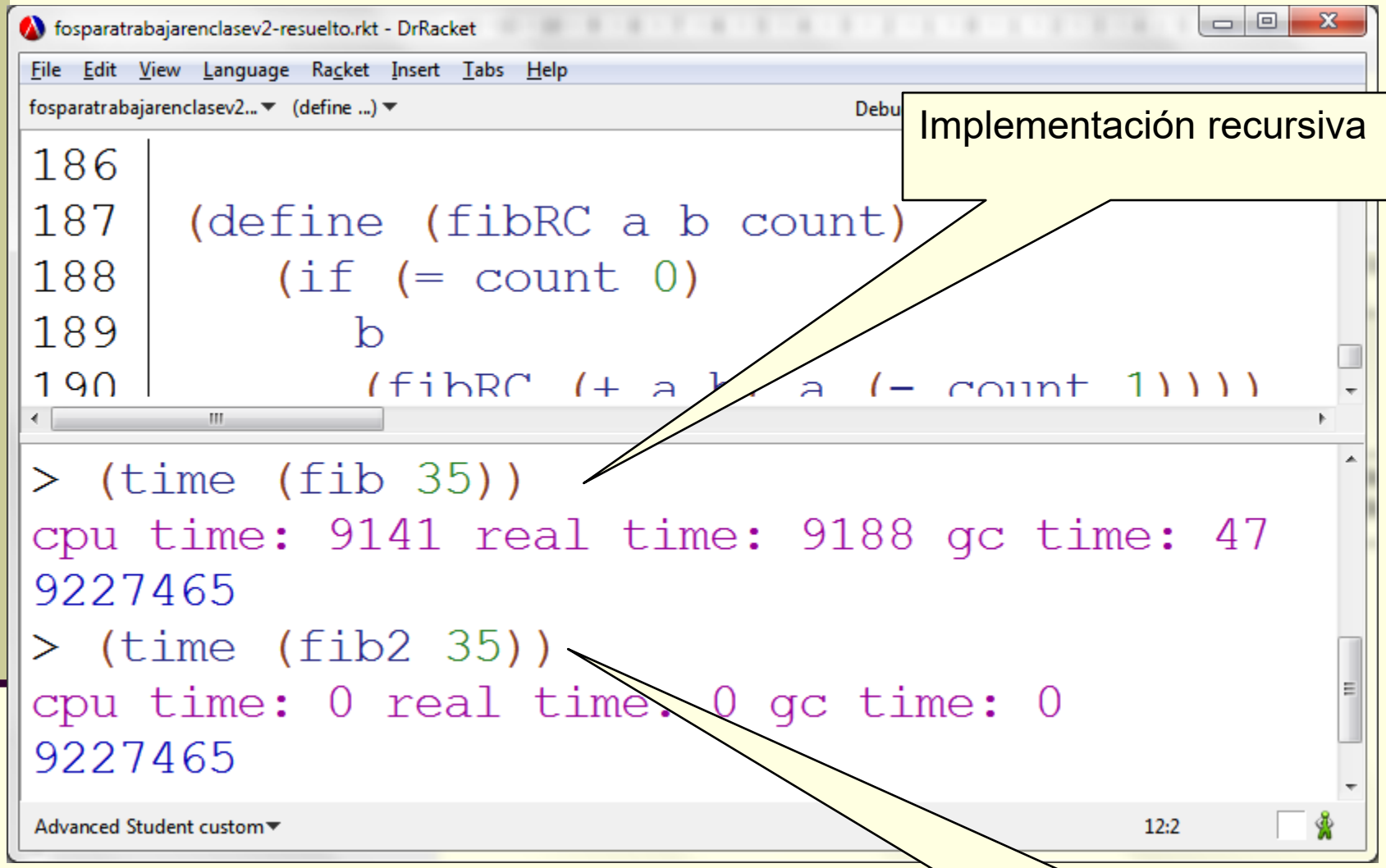
Definición recursiva

```
((define (fib n)
   (if (= n 0) 0
      (if (= n 1) 1
       (+ (fib (- n 1))
                (fib (- n 2))))))
```

Definición con recursión de cola

```
(define (fib2 n)
   (fibRC 1 0 n))

(define (fibRC a b count)
   (if (= count 0)
      b
       (fibRC (+ a b) a (- count 1))))
```


fosparatrabajarenclasev2-resuelto.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
fosparatrabajarenclasev2... (define ...)
186
187 (define (fibRC a b count)
188 (if (= count 0)
189 b
> (time (fib 35))
cpu time: 9141 real time: 9188 gc time: 47
9227465
> (time (fib2 35))
cpu time: 0 real time: 0 gc time: 0
9227465
Advanced Student custom
12:2
Implementación recursiva
Implementación con recursión de cola

Ejercicios

- Escribir utilizando recursion de cola las funciones:
  - (longitud L) que retorna la cantidad de elementos de la lista L
  - (invertir L) que devuelve la una lista con los elementos de la lista L en orden inverso.
- Ejercicios de examenes/parciales

Ejercicios

Defina una función de orden superior que recibe como parámetros una función (que retorna verdadero o falso) y una lista.

La función **(takewhile condicion L)** devuelve una lista con los n primeros elementos de la lista L para los que condicion retorne verdadero. Donde n es la posición del primer elemento para el que la evaluación de condicion retorne falso.

Por ejemplo:

(define mayor2 (lambda (x) (> x 2)))

(**takewhile** mayor2 '(5 6 7 2 8)) → '(5 6 7)

(**takewhile** mayor2 '(5 6 7 9 8)) → '(5 6 7 9 8)

(**takewhile** mayor2 '(1 2 3 3 2)) → '()

```
(define takewhile (lambda (test L)
                  (if (null? L) L
                      (if (test (car L)) (cons (car L)
                                               (takewhile test (cdr L)))
                          '()
))))
```

Ejercicios

De manera similar al punto anterior, defina la función (**dropwhile condition L)** que devuelve la lista L a la que se le eliminaron los primeros elementos que satisfacen la condición (esto es aquellos elementos que al aplicarle la función condition dan verdadero)

Por ejemplo:

(define menor4 (lambda (x) (< x 4)))
(dropwhile menor4 '(2 3 1 5 6))
‘(5 6)
(dropwhile menor4 '(2 3 1 5 6 1))
‘(5 6 1)
(dropwhile menor4 '(5 6 1 2 3))
‘ (5 6 1 2 3)

```
(define dropwhile (lambda (condicion lista)
                    (if (not (condicion (car lista)))  lista
                        (dropwhile condicion (cdr lista)))
                     ))
```

Ejercicios

Dada una función de un argumento *f* y un entero positivo *n*, se le solicita que defina la función de orden superior:

(repetir *f n*)

la cual retorna una función de un argumento que aplica *n* veces la función *f* . Por ejemplo:

```
(define cua (lambda (x) (* x x))
#cua
((repetir cua 1) 2)
4
((repetir cua 3) 2)
256
```

```
(define repetir (lambda (f n)
                  (lambda (x)
                    (if (= n 1)
                        (f x)
                        (f ((repetir f (- n 1)) x))))))
```