



Universidad
del Valle

Fundamentos
de
programación

Carlos Andrés
Delgado S.

Acumulación
del
conocimiento

Fundamentos de programación

Acumulación del conocimiento

carlos.andres.delgado@correounivalle.edu.co

Carlos Andrés Delgado S.

Facultad de Ingeniería. Universidad del Valle

Noviembre de 2016

Fundamentos
de
programación

Carlos Andrés
Delgado S.

Acumulación
del
conocimiento

1 Acumulación del conocimiento

Fundamentos
de
programación

Carlos Andrés
Delgado S.

Acumulación
del
conocimiento

1 Acumulación del conocimiento

Definición

En esta sección vamos a modificar la forma en que manejamos recursiones, hasta ahora podemos diseñar soluciones para:

- Una función de que eleve al cuadrado una lista de números
- Una función que sume los elementos de una lista.

Sin embargo, si nos solicitan invertir una lista ¿Podemos realizarlo?

Definición

Intentemos una solución:

```
;Invertir: lista-numeros -> lista-numeros
;Ejemplos: (invertir (list 1 2 3 4 5)) -> (list 5 4 3 2 1)
(define (invertir lst)
  (cond
    [(empty? lst) empty]
    [else (cons (invertir (rest lst)) (first lst))])
)
```

Handwritten red annotations:

- A wavy line under `rest` with the label `list` below it.
- A red circle around `first` with a line pointing to the label `num p/c` (number per cent).
- An arrow points from `num p/c` to the label `list` at the bottom right.

Definición

Para el ejemplo se obtiene:

```
;Invertir: lista-numeros -> lista-numeros  
;Ejemplos: (invertir (list 1 2 3 4 5)) -> (list 5 4 3 2 1 )  
(cons empty (cons 5 (cons 4 (cons 3 (cons 2 (cons 1 ) ) ) ) ) )
```

¡Tenemos una solución incorrecta!

Definición

Para solucionar este problema, debemos utilizar **acumuladores**

```
;Invertir: lista-numeros -> lista-numeros
;Ejemplos: (invertir (list 1 2 3 4 5)) -> (list 5 4 3 2 1 )
(define (invertir lst)
  (cond
    [(empty? lst) empty]
    [else (convertir-ultimo (first lst) (invertir (rest lst)))]
  )
)
;convertir-ultimo: Agrega un elemento al final de una lista
;convertir-ultimo: numero, lista-numeros -> lista-numeros
(define (convertir-ultimo an lst)
  (cond
    [(empty? lst) (list an)]
    [else (cons (first lst) (convertir-ultimo an (rest lst)))]
  )
)
```

convertir (list 2 3 1 7 6))

↓
(convertir-ultimo 2 (invertir (list 3 7 6)))

(convertir-ultimo 2 (convertir-ultimo 3 (invertir (list 1 7 6))))

(convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1 (invertir (list 7 6)))))

(convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1 (convertir-ultimo 7 (invertir (list 6))))))

convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1 (convertir-ultimo 7 (convertir-ultimo 6 empty)))))

convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1 (convertir-ultimo 7 (list 6)))))

convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1 (convertir-ultimo 7 (list 6)))))

convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1
(convertir-ultimo 6 (convertir-ultimo 7 empty))))))

convertir-ultimo 2 (convertir-ultimo 3 (convertir-ultimo 1
(convertir-ultimo 6 (list 7))))))

Definición

Analicemos su funcionamiento:

```
(invertir (list 1 2 3))  
;;Llamado  
(convertir-ultimo 1 (invertir (list 2 3)))  
(convertir-ultimo 1 (convertir-ultimo 2 (invertir (list 3))))  
(convertir-ultimo 1 (convertir-ultimo 2 (convertir-ultimo 3 (invertir empty)  
)))
```

Definición

Analicemos su funcionamiento:

```
(convertir-ultimo 1 (convertir-ultimo 2 (convertir-ultimo 3 empty)))  
(convertir-ultimo 1 (convertir-ultimo 2 (list 3)))  
(convertir-ultimo 1 (cons 3 (convertir-ultimo 2 empty)))  
(convertir-ultimo 1 (cons 3 (list 2)))  
(convertir-ultimo 1 (list 3 2))  
(cons 3 (convertir-ultimo 1 (list 2)))  
(cons 3 (cons 2 (convertir-ultimo 1 empty)))  
(cons 3 (cons 2 (list 1)))  
(list 3 2 1)
```

Definición

También podemos cambiar el diseño de nuestras funciones para usar acumuladores:

```
;;suma: lista-numeros -> numero
(define (suma lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst) (suma (rest lst)))]
  )
)
```

Definición

Usando abstracción de datos local:

```
;;suma: lista-numeros -> numero
(define (suma lst)
  (local (
    (define (suma-acc lista acc)
      (cond
        [(empty? lista) acc]
        [else (suma-acc (rest lista) (+ acc (first lista)))]
      )
    )
    (suma-acc lst 0)
  )
)
```

Definición

Observemos su funcionamiento

```
;;suma: lista-numeros -> numero  
(suma (list 2 4 6))  
(suma-acc (list 2 4 6) 0)  
(suma-acc (list 4 6) 2)  
(suma-acc (list 4) 6)  
(suma-acc empty 10)  
10
```

Ejercicio

Recordemos la definición de factorial:

```
;;factorial: numeros -> numero  
(define (factorial n)  
  (cond  
    [(zero? n) 1]  
    [else (* n (suma (- n 1)))]  
  )  
)
```

Diseñe una solución utilizando acumuladores

Ejercicio

Ahora una solución usando acumuladores:

```
;;factorialA: numeros -> numero
(define (factorialA n)
  (local
    (
      (define (fact-acc n acc)
        (cond
          [(zero? n) acc]
          [else (fact-acc (- n 1) (* n acc))])
      )
    )
    (fact-acc n 1)
  )
)
```


Ejercicios

Usando acumuladores diseñe:

- 1 Una función que sume los elementos pares de una lista
- 2 Una función que cuente los elementos de una lista

Ejercicio

Una función que sume los elementos pares de una lista

```
;;suma-lista: lista-numeros -> numero
(define (suma-lista lst)
  (local
    (
      (define (suma-acc lista acc)
        (cond
          [(empty? lista) acc]
          [else
           (cond
            [(even? (first lista)) (suma-acc (rest lista) (+ (first lista) acc))]
            [else (suma-acc (rest lista) acc)]
           )
          ]
        )
      )
    )
    (suma-acc lst 0)
  ))
```

Ejercicio

Una función que cuente los elementos de una lista

```
;;contar-lista: lista-numeros -> numero
(define (contar-lista lst)
  (local
    (
      (define (contar-acc lista acc)
        (cond
          [(empty? lista) acc]
          [else (contar-acc (rest lista) (+ 1 acc))])
      )
    )
    (contar-acc lst 0)
  ))
```

Ventajas

El diseño de los acumuladores mejora el funcionamiento de las funciones recursivas, ya que permite llevar un resultado parcial mientras se está calculando y no es necesario acordarse cual es el llamado anterior, observemos:

```
;Sumar una lista (list 1 2 3)
;Función suma recursiva
(suma (list 1 2 3))
(+ 1 (suma (list 2 3)))
(+ 1 (+ 2 (suma (list 3))))
(+ 1 (+ 2 (+ 3 0)))
;Función suma recursiva con acumuladores
(suma (list 1 2 3) 0)
(suma (list 2 3) 1)
(suma (list 3) 3)
(suma empty 6)
6
```

```
;;buscar-mayor-lista: lista de numeros -> numero
```

```
(define (buscar-mayor-lista lnum)  
  (cond  
    [(empty? lnum) (error "Debe al menos existir un numero")]  
    [(= (length lnum) 1) (first lnum)]  
    [else  
     (local  
       (define (buscar-mayor-aux lnumD mayor)  
         (cond  
           [(empty? lnumD) mayor]  
           [(> (first lnumD) mayor)  
            (buscar-mayor-aux lnumD (first lnumD))]  
           [else (buscar-mayor-aux lnumD mayor)])  
       )  
     )  
     (buscar-mayor-aux lnum (first lnum))  
    )  
  )  
)
```



Universidad
del Valle

¿Preguntas?

Fundamentos
de
programación

Carlos Andrés
Delgado S.

Acumulación
del
conocimiento

VAMO A

