EjercicioTransformacionResueltos...

Duvan02

Metodología de la Programación

1° Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

Ejercicio 10.

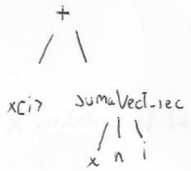
- Tupla de parámetros formales: $\tilde{x} = x, n, i$
- casobase: $i = 0$
- sol($\tilde{x}$) = 0
- suc($\tilde{x}$) = $x, n, i-1$
- comb($\tilde{x}$, y) = $x[i] + y$

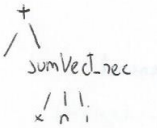
a) Generalización

- Función sumergida

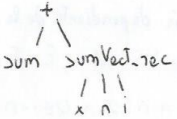
$$sumaVect\_rec(x,n,i) = x[i] + sumaVect\_rec(x,n,i-1)$$

- Árbol sintáctico (función sumergida)





- Árbol sintáctico con los parámetros de inmersión



- Función inmersora: $sumVectTRec(x,n,i,sum) = sum + sumVect\_rec(x,n,i)$
- Valores iniciales: Elemento neutro de la suma = 0
- Llamada inicial: $sumVectTRec(x,n,i,0) = 0 + sumVect\_rec(x,n,i)$
- Función llamada inicial.

```
//Cabecera: entero : sum (E Vect x, E entero: n, E entero: i)
//Precondición: x = A[1..n] ∧ n>0 ∧ 0≤i≤n
//Postcondición: devuelve el valor Σ_{α:1}^{i} x[α]
entero: sum (E vect: x, E entero: n, E entero: i)
  inicio
```

devolver sumVectRec(x, n, i, 0);

Fin_función

b) Desplegado

- Siguiendo la definición de fun_recFinal obtenida en la generalización y siguiendo el mismo análisis de caso que en la función recursiva no final, diseñar la función inmersora:

// Cabecera: entero: sumVectRec (E vect: x, E Entero: x, E entero: i, E entero: sum)

// Precondición: $x = A[1..n] \wedge n > 0 \wedge 0 \leq i \leq n$

// Postcondición: devuelve el valor: $\sum_{\alpha=1}^{i} x[\alpha]$

Entero: sumVectRec (E vect: x, E entero: x, E entero: i, E Entero: sum)

inicio

si i=0 entonces

devolver sum+0

si_no

devolver sum + x[i] + sumaVect_rec (x, n, i-1)

fin_si

fin_función.

- Reorganización del caso general:

(sum + x[i]) + sumaVect_rec(x, n, i-1)

Lo único que hemos hecho ha sido reorganizar los términos.

- Función recursiva final (aún dependiente de la función recursiva no final):

// Cabecera: Entero: sumVectRec (E vect: x, E Entero: n, E Entero: i, E Entero: sum)

// Precondición: $x = A[1..n] \wedge n > 0 \wedge 0 \leq i \leq n$

// Postcondición: devuelve el valor: $\sum_{\alpha=1}^{i} x[\alpha]$

Entero: sumVectRec (E Vect: x, E Entero: x, E Entero: i, E Entero: sum)

inicio

si i=0 entonces

devolver sum+0

si_no

devolver (sum + x[i]) + sumaVect_rec(x, n, i-1)

fin_si

fin_función

c) Plegado

- Sucesores: suc (sum) = sum + x[i]

- Versión Recursiva Final

```
//Cabecera: Entero : sumVectRec (E Vect:x, E Entero:n, E Entero i, E Entero:sum)
// Precondición: x = A[1..n] ∧ n>0 ∧ 0≤i≤n
// PostCondición: devuelve el valor Σ_{α:1}^{i} x[α]
Entero : sumVectRec (E Vect:x, E Entero:n, E Entero:i, E Entero: sum)
  inicio
    si i=0 entonces
      devolver sum+0
    si_no
      devolver sumVectRec (x,n,i-1, sum+x[i])
    fin_si
  fin_función
```

Transformación de función recursiva no final (RNF) a Iterativa

```
//Cabecera: Entero: sumIter (E Vect:x, E Entero: n, E Entero i)
// Precondición: x = A[1..n] ∧ n>0 ∧0≤i≤n
// PostCondición: devuelve el valor Σ_{α:1}^{i} x[α]
  var
    Entero c, res
  inicio
    c←0
    mientras (i=0) hacer
      c←c+1
      i←i-1
    fin_mientras
    res←0
    mientras c≠0 hacer
      c←c-1
      i←i+1
      res←res+x[i]
    fin_mientras
    devolver res
  fin_función
```

optimización del código

//Cabezera: Entero sumIter (E Vect:x, E Entero:x, E Entero:i)

//Precondición: x=A[1..n] ∧ n>0 ∧ 0≤i<n

//Postcondición: devuelve el valor $\sum_{\alpha=1}^{i} x[\alpha]$

```
Entero: sumIter (E Vect:x, E Entero:x, E Entero:i)
    var:
        entero: c, res
    inicio
        c ← n-i
        i ← 0
        res ← 0
        mientras c≠0 hacer
            c ← c-1
            i ← i+1
            res ← res + x[i]
        fin_mientras
        devolver res
fin_funcion
```

//Para optimizar la función lo que hemos hecho es eliminar el primer bucle, además hemos inicializado c al valor n-i y i a 0, para que la función se ejecute de manera correcta.

Transformación de Recursiva Final (RF) a Iterativa.

//Cabezera: Entero: sum (E Vect: x, E Entero: n, E Entero i)

//Precondición: x=A[1..n] ∧ n>0 ∧ 0≤i<n

//Postcondición: devuelve el valor $\sum_{\alpha=1}^{i} x[\alpha]$

```
Entero: sum (E Vect: x, E Entero: n, E Entero:i)
    inicio
        devolver sumIter(x, n, i, 0)
    fin_funcion
```

//Cabezera: Entero: sumIter (E Vect:x, E Entero:n, E entero:i, E Entero sum)

//Precondición: x=A[1..n] ∧ n>0 ∧ 0≤i<n

//Postcondición: devuelve el valor $\sum_{\alpha=1}^{i} x[\alpha]$

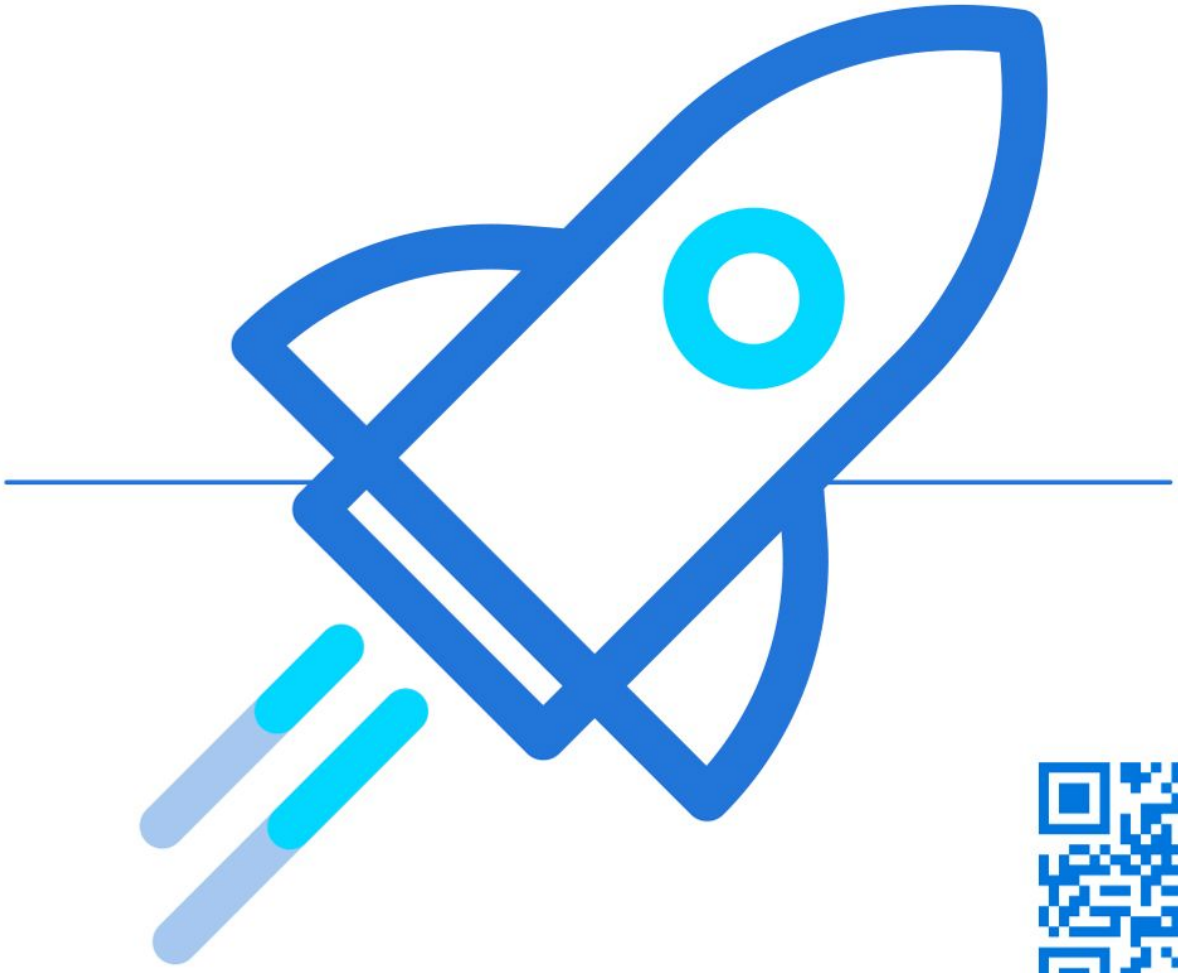
```
Entero: sumIter (E Vect: x, E Entero: n, E Entero: i, E Entero: sum)
 inicio
     mientras (i=0) hacer
       sum = sum + X[i]
       i = i-1
     fin_mientras
     devolver sum+0
fin. función
```

Optimización del código

```
//Cabezera: Entero : sumIter (E Vect: x, E Entero: n, E Entero: i, E Entero: sum)
//Precondición: x = A[1..n] ∧ n>0 ∧ 0≤i<n
//Postcondición: devuelve el valor $\sum_{\alpha=1}^{i} x[\alpha]$
Entero: sumIter (E Vect: x, E Entero: n, E Entero: i, E Entero: sum)
 inicio
    sum ← 0
    i ← n-1
    mientras (i=0) hacer
      sum = sum + X[i]
      i = i-1
    fin_mientras
    devolver sum+0
fin. función
```

//Para optimizar la función lo que hemos hecho ha sido inicializar los parámetros sum a 0 e i a n-1 para que la función funcione correctamente.

deja de imaginar
que trabajarás de
lo tuyo
hazlo posible.



encuentra curro

randstad

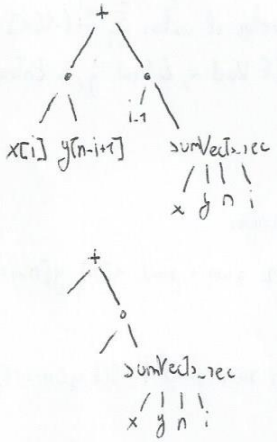
partner for talent.

Ejercicio 11.

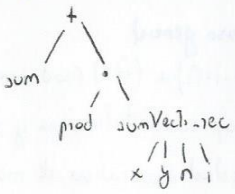
- Tupla de parámetros formales $\bar{x}$ = x, y, n, i
- caso base ($\bar{x}$) = i = n
- sol($\bar{x}$) = x[i] · y[n-i+1]
- suc($\bar{x}$) = x, y, n, i+1
- comb($\bar{x}$, j) = x[i] · y[n-i+1] + (i+1) · j

a) Generalización

- Función sumergida = sumVects_rec(x,y,n,i) = x[i] · y[n-i+1] + (i-1) · sumVects_rec(x,y,n,i+1)
- Árbol sintáctico (función sumergida)



- Árbol sintáctico con los parámetros de inmersión



- Función inmersora: sumVectRec(x,y,n,i,sum,prod) = sum + prod · sumVects_rec(x,y,n,i)
- Valores iniciales: Elemento neutro de la suma = 0
  Elemento neutro del producto = 1
- Llamada inicial: sumVectRec(x,y,n,i,0,1) = 0+1 · sumVects_rec(x,y,n,i)
- Función llamada inicial

// Cabecera: Entero: llamada (E Vect: x, E Vect: y, E Entero: n, E Entero: i)
// Precondición: x = A[1..n] ∧ y = B[1..n] ∧ i ≤ n ∧ i ≥ 1 ∧ n ≥ 0
// Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} \frac{\alpha!}{i!} (x[\alpha] \cdot y[n-\alpha+1])$
Entero: llamada (E Vect: x, E Vect: y, E Entero: n, E Entero: i)
  inicio

devolver sumVectRec(x,y,n,i,0,1)
fin_función

b) Desplegado

- Siguiendo la definición de sumRecFinal obtenida en la generalización y siguiendo el mismo análisis de casos que en la función recursiva no final, diseñar inmersora:

//Cabecera: Entero sumVectRec(E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum, E Entero:prod)
//Precondición: $x = A[1..n] \wedge y = B[1..n] \wedge i \leq n \wedge i \geq 1 \wedge n > 0$
//Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} \frac{\alpha!}{i!}(x[\alpha] \cdot y[n-\alpha+1])$
Entero sumVectRec (E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum, E Entero:prod)
  inicio
    si i=n entonces
      devolver sum + prod.x[i].y[n-i+1]
    si_no
      devolver sum + prod(x[i].y[n-i+1] + (i+1).sumVect_rec(x,y,n,i+1))
    fin_si
  fin_función

- Reorganización del caso general

$(sum + prod.x[i].y[n-i+1]) + (i+1).prod.sumVect\_rec(x,y,n,i+1)$

Se le aplica la propiedad distributiva y se reorganizan los términos, aplicando la propiedad asociativa, de manera similar al caso general. De esta manera se obtienen los sucesores de los parámetros de inmersión.

- Función recursiva final (aún dependiente de la función recursiva no final)

//Cabecera: Entero sumVectRec(E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum, E Entero:prod)
//Precondición: $x = A[1..n] \wedge y = B[1..n] \wedge i \leq n \wedge i \geq 1 \wedge n > 0$
//Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} \frac{\alpha!}{i!}(x[\alpha] \cdot y[n-\alpha+1])$
Entero: sumVectRec (E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum, E Entero:prod)

```
Entero: sumVectRec ( E Vect: x, E Vect: y, E Entero: n, E Entero: i,
                     E Entero: sum, E Entero: prod)
  inicio
    si i=n entonces
      devolver sum + prod.x[i].y[n-i+1]
    si_no
      devolver (sum + prod.x[i].y[n-i+1]) + (i+1).prod sumVectRec(x,y,n,i+1)
    fin_si
  fin_función
```

c) Plegado

- Sucesores

```
suc(sum): sum + prod.x[i].y[n-i+1]
suc(prod): (i+1).prod
```

- Versión Recursiva Final

```
//Cabezera: Entero: sumVectRec (E Vect: x, E Vect: y, E Entero: n, E Entero: i,
            E Entero: sum, E Entero: prod)
// Precondición: x = A[1..n] ∧ y = B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
// PostCondición: devuelve el valor: Σ_{α=1}^{n} α!/i! (x[α].y[n-α+1])
Entero: sumVectRec (E Vect: x, E Vect: y, E Entero: n, E Entero: i, E Entero: sum,
E Entero: prod)
  inicio
    si i=n entonces
      devolver sum + prod.x[i].y[n-i+1]
    si_no
      devolver sumVectRec ( x, y, n, i+1, sum + prod.x[i].y[n-i+1], (i+1).prod )
    fin_si
  fin_función
```

Transformación de Función Recursiva No Final (RNF) a Iterativa

```
//Cabezera: Entero: sumIter (E Vect: x, E Vect: y, E Entero: n, E Entero: i)
// Precondición: x = A[1..n] ∧ y = B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
// PostCondición: devuelve el valor: Σ_{α=1}^{n} α!/i! (x[α].y[n-α+1])
```

```
Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i)
  Var
    Entero c, res
  inicio
    c ← 0
    mientras (i=n) hacer
      c ← c+1
      i ← i+1
    fin_mientras
    res ← x[i].y[n-i+1]
    mientras c ≠ 0 hacer
      c ← c-1
      i ← i-1
      res ← x[i].y[n-i+1] + (i+1).res
    fin_mientras
    devolver res
  fin_función.
```

Optimización del código

```
//Cabecera: Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i)
//Precondición: x = A[1..n] ∧ y = B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
//Postcondición: devuelve el valor Σ_{α=1}^{n} (α!/i!) (x[α].y[n-α+1])
Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i)
  Var
    Entero c, res
  inicio
    c ← n-i
    i ← n
    res ← x[i].y[n-i+1]
    mientras c ≠ 0 hacer
      c ← c-1
      i ← i-1
      res ← x[i].y[n-i+1] + (i+1).res
    fin_mientras
    devolver res
  fin_función
```

// Para optimizar el código lo que hemos hecho ha sido eliminar el primer bucle e inicializar los valores de n e i.

Transformación de Recursiva Final (RF) a Iterativa.

```
//Cabecera: Entero: llamada (E Vect:x, E Vect:y, E Entero:n, E Entero:i)
//Precondición: x=A[1..n] ∧ y=B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
//Postcondición: devuelve el valor Σ_{α:1}^{n} α!/i! (x[α].y[n-α+1])
Entero: llamada (E Vect:x, E Vect:y, E Entero:n, E Entero:i)
  inicio
    devolver sumIter(x,y,n,i,0,1)
  fin_función

//Cabecera: Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero sum,
            E Entero: prod)
//Precondición: x=A[1..n] ∧ y=B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
//Postcondición: devuelve el valor Σ_{α:1}^{n} α!/i! (x[α].y[n-α+1])
Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum, E Entero:prod)
  inicio
    mientras (i=n) hacer
      sum ← sum + prod.x[i].y[n-i+1]
      prod ← (i+1).prod
      i ← i+1
    fin_mientras
    devolver sum + prod.x[i].y[n-i+1]
  fin-función
```

Optimización del código.

```
//Cabecera: Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum,
            E Entero:prod)
//Precondición: x=A[1..n] ∧ y=B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
//Postcondición: devuelve el valor Σ_{α:1}^{n} α!/i! (x[α].y[n-α+1])
Entero: sumIter(E Vect:x, E Vect:y, E Entero:n, E Entero:i, E Entero:sum, E Entero:prod)
  inicio
    sum ← 0
    prod ← 0
    mientras (i=n) hacer
```

```
        suma ← suma + prod.x[i].y[n-i+1]
        prod ← (i+1).prod
        i ← i+1
    fin-mientras
    devuelve suma + prod.x[i].y[n-i+1]
fin-función
```

// Para optimizar la función simplemente hemos inicializado a 0 los parámetros de suma y prod

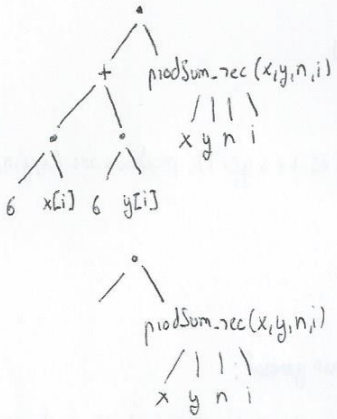
Ejercicio 12.

- Tupla de parámetros formales: x, y, n, i
- Caso base? $(\bar{x})$: $i > n$
- Sol $(\bar{x})$: 1
- Suc $(\bar{x})$: $x, y, n, i+1$
- Comb $(\bar{x}, \beta)$: $6 \cdot x[i] + 6 \cdot y[i] \cdot \beta$

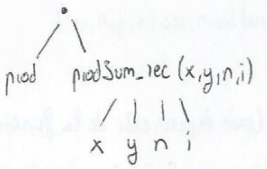
a) Generalización.

- Función sumergida: $prodSum\_rec(x, y, n, i) = (6 \cdot x[i] + 6 \cdot y[i]) \cdot prodSum\_rec(x, y, n, i+1)$

- Árbol sintáctico (función sumergida):



- Árbol sintáctico con los parámetros de inmersión:



- Función inmersora: $prodSum\_rec\_F(x, y, n, i, prod) = prod \cdot prodSum\_rec(x, y, n, i)$

- Valores iniciales: Elemento neutro del producto prod=1

- Llamada inicial: $prodSum\_rec\_F(x, y, n, i, 1) = 1 \cdot prodSum\_rec(x, y, n, i)$

// Cabezera: entero prod (E Vect: x, E vect: y, E entero: n, E entero: i)

// Precondición: $X = A[1...n] \wedge y = B[1...n] \wedge 1 \leq i \leq n \wedge n > 0$

// Poscondición: devuelve el valor de $\hat{\prod_{\alpha:i}} 6 \cdot (x[\alpha] + y[\alpha])$

```
Entero: función prod (E vect:x, E vect:y, E entero:n, E entero:i)
inicio
   devolver prodSum_rec_F(x,y,n,i,1)
fin_función
```

b) Desplegado

- Función inmersora:

```
// Cabecera: entero: prodSum_rec_F(E vect:x, E vect:y, E entero:n, E entero:i, E entero: prod)
// Precondición: x=A[1..n] ∧ y=B[1..n] ∧ 1≤i≤n ∧ n>0
// Postcondición: devuelve el valor ∏_{k=i}^{n} 6(x[i]+y[i])
Entero: función prodSum_rec_F(E vect:x, E vect:y, E entero:n, E entero:i, E entero: prod)
inicio
   si i>n entonces
      devolver 1.prod
   si_no
      devolver prod.(6xCi] + 6yCi]).prodSum_rec(x,y,n,i+1)
   fin_si
fin_función.
```

- Reorganización del caso general:

Usando las propiedades de la multiplicación se puede reorganizar los términos para obtener los nuevos sucesores de los parámetros de inmersión.

6(prod.(x[i]+y[i])).prodSum_rec(x,y,n,i+1)

- Función recursiva final (aún dependiente de la función recursiva no final)

```
// Cabecera: entero: prodSum_rec_F(E vect:x, E vect:y, E entero:n, E entero:i, E entero: prod)
// Precondición: x=A[1..n] ∧ y=B[1..n] ∧ 1≤i≤n ∧ n>0
// Postcondición: devuelve el valor ∏_{k=i}^{n} 6(x[i]+y[i])
entero: función prodSum_rec_F(E vect:x, E vect:y, E entero:n, E entero:i, E entero: prod)
 inicio
    Si i>n entonces
       devolver 1.prod
    si_no
       devolver 6.prod(x[i]+y[i]).prodSum_rec(x,y,n,i+1)
    fin_si
fin_función.
```

c) Plegado

- Obtener los sucesores de los parámetros de inmersión que deben ir en la llamada Recursiva Final

suc(prod) = 6 prod.(x[i] + y[i])

- Versión recursiva Final:

// Cabecera: entero prodSum_recF(E vect x, E vect y, E entero:n, E entero:i, E entero:prod)

// Precondición: x=A[1..n] ∧ y=B[1..n] ∧ 1≤i≤n ∧ n>0

// Postcondición: devuelve el valor $\prod_{\alpha:i}^{n} 6(x[i]+y[i])$

Entero: función prodSum_recF(E vect x, E vect y, E entero:n, E entero:i, E entero:prod)

inicio

si i>n entonces

devolver 1.prod

si_no

devolver prodSum_rec(x, y, n, i+1, 6 prod.(x[i] + y[i]))

fin_si

fin_función

- Transformación de Función Recursiva No Final a Iterativa

// Cabecera: entero prodSum_rec(E vect:x, E vect:y, E entero:n, E entero:i)

// Precondición: x=A[1..n] ∧ y=B[1..n] ∧ 1≤i≤n ∧ n>0

// Postcondición: devuelve el valor $\prod_{\alpha:i}^{n} 6(x[i]+y[i])$

Entero: función prodSum_rec(E vect:x, E vect:y, E entero:n, E entero:i)

Var

Entero c, res

inicio

mientras ¬(i>n) hacer

c ← c+1

i ← i+1

fin_mientras

res ← 1

mientras c ≠ 0 hacer

c ← c-1

i ← i-1

res ← 6(x[i]+y[i]).res

fin_mientras

devolver res

fin_función.

- Optimización del código

```
// Cabecera: entero prodSum_rec (E vect:x, E vect:y, E entero:n, E entero:i)
// Precondición: x=A[1...n] ∧ y=B[1...n] ∧ 1≤i≤n ∧ n>0
// Postcondición: devuelve el valor: ∏ 6(x[i]+y[i])  (a: i..n)
Entero: función prodSum_rec (E vect:x, E vect:y, E entero:n, E entero:i)
Var
    entero: c, res
inicio
    c ← n-i
    i ← n
    res ← 1
    mientras c≠0 hacer
        c ← c-1
        res ← 6.(x[i]+y[i]).res
        i ← i-1
    fin_mientras
    devolver res
fin_función
```

- Transformación de Recursiva Final a Iterativa.

```
// Cabecera: entero prod (E vect:x, E vect:y, E vect:n, E entero:i)
// Precondicion: X=A[1...n] ∧ y=B[1...n] ∧ 1≤i≤n ∧n>0
// Postcondición: devuelve el valor de ∏ 6(x[i]+y[i])  (a: i..n)
Entero: funcion prodSum_inicio (E vect:x, E vect:y, E entero:n, E entero:i)
inicio
    devolver
fin_función

// Cabecera: entero prodSum_recII(E vect:x, E vect:y, E entero:n, E entero:i, E entero:prod)
// Precondición: x=A[1...n] ∧ y=B[1...n] ∧ 1≤i≤n ∧n>0
// Postcondición: devuelve el valor ∏ 6(x[i]+y[i])  (a: i..n)
entero: función prodSum_rec (E vect:x, E vect:y, E entero:n, E entero:i, E entero:prod)
inicio
    mientras(i>n) hacer
        prod ← 6.prod(x[i]+y[i])
        i ← i+1
```

fin_mientras
devolver prod
fin_función

- Optimización del código

// Cabecera: entero prodSum_rec(E vect: x, E vect: y, E entero: n, E entero: i, E entero: prod)
// Precondición: x=A[1..n] ∧ y=B[1..n] ∧ 1≤i≤n ∧ n>0
// Postcondición: devuelve el valor $\prod_{x:i} 6(x[i]+y[i])$
entero: función prodSum_rec (E vect: x, E vect: y, E entero n, E entero i, E entero prod)
Var
  entero: prod
inicio
  prod ← 1
  mientras (i ≤ n) hacer
    prod ← 6 prod (x[i] + y[i])
    i ← i+1
  fin_mientras
  devolver prod
fin_mientras

Ejercicio 13.

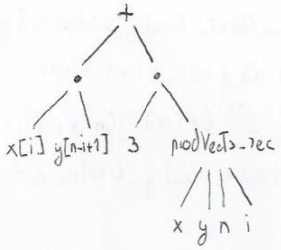
Esquema.

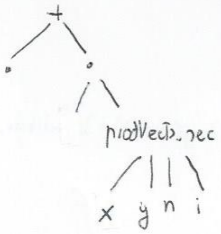
- Tupla de parámetros formales: $\bar{x} = x, y, n, i$
- Caso base? $(\bar{x}) = i = n+1$
- $Sol(\bar{x}) = 0$
- $Suc(\bar{x}) = x, y, n, i+1$
- $Comb(\bar{x}, j) = x[i] \cdot y[n-i+1] + 3.prodVects\_rec$

a) Generalización

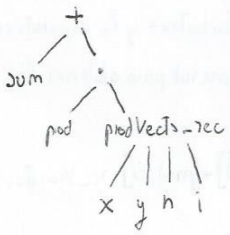
- Función sumergida = $x[i] \cdot y[n-i+1] + 3 prodVects\_rec$

- Árbol sintáctico =





- Árbol sintáctico con los parámetros de inversión =



- Función inmersora $prodVects\_rec(x, y, n, i, sum, prod) = sum + prod \cdot prodVects\_rec(x, y, n, i)$

- Valores iniciales = Elemento neutro de la suma = 0
  Elemento neutro del producto = 1

- Llamada inicial: prodVects_rec_Final(x,y,n,i,0,1) = 0+1 prodVects_rec(x,y,n,i)

- Función que realiza la llamada inicial a la función recursiva final (RF):

//Cabezera: entero: prod(E Vect: x, E Vect: y, E entero: n, E entero i)

//Precondición: $x=A[1...n] \wedge y=B[1...n] \wedge i \leq n \wedge i \geq 1 \wedge n>0$

//Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i}(x[\alpha].y[n-\alpha+1])$

entero: función prod(E Vect: x, E Vect: y, E entero: n, E entero i)

inicio

devolver prodVects_rec_Final(x,y,n,i,0,1)

fin_función

b) Desplegado

- Diseñar función inmersora:

//Cabezera: entero: prodVects_rec_Final(E Vect x, E Vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)

//Precondición: $x=A[1...n] \wedge y=B[1...n] \wedge i \leq n \wedge i \geq 1 \wedge n>0$

//Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i}(x[\alpha].y[n-\alpha+1])$

entero: función prodVects_rec_Final(E Vect: x, E Vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)

inicio

si i=n+1 entonces

devolver sum + prod. 0

si no

devolver sum + prod. (x[i].y[n-i-1] + 3. prodVects_rec(x,y,n,i+1))

fin_si

fin_función

- Reorganización del caso general:

Se le aplica la propiedad distributiva y la asociativa para luego reorganizarlo de manera similar al caso general para obtener los sucesores de los parámetros de inmersión.

[sum + prod.(x[i].y[n-i+1])] + [prod.3]. prodVects_rec(x,y,n,i+1)

- Función recursiva final (aún dependiente de la función recursiva no final)

//Cabezera: entero: prodVects_rec_Final(E Vect: x, E Vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)

//Precondición: $x=A[1...n] \wedge y=B[1...n] \wedge i \leq n \wedge i \geq 1 \wedge n>0$

//Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i}(x[\alpha].y[n-\alpha+1])$

entero: función prodVects_rec_Final (E vect x, E vect: y, E entero: n, E entero: i, E entero: sum, E entero prod)
inicio
  si i=n+1 entonces
    devolver sum + prod. 0
  si_no
    devolver [sum + prod. (x[i]. y[n-i+1])] + [prod. 5]. prodVects_rec(x, y, n, i+1)
  fin_si
fin_función.

c) Plegado

- Sucesores: · suc(sum) = sum + prod. x[i]. y[i-n+1]
  · suc(prod) = prod. 3

- Versión Recursiva Final:

// Cabecera: entero: prodVects_rec_Final (E Vect: x, E Vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod
// Precondición: x = A[1...n] ∧ y = B[1...n] ∧ i ≤ n ∧ i ≥ 1 ∧ n > 0
// Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i} (x[\alpha].y[n-\alpha+1])$
entero: función prodVects_rec_Final (E Vect: x, E Vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)
inicio
  si i=n+1 entonces
    devolver sum+prod. 0
  si_no
    devolver prodVects_rec_Final (x, y, n, i+1, sum+prod. x[i]. y[n-i+1], prod. 3)
  fin_si
fin. función.

- Transformación de una función Recursiva no Final a Iterativa.

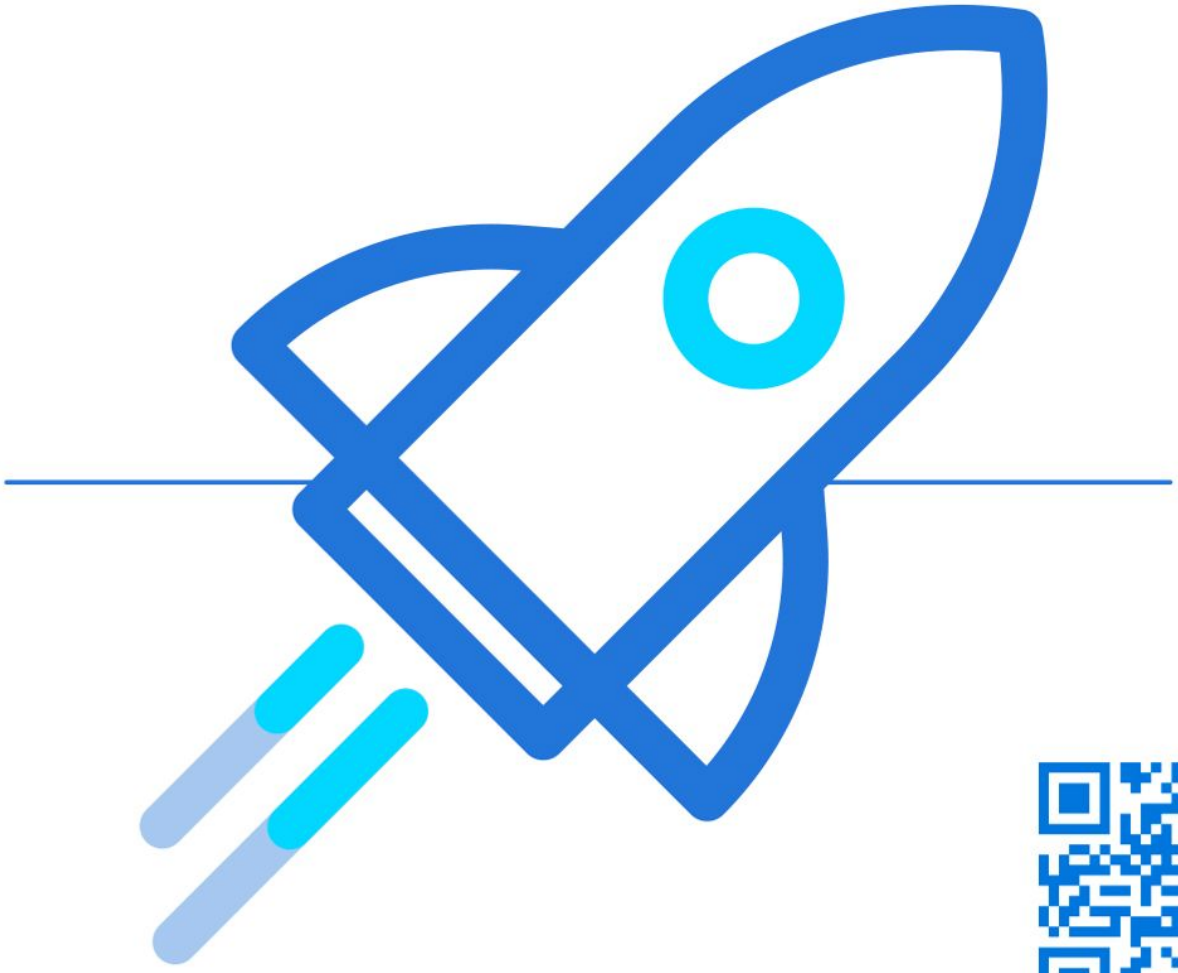
// Cabecera: entero: prodVects_rec_Iter (E Vect: x, E Vect: y, E entero: n, E entero: i)
// Precondición: x = A[1...n] ∧ y = B[1...n] ∧ i ≤ n ∧ i ≥ 1 ∧ n > 0
// Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i} (x[\alpha].y[n-\alpha+1])$
entero: prodVects_rec_Iter (E Vect: x, E Vect: y, E entero: n, E entero: i)
  Var
    entero c, res
  inicio
  c ← 0

```
mientras ¬(i=n+1) hacer
    c ← c+1
    i ← i+1
fin_mientras
res ← 0
mientras c≠0 hacer
    c ← c-1
    i ← i-1
    res ← X[i] . y[n-i+1] + 3.res
fin_mientras
devolver res
fin_función
```

- Optimización del código:

```
// Cabecera: entero prodVects_rec_Iter (E Vect x, E Vect y, E entero n, E entero i)
// Precondición: x=A[1..n] ∧ y=B[1..n] ∧ i≤n ∧ i≥1 ∧ n>0
// Postcondición: devuelve el valor de Σ_{α=i}^{n} 3^{α-i} (X[α].y[n-α+1])
entero función prodVects_rec_Iter (E Vect x, E Vect y, E entero n, E entero i)
Var
    entero c, res
inicio
    c ← n-i+1
    i ← n+1
    res ← 0
    mientras c≠0 hacer
        c ← c-1
        i ← i-1
        res ← X[i] . y[n-i+1] + 3res
    fin_mientras
    devolver res
fin_función
```

deja de imaginar
que trabajarás de
lo tuyo
hazlo posible.



encuentra curro

randstad

partner for talent.

- Transformación de Recursiva Final a Iterativa:

// Cabecera: entero: prod (E vect: x, E vect: y, E entero: n, E entero: i)
// Precondición: x = A[1...n] ∧ y = B[1...n] ∧ i ≤ n ∧ i ≥ 1 ∧ n > 0
// Postcondición: el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i} (x[\alpha] \cdot y[n-\alpha+1])$
entero: funcion prod (E vect: c, E vect: y, E entero: n, E entero: i)
inicio
  devolver prodVects_rec_I (x, y, n, i, 0, 1)
fin_funcion

// Cabecera: entero: prodVects_rec_I (E vect: x, E vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)
// Precondición: x = A[1...n] ∧ y = B[1...n] ∧ i ≤ n ∧ i ≥ 1 ∧ n > 0
// Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i} (x[\alpha] \cdot y[n-\alpha+1])$
Entero: funcion prodVects_rec_I (E vect: x, E vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)
Inicio
  mientras (i = n+1) hacer
    sum ← sum + prod . x[i] . y[n-i+1]
    prod ← prod . 3
    i ← i+1
  fin_mientras
  devolver sum + prod . 0
fin_funcion

- Optimización del código:

// Cabecera: entero: prodVects_rec_I (E vect: x, E vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)
// Precondición: x = A[1...n] ∧ y = B[1...n] ∧ i ≤ n ∧ i ≥ 1 ∧ n > 0
// Postcondición: devuelve el valor $\sum_{\alpha=i}^{n} 3^{\alpha-i} (x[\alpha] \cdot y[n-\alpha+1])$
entero: funcion prodVects_rec (E vect: x, E vect: y, E entero: n, E entero: i, E entero: sum, E entero: prod)
Var
  entero: sum, prod
inicio
  sum ← 0
  prod ← 1
  mientras (i = n+1) hacer
    sum ← sum + prod . x[i] . y[n-i+1]
    prod ← prod . 3
    i ← i+1
  fin_mientras
  devolver sum + prod . 0
fin_funcion