Paradigmas Fundamentales de Programación

Computaciones iterativas y recursivas: Ejemplos

Juan Francisco Díaz Frias

Maestría en Ingeniería, Énfasis en Ingeniería de Sistemas y Computación
Escuela de Ingeniería de Sistemas y Computación,
home page: http://eisc.univalle.edu.co
Universidad del Valle - Cali, Colombia

Modelos y Paradigmas de Programación

Plan

1 Programando con listas
- Tamaño, concatenación, inversión, ordenamiento

2 Programando con árboles
- Árbol binario ordenado

Plan

1 Programando con listas
- Tamaño, concatenación, inversión, ordenamiento

2 Programando con árboles
- Árbol binario ordenado

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

1. Programando con listas
   - Tamaño, concatenación, inversión, ordenamiento
2. Programando con árboles
   - Árbol binario ordenado

Programando con listas (1)

Siguiendo su estructura recursiva

La función consta de:

- Un caso básico, para listas de tamaño menor (digamos, de cero, uno o dos elementos). Para este caso la función calcula la respuesta directamente.
- Un caso recursivo, para listas más grandes. En este caso, la función calcula el resultado en términos de de sus resultados sobre una o más listas de menor tamaño.

Calculando el tamaño de una lista

```
fun {Tam Ls}
   case Ls
   of nil then 0
   [] _|Lr then 1+{Tam Lr}
   end
end
{Browse {Tam [a b c]}}
```

¿Es una computación iterativa?

Programando con listas (1)

Siguiendo su estructura recursiva

La función consta de:

- Un caso básico, para listas de tamaño menor (digamos, de cero, uno o dos elementos). Para este caso la función calcula la respuesta directamente.
- Un caso recursivo, para listas más grandes. En este caso, la función calcula el resultado en términos de de sus resultados sobre una o más listas de menor tamaño.

Calculando el tamaño de una lista

```
fun {Tam Ls}
   case Ls
   of nil then 0
   [] _|Lr then 1+{Tam Lr}
   end
end
{Browse {Tam [a b c]}}
```

¿Es una computación iterativa?

Programando con listas (1)

Siguiendo su estructura recursiva

La función consta de:

- Un caso básico, para listas de tamaño menor (digamos, de cero, uno o dos elementos). Para este caso la función calcula la respuesta directamente.
- Un caso recursivo, para listas más grandes. En este caso, la función calcula el resultado en términos de de sus resultados sobre una o más listas de menor tamaño.

Calculando el tamaño de una lista

```
fun {Tam Ls}
   case Ls
   of nil then 0
   [] _|Lr then 1+{Tam Lr}
   end
end
{Browse {Tam [a b c]}}
```

¿Es una computación iterativa?

Programando con listas (1)

Siguiendo su estructura recursiva

La función consta de:

- Un caso básico, para listas de tamaño menor (digamos, de cero, uno o dos elementos). Para este caso la función calcula la respuesta directamente.
- Un caso recursivo, para listas más grandes. En este caso, la función calcula el resultado en términos de de sus resultados sobre una o más listas de menor tamaño.

Calculando el tamaño de una lista

```
fun {Tam Ls}
   case Ls
   of nil then 0
   [] _|Lr then 1+{Tam Lr}
   end
end
{Browse {Tam [a b c]}}
```

¿Es una computación iterativa?

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con listas (2)

La concatenación

- Si `L=[`$l_1 \dots l_n$`]` y `M=[`$m_1 \dots m_p$`]` entonces `{Concat L M}` devuelve `[`$l_1 \dots l_n\ m_1 \dots m_p$`]`
- Dos propiedades:
  $\text{conc}(\texttt{nil}, m) = m$
  $\text{conc}(x \mid l, m) = x \mid \text{conc}(l, m)$

El programa resultante es

```
fun {Concat Ls Ms}
   case Ls
   of nil then Ms
   [] X|Lr then
      X|{Concat Lr Ms}
   end
end
```

¿Es una computación iterativa?

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con listas (2)

La concatenación

- Si `L=[`$l_1 \ldots l_n$`]` y `M=[`$m_1 \ldots m_p$`]` entonces `{Concat L M}` devuelve `[`$l_1 \ldots l_n\, m_1 \ldots m_p$`]`
- Dos propiedades:
  $\mathrm{conc}(\texttt{nil},\ m) = m$
  $\mathrm{conc}(x \mid l,\ m) = x \mid \mathrm{conc}(l, m)$

El programa resultante es

```
fun {Concat Ls Ms}
   case Ls
   of nil then Ms
   [] X|Lr then
      X|{Concat Lr Ms}
   end
end
```

¿Es una computación iterativa?

Modelos y Paradigmas de Programación

Programando con listas (2)

La concatenación

- Si `L=[`$l_1 \dots l_n$`]` y `M=[`$m_1 \dots m_p$`]` entonces `{Concat L M}` devuelve `[`$l_1 \dots l_n \; m_1 \dots m_p$`]`
- Dos propiedades:
  $\text{conc}(\texttt{nil}, m) = m$
  $\text{conc}(x \,|\, l, m) = x \,|\, \text{conc}(l, m)$

El programa resultante es

```
fun {Concat Ls Ms}
   case Ls
   of nil then Ms
   [] X|Lr then
      X|{Concat Lr Ms}
   end
end
```

¿Es una computación iterativa?

Programando con listas (3)

La función `Enesimo`

Dada una lista, calcular el elemento que está en la posición `N`:

```
fun {Enesimo Xs N}
   if N==1 then Xs.1
   elseif N>1 then
       {Enesimo Xs.2 N-1}
   end
end
```

- Funciona si `N` $>$ 0 y `N` $\leq$ `{Tam Xs}`
- Sino, se lanza una excepción.
- ¿Es una computación iterativa?

Invertir una lista

Primero una definición recursiva:

Programando con listas (3)

La función `Enesimo`

Dada una lista, calcular el elemento que está en la posición `N`:

```
fun {Enesimo Xs N}
   if N==1 then Xs.1
   elseif N>1 then
       {Enesimo Xs.2 N-1}
   end
end
```

- Funciona si `N` $>$ 0 y `N` $\leq$ `{Tam Xs}`
- Sino, se lanza una excepción.
- ¿Es una computación iterativa?

Invertir una lista

Primero una definición recursiva:

Programando con listas (3)

La función `Enesimo`

Dada una lista, calcular el elemento que está en la posición `N`:

```
fun {Enesimo Xs N}
   if N==1 then Xs.1
   elseif N>1 then
       {Enesimo Xs.2 N-1}
   end
end
```

- Funciona si `N` $>$ 0 y `N` $\leq$ `{Tam Xs}`
- Sino, se lanza una excepción.
- ¿Es una computación iterativa?

Invertir una lista

Primero una definición recursiva:

- La inversión de `nil` da `nil`.
- La inversión de `X|Xs` da `Z`, donde `Ys` es la inversión de `Xs`, y `Z` es la concatenación de `Ys` y `[X]`.

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con listas (3)

La función `Enesimo`

Dada una lista, calcular el elemento que está en la posición `N`:

```
fun {Enesimo Xs N}
   if N==1 then Xs.1
   elseif N>1 then
       {Enesimo Xs.2 N-1}
   end
end
```

- Funciona si `N` $>$ 0 y `N` $\leq$ `{Tam Xs}`
- Sino, se lanza una excepción.
- ¿Es una computación iterativa?

Invertir una lista

Primero una definición recursiva:

- La inversión de `nil` da `nil`.
- La inversión de `X|Xs` da `Z`, donde `Ys` es la inversión de `Xs`, y `Z` es la concatenación de `Ys` y `[X]`.

Modelos y Paradigmas de Programación

Programando con listas (3)

La función `Enesimo`

Dada una lista, calcular el elemento que está en la posición `N`:

```
fun {Enesimo Xs N}
   if N==1 then Xs.1
   elseif N>1 then
      {Enesimo Xs.2 N-1}
   end
end
```

- Funciona si `N` $>$ 0 y `N` $\leq$ `{Tam Xs}`
- Sino, se lanza una excepción.
- ¿Es una computación iterativa?

Invertir una lista

Primero una definición recursiva:

- La inversión de `nil` da `nil`.
- La inversión de `X|Xs` da `Z`, donde `Ys` es la inversión de `Xs`, y `Z` es la concatenación de `Ys` y `[X]`.

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con listas (4)

El programa resultante es

```
fun {Invertir Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      {Concat {Invertir Xr}
              [X]}
   end
end
```

Análisis

- Complejidad: n^2.
  ¡Uno esperaría menos!
- El tamaño de la pila crece con el tamaño de la lista de entrada, i.e., se define una computación recursiva que no es iterativa.

Modelos y Paradigmas de Programación

Programando con listas (4)

El programa resultante es

```
fun {Invertir Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      {Concat {Invertir Xr}
              [X]}
   end
end
```

Análisis

- Complejidad: n^2.
  ¡Uno esperaría menos!
- El tamaño de la pila crece con el tamaño de la lista de entrada, i.e., se define una computación recursiva que no es iterativa.
- ¿Se puede hacer mejor?

Programando con listas (4)

El programa resultante es

```
fun {Invertir Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      {Concat {Invertir Xr}
              [X]}
   end
end
```

Análisis

- Complejidad: n^2.
  ¡Uno esperaría menos!
- El tamaño de la pila crece con el tamaño de la lista de entrada, i.e., se define una computación recursiva que no es iterativa.
- ¿Se puede hacer mejor?

Modelos y Paradigmas de Programación

Programando con listas (4)

El programa resultante es

```
fun {Invertir Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      {Concat {Invertir Xr}
              [X]}
   end
end
```

Análisis

- Complejidad: n^2.
  ¡Uno esperaría menos!
- El tamaño de la pila crece con el tamaño de la lista de entrada, i.e., se define una computación recursiva que no es iterativa.
- ¿Se puede hacer mejor?

Programando con listas (4)

El programa resultante es

```
fun {Invertir Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      {Concat {Invertir Xr}
              [X]}
   end
end
```

Análisis

- Complejidad: n^2.
  ¡Uno esperaría menos!
- El tamaño de la pila crece con el tamaño de la lista de entrada, i.e., se define una computación recursiva que no es iterativa.
- ¿Se puede hacer mejor?

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- Entrada: Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `R`=|`Xs`|
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`I`, `Ys`) tal que |`Xs`| = I + |`Ys`| (Invariante)
- Estado Inicial: $I = 0,\ Ys = Xs$
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(I, Ys = \_|Yr) \rightarrow (I+1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- **Entrada:** Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- **Salida:** `R`=|`Xs`|
- **Idea:**

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- **Estado:** Tupla de la forma $(\texttt{I}, \texttt{Ys})$ tal que |`Xs`| = I + |`Ys`| (Invariante)
- **Estado Inicial:** $I = 0,\ Ys = Xs$
- **Estado Final:** `Ys=nil`
- **Transformación de estados:**

$$(I, Ys = \_|Yr) \to (I + 1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Modelos y Paradigmas de Programación

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- Entrada: Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `R`=|`Xs`|
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`I`, `Ys`) tal que |`Xs`| = I + |`Ys`| (Invariante)
- Estado Inicial: $I = 0,\ Ys = Xs$
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(I, Ys = \_|Yr) \rightarrow (I + 1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- Entrada:Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `R`=|`Xs`|
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma $(\texttt{I}, \texttt{Ys})$ tal que |`Xs`| = I + |`Ys`| (Invariante)
- Estado Inicial: $I = 0,\ Ys = Xs$
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(I, Ys = \_|Yr) \rightarrow (I+1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- Entrada: Una lista `Xs`=[e_1 e_2 $\cdots$ e_n]
- Salida: `R`=|`Xs`|
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`I`, `Ys`) tal que |`Xs`| = I + |`Ys`| (Invariante)
- Estado Inicial: $I = 0,\ Ys = Xs$
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(I, Ys = \_|Yr) \rightarrow (I + 1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- Entrada: Una lista `Xs`=[e_1 e_2 $\cdots$ e_n]
- Salida: `R`=|`Xs`|
- Idea:

$$\overbrace{e_1 \; e_2 \; \cdots \; e_i \; \underbrace{e_{i+1} \; \cdots \; e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`I`, `Ys`) tal que |`Xs`| = I + |`Ys`| (Invariante)
- Estado Inicial: $I = 0,\; Ys = Xs$
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(I, Ys = \_|Yr) \rightarrow (I + 1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Modelos y Paradigmas de Programación

Programando con listas (5)

Mejorando la eficiencia de `Tam`

- Entrada: Una lista `Xs`=[e_1 e_2 $\cdots$ e_n]
- Salida: `R`=|`Xs`|
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma $(\texttt{I}, \texttt{Ys})$ tal que |`Xs`| = I + |`Ys`| (Invariante)
- Estado Inicial: $I = 0,\ Ys = Xs$
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(I, Ys = \_|Yr) \rightarrow (I+1, Yr)$$

El programa resultante es:

```
fun {TamIt Xs}
   {Iterar
      t(0 Xs)
      fun {$ t(I Ys)}
         Ys==nil
      end
      fun {$ t(I Ys)}
         Yr in
            Ys=_|Yr
            t(I+1 Yr)
      end}
end
```

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- Entrada: Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `Zs`=[$e_n\ e_{n-1}\ \cdots\ e_1$]
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`Zs`, `Ys`) tal que `{Inv Xs}` = `Zs*Ys` (Invariante)
- Estado Inicial: `Zs=nil`, `Ys=Xs`
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(\texttt{Zs, Ys=e|Yr}) \rightarrow (\texttt{e|Zs, Yr})$$

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
            Ys=E|Yr
            t(E|Zs Yr)
      end}
end
```

Modelos y Paradigmas de Programación

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- Entrada: Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `Zs`=[$e_n\ e_{n-1}\ \cdots\ e_1$]
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma $(\texttt{Zs}, \texttt{Ys})$ tal que `{Inv Xs}`=`Zs`*`Ys` (Invariante)
- Estado Inicial: `Zs=nil`, `Ys=Xs`
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(\texttt{Zs}, \texttt{Ys=e|Yr}) \rightarrow (\texttt{e|Zs}, \texttt{Yr})$$

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
            Ys=E|Yr
            t(E|Zs Yr)
      end}
end
```

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- Entrada: Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `Zs`=[$e_n\ e_{n-1}\ \cdots\ e_1$]
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`Zs`, `Ys`) tal que `{Inv Xs}` = `Zs` * `Ys` (Invariante)
- Estado Inicial: `Zs=nil`, `Ys=Xs`
- Estado Final: `Ys=nil`
- Transformación de estados:

(`Zs`, `Ys=e|Yr`) $\rightarrow$ (`e|Zs`, `Yr`)

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
            Ys=E|Yr
            t(E|Zs Yr)
      end}
end
```

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- Entrada:Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `Zs`=[$e_n\ e_{n-1}\ \cdots\ e_1$]
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`Zs`, `Ys`) tal que `{Inv Xs}` = `Zs` * `Ys` (Invariante)
- Estado Inicial: `Zs=nil`, `Ys`= `Xs`
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(\texttt{Zs}, \texttt{Ys=e|Yr}) \rightarrow (\texttt{e|Zs}, \texttt{Yr})$$

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
           Ys=E|Yr
           t(E|Zs Yr)
      end}
end
```

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- **Entrada:** Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- **Salida:** `Zs`=[$e_n\ e_{n-1}\ \cdots\ e_1$]
- **Idea:**

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- **Estado:** Tupla de la forma (`Zs`, `Ys`) tal que `{Inv Xs}` = `Zs` * `Ys` (Invariante)
- **Estado Inicial:** `Zs=nil`, `Ys`= `Xs`
- **Estado Final:** `Ys=nil`
- **Transformación de estados:**

$$(\texttt{Zs}, \texttt{Ys=e|Yr}) \rightarrow (\texttt{e|Zs}, \texttt{Yr})$$

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
            Ys=E|Yr
            t(E|Zs Yr)
      end}
end
```

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- Entrada:Una lista `Xs`=[$e_1\ e_2\ \cdots\ e_n$]
- Salida: `Zs`=[$e_n\ e_{n-1}\ \cdots\ e_1$]
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`Zs`, `Ys`) tal que `{Inv Xs}` = `Zs` * `Ys` (Invariante)
- Estado Inicial: `Zs=nil`, `Ys`= `Xs`
- Estado Final: `Ys=nil`
- Transformación de estados:

$$(\texttt{Zs}, \texttt{Ys=e|Yr}) \rightarrow (\texttt{e|Zs}, \texttt{Yr})$$

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
            Ys=E|Yr
            t(E|Zs Yr)
      end}
end
```

Modelos y Paradigmas de Programación

Programando con listas (6)

Mejorando la eficiencia de `Invertir`

- Entrada: Una lista `Xs`=[e_1 e_2 $\cdots$ e_n]
- Salida: `Zs`=[e_n e_{n-1} $\cdots$ e_1]
- Idea:

$$\overbrace{e_1\ e_2\ \cdots\ e_i\ \underbrace{e_{i+1}\ \cdots\ e_n}_{\texttt{Ys}}}^{\texttt{Xs}}$$

- Estado: Tupla de la forma (`Zs`, `Ys`) tal que `{Inv Xs}` = `Zs`*`Ys` (Invariante)
- Estado Inicial: `Zs=nil`, `Ys=Xs`
- Estado Final: `Ys=nil`
- Transformación de estados:

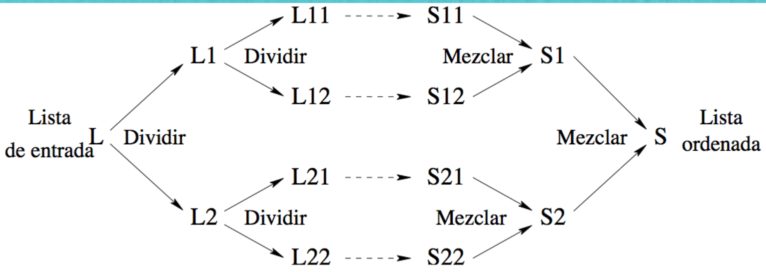
$$(\texttt{Zs},\texttt{Ys=e|Yr}) \rightarrow (\texttt{e|Zs},\texttt{Yr})$$

El programa resultante es:

```
fun {InvIt Xs}
   {Iterar
      t(nil Xs)
      fun {$ t(Zs Ys)}
         Ys==nil
      end
      fun {$ t(Zs Ys)}
         Yr E in
            Ys=E|Yr
            t(E|Zs Yr)
      end}
end
```

Modelos y Paradigmas de Programación

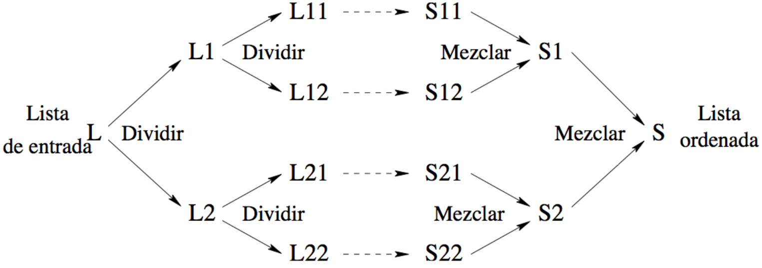
Programando con listas (7)



Dividir y Conquistar: `MergeSort`

- Dividir la lista en dos listas de menor tamaño, aproximadamente de la misma longitud.
- Utilizar el *mergesort* recursivamente para ordenar las dos listas de menor tamaño.
- Mezclar las dos listas ordenadas para obtener el resultado final.

Programando con listas (7)



Dividir y Conquistar: `MergeSort`

- Dividir la lista en dos listas de menor tamaño, aproximadamente de la misma longitud.
- Utilizar el *mergesort* recursivamente para ordenar las dos listas de menor tamaño.
- Mezclar las dos listas ordenadas para obtener el resultado final.

Modelos y Paradigmas de Programación

Programando con listas (8)

Dividir

```
proc {Dividir Xs ?Ys ?Zs}
   case Xs
   of nil then Ys=nil
               Zs=nil
   [] [X] then Ys=[X]
               Zs=nil
   [] X1|X2|Xr then
        Yr Zr in
         Ys=X1|Yr
         Zs=X2|Zr
         {Dividir Xr Yr Zr}
   end
end
```

Mezclar

```
fun {Mezclar Xs Ys}
   case Xs # Ys
   of nil # Ys then Ys
   [] Xs # nil then Xs
   [] (X|Xr) # (Y|Yr) then
      if X<Y then
       X|{Mezclar Xr
                  Ys}
      else
       Y|{Mezclar Xs
                  Yr}
      end
   end
end
```

Programando con listas (8)

Dividir

```
proc {Dividir Xs ?Ys ?Zs}
   case Xs
   of nil then Ys=nil
               Zs=nil
   [] [X] then Ys=[X]
               Zs=nil
   [] X1|X2|Xr then
        Yr Zr in
         Ys=X1|Yr
         Zs=X2|Zr
         {Dividir Xr Yr Zr}
   end
end
```

Mezclar

```
fun {Mezclar Xs Ys}
   case Xs # Ys
   of nil # Ys then Ys
   [] Xs # nil then Xs
   [] (X|Xr) # (Y|Yr) then
      if X<Y then
       X|{Mezclar Xr
                  Ys}
      else
       Y|{Mezclar Xs
                  Yr}
      end
   end
end
```

Modelos y Paradigmas de Programación

Plan

1. Programando con listas
   - Tamaño, concatenación, inversión, ordenamiento
2. Programando con árboles
   - Árbol binario ordenado

Programando con árboles (1)

Árboles

- Después de las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador.
- Un árbol es un nodo hoja o un nodo que contiene uno o más árboles.
- Una posible definición:

$$
\begin{array}{lcl}
\langle \mathrm{A} \rangle & ::= & \texttt{hoja} \\
 & | & \texttt{árbol}\,(\langle \mathrm{Valor} \rangle\ \langle \mathrm{A} \rangle_1 \cdots \langle \mathrm{A} \rangle_n)
\end{array}
$$

- Una lista tiene una estructura lineal; un árbol puede tener una estructura ramificada.

Muchas variedades de árboles

- Una lista es un árbol unario.
- En un árbol binario, los nodos que no son hojas tienen siempre exáctamente dos subárboles.

Programando con árboles (1)

Árboles

- Después de las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador.
- Un árbol es un nodo hoja o un nodo que contiene uno o más árboles.
- Una posible definición:

$$\begin{array}{lll} \langle A\rangle & ::= & \texttt{hoja} \\ & \mid & \texttt{árbol}\,(\langle \text{Valor}\rangle\ \langle A\rangle_1 \cdots \langle A\rangle_n) \end{array}$$

- Una lista tiene una estructura lineal; un árbol puede tener una estructura ramificada.

Muchas variedades de árboles

- Una lista es un árbol unario.
- En un árbol binario, los nodos que no son hojas tienen siempre exáctamente dos subárboles.
- En un árbol *n*-ario los nodos que no son hojas tienen siempre exáctamente *n* subárboles.
- En un árbol balanceado, todos los subárboles del mismo nodo tienen el mismo tamaño (i.e., el mismo número de nodos) o aproximadamente el mismo tamaño.
- Cada variedad de árbol tiene su propia clase de algoritmos para construirlos, recorrerlos y buscar información en ellos.

Programando con árboles (1)

Árboles

- Después de las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador.
- Un árbol es un nodo hoja o un nodo que contiene uno o más árboles.
- Una posible definición:
  $\langle \mathrm{A} \rangle ::=$ `hoja`
  $\mid$ `árbol` `(` $\langle \mathrm{Valor} \rangle\ \langle \mathrm{A} \rangle_1 \cdots \langle \mathrm{A} \rangle_n$ `)`
- Una lista tiene una estructura lineal; un árbol puede tener una estructura ramificada.

Muchas variedades de árboles

- Una lista es un árbol unario.
- En un árbol binario, los nodos que no son hojas tienen siempre exáctamente dos subárboles.
- En un árbol *n*-ario los nodos que no son hojas tienen siempre exáctamente *n* subárboles.
- En un árbol balanceado, todos los subárboles del mismo nodo tienen el mismo tamaño (i.e., el mismo número de nodos) o aproximadamente el mismo tamaño.
- Cada variedad de árbol tiene su propia clase de algoritmos para construirlos, recorrerlos y buscar información en ellos.

Programando con árboles (1)

Árboles

- Después de las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador.
- Un árbol es un nodo hoja o un nodo que contiene uno o más árboles.
- Una posible definición:

$$\begin{array}{lll} \langle \mathrm{A} \rangle & ::= & \texttt{hoja} \\ & \mid & \texttt{árbol}\,(\langle \mathrm{Valor} \rangle\ \langle \mathrm{A} \rangle_1 \cdots \langle \mathrm{A} \rangle_n) \end{array}$$

- Una lista tiene una estructura lineal; un árbol puede tener una estructura ramificada.

Muchas variedades de árboles

- Una lista es un árbol unario.
- En un árbol binario, los nodos que no son hojas tienen siempre exáctamente dos subárboles.
- En un árbol *n*-ario los nodos que no son hojas tienen siempre exáctamente *n* subárboles.
- En un árbol balanceado, todos los subárboles del mismo nodo tienen el mismo tamaño (i.e., el mismo número de nodos) o aproximadamente el mismo tamaño.
- Cada variedad de árbol tiene su propia clase de algoritmos para construirlos, recorrerlos y buscar información en ellos.

Modelos y Paradigmas de Programación

Programando con árboles (1)

Árboles

- Después de las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador.
- Un árbol es un nodo hoja o un nodo que contiene uno o más árboles.
- Una posible definición:
  $$\begin{array}{lll} \langle \mathrm{A} \rangle & ::= & \texttt{hoja} \\ & \mid & \texttt{árbol}\,(\langle \mathrm{Valor} \rangle\ \langle \mathrm{A} \rangle_1 \cdots \langle \mathrm{A} \rangle_n\,) \end{array}$$
- Una lista tiene una estructura lineal; un árbol puede tener una estructura ramificada.

Muchas variedades de árboles

- Una lista es un árbol unario.
- En un árbol binario, los nodos que no son hojas tienen siempre exáctamente dos subárboles.
- En un árbol *n*-ario los nodos que no son hojas tienen siempre exáctamente *n* subárboles.
- En un árbol balanceado, todos los subárboles del mismo nodo tienen el mismo tamaño (i.e., el mismo número de nodos) o aproximadamente el mismo tamaño.
- Cada variedad de árbol tiene su propia clase de algoritmos para construirlos, recorrerlos y buscar información en ellos.

Modelos y Paradigmas de Programación

Programando con árboles (1)

Árboles

- Después de las listas y las colas, los árboles son la estructura de datos recursiva más importante en el repertorio de un programador.
- Un árbol es un nodo hoja o un nodo que contiene uno o más árboles.
- Una posible definición:

$$\begin{array}{lll} \langle \text{A} \rangle & ::= & \texttt{hoja} \\ & | & \texttt{árbol}\,\texttt{(}\langle \text{Valor} \rangle\ \langle \text{A} \rangle_1 \cdots \langle \text{A} \rangle_n\texttt{)} \end{array}$$

- Una lista tiene una estructura lineal; un árbol puede tener una estructura ramificada.

Muchas variedades de árboles

- Una lista es un árbol unario.
- En un árbol binario, los nodos que no son hojas tienen siempre exáctamente dos subárboles.
- En un árbol *n*-ario los nodos que no son hojas tienen siempre exáctamente *n* subárboles.
- En un árbol balanceado, todos los subárboles del mismo nodo tienen el mismo tamaño (i.e., el mismo número de nodos) o aproximadamente el mismo tamaño.
- Cada variedad de árbol tiene su propia clase de algoritmos para construirlos, recorrerlos y buscar información en ellos.

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con árboles (2)

Árbol binario ordenado

- Es un árbol binario en el cual cada nodo que no es hoja incluye un par de valores:

  $\langle$ABO$\rangle$ ::= `hoja`
  | `árbol(`$\langle$VO$\rangle$ $\langle$V$\rangle$ $\langle$ABO$\rangle_1$ $\langle$ABO$\rangle_2$`)`

- El primer valor (la llave) $\langle$VO$\rangle$ es cualquier subtipo de $\langle$Valor$\rangle$ que sea totalmente ordenado.
- El segundo valor (la información) $\langle$V$\rangle$ es cualquier información.
- Un árbol binario es ordenado si para cada nodo que no es hoja, todas las llaves en el primer subárbol son menores que la llave del nodo, y todas las llaves en el segundo subárbol son mayores que la llave del nodo.
- Tres ops. básicas: uscar, insertar y borrar información.

Buscar

```
fun {Buscar X T}
   case T
   of hoja then
        noencontrado
   [] árbol(Y V T1 T2)
      andthen X==Y then
        encontrado(V)
   [] árbol(Y V T1 T2)
      andthen X<Y then
        {Buscar X T1}
   [] árbol(Y V T1 T2)
      andthen X>Y then
        {Buscar X T2}
   end
end
```

Programando con árboles (2)

Árbol binario ordenado

- Es un árbol binario en el cual cada nodo que no es hoja incluye un par de valores:

  $\langle \text{ABO} \rangle$ ::= `hoja`
  | `árbol(`$\langle \text{VO} \rangle$ $\langle \text{V} \rangle$ $\langle \text{ABO} \rangle_1$ $\langle \text{ABO} \rangle_2$`)`

- El primer valor (la llave) $\langle \text{VO} \rangle$ es cualquier subtipo de $\langle \text{Valor} \rangle$ que sea totalmente ordenado.
- El segundo valor (la información) $\langle \text{V} \rangle$ es cualquier información.
- Un árbol binario es ordenado si para cada nodo que no es hoja, todas las llaves en el primer subárbol son menores que la llave del nodo, y todas las llaves en el segundo subárbol son mayores que la llave del nodo.
- Tres ops. básicas: uscar, insertar y borrar información.

Buscar

```
fun {Buscar X T}
   case T
   of hoja then
        noencontrado
   [] árbol(Y V T1 T2)
      andthen X==Y then
        encontrado(V)
   [] árbol(Y V T1 T2)
      andthen X<Y then
        {Buscar X T1}
   [] árbol(Y V T1 T2)
      andthen X>Y then
        {Buscar X T2}
   end
end
```

Programando con árboles (3)

Insertar

```
fun {Insertar X V T}
   case T
   of hoja then
         árbol(X V hoja hoja)
   [] árbol(Y W T1 T2)
      andthen X==Y then
         árbol(X V T1 T2)
   [] árbol(Y W T1 T2)
      andthen X<Y then
         árbol(Y W {Insertar X V T1} T2)
   [] árbol(Y W T1 T2)
      andthen X>Y then
         árbol(Y W T1 {Insertar X V T2})
   end
end
```

Modelos y Paradigmas de Programación

Programando con árboles (4)

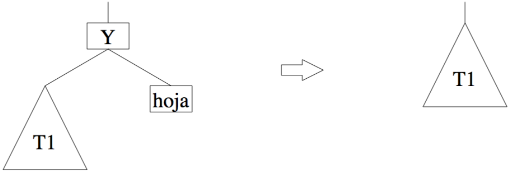
`Borrar` primera versión

```
fun {Borrar X T}
   case T
   of hoja then
        hoja
   [] árbol(Y W T1 T2)
      andthen X==Y then
        hoja
   [] árbol(Y W T1 T2)
      andthen X<Y then
        árbol(Y W {Borrar X T1} T2)
   [] árbol(Y W T1 T2)
      andthen X>Y then
        árbol(Y W T1 {Borrar X T2})
   end
end
```

¿Es correcto?

Programando con árboles (5)

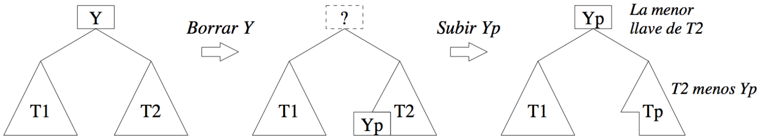
Borrar: caso fácil



Modelos y Paradigmas de Programación

Programando con árboles (6)

Borrar: caso difícil



Programando con árboles (7)

BorrarMenor

```
fun {BorrarMenor T}
   case T
   of hoja then
      nada
   [] árbol(Y V T1 T2) then
      case {BorrarMenor T1}
      of nada then
         Y#V#T2
      [] Yp#Vp#Tp then
         Yp#Vp#árbol(Y V Tp T2)
      end
   end
end
```

Modelos y Paradigmas de Programación

Universidad del Valle

Programando con árboles (8)

Borrar

```
fun {Borrar X T}
   case T
   of hoja then hoja
   [] árbol(Y W T1 T2) andthen X==Y then
      case {BorrarMenor T2}
      of nada then T1
      [] Yp#Vp#Tp then
         árbol(Yp Vp T1 Tp)
      end
   [] árbol(Y W T1 T2) andthen X<Y then
      árbol(Y W {Borrar X T1} T2)
   [] árbol(Y W T1 T2) andthen X>Y then
      árbol(Y W T1 {Borrar X T2})
   end
end
```