Paradigmas Fundamentales de Programación

Programación de alto orden

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

Universidad del Valle

Plan

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Operaciones básicas

- Abstracción procedimental: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
- Genericidad: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.
- Instanciación: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Operaciones básicas

- Abstracción procedimental: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
- Genericidad: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.
- Instanciación: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.
- Embebimiento: la capacidad de colocar valores de tipo procedimiento dentro de estructuras de datos.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Operaciones básicas

- Abstracción procedimental: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
- Genericidad: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.
- Instanciación: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.
- Embebimiento: la capacidad de colocar valores de tipo procedimiento dentro de estructuras de datos.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Operaciones básicas

- Abstracción procedimental: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
- Genericidad: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.
- Instanciación: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.
- Embebimiento: la capacidad de colocar valores de tipo procedimiento dentro de estructuras de datos.

Programación de alto orden

Definición

- Técnicas de programación disponibles cuando se usan valores de tipo procedimiento.
- Un procedimiento en el que ningún argumento es de tipo procedimiento se dice de primer orden.
- Un lenguaje que sólo permite esta clase de procedimientos se llama un lenguaje de primer orden.
- Un procedimiento que tiene al menos un procedimiento de primer orden como argumento se llama de segundo orden.
- Un procedimiento es de orden $n + 1$ si tiene al menos un argumento de orden n y ninguno de orden más alto.
- La programación de alto orden significa, entonces, que los procedimientos pueden ser de cualquier orden.

Operaciones básicas

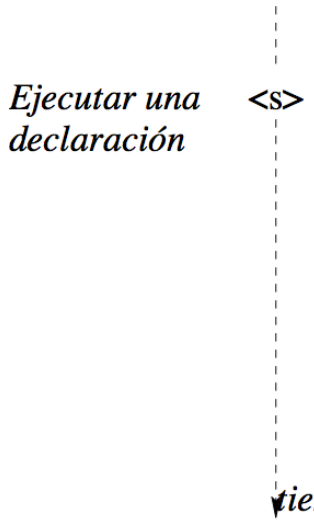
- Abstracción procedimental: la capacidad de convertir cualquier declaración en un valor de tipo procedimiento.
- Genericidad: la capacidad de pasar un valor de tipo procedimiento como argumento en una invocación a un procedimiento.
- Instanciación: la capacidad de devolver valores de tipo procedimiento como resultado de una invocación a un procedimiento.
- Embebimiento: la capacidad de colocar valores de tipo procedimiento dentro de estructuras de datos.

Modelos y Paradigmas de Programación

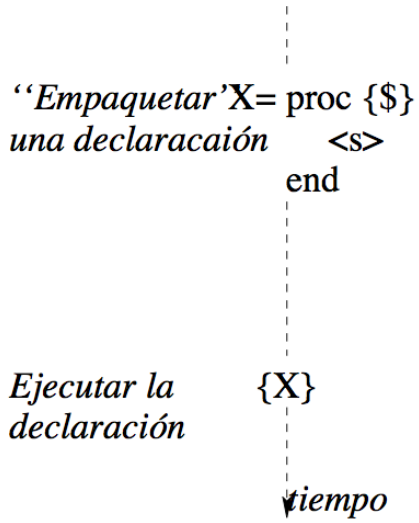
Universidad del Valle

Plan

Abstracción procedimental (1)



Ejecución normal



Ejecución retardada

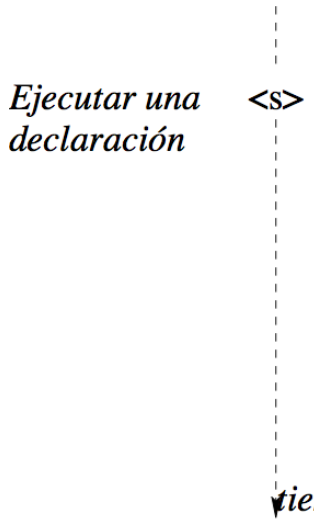
Ejecución retardada

- Como el valor de tipo procedimiento contiene un ambiente contextual, ejecutarlo produce exáctamente el mismo resultado que ejecutar ⟨d⟩.

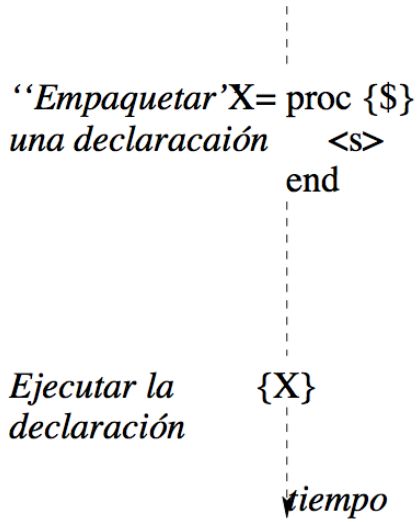
Importancia de la clausura

- La abstracción procedimental subyace a la programación de alto orden y a la programación orientada a objetos, y es la principal herramienta para construir abstracciones.

Abstracción procedimental (1)



Ejecución normal



Ejecución retardada

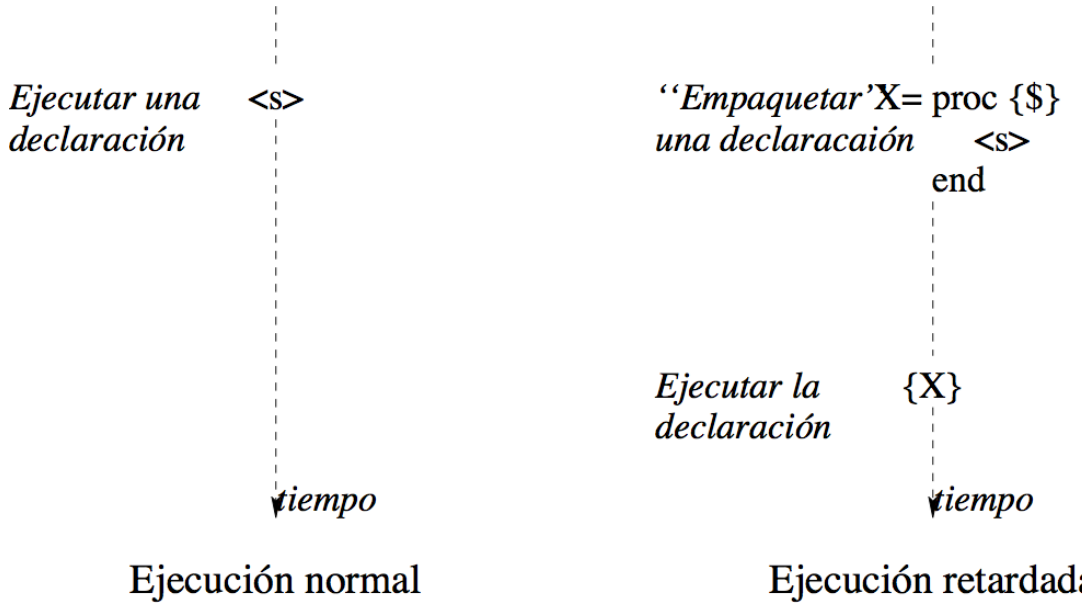
Ejecución retardada

- Como el valor de tipo procedimiento contiene un ambiente contextual, ejecutarlo produce exáctamente el mismo resultado que ejecutar ⟨d⟩.

Importancia de la clausura

- La abstracción procedimental subyace a la programación de alto orden y a la programación orientada a objetos, y es la principal herramienta para construir abstracciones.

Abstracción procedimental (1)



Ejecución retardada

- Como el valor de tipo procedimiento contiene un ambiente contextual, ejecutarlo produce exáctamente el mismo resultado que ejecutar ⟨d⟩.

Importancia de la clausura

- La abstracción procedimental subyace a la programación de alto orden y a la programación orientada a objetos, y es la principal herramienta para construir abstracciones.

Abstracción procedimental (2)

Calculando raíces: sin abstracción

```
local A=1.0 B=3.0 C=2.0
      D SolReal X1 X2 in
  D=B*B-4.0*A*C
  if D>=0.0 then
    SolReal=true
    X1=(~B+{Sqrt D})/(2.0*A)
    X2=(~B-{Sqrt D})/(2.0*A)
  else
    SolReal=false
    X1=~B/(2.0*A)
    X2={Sqrt ~D}/(2.0*A)
  end
  {Browse SolReal#X1#X2}
end
```

Calculando raíces: con abstracción

```
proc {EcCuad A B C
      ?SolReal ?X1 ?X2}
   D=B*B-4.0*A*C
in
if D>=0.0 then
  SolReal=true
  X1=(~B+{Sqrt D})/(2.0*A)
  X2=(~B-{Sqrt D})/(2.0*A)
else
  SolReal=false
  X1=~B/(2.0*A)
  X2={Sqrt ~D}/(2.0*A)
end
end
declare RS X1 X2 in
{EcCuad 1.0 3.0 2.0 RS X1 X2}
{Browse RS#X1#X2}
```

Abstracción procedimental (2)

Calculando raíces: sin abstracción

```
local A=1.0 B=3.0 C=2.0
      D SolReal X1 X2 in
  D=B*B-4.0*A*C
  if D>=0.0 then
    SolReal=true
    X1=(~B+{Sqrt D})/(2.0*A)
    X2=(~B-{Sqrt D})/(2.0*A)
  else
    SolReal=false
    X1=~B/(2.0*A)
    X2={Sqrt ~D}/(2.0*A)
  end
  {Browse SolReal#X1#X2}
end
```

Calculando raíces: con abstracción

```
proc {EcCuad A B C
      ?SolReal ?X1 ?X2}
   D=B*B-4.0*A*C
in
if D>=0.0 then
  SolReal=true
  X1=(~B+{Sqrt D})/(2.0*A)
  X2=(~B-{Sqrt D})/(2.0*A)
else
  SolReal=false
  X1=~B/(2.0*A)
  X2={Sqrt ~D}/(2.0*A)
end
end
declare RS X1 X2 in
{EcCuad 1.0 3.0 2.0 RS X1 X2}
{Browse RS#X1#X2}
```

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

Modelos y Paradigmas de Programación

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

- En Pascal y C, la implementación coloca parte del almacén en la pila semántica: guarda las variables locales.
- A las variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esto es mucho más sencillo que implementar la recolección de basura.

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

- En Pascal y C, la implementación coloca parte del almacén en la pila semántica: guarda las variables locales.
- A las variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esto es mucho más sencillo que implementar la recolección de basura.
- Desafortunadamente, se crean fácilmente referencias sueltas.

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

- En Pascal y C, la implementación coloca parte del almacén en la pila semántica: guarda las variables locales.
- A las variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esto es mucho más sencillo que implementar la recolección de basura.
- Desafortunadamente, se crean fácilmente referencias sueltas.
- Se puede aliviar: en lenguajes orientados a objetos, tales como C++ o Java, es posible que los objetos juegen el papel de los valores de tipo procedimiento.

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

- En Pascal y C, la implementación coloca parte del almacén en la pila semántica: guarda las variables locales.
- A las variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esto es mucho más sencillo que implementar la recolección de basura.
- Desafortunadamente, se crean fácilmente referencias sueltas.
- Se puede aliviar: en lenguajes orientados a objetos, tales como C++ o Java, es posible que los objetos juegen el papel de los valores de tipo procedimiento.

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

- En Pascal y C, la implementación coloca parte del almacén en la pila semántica: guarda las variables locales.
- A las variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esto es mucho más sencillo que implementar la recolección de basura.
- Desafortunadamente, se crean fácilmente referencias sueltas.
- Se puede aliviar: en lenguajes orientados a objetos, tales como C++ o Java, es posible que los objetos juegen el papel de los valores de tipo procedimiento.

Abstracción procedimental (3)

Formas restringidas de abstracción

- En C, todas las definiciones de procedimientos son globales (no pueden estar anidadas).
- En Pascal, las definiciones de procedimientos pueden estar anidadas, pero los valores de tipo procedimiento sólo se pueden usar en el mismo alcance de la definición del procedimiento, y allí sólo mientras el programa se está ejecutando en ese alcance.
- Se hace difícil "empaquetar" una declaración y ejecutarla en otro lugar.
- Es imposible programar abstracciones de control nuevas. Pascal y C proveen un conjunto predefinido de abstracciones de control (tales como ciclos, condicionales y excepciones).

El problema: la administración de la memoria

- En Pascal y C, la implementación coloca parte del almacén en la pila semántica: guarda las variables locales.
- A las variables locales se les asigna espacio a la entrada al procedimiento, el cual devuelven a la salida correspondiente. Esto es mucho más sencillo que implementar la recolección de basura.
- Desafortunadamente, se crean fácilmente referencias sueltas.
- Se puede aliviar: en lenguajes orientados a objetos, tales como C++ o Java, es posible que los objetos juegen el papel de los valores de tipo procedimiento.

Plan

Genericidad (1)

Qué es . . .

- Es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma.
- Decimos que la entidad ha sido abstraida hacia afuera del cuerpo de la función.
- La entidad específica se recibe como entrada cuando se invoca la función.

Ejemplo:

```
fun {SumList L}
   case L
   of nil then 0
   [] X|L1 then
      X+{SumList L1}
   end
end
```

Genericidad (1)

Qué es . . .

- Es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma.
- Decimos que la entidad ha sido abstraida hacia afuera del cuerpo de la función.
- La entidad específica se recibe como entrada cuando se invoca la función.

Ejemplo:

```
fun {SumList L}
   case L
   of nil then 0
   [] X|L1 then
      X+{SumList L1}
   end
end
```

- Si abstraemos: el número cero (0) y la operación de adición (+).

```
FoldR

fun {FoldR L F U}
   case L
   of nil then U
   [] X|L1 then
      {F X {FoldR L1
                F
                U}}
   end
end
```

Genericidad (1)

Qué es . . .

- Es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma.
- Decimos que la entidad ha sido abstraida hacia afuera del cuerpo de la función.
- La entidad específica se recibe como entrada cuando se invoca la función.

Ejemplo:

```
fun {SumList L}
   case L
   of nil then 0
   [] X|L1 then
      X+{SumList L1}
   end
end
```

- Si abstraemos: el número cero (0) y la operación de adición (+).

```
FoldR

fun {FoldR L F U}
   case L
   of nil then U
   [] X|L1 then
      {F X {FoldR L1
                  F
                  U}}
   end
end
```

Genericidad (1)

Qué es . . .

- Es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma.
- Decimos que la entidad ha sido abstraida hacia afuera del cuerpo de la función.
- La entidad específica se recibe como entrada cuando se invoca la función.

Ejemplo:

```
fun {SumList L}
   case L
   of nil then 0
   [] X|L1 then
      X+{SumList L1}
   end
end
```

- Si abstraemos: el número cero (0) y la operación de adición (+).

```
FoldR

fun {FoldR L F U}
   case L
   of nil then U
   [] X|L1 then
      {F X {FoldR L1
                  F
                  U}}
   end
end
```

Genericidad (1)

Qué es . . .

- Es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma.
- Decimos que la entidad ha sido abstraida hacia afuera del cuerpo de la función.
- La entidad específica se recibe como entrada cuando se invoca la función.

Ejemplo:

```
fun {SumList L}
   case L
   of nil then 0
   [] X|L1 then
      X+{SumList L1}
   end
end
```

- Si abstraemos: el número cero (0) y la operación de adición (+).

```
FoldR

fun {FoldR L F U}
   case L
   of nil then U
   [] X|L1 then
      {F X {FoldR L1
                  F
                  U}}
   end
end
```

Genericidad (1)

Qué es . . .

- Es convertir una entidad específica (i.e., una operación o un valor) en el cuerpo de la función, en un argumento de la misma.
- Decimos que la entidad ha sido abstraida hacia afuera del cuerpo de la función.
- La entidad específica se recibe como entrada cuando se invoca la función.

Ejemplo:

```
fun {SumList L}
   case L
   of nil then 0
   [] X|L1 then
      X+{SumList L1}
   end
end
```

- Si abstraemos: el número cero (0) y la operación de adición (+).

```
FoldR

fun {FoldR L F U}
   case L
   of nil then U
   [] X|L1 then
      {F X {FoldR L1
                  F
                  U}}
   end
end
```

Genericidad (2)

`FoldR`: abstracción de ciclos

La función `{FoldR L F U}` hace lo siguiente:

```
{F X1
   {F X2
      {F X3
         ... {F Xn
                U} ...}}}
```

Ejemplo de aplicación

```
fun {SumList L}
   {FoldR L
      fun {$ X Y}
         X+Y end
      0}
end
```

Otro ejemplo: `ProductList`

```
fun {ProductList L}
   {FoldR L
      fun {$ X Y}
         X*Y end
      1}
end
```

Y otro más

```
fun {Some L}
   {FoldR L
      fun {$ X Y}
         X orelse Y
      end
      false}
end
```

Genericidad (2)

`FoldR`: abstracción de ciclos

La función `{FoldR L F U}` hace lo siguiente:

```
{F X1
   {F X2
      {F X3
         ... {F Xn
                U} ...}}}
```

Ejemplo de aplicación

```
fun {SumList L}
   {FoldR L
           fun {$ X Y}
               X+Y end
           0}
end
```

Otro ejemplo: `ProductList`

```
fun {ProductList L}
   {FoldR L
           fun {$ X Y}
               X*Y end
           1}
end
```

Y otro más

```
fun {Some L}
   {FoldR L
           fun {$ X Y}
             X orelse Y
           end
           false}
end
```

Genericidad (2)

`FoldR`: abstracción de ciclos

La función `{FoldR L F U}` hace lo siguiente:

```
{F X1
   {F X2
      {F X3
         ... {F Xn
                U} ...}}}
```

Ejemplo de aplicación

```
fun {SumList L}
   {FoldR L
           fun {$ X Y}
               X+Y end
           0}
end
```

Otro ejemplo: `ProductList`

```
fun {ProductList L}
   {FoldR L
           fun {$ X Y}
               X*Y end
           1}
end
```

Y otro más

```
fun {Some L}
   {FoldR L
           fun {$ X Y}
              X orelse Y
           end
           false}
end
```

Genericidad (2)

`FoldR`: abstracción de ciclos

La función `{FoldR L F U}` hace lo siguiente:

```
{F X1
   {F X2
      {F X3
         ... {F Xn
                U} ...}}}
```

Ejemplo de aplicación

```
fun {SumList L}
   {FoldR L
           fun {$ X Y}
               X+Y end
           0}
end
```

Otro ejemplo: `ProductList`

```
fun {ProductList L}
   {FoldR L
           fun {$ X Y}
               X*Y end
           1}
end
```

Y otro más

```
fun {Some L}
   {FoldR L
           fun {$ X Y}
             X orelse Y
           end
           false}
end
```

Genericidad (3)

MergeSort Genérico

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

MergeSort Genérico

```
   ...
   fun {MS Xs}
      case Xs
      of nil then nil
      [] [X] then [X]
      else Ys Zs in
         {Dividir Xs Ys Zs}
         {Mezclar {MS Ys}
                  {MS Zs}}
      end
   end
in
   {MS Xs}
end
```

Genericidad (3)

MergeSort Genérico

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

MergeSort Genérico

```
      ...
      fun {MS Xs}
         case Xs
         of nil then nil
         [] [X] then [X]
         else Ys Zs in
            {Dividir Xs Ys Zs}
            {Mezclar {MS Ys}
                     {MS Zs}}
         end
      end
   in
      {MS Xs}
   end
```

Modelos y Paradigmas de Programación

Genericidad (4)

Map

```
fun {Map Xs F}
   case Xs
   of nil then
      nil
   [] X|Xr then
      {F X}|{Map Xr F}
   end
end
```

Filter

```
fun {Filter Xs F}
   case Xs
   of nil then
      nil
   [] X|Xr andthen
      {F X} then
      X|{Filter Xr F}
   [] X|Xr then
      {Filter Xr F}
   end
end
```

Genericidad (4)

Map

```
fun {Map Xs F}
   case Xs
   of nil then
      nil
   [] X|Xr then
      {F X}|{Map Xr F}
   end
end
```

Filter

```
fun {Filter Xs F}
   case Xs
   of nil then
      nil
   [] X|Xr andthen
      {F X} then
      X|{Filter Xr F}
   [] X|Xr then
      {Filter Xr F}
   end
end
```

Plan

Instanciación

Qué es instanciar . . .

- Es la capacidad de devolver un valor de tipo procedimiento como resultado.
- Suponga que `Ordenar` recibe dos entradas, una lista `L` y una función booleana `F`, y devuelve una lista ordenada.
- `CrearOrdenamiento` toma una función booleana de comparación `F` y devuelve una función de ordenamiento que utiliza a `F` como función de comparación.

```
CrearOrdenamiento

fun {CrearOrdenamiento F}
   fun {$ L}
      {Ordenar L F}
   end
end
```

Cada invocación a `CrearOrdenamiento` crea una instancia.

Instanciación

Qué es instanciar . . .

- Es la capacidad de devolver un valor de tipo procedimiento como resultado.
- Suponga que `Ordenar` recibe dos entradas, una lista `L` y una función booleana `F`, y devuelve una lista ordenada.
- `CrearOrdenamiento` toma una función booleana de comparación `F` y devuelve una función de ordenamiento que utiliza a `F` como función de comparación.

`CrearOrdenamiento`

```
fun {CrearOrdenamiento F}
   fun {$ L}
      {Ordenar L F}
   end
end
```

Cada invocación a `CrearOrdenamiento` crea una instancia.

Plan

Embebimiento

Es la capacidad de colocar los valores de tipo procedimiento en estructuras de datos.

Usos

- Evaluación perezosa explícita, también llamada *evaluacion retardada*. Construir estructuras de datos por demanda.
- Módulos. Un módulo es un registro que agrupa un conjunto de operaciones relacionadas.
- Componentes de Software. Un componente de *software* es un procedimiento genérico que recibe un conjunto de módulos como argumentos de entrada y devuelve un nuevo módulo.

Embebimiento

Es la capacidad de colocar los valores de tipo procedimiento en estructuras de datos.

Usos

- Evaluación perezosa explícita, también llamada *evaluacion retardada*. Construir estructuras de datos por demanda.
- Módulos. Un módulo es un registro que agrupa un conjunto de operaciones relacionadas.
- Componentes de *Software*. Un componente de *software* es un procedmiento genérico que recibe un conjunto de módulos como argumentos de entrada y devuelve un nuevo módulo.

Embebimiento

Es la capacidad de colocar los valores de tipo procedimiento en estructuras de datos.

Usos

- Evaluación perezosa explícita, también llamada *evaluacion retardada*. Construir estructuras de datos por demanda.
- Módulos. Un módulo es un registro que agrupa un conjunto de operaciones relacionadas.
- Componentes de *Software*. Un componente de *software* es un procedmiento genérico que recibe un conjunto de módulos como argumentos de entrada y devuelve un nuevo módulo.

Embebimiento

Es la capacidad de colocar los valores de tipo procedimiento en estructuras de datos.

Usos

- Evaluación perezosa explícita, también llamada *evaluacion retardada*. Construir estructuras de datos por demanda.
- Módulos. Un módulo es un registro que agrupa un conjunto de operaciones relacionadas.
- Componentes de *Software*. Un componente de *software* es un procedmiento genérico que recibe un conjunto de módulos como argumentos de entrada y devuelve un nuevo módulo.

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Abstracciones de ciclos (1)

Los ciclos . . .

- en el modelo declarativo tienden a ser pródigos en palabras porque necesitan invocaciones recursivas explícitas.
- son más concisos si se les define como abstracciones.
- Diferentes tipos de ciclos: “para”, sencillos, sobre enteros y sobre listas, y con acumuladores para volverlos más útiles.

Ciclos sobre enteros y listas

Ciclo sobre enteros

```
{For A B S P}
   {P A    }
   {P A+S  }
   {P A+2*S}
      .
      .
      .
   {P A+n*S}
```

*(si S>0: tantas veces como A+n*S=<B)*
*(si S<0: tantas veces como A+n*S>=B)*

Ciclo sobre listas

```
{ForAll L P}
   {P X1}
   {P X2}
   {P X3}
      .
      .
      .
   {P Xn}
```

(donde L=[X1 X2 ... Xn])

Abstracciones de ciclos (1)

Los ciclos . . .

- en el modelo declarativo tienden a ser pródigos en palabras porque necesitan invocaciones recursivas explícitas.
- son más concisos si se les define como abstracciones.
- Diferentes tipos de ciclos: “para”, sencillos, sobre enteros y sobre listas, y con acumuladores para volverlos más útiles.

Ciclos sobre enteros y listas

Ciclo sobre enteros

```
{For A B S P}
   {P A    }
   {P A+S  }
   {P A+2*S}
      .
      .
      .
   {P A+n*S}
```

*(si S>0: tantas veces como A+n*S=<B)*
*(si S<0: tantas veces como A+n*S>=B)*

Ciclo sobre listas

```
{ForAll L P}
   {P X1}
   {P X2}
   {P X3}
      .
      .
      .
   {P Xn}
```

(donde L=[X1 X2 ... Xn])

Abstracciones de ciclos (1)

Los ciclos . . .

- en el modelo declarativo tienden a ser pródigos en palabras porque necesitan invocaciones recursivas explícitas.
- son más concisos si se les define como abstracciones.
- Diferentes tipos de ciclos: “para”, sencillos, sobre enteros y sobre listas, y con acumuladores para volverlos más útiles.

Ciclos sobre enteros y listas

Ciclo sobre enteros

```
{For A B S P}
   {P A    }
   {P A+S  }
   {P A+2*S}
      .
      .
      .
   {P A+n*S}
```

*(si S>0: tantas veces como A+n*S=<B)*
*(si S<0: tantas veces como A+n*S>=B)*

Ciclo sobre listas

```
{ForAll L P}
   {P X1}
   {P X2}
   {P X3}
      .
      .
      .
   {P Xn}
```

(donde L=[X1 X2 ... Xn])

Abstracciones de ciclos (1)

Los ciclos . . .

- en el modelo declarativo tienden a ser pródigos en palabras porque necesitan invocaciones recursivas explícitas.
- son más concisos si se les define como abstracciones.
- Diferentes tipos de ciclos: “para”, sencillos, sobre enteros y sobre listas, y con acumuladores para volverlos más útiles.

Ciclos sobre enteros y listas

Ciclo sobre enteros

```
{For A B S P}
   {P A    }
   {P A+S  }
   {P A+2*S}
      .
      .
      .
   {P A+n*S}
```

*(si S>0: tantas veces como A+n*S=<B)*
*(si S<0: tantas veces como A+n*S>=B)*

Ciclo sobre listas

```
{ForAll L P}
   {P X1}
   {P X2}
   {P X3}
      .
      .
      .
   {P Xn}
```

(donde L=[X1 X2 ... Xn])

Abstracciones de ciclos (2)

Ciclo sobre enteros

```
proc {For A B S P}
   proc {CicloAsc C}
      if C=<B then {P C}
                   {CicloAsc C+S}
      end
   end
   proc {CicloDesc C}
      if C>=B then {P C}
                   {CicloDesc C+S}
      end
   end
in
   if S>0 then {CicloAsc A} end
   if S<0 then {CicloDesc A} end
end
```

Ciclo sobre listas

```
proc {ForAll L P}
   case L
   of nil then
      skip
   [] X|L2 then
      {P X}
      {ForAll L2
             P}
   end
end
```

Abstracciones de ciclos (2)

Ciclo sobre enteros

```
proc {For A B S P}
   proc {CicloAsc C}
      if C=<B then {P C}
                   {CicloAsc C+S}
      end
   end
   proc {CicloDesc C}
      if C>=B then {P C}
                   {CicloDesc C+S}
      end
   end
in
   if S>0 then {CicloAsc A} end
   if S<0 then {CicloDesc A} end
end
```

Ciclo sobre listas

```
proc {ForAll L P}
   case L
   of nil then
      skip
   [] X|L2 then
      {P X}
      {ForAll L2
              P}
   end
end
```

Abstracciones de ciclos (3)

Desventajas ciclos sencillos

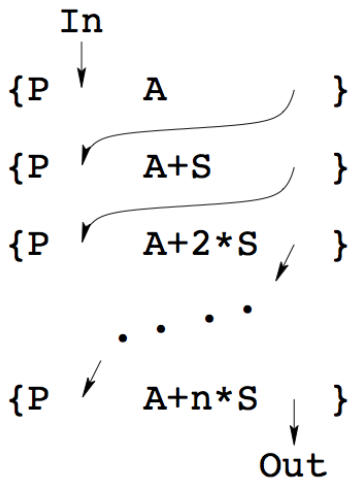
Sólo repiten una acción sobre diferentes argumentos, pero no calculan ningún resultado.

Capacidad de acumulación

Es importante extenderlo con acumuladores para que sean útiles en el modelo declarativo.

Ciclo de acumulación sobre enteros

```
{ForAcc A B S P In Out}
```

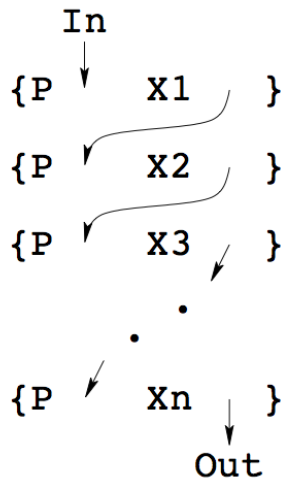


*(si S>0: tantas veces como A+n*S=<B)*

*(si S<0: tantas veces como A+n*S>=B)*

Ciclo de acumulación sobre listas

```
{ForAllAcc L P In Out}
```



(donde L=[X1 X2 ... Xn])

Abstracciones de ciclos (4)

Ciclos de acum. sobre enteros

```
proc {ForAcc A B S P In ?Out}
   proc {CA C In ?Out}
    Mid in
    if C=<B then {P In C Mid}
                 {CA C+S Mid Out}
    else In=Out end
   end
   proc {CD C In ?Out}
    Mid in
    if C>=B then {P In C Mid}
                 {CD C+S Mid Out}
    else In=Out end
   end
in
 if S>0 then {CA A In Out} end
 if S<0 then {CD A In Out} end
end
```

Ciclos de acum. sobre listas

```
proc {ForAllAcc L P In ?Out}
   case L
   of nil then
    In=Out
   [] X|L2 then
    Mid in
      {P In X Mid}
      {ForAllAcc L2
                 P
                 Mid
                 Out}
   end
end
```

Abstracciones de ciclos (4)

Ciclos de acum. sobre enteros

```
proc {ForAcc A B S P In ?Out}
   proc {CA C In ?Out}
    Mid in
    if C=<B then {P In C Mid}
                 {CA C+S Mid Out}
    else In=Out end
   end
   proc {CD C In ?Out}
    Mid in
    if C>=B then {P In C Mid}
                 {CD C+S Mid Out}
    else In=Out end
   end
in
 if S>0 then {CA A In Out} end
 if S<0 then {CD A In Out} end
end
```

Ciclos de acum. sobre listas

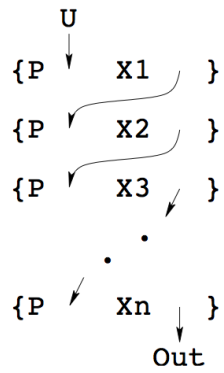
```
proc {ForAllAcc L P In ?Out}
    case L
    of nil then
     In=Out
    [] X|L2 then
     Mid in
       {P In X Mid}
       {ForAllAcc L2
                  P
                  Mid
                  Out}
    end
end
```

Abstracciones de ciclos (5)

Acumuladores sobre listas

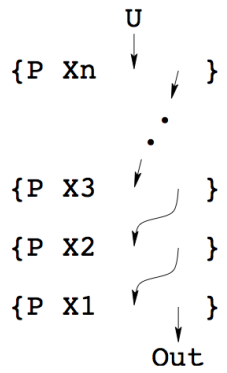
Plegado desde la izquierda

```
{FoldL L P U Out}
```



Plegado desde la derecha

```
{FoldR L P U Out}
```



Sea $l = [x_1\ x_2\ x_3\ \cdots\ x_n]$. Plegar l con el operador infijo f y el elemento neutro u, produce:

- $((\cdots(((u\ f\ x_1)\ f\ x_2)\ f\ x_3)\ f\ \cdots\ x_{n-1})\ f\ x_n)$, o
- $(x_1\ f\ (x_2\ f\ (x_3\ f\ \cdots\ (x_{n-1}\ f\ (x_n\ f\ u))\cdots)))$

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
   if P<1000 then
      {Browse N}
   end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
   thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
       if P<1000 then
          {Browse N}
       end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
   thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
       if P<1000 then
         {Browse N}
       end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
   thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
       if P<1000 then
         {Browse N}
       end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
 thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
       if P<1000 then
         {Browse N}
       end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
 thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
       if P<1000 then
         {Browse N}
       end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
 thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.

Abstracciones de ciclos (6)

Soporte lingüístico

- `for I in A..B do ⟨d⟩ end`

  o:

  `for I in A..B;S do ⟨d⟩ end`

  cuando `S` es diferente de 1.
- `for X in L do ⟨d⟩ end`
- Aprovechando los patrones:

```
for obj(nombre:N
        precio:P
        coordenadas:C)
    in L do
       if P<1000 then
         {Browse N}
       end
end
```

Ciclos declarativos versus imperativos

- El contador de un ciclo imperativo es una variable asignable a la cual se le asigna un valor diferente en cada iteración.
- En un ciclo declarativo en cada iteración se declara una nueva variable.
- No es asignación destructiva en absoluto.
- Es posible ejecutar las iteraciones de un ciclo declarativo concurrentemente sin cambiar el resultado final del ciclo. Por ejemplo:

```
for I in A..B do
 thread ⟨d⟩ end
end
```

ejecuta todas las iteraciones concurrentemente, pero cada una de ellas accede aún al valor correcto de `I`.