Tema 7: Programación imperativa

Sesión 20: Programación imperativa

Indice

- Historia de la programación imperativa
- Características principales
- Datos mutables
- Estructuras de control
- Modelo de entornos

Orígenes de la programación imperativa

- La programación imperativa es la forma natural de programar un computador, es el estilo de programación que se utiliza en el ensamblador, el estilo más cercano a la arquitectura del computador
- Características de la arquitectura clásica de Von Newmann:
  - memoria donde se almacenan los datos (referenciables por su dirección de memoria) y el programa
  - unidad de control que ejecuta las instrucciones del programa (contador del programa)
- Los primeros lenguajes de programación (como el Fortran) son abstracciones del ensamblador y de esta arquitectura, lenguajes más modernos como el BASIC o el C han continuado esta idea

Programación procedural

- Uso de procedimientos y subrutinas
- Los cambios de estado se localizan en estos procedimientos
- Los procedimientos especifican parámetros y valores devueltos (un primer paso hacia la abstracción y los modelos funcionales y declarativos)
- Lenguajes: ALGOL

Programación estructurada

- Artículo a finales de los 60 de Edsger W. Dijkstra: GOTO statement considered harmful en el que se arremete contra la sentencia GOTO de muchos lenguajes de programación de la época
- La programación estructurada mantiene la programación imperativa, pero haciendo énfasis en la necesidad de que los programas sean correctos (debe ser posible de comprobar formalmente los programas), modulares y mantenibles.
- Lenguajes: Pascal, ALGOL 68, Ada

Programación Orientada a Objetos

- La POO también utiliza la programación imperativa, aunque extiende los conceptos de modularidad, mantenibilidad y estado local
- Se populariza a finales de los 70 y principios de los 80

Características principales de la programación imperativa

- La computación se realiza cambiando el estado del programa por medio de sentencias que definen pasos de ejecución del computador
  - Estado del programa modificable
  - Sentencias de control que definen pasos de ejecución

Conceptos de estado de un programa

- Modificación de datos
- Almacenamiento de datos en variables con valor y referencia
- Igualdad de valor y de referencia

Modificación de datos

- Uno de los elementos de la arquitectura de Von Newmann es la existencia de celdas de memoria referenciables y modificables
- El ejemplo típico de este concepto en los lenguajes de programación es el array: una estructura de datos que se almacena directamente en memoria y que puede ser accedido y modificado.
- En Scala los arrays son tipeados, mutables y de tamaño fijo

```
val unoDosTres = Array("uno","dos","tres")
unoDosTres(2) = unoDosTres(0)
```


Modificación de datos

- Un ejemplo de una función que define un parámetro array en Scala:

```
def llenaArray(array: Array[String]) = {
   var i = 1
   while (i < array.length) {
      array(i) = array(0)
      i += 1
   }
}
```

Almacenamiento de datos en variables

- Todos los lenguajes de programación definen variables que contienen datos
- Las variables pueden mantener **valores** (tipos de valor o value types) o **referencias** (tipos de referencia o reference types)
- En C, C++ o Java, los datos **primitivos** como int o char son de tipo **valor** y los objetos y datos **compuestos** son de tipo **referencia**
- La asignación de un valor a una variable tiene implicaciones distintas si el tipo es de valor (se copia el valor) o de referencia (se copia la referencia)

Almacenamiento de datos en variables

- En Scala, y en Scheme, se utiliza la semántica de copia de **valor** para los tipos de datos **simples**

```
var x = 10
var y = x
x = 20
y
```

```
(define x 10)
(define y x)
(define x 20)
y
```

- La semántica de copia de **referencia** se utiliza para los tipos **compuestos**

```
var x = Array(1,2,3,4)
var y = x
x(0) = 10
x
y
```

- La función anterior `llenaArray` recibe un tipo de referencia como parámetro y lo modifica

Igualdad de valor y referencia

- Todos los lenguajes de programación imperativos que permite la distinción entre valores y referencias implementan dos tipos de igualdad entre variables
- Igualdad de valor (el contenido de los datos de las variables es el mismo)
- Igualdad de referencia (las variables tienen la misma referencia)
- Igualdad de referencia => Igualdad de valor (pero al revés no)

Sentencias de control

- La ejecución de pasos de ejecución va modificando el contador de programa que indica la siguiente instrucción a ejecutar
- Sentencias de control:
  - de secuencia: definen instrucciones que son ejecutadas una detrás de otra de forma síncrona
  - de selección: definen una o más condiciones que determinan las instrucciones que se deberán ejecutar
  - de iteración: definen instrucciones que se ejecutan de forma repetitiva hasts que se cumple una determinada condición
- Llamada a una función: se modifica el contador de programa

Mutadores en Scheme y Scala

- Vamos a ver con más detalle ejemplos de mutadores en Scheme y Scala
- Mutadores en Scheme: `set!`, `set-car!`, `set-cdr!`
- Mutadores en Scala: cualquier clase es mutable por defecto, muchas colecciones mutables. Veremos un ejemplo: ListBuffer

Formas especiales `set!` en Scheme

- Operadores de mutación en Scheme:

```
(set! <simbolo> <expresión>)
(set-car! <pareja> <expresión>)
(set-cdr! <pareja> <expresión>)
```

- Ejemplo de mutación y efecto lateral:

```
(define a (cons 1 2))
(define b a)
(set-car! a 10)
a
b
```

Igualdad de valor y referencia en Scheme

- La igualdad de referencia se comprueba con la función `eq?: (eq? x y)` devuelve #t cuando x e y apuntan al mismo dato
- La igualdad de contenido se comprueba con la función `equal?: (equal? x y)` devuelve #t cuando x e y contienen el mismo valor
- Si dos variables son eq? también son equal?.

```
(define a (cons 1 2))
(define b (cons 1 2))
(define c a)
(equal? a b)
(equal? a c)
(eq? a b)
(eq? a c)
```


Ejemplo: lista ordenada

- Ventaja de los operadores de mutación: actualización eficiente de estructuras de datos
- Ejemplo: inserción en una lista ordenada en Scheme

```
(define (make-olist)
   (list '*list*))
```

Ejemplo: lista ordenada

- Ventaja de los operadores de mutación: actualización eficiente de estructuras de datos
- Ejemplo: inserción en una lista ordenada en Scheme

```
(define (make-olist)
   (list '*list*))
```

```
(define (insert! n olist)
   (cond
      ((null? (cdr olist)) (set-cdr! olist (cons n '())))
      ((< n (cadr olist)) (set-cdr! olist (cons n (cdr olist))))
      ((= n (cadr olist)) #f) ; el valor devuelto no importa
      (else (insert! n (cdr olist)))))
```


Ejemplo: tabla hash

- Tabla hash definida con una lista de asociación:

Ejemplo: tabla hash

- Tabla hash definida con una lista de asociación:

```
(define (make-table)
   (list '*table*))

(define (get key table)
   (let ((record (assq key (cdr table))))
      (if (not record)
         #f
         (cdr record))))

(define (put key value table)
   (let ((record (assq key (cdr table))))
      (if (not record)
         (set-cdr! table
            (cons (cons key value) (cdr table)))
         (set-cdr! record value))))
```

Tema 7: Programación imperativa

Sesión 21: Programación imperativa (2)

Indice

- Historia de la programación imperativa
- Características principales
- **Datos mutables** en Scheme y **en Scala**
- **Estructuras de control**
- Modelo de entornos

Igualdad de valor y referencia en Scheme

- La igualdad de referencia se comprueba con la función `eq?: (eq? x y)` devuelve #t cuando x e y apuntan al mismo dato
- La igualdad de contenido se comprueba con la función `equal?: (equal? x y)` devuelve #t cuando x e y contienen el mismo valor
- Si dos variables son eq? también son equal?.

```
(define a (cons 1 2))
(define b (cons 1 2))
(define c a)
(equal? a b)
(equal? a c)
(eq? a b)
(eq? a c)
```


Igualdad de valor y referencia en Scala

- La igualdad de referencia se comprueba con la función `eq`: `x.eq(y)` devuelve `true` cuando x e y apuntan al mismo dato
- La igualdad de contenido se comprueba con la función `equals`: `x.equals(y)` devuelve `true` cuando x e y contienen el mismo valor

```
val x = List(1,2,3)
val y = List(1,2,3)
x.equals(y) --> true
x.eq(y) --> false
val z = x
x.equals(z) --> true
x.eq(z) --> true
```

Ejemplos de mayor eficiencia con los datos mutables en Scheme: inserción en lista ordenada

- Ventaja de los operadores de mutación: actualización eficiente de estructuras de datos
- Ejemplo: inserción en una lista ordenada en Scheme

```
(define (make-olist)
   (list '*list*))
```


Ejemplos de mayor eficiencia con los datos mutables en Scheme: inserción en lista ordenada

- Ventaja de los operadores de mutación: actualización eficiente de estructuras de datos
- Ejemplo: inserción en una lista ordenada en Scheme

```
(define (make-olist)
   (list '*list*))
```

```
(define (insert! n olist)
   (cond
      ((null? (cdr olist)) (set-cdr! olist (cons n '())))
      ((< n (cadr olist)) (set-cdr! olist (cons n (cdr olist))))
      ((= n (cadr olist)) #f) ; el valor devuelto no importa
      (else (insert! n (cdr olist)))))
```

Ejemplo: tabla hash

- Tabla hash definida con una lista de asociación:

Ejemplo: tabla hash

- Tabla hash definida con una lista de asociación:

```
(define (make-table)
   (list '*table*))

(define (get key table)
   (let ((record (assq key (cdr table))))
      (if (not record)
         #f
         (cdr record))))

(define (put key value table)
   (let ((record (assq key (cdr table))))
      (if (not record)
         (set-cdr! table
            (cons (cons key value) (cdr table)))
         (set-cdr! record value))))
```

Datos mutables en Scala

- Objetos
- Colecciones mutables
- List Buffer

Ejemplo de `ListBuffer`

```
import scala.collection.mutable.ListBuffer
val buf = new ListBuffer[Int]
buf += 1
buf += 2
3 +=: buf
```

- ListBuffer es un tipo de referencia

```
val buf = new ListBuffer[Int]
val buf2 = buf
buf2 += 4
```

Operaciones importantes de ListBuffer

- Añadir:

```
val buf = new ListBuffer[Int]
buf += 1
val buf2 = new ListBuffer[Int]
buf2 ++= List(2,3,4)
buf ++= buf2
```

- Eliminar elementos:

```
val buf = new ListBuffer[String]
buf ++= List("Paris","Madrid","Londres")
buf -= "Paris"
```


Operaciones importantes de ListBuffer

- Función `indexOf`:

```
val buf = new ListBuffer[String]
buf ++= List("Paris","Madrid","Londres")
buf.indexOf("Madrid")
```

- Función `update`:

```
val buf = new ListBuffer[String]
buf ++= List("Paris","Madrid","Londres")
buf.update(1,"Barcelona")
```

Estructuras de control

- Secuencia
- Selección
- Iteración

Secuencia

- Scheme:

```
(define x 3)
(define y 5)
(define (cambia-vars a b)
   (set! x a)
   (set! y (+ a b x))
   (+ x y))
```

- Scala:

```
var x=3
var y=5
def cambiaVars(a: Int, b: Int) = {
   x=a
   y=a+b+x
   x+y }
```

Selección `if`

- La sentencia `if` de Scala devuelve el resultado de la última expresión evaluada

```
val filename = if (!args.isEmpty) args(0)
               else "default.txt"
println(filename)
```


Sentencia `match`

- Con efectos laterales:

```
def pruebaMatch(str: String) = {
   str match {
      case "salt" => println("pepper")
      case "chips" => println("salsa")
      case "eggs" => println("bacon")
      case _ => println("huh?")
   }
}
```

Sentencia `match`

- Devolviendo un valor:

```
def pruebaMatch2(str: String): String = {
   str match {
      case "salt" => "pepper"
      case "chips" => "salsa"
      case "eggs" => "bacon"
      case _ => "huh?"
   }
}
```

Tema 7: Programación imperativa

Sesión 22: Programación imperativa (3)

Indice

- Historia de la programación imperativa
- Características principales
- Datos mutables en Scheme y en Scala
- **Estructuras de control**
- **Modelo de entornos**

Sentencias de repetición: `while`

- `while:`

```
def gcdLoop(x: Long, y: Long): Long = {
   var a = x
   var b = y
   while (a != 0) {
      val temp = a
      a = b % a
      b = temp
   }
   b
}
```


Sentencias de repetición: `while`

- `do-while:`

```
def leerEntrada() = {
   var line = ""
   do {
      line = readLine()
      println("Read: " + line)
   } while (line != "")
}
```

Expresiones `for`

- Bucles `for` con contador:

Expresiones `for`

- Bucles `for` con contador:

```
def numsCuadrados(n: Int): (Array[Int],Array[Int]) =
{
   val nums = new Array[Int](n)
   val cuadrados = new Array[Int](n)
   for (i <- 0 until n) {
      nums(i) = i
      cuadrados(i) = i*i
   }
   (nums,cuadrados)
}
```

Expresiones `for`

- Bucles `for` anidados:

```
def divisorPattern(n: Int) = {
   for (i <- 1 to n) {
      for (j <- 1 to n) {
         if ((i % j == 0) || (j % i) == 0)
            print("* ")
         else
            print("  ")
      }
      println(i);
   }
}
```


Expresiones `for`

- Iterando por colecciones:

```
def printFiles() = {
   val filesHere: Array[java.io.File] = (new java.io.File(".")).listFiles
   for (file <- filesHere)
      println(file)
}
```

Expresiones `for`

- Filtrado en el `for`:

```
def printFiles2() = {
   val filesHere: Array[java.io.File] =
      (new java.io.File(".")).listFiles
   for (file <- filesHere if file.getName.endsWith(".scala"))
      println(file)
}
```


Expresiones `for`

- Múltiples condiciones en el `for`:

```
for (
   file <- filesHere
   if file.isFile;
   if file.getName.endsWith(".scala")
) println(file)
```

Expresiones `for`

- Iteración anidada:

```
def fileLines(file: java.io.File) =
   scala.io.Source.fromFile(file).getLines.toList

def grep(pattern: String) =
   for (
      file <- filesHere
      if file.getName.endsWith(".scala");
      line <- fileLines(file)
      if line.trim.matches(pattern)
   ) println(file +": "+ line.trim)

grep(".*gcd.*")
```


Expresiones `for`

- Produciendo una nueva colección: **`yield`**

```
def scalaFiles = for {
   file <- filesHere
   if file.getName.endsWith(".scala")
} yield file
```

Modelo de entornos

- El modelo de sustitución visto en la programación funcional no es suficiente para la programación imperativa
- Necesitamos un modelo en el que las variables queden ligadas a su valor en un ámbito o entorno
- Lo llamamos **modelo de entornos**

Ámbito de variables

- Ámbito: colección de asociaciones entre **nombres de variables** y valores o referencias
- Se crea una nueva variable cuando definimos una variable (en Scala con `var` o `val`, en Scheme con `define`)
- Las variables pueden cambiar de valor como resultado de la ejecución de sentencias de asignación
- Las variables se evalúan con su último valor asignado
- Por defecto existe el **entorno global** en el que se ejecutan las expresiones del intérprete

Ejemplo

- Dibujamos el entorno resultado de ejecutar las siguientes sentencias:

- En Scheme:

```
(define x 1)
(define y (+ x 2))
(set! x 5)
(set! y (+ x y))
```

- En Scala:

```
var x=1
var y=x+2
x=5
y=x+y
```


Ámbitos e invocación de funciones

- ¿Qué sucede con los entornos cuando se invoca una función? Vamos a investigarlo.

- ¿El ámbito global está accesible desde el cuerpo de la función? ¿Se pueden cambiar variables definidas en el ámbito global?

- Comprobamos qué pasa con el siguiente ejemplo:

```
var x = 10

def cambiaX() = {
   x = x+1
   x
}

cambiaX()
x
```

```
(define x 10)

(define (cambia-x)
   (set! x (+ x 1))
   x)

(cambia-x)
x
```

Ámbitos e invocación de funciones

- ¿Qué pasa si creamos una variable en la función? ¿Es accesible desde el ámbito global? ¿Y si tiene el mismo nombre que una variable del ámbito global? Vamos a comprobarlo ejecutando los ejemplos y dibujando los ámbitos creados:

```
(define (crea-vars)
   (define x 10)
   (define y (+ x 20))
   (+ x y))

(define x 0)
(crea-vars)
x
y
```

```
def creaVars() = {
   var x=10
   var y=x+20
   x+y
}

var x=0
creaVars()
x
y
```


Resumen de lo que hemos visto

- El ámbito global es accesible desde invocaciones a funciones
- La invocación a una función crea un ámbito local **contenido** en el ámbito global
- Las variables creadas en ese ámbito local no son accesibles desde el ámbito global

Tema 7: Programación imperativa

Sesión 23: Programación imperativa (4)

Indice

- Historia de la programación imperativa
- Características principales
- Datos mutables en Scheme y en Scala
- Estructuras de control
- **Modelo de entornos**

Repaso del modelo de entornos visto hasta ahora

- Un entorno contiene un conjunto de nombres de variables y valores asociadas a ellas.
- Por defecto existe un entorno denominado **entorno global** en el que se almacenan todas las variables creadas en el intérprete (tanto de Scheme como de Scala).
- La invocación a una función crea un entorno local **contenido** en el entorno global. Las sentencias de la función se ejecutan en este entorno local. Las variables creadas en estas sentencias se guardan en el entorno en el que se ejecutan (el local).
- Las variables del **entorno padre** en el que está contenido el entorno local son también accesibles desde las sentencias ejecutadas en el entorno local (siempre que tengan un nombre distinto de las variables locales).

Ejemplo

- Representa los entornos creados con la ejecución de las siguientes instrucciones:

```
;; Scheme

(define x 10)
(define (cambia-x)
   (set! x (+ x 1))
   x)

(cambia-x)
x
```

```
// Scala

var x = 10
def cambiaX(y: Int) = {
   x = x+y
   x
}

cambiaX()
x
```

Objetos creados en el entorno local

```
;; Scheme

(define (crea-pareja)
   (define a (cons 1 2))
   a)

(define b (crea-pareja))
```

```
// Scala

def creaListBuffer() = {
   var a = new ListBuffer[String]
   a += "Nueva York"
   a
}

var b = creaListBuffer()
```

- El objeto devuelto se crea en el entorno local y se referencia también desde el global

¿Qué sucede si el objeto creado en el entorno local es una función?

- Vamos a ver lo que pasa cuando se combinan funciones de orden superior y entornos
- Creamos una función en el entorno local y la devolvemos al entorno global

```
(define (make-contador)
   (define x 0)
   (lambda ()
      (set! x (+ x 1))
      x))

(define x 10)
(define f (make-contador))
(f)
(f)
x
```

```
def makeContador() = {
   var x = 0
   () => {
      x = x+1
      x
   }
}

var x = 10
f = makeContador()
f()
f()
x
```

¿Qué sucede si el objeto creado en el entorno local es una función?

- Estas funciones creadas en el entorno local se denominan *closures*
- Precisamente el nombre *closure* viene del hecho de que la función que se devuelve viene "acompañada" de un entorno particular que puede utilizar

```
(define (make-contador)
   (define x 0)
   (lambda ()
      (set! x (+ x 1))
      x))

(define x 10)
(define f (make-contador))
(f)
(f)
x
```

```
def makeContador() = {
   var x = 0
   () => {
      x = x+1
      x
   }
}

var x = 10
f = makeContador()
f()
f()
x
```


Reglas del funcionamiento de closures en el modelo de entornos

- **Regla de creación de closures**: Cuando se crea una función anónima en un entorno local y se devuelve como resultado, la función queda asociada al entorno local en que se ha creado. Una posterior invocación a la función anónima se ejecutará dentro de este entorno local.
- **Regla de invocación de funciones**: Cuando se invoca a una función se crea un entorno local dentro del entorno **en el que la función se creó** y se ejecutan todas sus sentencias en este entorno local.

Estado local

- En el ejemplo anterior, la variable `x` es un estado local a la función que se devuelve
- El estado se mantiene entre distintas invocaciones a `f` (diferencia con el paradigma funcional)
- Se crean tantos ámbitos locales como invocaciones a `makeContador`
- Combinando closures y programación imperativa es posible definir estado local asociado a funciones (un enfoque distinto del de programación orientada a objetos)

Estado local

```
contador1 = makeContador()
contador2 = makeContador()
contador1()
contador1()
contador1()
contador2()
```

Compartiendo estado entre ámbitos

- Un último ejemplo, en el que comprobamos que el estado local puede ser accesible desde otros ámbitos (el global, en el caso del ejemplo)

```
def makeClosure() = {
   var b = new ListBuffer[String]
   (s: String) => {
      b += s
      b
   }
}
var g = makeClosure()
var buf = g("Londres")
buf += "Roma"
g("París")
```


Compartiendo estado entre ámbitos

- Un último ejemplo, en el que comprobamos que el estado local puede ser accesible desde otros ámbitos (el global, en el caso del ejemplo)

```
def makeClosure() = {
   var b = new ListBuffer[String]
   (s: String) => {
      b += s
      b
   }
}
var g = makeClosure()
var buf = g("Londres")
buf += "Roma"
g("París")
```

- ¿Cuál es el tipo de g?
- ¿Qué pasa cuando invocamos a g?
- ¿En qué ámbito reside el listBuffer? ¿Quién tiene acceso a él?
- ¿Cómo se dibujarían los ámbitos resultantes?
- ¿Podrías hacer un ejemplo similar en Scheme?

Resumen del modelo de entornos

- Las variables se definen en entornos, asociando un valor a su nombre.
- Por defecto existe un entorno global.
- Las instrucciones se ejecutan en los entornos. Cuando se referencia una variable en un entorno se devuelve su valor definido en ese entorno. Si la variable no existe en ese entorno, se busca en su entorno padre, hasta que se llega al entorno global.
- Una invocación a una función crea un ámbito local, dentro del ámbito en el que se creó la función.
- El cuerpo de la función se ejecuta en el ámbito local creado por su invocación.

Un último ejemplo

- ¿Cómo podríamos modificar la función `makeContador` para conseguir que todos los incrementos de los contadores se guarden en alguna variable compartida y local?