Taller 2: Funciones de alto orden

Juan Francisco Díaz Frias Emily Núñez

Septiembre 2024

1. Ejercicio de programación: Simular un sumador de n dígitos a partir de compuertas lógicas sencillas

Los computadores están hechos a partir de circuitos lógicos. Los circuitos se construyen a partir de ensamblar componentes. Los componentes pueden ser sencillos o compuestos de componentes sencillos.

En este ejercicio vamos a programar los componentes más sencillos como son las *compuertas lógicas*. Hay 3 compuertas lógicas básicas: la compuerta *not*, la compuerta *and* y la compuerta *or*. Cada compuerta se puede ver como una caja negra que tiene unas entradas y una salida. A esas cajas negras se les denominan *Chips*.

Gráficamente, estas compuertas tienen unos íconos que las representan, así:

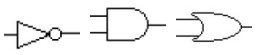


Figura 1: Compuertas *not*, *and* y *or* respectivamente

Cada una de las compuertas recibe unas señales de entrada (bits, 0 ó 1) y produce una señal de salida (la función lógica correspondiente). Tomando las entradas como bits, las salidas se pueden calcular con operaciones aritméticas sencillas, así:

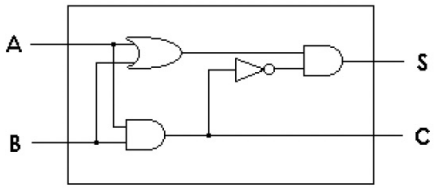
- Compuerta *not*: Si recibe una señal de entrada x su salida es $1 - x$.
- Compuerta *and*: Si recibe dos señales de entrada x, y su salida es xy.
- Compuerta *or*: Si recibe dos señales de entrada x, y su salida es $x + y - x * y$.

Con estas compuertas sencillas, se pueden construir *Chips* que corresponden a componentes más complejos.

Un ejemplo es el *semisumador* que es un *chip* que recibe dos señales (bits) A, B y devuelve dos señales S, C donde S corresponde a la suma de los dos dígitos binarios, y C corresponde al acarreo. Su funcionamiento se especifica en la siguiente tabla, y al lado se presenta cómo se construiría con las compuertas sencillas:

Cuadro 1: Semisumador

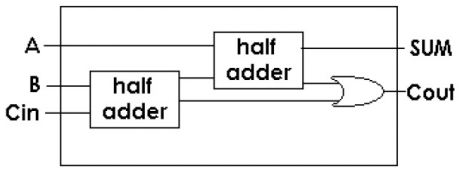
A	B	C	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

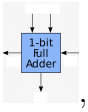


Otro ejemplo es el *sumador completo* que es un *chip* que recibe dos señales (bits) A, B y una señal de un acarreo entrante Cin y devuelve dos señales $SUM, Cout$ donde SUM corresponde a la suma de los dos dígitos binarios teniendo en cuenta el acarreo entrante, y $Cout$ corresponde al nuevo acarreo. Su funcionamiento se especifica en la siguiente tabla, y al lado se presenta cómo se construiría con las compuertas sencillas y dos *semisumadores*:

Cuadro 2: Sumador completo

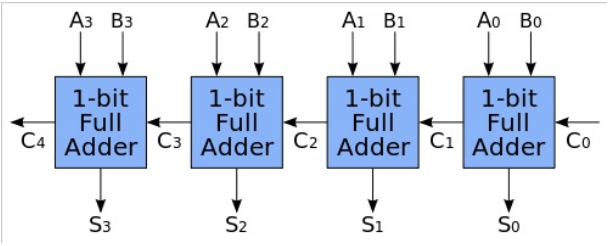
A	B	Cin	$Cout$	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Se pueden usar n *sumadores completos* como cajas negras: 

, para construir un

nuevo *Chip* sumador de números binarios de n dígitos, así:



El objetivo de este taller es volverse competentes en desarrollar funciones de alto orden. Vamos a usarlas para representar los *Chips* y construir las compuertas sencillas mencionadas, así como *semisumadores*, *sumadores completos* y ensamblar un sumador de n dígitos binarios, con $n \geq 1$. A este último lo llamaremos el *sumador-n*.

Comenzaremos por representar cada *Chip* como un valor del tipo $List[Int] => List[Int]$ es decir como una función que recibe una lista de bits de entrada y produce una lista de bits de salida.

```
type Chip = List[Int] => List[Int]
```

Note que las compuertas lógicas *and* y *or* se pueden ver como un *Chip* que recibe una lista de entrada de dos bits y devuelve una lista de salida con un solo bit. Por su lado, la compuerta lógica *not* se puede ver como un *Chip* que recibe una lista de entrada de un bit y devuelve una lista de salida con un bit.

El *semisumador* se puede ver como un *Chip* que recibe una lista de entrada de dos bits y devuelve una lista de salida con dos bits.

El *sumador completo* se puede ver como un *Chip* que recibe una lista de entrada de tres bits y devuelve una lista de salida con dos bits.

Y, por último, el *sumador-n* se puede ver como un *Chip* que recibe una lista de entrada de $2n$ bits (los n bits de un número binario y los n bits del otro) y devuelve una lista de salida con $n+1$ bits correspondientes al acarreo final y los resultados de las sumas de los bits en el orden correspondiente.

Restricción sobre uso de métodos de listas: Hay cuatro métodos que se proveen en List[Int] que pueden ser útiles para este ejercicio:

- l.head: Int (devuelve el primer elemento de la lista l)
- l.tail: List[Int] (devuelve la lista sin el primer elemento l)
- l.take(n):List[Int] (devuelve los primeros n elementos de l)
- l.drop(n):List[Int] (devuelve los útlimos n elementos de l)

En general sólo necesitan los dos primeros métodos. Los dos últimos sólo los necesitan para implementar el *sumador-n*. El uso de cualquier otro método debe ser autorizado por el profesor.

1.1. Creación de las compuertas sencillas

1.1.1. Creando compuertas unarias

Implemente la función `crearChipUnario` que dada una función unaria de tipo $Int => Int$ devuelva un *Chip* correspondiente a procesar el único bit de la lista de entrada como lo indica la función unaria de entrada:

```
  def crearChipUnario(funcion:Int=>Int): Chip =
  // Dada la funcion que calcula una operacion logica unaria (como la negacion),
  // devuelve la representacion del Chip respectivo
...
    }
```

Por ejemplo, para crear el *Chip* correspondiente a la compuerta lógica *not* se usaría la función así:

```
val chip_not = crearChipUnario((x:Int) => (1-x))
chip_not(List(1))
chip_not(List(0))
```

y el resultado al correrlo sería:

```
val chip_not: Circuitos.Chip = Circuitos.package <function>
val res0: List[Int] = List(0)
val res1: List[Int] = List(1)
```

1.1.2. Creando compuertas binarias

Implemente la función `crearChipBinario` que dada una función binaria de tipo $(Int, Int) => Int$ devuelva un *Chip* correspondiente a procesar el único bit de la lista de entrada como lo indica la función binaria de entrada:

```
  def crearChipBinario(funcion:(Int,Int)=>Int): Chip = {
    // Dada la funcion que calcula una operacion logica binaria,
    // devuelve la representacion del Chip respectivo
...
  }
```

Por ejemplo, para crear los *Chips* correspondientes a las compuertas lógicas *and* y *or* se usaría la función así:

```
val chip_and = crearChipBinario((x:Int, y:Int) => (x*y))
chip_and(List(1,1))
chip_and(List(1,0))

val chip_or = crearChipBinario((x:Int, y:Int) => x + y - (x*y))
chip_or(List(1,1))
chip_or(List(1,0))
```

y el resultado al correrlo sería:

```
val chip_and: Circuitos.Chip = Circuitos.package <function>
val res2: List[Int] = List(1)
val res3: List[Int] = List(0)

val chip_or: Circuitos.Chip = Circuitos.package <function>
val res4: List[Int] = List(1)
val res5: List[Int] = List(1)
```

1.2. Creando *semisumadores*

Implemente la función `half_adder` que construya un *Chip* correspondiente a un *semisumador*:

```
  def half_adder:Chip = {
    // Devuelve el Chip correspondiente a un half_adder
...
  }
```

Por ejemplo, para crear un *semisumador* se usaría la función así:

```
val ha=half_adder
ha(List(0,0))
ha(List(0,1))
ha(List(1,0))
ha(List(1,1))
```

y el resultado al correrlo sería:

```
val ha: Circuitos.Chip = Circuitos.package<function>
val res6: List[Int] = List(0, 0)
val res7: List[Int] = List(0, 1)
val res8: List[Int] = List(0, 1)
val res9: List[Int] = List(1, 0)
```

1.3. Creando *sumadores completos*

Implemente la función `full_adder` que construya un *Chip* correspondiente a un *sumador completo*:

```
  def full_adder:Chip = {
    // Devuelve el Chip correspondiente a un full_adder
...
  }
```

Por ejemplo, para crear un *sumador completo* se usaría la función así:

```
val fa=full_adder
fa(List(1,1,0))
fa(List(1,1,1))
fa(List(0,1,0))
fa(List(0,1,1))
```

y el resultado al correrlo sería:

```
val fa: Circuitos.Chip = Circuitos.package<function>
val res10: List[Int] = List(1, 0)
val res11: List[Int] = List(1, 1)
val res12: List[Int] = List(0, 1)
val res13: List[Int] = List(1, 0)
```

1.4. Construyendo un *sumador-n*

Implemente la función `adder` que, dado $n \geq 1$ construya un *Chip* correspondiente a un *sumador-n*:

```
  def adder(n:Int):Chip = {
```

```
        // Devuelve un Chip que recibe 2n bits de entrada (los primeros n bits corresponden
        // al primer numero a sumar y los segundos n bits corresponden al segundo numero a sumar
        // y devuelve n+1 bits de salida, donde el primero es el acarreo de la suma y los otros n
        // corresponden a los n bits de salida
...
  }
```

Por ejemplo, para crear un *sumador*-1 de números de un bit de longitud, se usaría la función así:

```
val add_1 = adder(1)
add_1(List(1) ++ List(1))
add_1(List(0) ++ List(0))
add_1(List(1) ++ List(0))
add_1(List(0) ++ List(1))
```

y el resultado al correrlo sería:

```
val add_1: Circuitos.Chip = Circuitos.package<function>
val res14: List[Int] = List(1, 0)
val res15: List[Int] = List(0, 0)
val res16: List[Int] = List(0, 1)
val res17: List[Int] = List(0, 1)
```

Por ejemplo, para crear un *sumador*-2 de números de dos bits de longitud, se usaría la función así:

```
val add_2 = adder(2)
add_2(List(1,0) ++ List(0,1))
add_2(List(1,1) ++ List(0,1))
```

y el resultado al correrlo sería:

```
val add_2: Circuitos.Chip = Circuitos.package<function>
val res18: List[Int] = List(0, 1, 1)
val res19: List[Int] = List(1, 0, 0)
```

Por ejemplo, para crear un *sumador*-3 de números de tres bits de longitud, se usaría la función así:

```
val add_3 = adder(3)
add_3(List(1,0,1) ++ List(0, 0 , 0))
add_3(List(1,0,1) ++ List(1, 0 , 1))
```

y el resultado al correrlo sería:

```
val add_3: Circuitos.Chip = Circuitos.package<function>
val res20: List[Int] = List(0, 1, 0, 1)
val res21: List[Int] = List(1, 0, 1, 0)
```

Por ejemplo, para crear un *sumador*-4 de números de cuatro bits de longitud, se usaría la función así:

```
val add_4= adder(4)
add_4(List(1,0,1,1) ++ List(1, 0 , 1, 0))
```

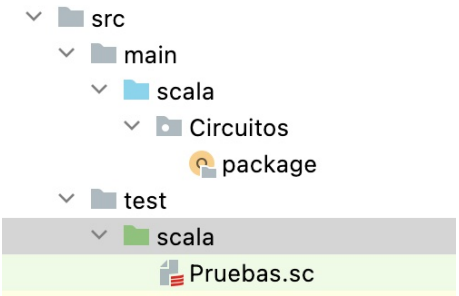
y el resultado al correrlo sería:

```
val add_4: Circuitos.Chip = Circuitos.package<function>
val res22: List[Int] = List(1, 0, 1, 0, 1)
```

2. Entrega

2.1. Paquete *Circuitos* y *worksheet* de pruebas

Usted deberá entregar dos archivos `package.scala` y `pruebas.sc` los cuales harán parte de su estructura de proyecto `IntellijIdea`, como se muestra en la figura a continuación:



Las funciones correspondientes a cada ejercicio, `crearChipUnario`, `crearChipBinario`, `half_adder`, `full_adder`, y `adder`, deben ser implementadas en un paquete de Scala denominado `Circuitos`. En ese paquete debe venir un archivo denominado `package.scala` que debe tener la forma siguiente:

```
package object Circuitos {

  type Chip = List[Int] => List[Int]


  def crearChipBinario(funcion:(Int,Int)=>Int): Chip = {
    // Dada la funcion que calcula una operacion logica binaria,
    // devuelve la representacion del Chip respectivo
...
  }

  def crearChipUnario(funcion:Int=>Int): Chip = {
  // Dada la funcion que calcula una operacion logica unaria (como la negacion),
  // devuelve la representacion del Chip respectivo
...
}

  def half_adder:Chip = {
    // Devuelve el Chip correspondiente a un half_adder
...
  }

  def full_adder:Chip = {
    // Devuelve el Chip correspondiente a un full_adder
...
  }

  def adder(n:Int):Chip = {
          // Devuelve un Chip que recibe una lista de 2n bits de entrada (los primeros n bits corresponden
          // al primer numero a sumar y los segundos n bits corresponden al segundo numero a sumar
          // y devuelve n+1 bits de salida, donde el primero es el acarreo de la suma y los otros n
          // corresponden a los n bits de salida
...
}
```

Dicho paquete será usado en un *worksheet* de Scala con casos de prueba. Estos casos de prueba deben venir en un archivo denominado `pruebas.sc` . Un ejemplo de un tal archivo es el siguiente:

```
import Circuitos._

val chip_not = crearChipUnario((x:Int) => (1-x))
chip_not(List(1))
chip_not(List(0))

val chip_and = crearChipBinario((x:Int, y:Int) => (x*y))
chip_and(List(1,1))
chip_and(List(1,0))

val chip_or = crearChipBinario((x:Int, y:Int) => x + y - (x*y))
chip_or(List(1,1))
chip_or(List(1,0))

half_adder(List(0,0))
half_adder(List(0,1))
half_adder(List(1,0))
half_adder(List(1,1))

full_adder(List(1,1,0))
full_adder(List(1,1,1))
full_adder(List(0,1,0))
full_adder(List(0,1,1))

adder(1)(List(1) ++ List(1))
adder(1)(List(0) ++ List(0))
adder(1)(List(1) ++ List(0))
adder(1)(List(0) ++ List(1))

adder(2)(List(1,0) ++ List(0,1))
adder(2)(List(1,1) ++ List(0,1))

adder(3)(List(1,0,1) ++ List(0, 0 , 0))
adder(3)(List(1,0,1) ++ List(1, 0 , 1))

val ad4= adder(4)
ad4(List(1,0,1,1) ++ List(1, 0 , 1, 0))
```

2.2. Informe del taller - secciones

Todo taller debe venir acompañado de un informe en formato pdf. El informe debe contener la información explícita solicitada en el enunciado.

Para este caso, el informe del taller debe contener al menos tres secciones: informe de funciones de alto orden, informe de corrección y conclusiones.

2.2.1. Informe de funciones de alto orden

Tal como se ha visto en clase, usar funciones de alto orden significa usarlas como parámetro, de forma anónima o como respuesta. Indique en una tabla, para cada función realizada, cuáles de estas tres formas de uso se utilizaron.

2.2.2. Informe de corrección

Es muy importante reflexionar sobre la corrección del código entregado. Para ello se deberá argumentar sobre la corrección de los programas entregados, y también deberá entregar un conjunto de pruebas. Todo esto lo consigna en esta sección del informe, dividida de la siguiente manera:

Argumentación sobre la corrección Para cada función, `crearChipUnario`, `crearChipBinario`, `half_adder`, `full_adder`, y `adder`, argumente lo más formalmente posible por qué es correcta. Utilice inducción o inducción estructural donde lo vea pertinente. Estas argumentaciones las consigna en esta sección del informe.

Casos de prueba Para cada función se requieren mínimo 5 casos de prueba donde se conozca claramente el valor esperado, y se pueda evidenciar que el valor calculado por la función corresponde con el valor esperado en toda ocasión.

Una descripción de los casos de prueba diseñados, sus resultados y una argumentación de si cree o no que los casos de prueba son suficientes para confiar en la corrección de cada uno de sus programas, los registra en esta sección del informe. Obviamente, esta parte del informe debe ser coherente con el archivo de pruebas entregado, `pruebas.sc`.

2.3. Fecha y medio de entrega

Todo lo anterior, es decir los archivos `package.scala`, `pruebas.sc`, e `Informe del taller`, debe ser entregado vía el campus virtual, a más tardar a las 10 a.m. del jueves 19 de septiembre de 2024, en un archivo comprimido que traiga estos tres elementos.

Cualquier indicación adicional será informada vía el foro del campus asociado a *programación funcional*.

3. Evaluación

Cada taller será evaluado tanto por el profesor del curso con ayuda del monitor, como por 2 compañeros que hayan entregado el taller. A este tipo de evaluación se le conoce como *Coevaluación*.

El objetivo de la coevaluación es lograr aprendizajes a través de:

- La lectura de lo que otros compañeros hicieron ante el mismo reto. Esto permite contrastar las soluciones propias con las de otros, y aprender de ellas, o compartir mejores maneras de hacer algo con otros.
- Retroalimentar a los compañeros que fueron asignados para evaluar. Al escribir la percepción que tenemos sobre el trabajo del otro, podemos aprender de cómo lo hicieron, y dar indicaciones al otro sobre otra formas de hacer lo mismo o, incluso, felicitarlo por la solución que presenta.
- La lectura de las retroalimentaciones de mis compañeros o del profesor/monitor.

La calificación de cada taller corresponderá entonces:

- en un 80 % a la calificación ponderada que reciba del profesor/monitor, vía una rúbrica de evaluación (pesa 5 veces lo que pesa la de otro estudiante) y de los tres compañeros asignados para evaluarlo.
- en un 20 % a la calificación que el sistema hará del trabajo de evaluación asignado. El Sistema tiene un método inteligente para estimar esa calificación, a partir de las evaluaciones realizadas por cada estudiante y por el profesor/monitor.