INFORME DEL TALLER 2

MAYRA ALEJANDRA SANCHEZ SALINAS (2040506)

FUNDAMENTOS DE PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

JUAN FRANCISCO DIAZ FRIAS

UNIVERSIDAD DEL VALLE
SANTIAGO DE CALI, VALLE DEL CAUCA
2022

- Definición del tipo Conj

```
/*
Funcion caracteristica
*/
 type Conj = Int => Boolean
```

- Definición de la función pertenece

```
/*
Funcion que verifica si un numero pertenece o no a un conjunto
*/
def pertenece(elem: Int, s: Conj): Boolean = s(elem)
```

INFORME DE FUNCIONES DE ALTO ORDEN

Funciones básicas sobre conjuntos

1. **Conjunto unitario**

```
/*
 * Devuelve el conjunto con un solo elemento
 */
def conjuntoUnitario(elem: Int): Conj = {
 (x: Int) => x == elem
}
```

En esta función lo que se hace es validar si el valor que se ingreso es igual a lo que se debe ingresar, por ejemplo, esta función no va a recibir algo que no sea un entero, ejemplo: Si se ingresa una cadena de caracteres se arroja *false*, en cambio si se ingresa un número ya sea desde - infinito hasta + infinito es *true*

2. **Unión, intersección y diferenciación**
   - **Unión**

```
/*
 * Devuelve la union de dos conjuntos
 * el conjunto de todos los elemntos que estan en 's' o en 't'
 */
def union (s: Conj, t: Conj): Conj = {
 (x: Int) => s(x) || t(x)
}
```

En esta función lo que se hace es verificar que el elemento x está en el conjunto s o en el conjunto t.

Por ejemplo, ingresamos un valor x en este caso 3 y tenemos dos conjuntos s y t los cuales contienen los siguientes elementos, s : { 1, 2, 3 } y t : { 2 } la respuesta será de tipo Booleano *true* ya que el x = 3 se encuentra en el conjunto s, en cambio si el valor es de x = 4 la respuesta será *false* ya que ese elemento x no pertenece al conjunto s o t

   - **Intersección**

```
/*
 * Devuelve la interseccion de dos conjuntos,
 * el conjunto de todos los elementos que estan en ´s´y en ´t´
 */
def interseccion (s: Conj, t: Conj): Conj = {
 (x: Int) => s(x) && t(x)
}
```

En esta función lo que se hace es verificar que el elemento x está en el conjunto s y en el conjunto t.

Por ejemplo, ingresamos un valor x en este caso 7 y tenemos dos conjuntos s y t los cuales contienen los siguientes elementos, s : { 1, 2, 3, 4, 5, 6, 7 } y t : { 2, 7 } la respuesta será de tipo Booleano *true* ya que el x = 7 se encuentra en el conjunto s y en el t, en cambio si el valor es de x = 9 la respuesta será *false* ya que ese elemento x no pertenece al conjunto s ni al conjunto t

- **Diferenciación**

```
/*
 * Devuelve la diferencia de dos conjuntos,
 * el conjunto de todos los elementos de ´s´ que no están en ´t´
 */
def dif (s: Conj, t: Conj): Conj = {
 (x: Int) => s(x) && !t(x)
}
```

En esta función lo que se hace es verificar que el elemento x está en el conjunto s y no este en el conjunto t.

Por ejemplo, ingresamos un valor x en este caso 5 y tenemos dos conjuntos s y t los cuales contienen los siguientes elementos, s : { 1, 2, 3, 4, 5, 6, 7 } y t : { 2 } la respuesta será de tipo Booleano *true* ya que el x = 5 se encuentra en el conjunto s y no en el conjunto t, en cambio si el valor de es x = 2 la respuesta será *false* ya que ese elemento x pertenece al conjunto s y al conjunto t

3. **Filtrar**

```
/*
 * Devuelve el subconjunto de elementos de ´s´ para los cuales ´p´ se cumple
 */
def filtrar (s: Conj, p: Int => Boolean): Conj = {
 interseccion(s,p)
}
```

En esta función se llama la función *intersección* que se creó anteriormente, en ella se ingresa el conjunto s y un número p, con el cual lo que se logra es verificar si el número ingresado en p se encuentra en el conjunto s, si es así la respuesta es *true* y si no es así entonces es *false*

Consultas y transformaciones de conjuntos

1. **Para todo**

```
Los limites para ´para_todo´ y ´existe´ son +/- 1000
 */
val limite = 1000
/*
Calcula si todos los elementos de ´s´ que estan en los limites satisfacen ´p´
 */
def paratodo(s: Conj, p: Int => Boolean): Boolean = {
 def paratodoF(elem: Int): Boolean = {
  if (s(elem) && !p(elem)) false
  else if (elem > limite || elem < -limite) true
  else paratodoF(elem-1)
 }
 paratodoF(limite)
}
```

Este código funciona de forma recursiva en donde se recorre el conjunto s para verificar que los elementos de s que están en los límites de existencia para los que se cumplen p.

2. **Existe**

```
/*
Calcula si existe algun elemento de ´s´dentro de los limites que satisfaga ´p´
 */
def existe(s: Conj, p: Int => Boolean): Boolean = {
 !paratodo(s,(elem: Int) => !pertenece(elem,p))
}
```

En esta función se verifica si existe un elemento del conjunto s para los cuales p se cumple satisfactoriamente dentro de los límites.

3. **Map**

```
/*
Devuelve el conjunto transformado aplicando ´f´ a cada elemento de ´s´
 */
def map(s: Conj, f: Int => Int): Conj = {
  (x => existe(s, (y: Int) => f(y) == x))
}
```

Por último esta función verifica si existe un elemento del conjunto s para los cuales f se satisface dentro de los límites.

Funciones	Parámetro	Anonima	Respuesta
Conjunto unitario	**X**		
Unión, intersección y diferenciación	**X**		
Filtrar			**X**
Para todo			**X**
Existe		**X**	
Map			**X**

INFORME DE CORRECCIÓN

Funciones básicas sobre conjuntos

1. **Conjunto unitario**

   Sea elem: Int & x: Int <-> *true*
   if (elem: Int & x: otro tipo) <-> *false*

   Esto quiere decir que el valor ingresado debe ser únicamente un entero para ser true de resto es falso

2. **Unión, intersección y diferenciación**

   - **Unión**

     Sea s : Conj & t : Conj <->
     s : { 1 , 2 , 3 } & t : { 3, 6 } <->
     s & t : { 1, 2, 3, 6 }

     Para ello se necesita saber qué elementos se encuentran tanto en los conjuntos s y t, después hacer la unión, por ende (x: Int) => s(x) || t (x), sirve para tener en x los valores del elemento de los conjuntos

   - **Intersección**

     Sea s : Conj & t : Conj <->
     s : { 1 , 3 , 4 } & t : { 3, 5 } <->
     s & t : { 3 }

     Para ello se necesita saber qué elementos se encuentran tanto en los conjuntos s y t, después hacer la intersección, por ende (x: Int) => s(x) && t (x), sirve para tener en x los valores del elemento de los conjuntos y devolver solamente los elementos iguales en los dos conjuntos a comparar

   - **Diferenciación**

     Sea s : Conj & t : Conj <->
     s : { 1 , 3 , 6 } & t : { 3, 4 } <->
     s & t : { 1, 6 }

     Para ello se necesita saber qué elementos se encuentran tanto en los conjuntos s y t, después hacer la diferencia, por ende (x: Int) => s(x) && !t (x), sirve para tener en x los valores del elemento de los conjuntos y devolver solamente los elementos de s que no se encuentren en t.

3. **Filtrar**
   Sea s : Conj & p : Int <->
   s: {1, 2, 3, 4, 5} & p: 5 <->
   true

   Para ello se necesita saber qué elementos se encuentran en el conjunto s, después mirar qué valor hay en p y verificar si el valor p se encuentra en s.

Consultas y transformaciones de conjuntos

4. **Para todo**

   Sea s: Conj & p: Int => Booleano <->
   si existe un elemento que verifica los p que no cumplen en s y el elemento aumenta hasta llegar a +/- 1000 <->
   true

5. **Existe**

   Sea s: Conj & p: Int => Booleano <->
   Es *true* <->
   !paratodo (s, (elem: Int) => !pertenece(elem, p))

   si existe un elemento del conjunto s para el cual p se satisface dentro de los límites

6. **Map**
   Sea s: Conj & f: Int => Int <->
   Se define una función secundaria la cual ingresa como parámetro un a1:Int y se llama a la función existe <->
   s: Conj , (a2 : Int) => f (a2) == a1

   Es decir, lo que se ingrese de f se aplicará al conjunto s

CASOS DE PRUEBA

1. Se definen las siguientes variables

```
val s1 = conjuntoUnitario( elem = 1)
val s2 = conjuntoUnitario( elem = 2)
val s3 = conjuntoUnitario( elem = 3)
val s4 = conjuntoUnitario( elem = 4)
val s5 = conjuntoUnitario( elem = 5)
val s6 = union (s1 , s3)
val s7 = union (s2 , s3)
val s8 = union (s3 , s4)
val s9 = union (s4 , s5)
val s10 = union (s3, s5)
val s11 = interseccion(s6,s7)
val s12 = interseccion(s7,s8)
val s13 = interseccion(s8,s9)
val s14 = interseccion(s9,s10)
val s15 = interseccion(s10,s6)
val s16 = dif(s6,s7)
val s17 = dif(s7,s8)
val s18 = dif(s8,s9)
val s19 = dif(s9,s10)
val s20 = dif(s10,s6)
```

Las cuales servirán para usar en los demás casos de prueba ya que todas las funciones están correlacionadas

```
conjComoCadena(s1)
conjComoCadena(s2)
conjComoCadena(s3)
conjComoCadena(s4)
conjComoCadena(s5)
conjComoCadena(s6)
conjComoCadena(s7)
conjComoCadena(s8)
conjComoCadena(s9)
conjComoCadena(s10)
conjComoCadena(s11)
conjComoCadena(s12)
conjComoCadena(s13)
conjComoCadena(s14)
conjComoCadena(s15)
conjComoCadena(s16)
conjComoCadena(s17)
conjComoCadena(s18)
conjComoCadena(s19)
conjComoCadena(s20)
```

En esta parte de los casos de prueba se ponen la variables creadas dentro de la función conjComoCadena la cual fue creada de la siguiente forma

```
def conjComoCadena( s : Conj): String = {
  val xs = for (i <- 0 ≤ to ≤ 10 if pertenece(i,s)) yield i
  xs.mkString("{",",","}")
}
```

Funciones básicas sobre conjuntos

1. **Conjunto unitario**

```
pertenece ( elem = 1, s1)
assert(pertenece ( elem = 1, s1), "Conjunto_Unitario")
```

```
val res20: Boolean = true
```

En este caso si verifica si el elemento se encuentra en el conjunto por lo que es *true* ya si se pone otro número no correspondiente al conjunto arroja *false*

2. **Unión, intersección y diferenciación**

- **Unión**

```
// Prueba de uniones
pertenece(  elem = 1 , s6 )
pertenece(  elem = 2 , s6 )
!pertenece(  elem = 3 , s7 )
pertenece(  elem = 1 , s7 )
pertenece(  elem = 3 , s8 )
!pertenece(  elem = 2 , s8 )
pertenece(  elem = 2 , s9 )
pertenece(  elem = 3 , s9 )
!pertenece(  elem = 1 , s10 )
```

```
val res22: Boolean = true
val res23: Boolean = false
val res24: Boolean = false
val res25: Boolean = false
val res26: Boolean = true
val res27: Boolean = true
val res28: Boolean = false
val res29: Boolean = false
val res30: Boolean = true
```

Se presentan una serie de pruebas y como ejemplo para argumentar que esta correcto el código vamos a tomar la unión s8 que es unir s3 y s4, s8: {3, 4} pertenece (3, s8) <-> true ya que 3 se encuentra en la unión

- **Intersección**

```
// Prueba de intersecciones
pertenece( elem = 1 , s11 )
!pertenece( elem = 2 , s11 )
!pertenece( elem = 3 , s12 )
pertenece( elem = 2 , s12 )
!pertenece( elem = 1 , s13 )
pertenece( elem = 3 , s13 )
!pertenece( elem = 3 , s14 )
pertenece( elem = 3 , s15 )
!pertenece( elem = 1 , s15 )
```

```
val res31: Boolean = false
val res32: Boolean = true
val res33: Boolean = false
val res34: Boolean = false
val res35: Boolean = true
val res36: Boolean = false
val res37: Boolean = true
val res38: Boolean = true
val res39: Boolean = true
```

En este caso como en el de unión se debe coger el valor por la uniones, para este ejemplo se escogerá s13 para la demostración, s3 = intersección (s8, s9), entonces s8 sabemos que es {3, 4} y s9 es {4,5}, entonces pertenece(3, s13) como la intersección entre s8 y s9 es {4} por ende esto es *falso*, si fuera pertenece(4, s13) la respuesta sería *true*

- **Diferenciación**

```
// Prueba de diferenciacion
pertenece( elem = 1 , s16 )
!pertenece( elem = 2 , s17 )
!pertenece( elem = 3 , s18 )
pertenece( elem = 2 , s19 )
!pertenece( elem = 1 , s20 )
pertenece( elem = 3 , s16 )
!pertenece( elem = 3 , s17 )
pertenece( elem = 3 , s18 )
!pertenece( elem = 1 , s20 )
```

```
val res40: Boolean = true
val res41: Boolean = false
val res42: Boolean = false
val res43: Boolean = false
val res44: Boolean = true
val res45: Boolean = false
val res46: Boolean = true
val res47: Boolean = true
val res48: Boolean = true
```

Lo mismo sucede en este caso pero en forma de diferencia, vamos a coger entonces s20 que es la diferencia entre s10 y s6, s10 = unión (s3, s5) = {3, 5} y s6 = unión (s1, s3) = {1, 3}, la diferencia entre estos dos es igual a {5}, en este caso tenemos un no pertenece = !pertenece (1, s20) por lo que es true ya que el único elemento entre la diferencia de s10 y s6 es {5}

3. **Filtrar**

```
/**
 * Prueba de filtros
 */
val s21 = filtrar ( union( s6 , s7 ) , ( x : Int ) => ( x % 2) == 0)
conjComoCadena ( s21 )
!pertenece( elem = 1 , s21 )
!pertenece( elem = 3 , s21 )
pertenece( elem = 2 , s21 )
```

```
val s21: ConjuntosF.Conj = ConjuntosF.package$<function>
val res49: String = {2}
val res50: Boolean = true
val res51: Boolean = true
val res52: Boolean = true
```

Como se puede ver en el caso de prueba se debe filtrar
s6 = unión (s1, s3) = {1, 3} y s7 = unión (s2, s3) = {2, 3} que es {2} ahora ese conjunto se evalúa en si pertenece o no y la única forma en la que que acierta en true es cuando pertenece (2, s21)

Consultas y transformaciones de conjuntos

4. Para todo y existe

```
/**
 * Prueba de para_todo y existe
 */
!paratodo( union( s6 , s7 ) , ( x : Int ) => ( x % 2) == 0 )
existe( union( s6 , s7 ) , ( x : Int ) => ( x % 2) == 0 )
paratodo( s7 , ( x : Int ) => ( x % 2) == 1 )
```

```
val res53: Boolean = true
val res54: Boolean = true
val res55: Boolean = false
```

La función para todo y existe depende de un conjunto que se presente en relación de una función y un límite x dado.

paratodo (s7, (x: Int) => (x % 2) == 1)

Es falso porque p es diferente de los elementos del conjunto

5. Map

```
/**
 * Prueba de map
 */
val s22 = map( union ( s4 , s5 ) , ( x : Int ) => x * x )
conjComoCadena ( s22 )
!pertenece( elem = 2 , s22 )
!pertenece( elem = 3 , s22 )
!pertenece( elem = 5 , s22 )
!pertenece( elem = 6 , s22 )
!pertenece( elem = 7 , s22 )
!pertenece( elem = 8 , s22 )
pertenece( elem = 1 , s22 )
pertenece( elem = 4 , s22 )
pertenece( elem = 9 , s22 )
```

```
val s22: ConjuntosF.Conj = ConjuntosF.package$<function>
val res56: String = {}
val res57: Boolean = true
val res58: Boolean = true
val res59: Boolean = true
val res60: Boolean = true
val res61: Boolean = true
val res62: Boolean = true
val res63: Boolean = false
val res64: Boolean = false
val res65: Boolean = false
```

Lo que hace el map es transformar los elementos dependiendo del segundo parámetro, ya que multiplicado al cuadrado cada término del conjunto nos da vacio, entonces todos los !pertenece dan *true* y los pertenece *false*