
Paradigmas de Programación

(TB025) Curso Cano

# Trabajo Práctico 2
# Intérprete de Calculo Lambda no tipado

7 de junio de 2024


| Thiago Pacheco | Matias Bartellone | Ivan Maximoff | Andrea Figueroa |
|---|---|---|---|
| 111003 | 110484 | 110868 | 110450 |

# 1. Objetivo del trabajo

El presente trabajo práctico tiene como objetivos aplicar los conocimientos adquiridos en la cursada respecto a cálculo lambda no tipado mediante la implementación de un intérprete de expresiones lambda programado dentro del paradigma funcional, procurando respetar las buenas prácticas de programación que se nos enseñaron.

De manera que al implementar el intérprete se deben demostrar y plasmar conocimientos relativos a:

- Expresiones lambda, reducciones beta, transformaciones alpha, variables libres de una expresión y árboles de sintaxis abstracta.
- Funciones como bloques de construcción básicos del programa, composición de funciones, inmutabilidad de los datos, funciones puras e impuras, entre otros relativos al paradigma funcional.

# 2. Supuestos

Se asume que la entrada al interprete son únicamente expresiones lambda válidas, las cuales están conformadas por los símbolos: "$\lambda$","$\_$", ".", "(", ")", además de los strings a elección que representan los nombres de las variables, dichos símbolos se asume están en el formato indicado por la cátedra de manera que se representen variables, abstracciones o aplicaciones válidas.

Por lo mencionado en el anterior supuesto nos tomamos la libertad de omitir alguna implementación relativa a validaciones a las expresiones ingresadas, lo que tiene implicancias en el código como que algunos pattern matchings en el código no se hayan implementado exhaustivos.

Además se asumió que la estrategia de reducción por defecto, es call-by-name, lo cual no modifica que el intérprete soporta todas las estrategia de reducción enseñadas por la cátedra. El supuesto más que nada es precuatorio ya que es de nuestro conocimiento que dentro de las expresiones que puede recibir nuestro intérprete se pueden incluir expresiones que no tienen forma normal, es decir que no se pueden normalizar mediante reducciones beta finitas, al ser la estrategia call by name la estrategia por default, garantizamos que el intérprete en su estado por defecto realice las reducciones de una expresión y de poderse evitar, no se entre en bucles infinitos.

# 3. Consideraciones

A continuación exponemos algunas consideraciones que surgieron en el desarrollo del trabajo práctico:

- La primera apreciación que resultó evidente, es que la implementación de un programa netamente funcional no es usual y aún más importante en nuestro caso no resulta asequible, ya que para que imlementar un intérprete funcional resulta **imprescindible la interacción con el exterior**, pues es necesario el uso de funciones impuras como *scala.io.StdIn.readLine()* para leer la entrada estándar y mostrar en pantalla el resultado. Sin embargo, dado que uno de los principales objetivos del trabajo práctico es llevar a cabo el diseño haciendo uso de los conceptos de programación funcional, tomamos como directriz preponderante el **aislamiento** de aquellas **funcionalidades** que **choquen** con lo netamente **funcional**.
- Además, si bien se indicó que el parser sería el modulo que se encargase de convertir un AST (implementado como un arbol conformado por Expresiones) a su representación a Strings, se decidió que el módulo de parser se aboque a desglosar las expresiones en los datos que ya venia manejando, es decir tokens y strings. Por lo que, de manera semejante a la funcionalidad que ya tenia el módulo del lexer (interpretar strings y convetir los operadores a tokens) se encomendó a este módulo el camino de vuelta, lo que es, traducir listas de los tokens y strings a un único string que denote el resultado final que se mostrará por pantalla.

# 4. Diseño

El programa se diseñó manteniendo la estructura general presentada durante las clases, dividiendo a grandes rasgos el programa en 3 componentes escenciales: Lexer, Parser y Reductor, dichas componentes estan constituidas por funciones concretas, de manera que, mediante la composicion de estas, se implementa el intérprete.

# 5. Detalles de implementación

## Generales:

### Lectura de input

Como mencionamos previamente, uno de nuestros objetivos planteados para la implementación del programa fue el de aislar aquellos "comportamientos" que no estén contemplados dentro del paradigma funcional que el mismo requiera, esta especificiacion se menciona en el presente informe ya que consideramos importante explicar cómo manejamos la situación concerniente a la lectura de la entrada al programa considerando lo planteado. Para leer la entra entrada estandar, inicialmente se buscó aislar el contacto con el exterior (entrada de input) a unicamente el main, de manera que las otras funciones que se usan en este no dependan ni tengan la necesidad de usar directamente estas funciones impuras. Sin embargo, dado que dentro del paradigma funcional el uso de estructuras de control de flujo no tiene cabida ya que estas dependen de variables de control que se modifican en cada iteración y justamente el paradigma se basa fuertemente en la inmutabilidad de los valores, la forma implementadapara seguir recibiendo la entrada estándar se tuvo que pensar de manera recursiva, por lo que, en vista de que el puente de comunicación entre lo obtenido en una llamada recursiva a otra llamada recursiva es el uso de los parámetros, nos vimos en la necesidad de delegar el uso de la función *readLine* a una función leerInput, de manera que al llamar recursivamente para seguir leyendo la entrada, podemos contar y conservar la información relevante de la anterior llamada recursiva (el seteo de cómo y qué se quiere obtener de las siguientes expresiones lambda que se reciban) mediante los parámetros (los cuales evidentemente el main no podría recibir). Lo mencionado se reflejó en el código de la siguiente manera:

```
def main(args: Array[String]): Unit = {
        leerInput(callByName)
}
def leerInput(reductor : Expresion => Expresion): Unit = {
    val input = scala.io.StdIn.readLine()
    ...
```

De manera análoga se manejó el uso de las funciones para mostrar los resultados de la salida del programa.

Cabe recalcar que, dado que las funcionalidades requeridas para que el intérprete se comunique con el exterior ya se expusieron y se indicaron a dónde se aislaron, los módulos que se expondrán a continuación y las funciones que conforman estos sí respetan el concepto de "funciones puras".

## Concernientes al modelo

- Para modelar los **operadores** que se contemplan en las expresiones lambda, las cuales inicialmente determinan cómo el intérprete manejará la expresión, usamos enums. Dentro de estos operadores están: *LAMBDA, PUNTO, PAREN_IZQ, PAREN_DER, ESPACIO*, que vinvulan lo que se entenderán como abstracciones o aplicaciones según sea el caso.
- Para representar las tres definiciones recursivas que componen lo que es una **expresión lambda**, definimos estas mediante clases especiales, *case class*, que Scala nos proporciona, mismas que nos permiten usar un concepto clave en la programación funcional que es el pattern matching, además de compatibilizar con la característica de la inmutabilidad de los

datos. Con lo que, dentro del modelo de las expresiones lambda nos manejamos con las siguientes case clases: $Variable(nombre : String)$, $Abstraccion(variable : Variable, cuerpo : Expresion)$, $Aplicacion(funcion : Expresion, argumento : Expresion)$

- Además, dado que el intérprete también ha de poder manjear una expresión para obtener sus variables libres se decidió incluir la representacion de estas dentro del modelo de Expresiones a las variables libres mediante una case class $VarLibres(variables : Set[String])$ de manera que también se identifique como una Expresión, esto para que empalme con el tipado $Expresion => Expresion$ que cumplen las funciones que se encargan de obtener, de una expresión lambda, lo que se espera por salida. (Véase más al respecto de estas funciones en la seccion de detalles de implementacion concernientes a la reducción de expresiones lambda)

## Concernientes al Lexer

El módulo lexer abarca lo relativo a la conversión de la entrada estándar recibida como un string, a los valores que el intérprete sabe cómo interpretar (valga la redundancia) para aplicar reduciones beta o de ser el caso obtener las variables libres. Esto mismo es llevado a cabo por la función presente en el módulo mencionado:

```
def leerEcuacion(ecuacion: String): List[Operador | String]
```

En adición, como ya mencionamos en la sección *Consideraciones*, el lexer se encarga de lo análogo al camino de vuelta de la anterior funcionalidad, es decir, que transforma una secuencia de tokens y strings en un valor que unifica estos en un string. En el código esta funcionalidad es llevada a cabo por la función:

```
def leerTokens(tokens: List[Operador | String]): String
```

Estas mismas funcionalidades trabajan en composición con otras funciones determinísticas que cargan con la lógica de la identificación respectiva de tokens con strings y viceversa.

## Concernientes al Parser

El módulo *Parser* se aboca a lo relativo al paso de la representacion de una expresión como una Lista de tokens y strings a la representación de la misma en un arbol de sintaxis abstracta implementada como un arbol de expresiones, en la que finalmente las variables serían las análogas a las hojas del árbol y los tokens son los que determinan cómo se va armando el arbol. Así mismo también cuenta con la funcionalidad para el camino de vuelta, es decir, dado un Arbol de Sintaxis Abstracta se devuele la representación de la expresión lambda como una lista de tokens y strings.

## Concernientes a la reducción de expresiones

En general el módulo contiene la implementacion de las funcionalidades que el intérprete soporta para obtener algún valor en base a un AST de una expresión lambda, las cuales son: obtener el resultado de aplicar reducciones beta y obtener las variables libres de dicha expresión. Las funcionalidades concretas que se mencionaron son implementadas mediante las funciones $callByName$, $callByValue$ y $freeVariables$, estas cumplen con el tipado $Expresion => Expresion$ y todas son consideradas como reductores. Con lo que, recibiendo por parámetro la función que obtenga lo que se desea de la expresion lambda y la expresión misma, el reductor se aboca a devolver el resultado de aplicarle al reductor de tipo $Expresion => Expresion$ el argumento $Expresion$.

## 5.1. Interfaz

El intérprete funciona mediante una interfaz de linea de comando, dichos comandos pueden representar las expresiones lambda a reducir, así como la configuración de qué se quiere obtener de

las expresiones lambda que se vayan a ingresar mediante el uso de la palabra “set” y la configuración a elegir.

- **“Set call-by-name”**: Establece la configuración de que lo que se quiere obtener de las expresiones lambda que se fueran a ingresar en adelante es el resultado de aplicarle reducciones beta a las mismas mediante la estrategia call-by-name.
- **“Set call-by-value”**: Establece la configuración de que lo que se quiere obtener de las expresiones lambda que se fueran a ingresar en adelante es el resultado de aplicarle reducciones beta a las mismas mediante la estrategia call-by-value.
- **“Set free-variables”**: Establece la configuración de que lo que se quiere obtener de las expresiones lambda que se fueran a ingresar es el resultado de la recopilación de las variables libres de las expresiones lambda que se fuesen a ingresar en adelante.
- **Dada una expresion lambda** Se mostrará en pantalla el resultado de la aplicación de la expresión a la función que se determinó previamente respectiva a qué se quiere obtener de las siguientes expresiones recibidas, de no haberse especificado por defecto se muestra el resultado de las reducciones beta aplicadas con la estrategia call by name.

# 6. Conclusiones

En conlcusión, si bien el desarrollar un programa orientado al paradigma funcional resultó algo nuevo para todos los integrantes del grupo, ya que no es frecuente su uso en contextos tradicionales y habituales de programación, una vez comprendidos los conceptos relativos al cálculo lambda resulta más asimilable el enfoque y protagonismo que el paradigma busca darle a las funciones, así como también la importancia y beneficios de la inmutabilidad de los datos.

Cabe mencionar que las características y conceptos que el paradigma trae a colación resultaron particularmente útiles para contextos como lo es la implementación de un intérprete, pues en este, dada una entrada, se busca obtener un mismo valor de forma determinística, además la funcionalidad global del programa se puede conseguir mediante la composicion de funciones.