INFORME TRABAJO PRÁCTICO 2
PARADIGMAS DE PROGRAMACIÓN
CURSO CANO, RAIK, BRASBURG

Integrantes: **Nicolás Khalil Chaia (110768),**
Gonzalo Nicolas Crudo (110816),
Nicolás Daniel Graeff Dieguez (110304),
Facundo Nazareno Lescano (110784)

FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

INTRODUCCIÓN

El trabajo consiste en realizar un intérprete de Cálculo Lambda no tipado utilizando el paradigma funcional. El mismo debe poder tener las siguientes funcionalidades:

- Poder reducir expresiones dadas ya sea mediante la estrategia de "*Call by name*" o "*Call by value*".
- Debe realizar las conversiones alpha que sean necesarias.
- Devolver el árbol de sintaxis abstracta, *AST*, de la expresión.
- Encontrar las variables libres y ligadas

El programa se dividió en 3 módulos, cada uno con su propósito y de suma importancia para el funcionamiento dado que están relacionados entre sí. Estos son:

- Lexer
- Parser
- Reductor

LEXER

El "*Lexer*" es el primer módulo que fue desarrollado, el mismo consiste en poder devolver una serie de tokens dada una expresión (*tokenizar*). Cada token hereda de una *sealed trait* llamada *CalculoLambda*
Cada elemento de una expresión lambda fue representado con un token, estos elementos pueden ser:

- Variables: Son un símbolo que representa un valor o una expresión. Las variables en el cálculo lambda son fundamentales porque permiten la abstracción y la aplicación de funciones.
- Paréntesis: Se utilizan para agrupar expresiones y definir el orden de evaluación. En el código se hizo una distinción entre paréntesis derechos e izquierdos
- Espacios: Se utilizan para representar una aplicación de dos cosas en una expresión.
- Punto: Se utilizan para separar el parámetro de la función de su cuerpo.
- Funciones: Comienzan con un λ de una variable que representa el parámetro (argumento) y de un punto junto con el propio cuerpo de la misma.
- Símbolo λ: Representa una abstracción de función.
- Aplicaciones: Es el proceso de evaluar en una función a un argumento.
- Vacío: TBD

Una vez definidos los elementos posibles en una expresión, en la siguiente tabla se puede encontrar cada elemento y su respectivo token:

Elemento	Token
Variables	VAR(name: String)
Paréntesis derecho	RPAR()
Paréntesis Izquierdo	LPAR()
Espacios	SPACE()
Puntos	DOT()
Funciones	LAMBDA(arg: String, body: CalculoLambda)
Símbolo Lambda	LAMBDA str()
Aplicaciones	APP(f: Cálculo Lambda, v: Cálculo Lambda)
Vacío	NIL()

Dada una expresión el Lexer va a descomponer cada elemento en tokens y devolver una lista con los mismos.

PARSER

Este módulo recibe una secuencia de tokens y, mediante ciertas reglas indicadas en la consigna, devuelve el árbol de sintaxis abstracta de la expresión.
Este AST será utilizado por el reductor para realizar todas las tareas que pida el usuario. Además el parser tendrá la sub-tarea de devolver las expresiones finales ("*desparsear*") como también devolver la expresión en caso de recibir un AST.

REDUCTOR

El reductor es el encargado de, valga la redundancia, reducir una expresión. Esto en principio no parece algo muy complicado pero es importante tener en cuenta ciertas consideraciones. La principal es darse cuenta de que puede darse un caso en el cual haya conflictos de variables, dos variables distintas con el mismo nombre donde una sea libre y la otra ligada. Es por esto que el reductor cuenta con una serie de funciones que permite realizar la conversión alpha para poder desambiguar estos casos, lo cual significó también tener que desarrollar un método para poder identificar las variables libres y ligadas de la expresión.

Una vez realizada la conversión alpha se puede ya comenzar a hablar de lo que sería propiamente la reducción de una expresión lambda.
Tenemos dos caminos:

- Estrategia *Call by Name* (CBN): Los argumentos se pasan textualmente sin evaluación previa. Se evalúan sólo cuando se utilizan en el cuerpo de la función.
- Estrategia *Call by Value* (CBV): Los argumentos se evalúan antes de pasarlos a la función. Cabe destacar que en esta estrategia puede pasar que tengamos ciclos, lo cual puede significar que una expresión que si tenía una forma reducida, la llamamos *redex,* no se pueda encontrar dado que estamos infinitamente reduciendo algo que siempre queda igual. Por otro lado, usando la primera estrategia este problema no ocurre.

El programa por defecto está configurado para reducir usando CBN aunque si se escribe como entrada "*set call-by-value*" el método de reducción será el de CBV.
Una vez seleccionada la estrategia y dada la expresión, el programa la devuelve reducida.

CONCLUSIONES:

Finalmente podemos obtener varias conclusiones, algunas relacionadas con el paradigma funcional y otras con el trabajo en sí.

La primera con respecto al trabajo es que, si bien no consta de un código muy complejo, arrancarlo desde cero y comenzar a modelar la idea fue complicado. Sumado también a una serie de casos bordes que surgieron en todo el desarrollo de las partes encargadas de la reducción y de la conversión de variables. Esto nos permitió tanto profundizar como demostrar los conceptos del cálculo lambda aprendidos en clase.

Luego con respecto al paradigma funcional, podemos destacar que se trata efectivamente de cambiar esa forma de programar, esencialmente el aspecto de tener variables inmutables, que eso tambien significa una ventaja a la vez pues la ausencia de efectos secundarios reduce la probabilidad de cometer errores. Tambien pensar al codigo pura y exclusivamente como funciones tiene cierta dificultad en un comienzo porque pueden surgir ideas intuitivas como la de usar objetos, que no siguen al paradigma funcional.

Por ultimo queremos destacar lo interesante que fue haber hecho el trabajo ya que, como ocurrió con el trabajo de POO, descubrimos una nueva faceta de la programacion que no conociamos y pudimos plasmarla en un proyecto que combina habilidades tanto para programar de manera funcional como en el apartado de Calculo Lambda.