TP1: MonaLambda

Objetivos

- Analizar el problema planteado correctamente.
- Diseñar una solución utilizando correctamente los conceptos de **programación funcional**.
- Demostrar el conocimiento de los conceptos de Cálculo Lambda.
- Seguir las buenas prácticas de programación.

Indicaciones del trabajo

Grupos de trabajo

El trabajo se desarrollará de forma grupal, en grupos de máximo 4 alumnos. El trabajo práctico debe realizarse utilizando Git y Github como herramienta de colaboración.
Todos los alumnos integrantes del grupo **deben** participar activamente del desarrollo del trabajo de forma equitativa. Se espera que la participación sea equitativa, queda a criterio de cada grupo la forma de trabajo y organización.

Tecnología

El trabajo debe realizarse en *Scala 3*, utilizando cualquier versión estable de esta versión. Para comenzar el trabajo se debe clonar este repositorio y programar la solución siguiendo la estructura de los archivos. Seguir el *README* del repositorio para poder correr las pruebas provistas por la cátedra. Los archivos que ya se encuentran en el repositorio son mínimos, se espera que cada grupo agregue las pruebas que considere necesario.

Tiempo de trabajo

Se contarán con aproximadamente 4 semanas completas para llevar a cabo el desarrollo del TP. Transcurrido el tiempo previsto para la realización del trabajo práctico debe realizarse la **entrega**.

Modalidad de entrega

Para realizar la entrega se debe:

- Github:
  - El repositorio tiene que ser privado y debe estar su corrector invitado al mismo.
  - Para realizar la entrega se debe crear un branch llamado `tp-1`, con el código correspondiente a la entrega. No se debe modificar más luego de la fecha de entrega.

    - Se debe contar con un `README.md` que explique cómo correr el programa "desde cero", cómo instalar las dependencias y otras explicaciones pertinentes. El código debe poder correrse con un comando que compile el código y lo ejecute, **no es válido subir un ejecutable y que el comando simplemente lo corra**. Las instrucciones deben ser genéricas, no pueden depender de un IDE puntual.
- Mail: Mandar un mail a algoritmos3.fiuba@gmail.com indicando número de grupo y sus integrantes, aclarando cuál es la branch y el repositorio de la entrega. El mail debe llevar adjunto el informe en formato PDF.

Fecha de entrega

La entrega definitiva, que debe alinearse a lo especificado en la Modalidad de entrega, debe realizarse con anterioridad a las 23:59hs del 05/05/2025.

Enunciado: Intérprete de Cálculo Lambda

En esta ocasión se pide implementar un intérprete de Cálculo Lambda no tipado **usando exclusivamente el paradigma funcional**. Un intérprete es un programa que permite leer y ejecutar código ingresado por el usuario. Dada la complejidad que puede llegar a tener su implementación, se pide estructurar el código usando al menos los siguientes componentes:

Lexer	El Lexer recibe un string que representa una expresión lambda y devuelve una secuencia de tokens
Parser	El Parser recibe una secuencia de tokens y devuelve el Árbol de Sintaxis Abstracta (*Abstract Syntax Tree - AST*). Además, debe poder recibir un AST y devolver la representación como string de esa expresión lambda.
Reductor	El Reductor recibe un AST que representa una expresión lambda, y devuelve el AST resultante de reducir la expresión recibida. Además, debe poder recibir un AST y devolver todas las variables libres.

Procesamiento de la entrada

Lexer

El Lexer, o Tokenizador, es un módulo que recibe una secuencia de caracteres que representan una expresión lambda, y los convierte en una secuencia de Tokens.

Un ejemplo del input que puede recibir el Lexer es el siguiente:

```
λx.(λy.(y)) (λx.(x x) λx.(x x))
```

Y deberá poder interpretar cada uno de los símbolos posibles que pueden llegar a conformar una expresión lambda, y representarlos usando Tokens. Los caracteres que se deben poder soportar, son:

"λ"	El símbolo lambda representa el comienzo de una abstracción
" "	El espacio permite separar el argumento de la función en una aplicación
"."	El punto permite separar el argumento del cuerpo de una abstracción
"("	El paréntesis izquierdo permite agrupar subexpresiones
")"	El paréntesis derecho permite representar el final de una subexpresión
string	Cualquier otro string distinto de los anteriores, debe ser interpretado como una variable

Parser

El Parser es un módulo que recibe una secuencia de tokens, realiza un análisis de los tokens recibidos y devuelve el Árbol de Sintaxis Abstracta (Abstract Syntax Tree - AST). Durante el análisis de los tokens, el Parser debe seguir una serie de reglas establecidas por la gramática que define las expresiones lambdas.

```
<λexp> ::=  | <var>                                  #Variable
            | <LAMBDA> <var> <DOT> <λexp>            #Abstracción
            | <LPAR> <λexp> <SPACE> <λexp> <RPAR>    #Aplicación
```

La gramática anterior nos indica que las expresiones lambda pueden ser de tres tipos:

- Una variable, representada por un string.
    - Puede ser de una o múltiples letras.
- Una abstracción, compuesta por una variable que representa el argumento de la función, y otra expresión lambda que representa el cuerpo de la función.
- Una aplicación, compuesta por una expresión lambda que representa la función a ser llamada, y otra expresión lambda que representa el argumento pasado a la función.

Para facilitar la gramática, el intérprete puede suponer que todas las abstracciones lambda tendrán su cuerpo encerrado entre paréntesis. Es decir, expresiones como:

```
(λx.λy.y x) a
```

No serían procesadas, a menos que se implementen correctamente las reglas de precedencia y asociatividad de las abstracciones. Por eso dicha expresión, en la gramática simplificada que se menciona, se recibiría de la siguiente manera:

```
(λx.(λy.(y x))) a
```

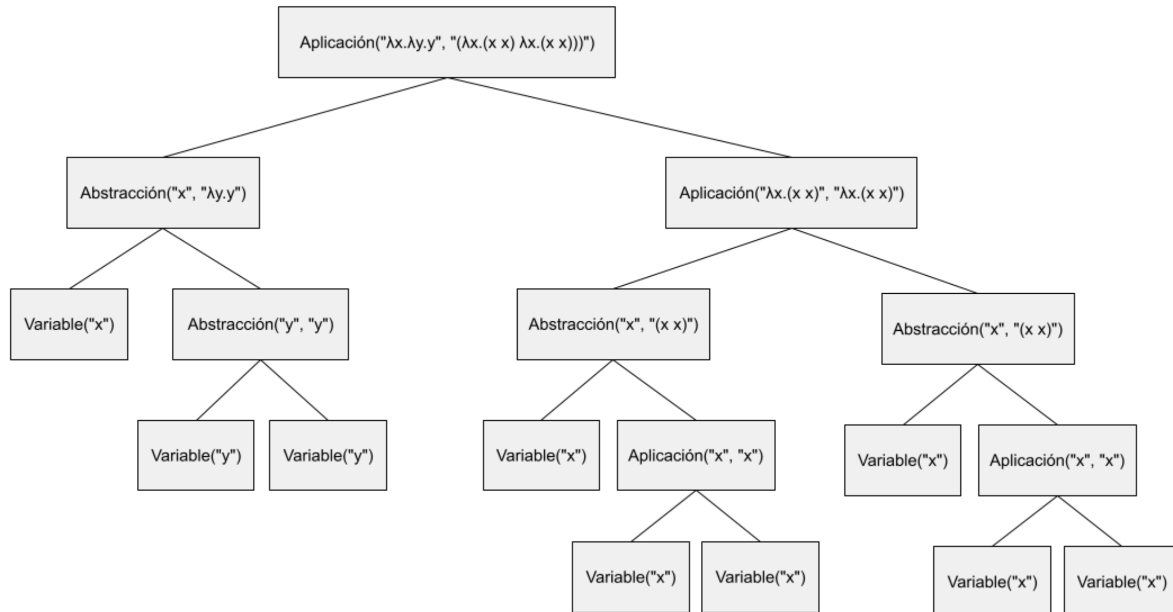
Sin embargo, las reglas de asociatividad de las aplicaciones **sí** hay que contemplarlas. Eso significa que estas primeras 2 expresiones deban ser equivalentes, mientras que la 3ra no, pues contiene paréntesis que explícitamente agrupan a 2 variables:

```
λx.(λy.(y x x))    → paréntesis implícitos por regla asociativa
λx.(λy.((y x) x))  → paréntesis explícitos

λx.(λy.(y (x x)))
```

Con dicho concepto de la gramática, el ejemplo usado en la sección de Lexer tendría un AST correspondiente con la siguiente forma:

```
λx.(λy.(y)) (λx.(x x) λx.(x x))
```



Además de la funcionalidad principal descrita anteriormente, el Parser debe poder recibir un AST que represente una expresión lambda y devolver la representación en string de esa expresión lambda.

Operaciones

Procesador de variables

El procesador de variables debe ser capaz de tomar un AST que representa una expresión lambda y retornar el conjunto de sus variables libres y de variables ligadas.
Para la expresión:

```
(λx.(λy.(y))) a b
```

Debe retornar:

```
Variables libres: {a,b}
Variables ligadas: {x,y}
```

Nota: Puede haber casos donde una variable esté en ambos conjuntos, por temas de nombres de variables ligadas.

Reductor

El Reductor es un módulo que recibe un AST que representa una expresión lambda, y devuelve el AST resultante de reducir la expresión recibida. Para ello el Reductor debe aplicar los métodos de conversión-α y reducción-□ para normalizar la expresión lambda a la mínima expresión.

A la hora de realizar la reducción, el Reductor debe soportar las siguientes estrategias de reducción:

- Orden Normal ("Normal Order") o "Call-by-name": Esta estrategia de reducción consiste en siempre reducir la aplicación más a la izquierda/más externa posible, reduciendo los argumentos lo más tarde posible.
- Orden de Aplicación ("Applicative Order") o "Call-by-value": Esta estrategia de reducción consiste en siempre ejecutar la aplicación más a la derecha/más interna posible, reduciendo los argumentos lo antes posible.

Ambas estrategias deberán reducir la expresión lambda a su mínima expresión, de ser posible. La primera estrategia, en caso de ser posible, siempre encontrará la normalización. Mientras que la segunda estrategia, no necesariamente encontrará la expresión mínima. Para el ejemplo de las secciones anteriores,

```
λx.(λy.(y)) (λx.(x x) λx.(x x))
```

En el caso de la estrategia "Call-by-name", el resultado debería ser:

```
λy.(y)
```

Pero en el caso de la estrategia "Call-by-value" debería generar una recursión infinita debido al ciclo infinito causado por `λx.(x x) λx.(x x)`.

NOTA: de realizarse una conversión-α, se espera que la nueva variable tenga el mismo nombre que antes, pero sumándole un apóstrofe (`'`).

Etiquetas de expresiones

Para poder facilitar la reutilización de ciertas expresiones y así construir expresiones más grandes, debe ser posibles definir **etiquetas** a las que le puedo asignar una subexpresión, para luego ser utilizada en otras expresiones utilizando un signo `!`.

Un ejemplo de esto puede ser el siguiente:

```
IDENTIDAD=λx.(x)
APLICAR=(λf.(λx.(f x)))
MAIN=!APLICAR !IDENTIDAD a
```

Dichas etiquetas, de existir, se reemplazarán por su contenido, dando lugar a que `MAIN` se entienda como:

```
MAIN=(λf.(λx.(f x))) λx.(x) a
```

Nota: Las etiquetas pueden utilizarse cada vez que se quiera.

Al definir cada etiqueta, se debe leer y generar el AST de la expresión que se le asigna, para asegurarse de que sea una expresión válida. De no serlo, el programa debe lanzar un error.

No se pueden redefinir etiquetas. De volver a definir una con el mismo nombre que otra anterior, el programa debe lanzar un error indicando que se repitieron nombres de etiqueta.

Desarrollo e Implementación

Interfaz de usuario

Al correr el intérprete, este comenzará a leer líneas de la entrada estándar. En ese momento, el usuario puede realizar cualquiera de las siguientes acciones:

- **Definir una etiqueta**:
  - Se realiza con la nomenclatura expresada en la sección de etiquetas
  - Ejemplo:
    - `ETIQUETA=λy.(y)`
- **Calcular variables libres y ligadas de una expresión**:
  - Se hace con la palabra clave `calc_vars`, seguido de una expresión lambda o una etiqueta
  - Ejemplo:
    - `calc_vars (λx.(λy.(y))) a b`
- **Reducir una expresión lambda:**
  - Se indica con la palabra clave `reduce`.
  - Posteriormente, se indica la estrategia de reducción a utilizar. Puede ser `cbv` o `cbn`, dependiendo si se quiere usar call-by-value o call-by-name, respectivamente.
  - Opcionalmente, se puede indicar la keyword `debug`, que debe mostrar el paso a paso de la reducción. Mostrando cómo queda la expresión luego de cada sustitución, renombre de argumentos, etc.
  - Finalmente, se indica la expresión lambda o etiqueta que contenga la expresión que se desee reducir.
  - Ejemplos:
    - `reduce cbv debug !ETIQUETA`
    - `reduce cbn (λx.(λy.(y)) (λx.(x x) λx.(x x)))`

Unas secuencia de instrucciones y resultados posibles puede ser las siguientes:

```
$> MI_EJEMPLO=(λx.(λy.(y)) (λx.(x x) λx.(x x)))
$> calc_vars !MI_EJEMPLO
Variables libres: {}
Variables ligadas: {x,y}
$> reduce cbn !MI_EJEMPLO
λy.(y)
```

```
$> IDZ=λz.(z)
$> calc_vars (λf.(f b)) (λx.(!IDZ) z) a
Variables libres: {b,z,a}
Variables ligadas: {f,x,z}
```

```
$> reduce cbv debug (λf.(f b)) (λx.(!IDZ) z) a
EXPR = (λf.(f b)) (λx.(λz.(z)) z) a
CONV[z -> z'] = (λf.(f b)) (λx.(λz'.(z')) z) a
SUST[x:=z] = (λf.(f b)) λz'.(z') a
SUST[f:=λz'.(z')] = (λz'.(z') b) a
SUST[z':=b] = b a
b a
```

Informe

Se debe entregar un informe detallando la solución y explicando las decisiones tomadas. El informe debe contener las hipótesis tomadas durante la realización de este trabajo práctico. No olvidar agregar una sección para las conclusiones.

Criterios de aprobación

Para que un TP se considere aprobado deberá al menos cumplir con los siguientes criterios:

- La implementación debe seguir los lineamientos del paradigma funcional.
- El intérprete deberá poder soportar variables de al menos un único carácter.
- El intérprete deberá poder devolver las variables libres y ligadas de una expresión.
- El intérprete deberá poder retornar el resultado de reducir expresiones utilizando al menos una de las dos estrategias pedidas (call-by-name o call-by-value)
- El intérprete deberá poder reducir cualquier expresión que no tenga conflictos de variables (es decir, que no requiera una conversión alpha en su desarrollo).

Si bien estas son las funcionalidades mínimas requeridas para la aprobación, un TP se considerará completo solo si cumple con todos los requerimientos descritos en las secciones anteriores.

Testing

Expresiones posibles:

`λx.(λy.(y)) (λx.(x x) λx.(x x))`	`λy.y (solo cbn)`
`(λx.(λy.(x))) y`	`λy'.y`
`(λf.(f λx.(λy.(x)))) (λx.(λy.(λf.(f x y))) a b)`	`a`
`(λx.(λx.(y x))) z`	`λx.(y x)`
`λx.(λy.(y) x)`	`λx.(x)`
`(λx.(x) y) (λy.(y) z)`	`y z`
`λy.(λx.(y y)) a (λx.(x x))`	`a a`
`λy.(λx.(y y)) a (λx.(x x) λx.(x x))`	`a a (solo cbn)`
`λx.(λy.(x)) (y a)`	`λy'.(y a)`