Intérprete de Cálculo Lambda

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

Diseño

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.

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 representar el comienzo de una aplicación
")"	El paréntesis derecho permite representar el final de una aplicación
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.
- 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.

*Nota: En la gramática provista, sólo se soportan paréntesis para agrupar una expresión lambda que representa una aplicación. Esta gramática difiere de la vista en clase, en la que podían ser usados para agrupar otro tipo de expresiones. Sin embargo, para facilitar la implementación del intérprete y evitar complejidades asociadas a las reglas de asociatividad y precedencia**, se deberá asumir que sólo se usarán paréntesis durante la aplicación.***

Respetando la gramática anterior, la siguiente expresión **no** sería válida. Ya que hay una aplicación sin paréntesis a su alrededor.

```
λx.λy.y x
```

Nuevamente, si bien esta gramática difiere de la vista en clase, permite facilitar la implementación y evitar implementar las reglas de asociatividad y precedencia. Por lo tanto, para que la expresión sea válida se deberán agregar paréntesis delimitando las expresiones lambda que componen la aplicación. En el siguiente ejemplo, la aplicación es **(y x)**:

```
λx.λy.(y x)
```

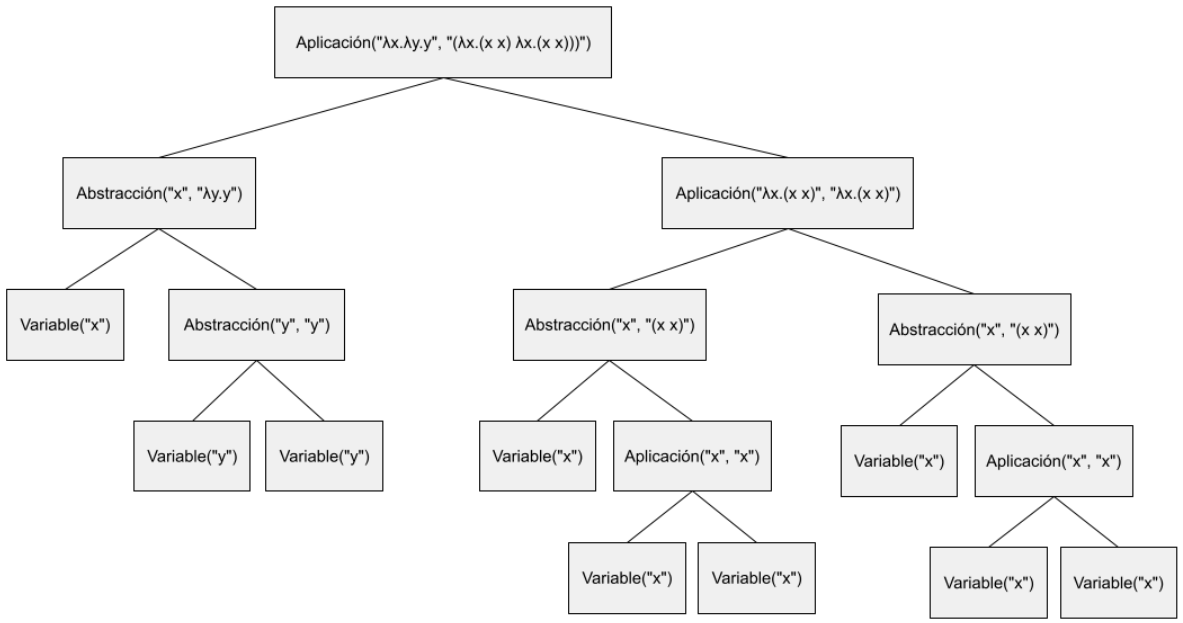
Sin embargo, en el siguiente ejemplo, la aplicación es **(λx.λy.y x)**:

```
(λx.λy.y x)
```

Siguiendo el ejemplo usado en la sección anterior,

```
(λx.λy.y (λx.(x x) λx.(x x)))
```

el AST correspondiente debería tener la forma:



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

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 sustitución-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 o 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 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)).

Además de la funcionalidad principal descripta anteriormente, el Reductor debe poder recibir un AST que represente una expresión lambda y devolver todas las variables libres de esa expresión.

Desarrollo e Implementación

Interfaz de usuario

El intérprete deberá esperar a que el usuario ingrese una expresión lambda y deberá mostrar el resultado de reducir la expresión lambda, luego deberá volver a pedir un input al usuario hasta que ingrese "exit".

```
$> (λx.λy.y (λx.(x x) λx.(x x)))
λy.y
$> exit
```

Por default el intérprete deberá usar la estrategia de reducción “Call-by-name”, pero se deberá poder modificar la estrategia usando el siguiente comando:

```
$> set call-by-value
```

Una vez modificada la estrategia de reducción, todas las siguientes reducciones usarán la estrategia “Call-by-value”, hasta que se vuelva a modificar la estrategia mediante:

```
$> set call-by-name
```

También se deberá poder pedirle al intérprete que devuelva todas las variables libres de una cierta expresión. Para eso se deberá modificar el modo usando el siguiente comando:

```
$> set free-variables
```

Una vez modificado el modo, al ingresar una expresión se deberán mostrar todas las variables libres. Por ejemplo:

```
$> (λf.λx.(y x) z)
{y,z}
```

Validaciones

No será necesaria la implementación de validaciones del input del usuario, es decir, se podrá asumir que el usuario solo ingresará los comandos mencionados anteriormente y las expresiones lambda ingresadas serán siempre válidas y respetarán la gramática detallada previamente.

Testing

El intérprete deberá soportar los siguientes casos de prueba:

	`call-by-name`	**`call-by-value`**
(λx.λy.y (λx.(x x) λx.(x x)))	λy.y	Recursión infinita
(λx.λy.x y)	λy*.y	λy*.y
(λf.(f λx.λy.x) ((λx.λy.λf.((f x) y) a) b))	a	a
(λx.λx.(y x) z)	λx.(y x)	λx.(y x)

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 las siguientes funcionalidades mínimas:

- El intérprete deberá poder soportar variables de un único carácter
- El intérprete deberá poder reducir expresiones usando una de las dos estrategias vistas (call-by-name o call-by-value)
- El intérprete deberá poder reducir expresiones que no tengan conflictos de variables
- El intérprete deberá poder devolver las variables libres de una expresión

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. Un TP incompleto, no será considerado para la promoción.

Además, para que un TP se considere aprobado, se deberán cumplir los siguientes requisitos:

- La implementación deberá respetar estrictamente el paradigma funcional.
- La implementación deberá contar al menos con los tres módulos detallados en la sección de “Diseño”.
- La implementación deberá respetar con la interfaz de usuario especificada en la sección de “Interfaz de usuario”.
- La implementación deberá pasar los casos de prueba especificados en la sección de “Testing” y los casos de prueba internos de los docentes.

Formato de la entrega

Para realizar la entrega se debe mandar un mail a algoritmos3.fiuba@gmail.com indicando los integrantes del grupo adjuntando el informe e indicando cual es el branch y el repositorio de la entrega. El repositorio tiene que ser privado. El trabajo práctico debe realizarse utilizando Git y Github. Para realizar la entrega se debe crear un branch llamado ”tp-x’ (en este caso tp-1) y no se debe modificar más luego de enviado el mail. Se recomienda utilizar un branch común de trabajo, por ejemplo ”main”, y crear el branch una vez que se decida hacer la entrega. Se debe dar permisos en el repositorio a los 3 docentes del curso. Se debe contar con un README que explique cómo correr el programa ”desde cero”, cómo instalar las dependencias y como correr el juego de forma clara y única. 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**.

Fecha de entrega

07/06/2024