**Proyecto #1 –LAMBDA**

**Trabajo grupal hasta 2 personas**

**Grupos:** No se permite que los grupos se copien entre sí. Cada individuo debe entender TODO el programa. Recomiendo que planifiquen juntos y trabajen juntos.

1. **Objetivo**

Practicar nuestro conocimiento de Racket con un ejemplo práctico. En este caso un programa que sabe desarrollar operaciones **POW, SUB, PRED, ISZERO, IFTHENELSE y MULT** en Racket utilizando cálculo lambda.

1. **Implementación**

Para la implementación puede utilizar programación aplicativa y/o recursión. Intente encontrar la forma más simple de hacerlo.

Sugerencia: Utilice funciones para imprimir en la pantalla

**Esta página** es el punto de partida para este proyecto <http://en.wikipedia.org/wiki/Lambda_calculus>

Es recomendable resolver las fórmulas en papel y entender bien el funcionamiento, antes de programar.

1. **Introducción:**

Su misión es escribir un programa que resuelve expresiones lambda. Y más adelante que permita definir instrucciones complejas como SUMA, MULTIPLICACIÓN, o IFTHENELSE, etc.

El primer paso es definir alguna representación:

***(λ x. x)***

No es muy compatible con racket, entonces en vez de utilizar eso utilizaremos la sintaxis:

**(L x \_ x)**

Es decir, en vez de ***λ*** usamos **L**. En vez del punto usamos: \_

No se olvide de nuestra gramática básica que define una expresión como:

**<expresion> ::= <identificador>**

**<expresion> ::= (L <identificador> \_ <expresion>)**

**<expresion> ::= (<expresion> <expresion>)**

Donde **<identificador>** representa cualquier símbolo como X, Y, XX, HOLA, etc.

1. **Instrucciones:**
2. **Lea el programa esqueleto (ejemplo proporcionado)**

Notará que el proceso básico es:

(define (inicio )

(cond [(null (imprimir(evaluar(validar-sintaxis(leer)))))

Nil]

[else inicio])))

Es decir que primero lee, luego valida la sintaxis, evalúa e imprime el resultado. Si el resultado no es nil entonces repite el proceso.

1. **Validar sintaxis (utilizando la gramática formal)**

Escribir la función VALIDAR-SINTAXIS que devuelve #t o null, de acuerdo a la validación de sintaxis.

Ej:

**> (VALIDAR-SINTAXIS '*(L x \_ ( x 2))* )**

***T***

**> (VALIDAR-SINTAXIS '*(L x \_ ( x 2) \_ x)* )**

***NIL***

1. **Implemente la función NUMERO-A-FORMULA**
2. **Implemente FORMULA-A-NUMERO**
3. **Implemente SUSTITUIR-NUMEROS**
4. **Implemente REDUCCION-BETA**

Reducción beta consiste principalmente en el proceso de sustituir las variables dependientes en el cuerpo de una función lambda cuando se aplica el argumento recibido.

**Ej.:**

**> ((L x \_ x) 2)**

***2***

donde se sustituye el parámetro 2 en la variable “m”.

**Ej:**

***> ((L m \_ (L n \_ (L p \_ (q (p m))) 2)***

***(L n \_ (L p \_ (q (p 2))***

1. **Implemente EVALUAR**

Evaluar básicamente hace todo el proceso. Primero sustituye números por formulas, luego hace reducción beta (repetidamente) hasta no poder más, y finalmente sustituye las formulas por números (donde puede). Durante todo esto va imprimiendo explicaciones de lo que hace.

1. **Agregue funcionalidad a la sintaxis del lenguaje**

Básicamente permita que se puedan definir un alias para expresiones, por ejemplo:

**(SUMA := (L m \_ (L n \_ (L f \_ (L x \_ ((m f) ((n f) x)) ) ) ) ))**

Aquí SUMA se vuelve equivalente a la formula (L m \_ (L n \_ (L f \_ (L x \_ ((m f) ((n f) x)) ) ) ) ) y puede ser utilizado en su lugar. Por ejemplo, después de eso podemos decir:

**((SUMA 1) 2)**

En vez de

**((L m \_ (L n \_ (L f \_ (L x \_ ((m f) ((n f) x)) ) ) ) ) 1) 2)**

Así podemos definir cualquier cosa como TRUE, FALSE, IFTHENELSE, etc.

Habrá que cambiar:

* El verificador de sintaxis para que acepte expresiones como esta
* Utilice SETF para asociar el alias con la expresión o alternativamente puede utilizar listas de listas
* Crea una nueva función para hacer sustituciones para poder sustituir los alias por su expresión
* Modifique el evaluador para que funcione con estas expresiones

1. **Escriba una función interesante con cálculo lambda**

Función exponencial, o algo más interesante combinando definiciones interesantes. Es decir, no hace falta cambiar el evaluador. solo hay que utilizar la funcionalidad de la parte (7) para hacer esto.

1. **Permita sintaxis alternativa**

Ejs:

(L x \_ x x) en vez de (L x \_ (x x))

(L f x \_ f x) en vez de (L f \_ (L x \_ (f x)))