# PROGRAMACIÓN FUNCIONAL Y LAMBDA CÁLCULO

*Jonatán Gómez Perdomo* -Wilson Castro Rojas** - Alexander Cardona López*
*Ingenieros de Sistemas*
*Universidad Nacional de Colombia, Santafé de Bogotá.*

**Resumen**

En la primera parte de este artículo se explican algunos conceptos de los lenguajes de programación, haciendo énfasis en las características y propiedades de los lenguajes de programación funcional. En la segunda, se realiza una introducción al lambda cálculo puro, su notación, axiomas y reglas elementales.

## INTRODUCCIÓN

El manejo que se le da a las funciones en los lenguajes imperativos hace muy complicado (o imposible) realizar operaciones que matemáticamente son importantes, tales como la composición de funciones. Los lenguajes funcionales surgen como una forma de llenar este vacío, dándoles a las funciones un valor de primer orden, gracias a lo cual éstas pueden tratarse como cualquier dato, permitiéndose incluso utilizar una función como parámetro de otra función.

Un programa desarrollado en lenguaje funcional se denomina programa funcional. Un programa funcional se explica como un conjunto de composición reiterada de funciones que resuelven los problemas en un estilo particular. Un programa funcional puede escribirse en varios lenguajes como LISP, SML, Miranda, SCHEME (dialecto de LISP) y

**Docente de la Universidad Nacional de Colombia*
***Docente Ocasional Universidad Nacional de Colombia*

otros. La implementación de lenguajes funcionales conlleva la construcción de una sintaxis y semántica fáciles de definir a partir del cálculo lambda. Si un lenguaje funcional permite realizar asignación y evaluación de expresiones no es puro, de otra forma es puro.

El cálculo lambda permite estudiar las propiedades de las funciones, de tal forma que se constituye en la teoría que formaliza los lenguajes funcionales, facilitando la rigurosa evaluación y prueba de expresiones. La forma elemental de ver el cálculo lambda puro es: $\lambda x.M$ que constituyen las llamadas abstracciones, donde $x$ es el parámetro de la función y $M$ el cuerpo. $MN$ es la aplicación de la función $M$ a $N$. Visto como una gramática de términos es $M := x|(M_1 M_2)|(\lambda x.M)$ .

## I. SINTAXIS Y SEMÁNTICA

En ciencias de la computación uno de los fundamentos para el desarrollo de lenguajes de programación consiste en el estudio de la forma de escribir los programas y el significado e interpretación de los mismos.

### *A. Forma de Backus - Naur*

Todos los elementos que componen un lenguaje, junto con sus relaciones, hacen deducir un conjunto de reglas que lo determinan. Los símbolos atómicos de un lenguaje se conocen como *componentes léxicos* o *símbolos terminales;* los símbolos no terminales son llamados constructores de un lenguaje. El símbolo no terminal que representa al constructor principal de un lenguaje se denomina *símbolo no terminal inicial.*

Una de las gramáticas independientes de contexto más usadas para especificar la sintaxis de un lenguaje de programación es la gramática BNF (Backus-Naur Form), la cual se muestra en el siguiente ejemplo:

***Ejemplo*** **:**

Describir la sintaxis de los números reales como 3.14159 con una parte entera, un punto decimal y una parte fraccionaria.

Los números reales positivos pueden definirse por las siguientes reglas:

- Un número real positivo está formado por dos secuencias de dígitos separadas por un punto. La primera es finita.
- Una secuencia de dígitos está compuesta por uno o más dígitos.
- Un dígito es 0,1,2,3,4,5,6,7,8 ó 9.

Usando BNF:

```
<número-real>          ::= <secuencia-dígitos>.
                           <secuencia-dígitos>
<secuencia-dígitos>    ::= <dígito>|<dígito>
                           <secuencia-dígitos>
<dígito>               ::= 0|1|2|3|4|5|6|7|8|9
```

en donde

```
<...>  : encierran variables representando
         constructores
::=    : se lee como "es"
|      : se lee como "o", - disyunción -.
```

Cada opción separada por | es una regla distinta; por ejemplo:

```
<secuencia-dígitos> ::= <dígito>|<dígito>
                        <secuencia-dígitos>
```

Se puede escribir como:

```
<secuencia-dígitos> ::= <dígito>
<secuencia-dígitos> ::= <dígito>
                        <secuencia-dígitos>
```

En este caso, los símbolos terminales son los dígitos (0,1,2, etc.) y el punto. Los constructores son los símbolos no terminales, como <secuencia-dígitos>.

A menos que se afirme otra cosa, las producciones del símbolo no terminal inicial son las primeras en aparecer. Además, hay que aclarar que una cadena es una secuencia finita de símbolos terminales con una longitud igual al número de símbolos. La cadena vacía tiene longitud cero.

Una expresión es una secuencia de símbolos que describen un cómputo. Las más importantes

clases de símbolos son constantes, variables,operadores y paréntesis [5]. La notación de las expresiones es importante debido que dependiendo de la adopción de una determinada notación ,un *intérprete* lee las expresiones y puede evaluar si una expresión pertenece o no al lenguaje. Por ejemplo, un operador *binario* se aplica a dos operandos en notación *infija* cuando el operador se coloca entre los operandos *(a*b)*, *prefija* cuando el operador aparece primero *(*ab)* y *posfija* cuando el operador va al final *(ab*)*.

### *B. Sintaxis*

La sintaxis de un lenguaje especifica cómo se construyen los programas, en dicho lenguaje. La estructura sintáctica, es decir la estructura impuesta por la sintaxis del lenguaje, constituye la herramienta fundamental para trabajar con éste. Por eso se ha utilizado para describir constructores y reglas para entender los programas escritos en un determinado lenguaje.

> ***Definición*:** El constructor de un lenguaje es una estructura sintáctica o conjunto de estructuras de un lenguaje, que sirven para expresar una clase particular de operaciones [3].

La sintaxis desempeña dos funciones principales en los lenguajes de programación: primero, la *sintaxis abstracta de un lenguaje* identifica los componentes significativos de cada constructor. Las descripciones de lenguajes y las implementaciones están organizadas alrededor de las sintaxis abstractas. Segundo, la *sintaxis concreta* de un lenguaje describe su representación escrita, incluyendo detalles como la colocación de palabras claves y los signos de puntuación [4].

### *C. Semántica*

Inicialmente se habló de la *semántica* en trabajos que se referían al estudio del cambio de significado de las palabras. Hoy generalmente se define como el estudio de las relaciones entre las palabras y las instrucciones de un lenguaje escrito o hablado y su significado. En áreas del conocimiento como la lingüística y la filosofía, la palabra semántica tiene una aplicación un tanto distinta, dado que interesa más el análisis del lenguaje natural.

Semántica se entiende como el significado de sentencias en el lenguaje formal de la lógica matemática [2], o también como el desarrollo de formas para expresar lenguajes usados para programación de computadores digitales. Esta última caracterización es más conocida como semántica de los lenguajes de programación, los cuales tradicionalmente se han basado en sentencias imperativas. En contraste, las sentencias de la lógica matemática intentan plantear axiomas, reglas, declaraciones, proposiciones, etcétera.

El significado de las palabras, frases, expresiones o signos que determinan una idea o cosa material, debe ser especificado en los lenguajes para determinar el significado de un programa. Como pueden darse varias especificaciones distintas, dependiendo del lenguaje, una misma sintaxis puede tener semánticas distintas.

En programación es un problema la determinación de los valores de las variables así como la definición del ambiente (ámbito), en el cual el lenguaje puede entender y reconocer una expresión, una variable o una función. Por tal razón, en los lenguajes existen reglas para la determinación del ámbito, las cuales son de dos clases: de *ámbito dinámico*, cuando para cualquier estado en que se encuentre un programa las variables pueden tomar valores distintos; de *ámbito estático*, cuando una vez definido un valor para una variable, la aplicación de ese valor se mantendrá en cualquier estado del programa a partir de la aplicación del valor. Además, todas las funciones poseen una distinción entre lo que es el ambiente en el cual se define la función (ambiente de definición) y el ambiente en el cual se aplica ésta (ambiente de activación).

El análisis semántico ayuda a la estandarización de terminologías y a la identificación de similitudes y diferencias entre lenguajes. Esto le permite al diseñador encontrar restricciones indeseables, incompatibilidades, ambigüedades, etc., mediante el uso de una formalización rigurosa y pruebas de las propiedades semánticas del lenguaje.

## II. LENGUAJES FUNCIONALES

Los lenguajes de programación se dividen principalmente en dos clases según la forma en que el lenguaje permita definir la obtención de los resultados deseados.

***Lenguajes declarativos (no procedimentales):*** un programa afirma explícitamente lo que requiere que exhiba el resultado, pero no afirma cómo debe obtenerse el resultado; acepta cualquier forma de producir un resultado que muestre las propiedades requeridas [3].

***Lenguajes imperativos (procedimentales):*** es el caso contrario de los lenguajes declarativos. Se define explícitamente cómo será obtenido el resultado, pero no se define explícitamente qué propiedades se espera que exhiba el resultado [3].

### *Características*

El lambda cálculo, desarrollado por el lógico matemático Alonso Church [1] ha servido dentro de la formalización de modelos matemáticos, para el diseño de lenguajes de programación, en especial como la base de los lenguajes de programación funcional.

Un lenguaje funcional puede entenderse como una subclase de los lenguajes declarativos. Un programa escrito en un lenguaje funcional está compuesto de un conjunto de ecuaciones basadas en el uso de valores, funciones y recursión. Estas ecuaciones operan con funciones y valores evaluados a partir de funciones primitivas y valores provistos por el lenguaje.

### *Ejemplo*

Evaluar la suma de los enteros de 1 a 10.

En un lenguaje *imperativo* se podría dar solución al problema usando un ciclo finito, que consiste en la actualización repetida del valor contenido en un contador y la variable acumuladora:

```
total = 0;
for (i=1; i<=10; ++i)
{
        total = total+i;
}
```

En este caso se trabaja con la *asignación* de un valor a la variable *total,* cada vez que cumple una iteración del ciclo, hasta que al final del ciclo esta variable contenga el valor total de la suma.

En el caso de un lenguaje declarativo, el programa puede expresarse sin actualización de variables. Así:

```
sum[1..10].
```

Expresa la suma, donde [1..10] es una expresión que representa la lista de enteros de 1 a 10, mientras *sum* es una función que puede usarse para calcular la suma de una lista arbitraria de valores.

En el caso particular de *lenguajes funcionales* como ML o SCHEME, es común encontrar la solución del problema utilizando un ciclo finito. En ML se hace de la siguiente manera, aunque la actualización de variables no es necesaria:

```
let fun sum i tot = if(i=0) then tot else sum(i-1)
(tot+i)
in sum 10 0
end
```

Donde se define una función *sum,* la cual tiene como resultado *tot* si i=0; en caso contrario la función se "llama" a si misma en forma recursiva con argumentos *(i-1) y (tot+i).*

En el ejemplo se observa que un resultado puede obtenerse, utilizando un lenguaje declarativo (por ejemplo uno funcional) o uno imperativo; la cuestión es definir en qué lenguaje se implementa.

Otra de las características importantes de los lenguajes funcionales es que los usuarios no tienen que preocuparse por el almacenamiento de datos, ya que se hace *manejo de almacenamiento implícito,* que consiste en que ciertas operaciones integradas de los datos asignan almacenamiento en el momento necesario. El almacenamiento que se vuelve inaccesible se libera, en forma automática.

Esta ausencia de código explícito para la liberación de memoria hace a los programas más sencillos y cortos, pero el lenguaje debe estar capacitado para recuperar memoria que se vuelve innecesaria.

## III. PROGRAMACIÓN FUNCIONAL

Cuando se habla de programación funcional, hay que entender que es un mecanismo adoptado para resolver problemas de programación utilizando un lenguaje funcional, es decir, trabajando con la sintaxis y con la semántica que definen ese lenguaje.

En programación funcional, una función puede ser el valor de una expresión, puede pasarse como argumento y puede colocarse en una base de datos, esto permite la creación de operaciones sobre colecciones de datos; que en lenguajes imperativos son muy difíciles de realizar o imposibles.

Existen distintas opiniones, incluso dentro de la comunidad de programación funcional, respecto a su definición; hasta el momento se propone la siguiente:

> ***Definición:*** programación funcional es un estilo de programación que enfatiza la evaluación de expresiones, más que la ejecución de comandos. Las expresiones en los lenguajes funcionales se forman mediante el uso de funciones que combinan valores básicos [6].

El estilo de programación funcional se caracteriza por la ejecución o evaluación de funciones, que desde un planteamiento general resuelven problemas con base en una composición reiterada de funciones, partiendo de que las composiciones más internas son funciones elementales (primitivas), como por ejemplo la suma de dos enteros. En términos de composición de funciones lo anterior significa:

$$f_1\left(f_2\left(f_3\left(f_4\left(\ldots\left(f_k\right)\ldots\right)\right)\right)\right)$$

Aquí la composición más interna (la más básica) es la que tiene que resolverse primero: en términos computacionales los niveles más internos de ejecución de funciones son los primeros en procesarse y los resultados de estas evaluaciones son necesariamente utilizadas en los niveles superiores.

***Ejemplo***

5+7*(4-30/(4-19)), se puede ver como:
+(5,*(7,-(4,/(30,-(4,19)))))

En donde la función más interna - *(4,19)* sería la primera en resolverse. El resultado de esta expresión es *47.*

## IV. LAMBDA CÁLCULO

El lambda cálculo surge antes que los lenguajes de programación de alto nivel a raíz del estudio realizado por Alonso Church[1] en la década del 30, que inicialmente se interesaba en el análisis de las funciones. La relevancia del lambda cálculo en ciencias de la computación sólo se apreció en 1960, cuando las propiedades básicas en lenguajes de alto nivel se estudiaron, observándose el potencial que tiene el lambda cálculo en la especificación de los lenguajes de programación. Desde entonces, el lambda cálculo es una teoría fundamental de *aplicación* y *abstracción* para el estudio de tipos[1] ya que puede verse como una gramática para términos de un lenguaje, es decir, el lambda cálculo permite decidir si una expresión pertenece o no al lenguaje.

El lambda cálculo puro[2] tiene una sintaxis pequeña compuesta solamente de tres constructores: variables, aplicación de funciones y creación de funciones.

---

***1. El tipo indica los valores que una expresión está en capacidad de representar y las operaciones que sobre ellas se pueden aplicar. Dado que como principio generalizado en el diseño de lenguajes toda expresión debe tener un tipo, entonces los tipos son un mecanismo para clasificar expresiones. Los tipos surgen de diferentes necesidades, como por ejemplo, el nivel de máquina, dado que los compiladores necesitan información sobre el tipo para poder generar expresiones en código de máquina, nivel de lenguaje cuando los tipos estructurados se construyen a partir de tipos básicos y nivel de usuario cuando el usuario puede definir sus propios tipos dependiendo del problema.***

***2. El lambda cálculo puro no tiene tipos. Las funciones pueden aplicarse sin restricciones, pero teniendo en cuenta el ámbito, el paso de parámetros y las estrategias de evaluación.***

Como se había mencionado, el lambda cálculo puro es una gramática de términos, la cual se especifica así:

$M := x|(M_1 M_2)|(\lambda x.\ M)$ que dice que un término puede ser una variable $x$, una aplicación $(M_1 M_2)$ de la función $M_1$ a $M_2$, o una abstracción $(\lambda x.M)$.

Se escribe:

$\lambda x.\ M$ : función con parámetro $x$ y cuerpo $M$.

Note que las funciones se escriben junto a sus argumentos.

$\lambda x.M \Leftrightarrow$ abstracción $\Leftrightarrow$ definición de función.

"Determinación de parámetros y cuerpo de la función"

$xy \Leftrightarrow$ aplicación $\Leftrightarrow$ invocación.

"Se está invocando la función $x$ con parámetro $y$, donde $y$ es una función o una variable"

Se usan las letras $f, x, y, z$ para variables y $M, N, P, Q$ para términos. La letra $c$ se utiliza para constantes básicas y funciones constantes.

***Ejemplo***[3]

$(\lambda x.x*x)5$ : Es una función que toma el valor 5 y le aplica el cuerpo "$x*x$", haciendo corresponder 5*5.(ver nota de pie 4).

"La función $\lambda x.x*x$ se aplica a cinco" se escribe $(\lambda x.x*x).5$ y las fórmulas de este tipo se denominan *términos*.

Un lenguaje de programación funcional es esencialmente un lambda cálculo con constantes apropiadas. Un lambda cálculo con tipos se tiene cuando se asocia un tipo a cada término[5].

Para profundizar en la noción de *abstracción* y *aplicación* se presenta una revisión de la llamada Igualdad Beta.

***A. Igualdad en expresiones lambda***

Escribimos $M =_\beta N$ si $M$ y $N$ son iguales según la igualdad beta (la cual se explica posteriormente) y se dice que tienen el mismo valor. Dicho de otra forma, es aplicar una abstracción $\lambda x.M$ a un argumento $N$. Se observan aquí las nociones de invocación a función y el paso de parámetros en lenguajes de programación.

***Ejemplo***

Observemos de nuevo la función cuadrado escrita en ML [4]

***fun*** *cuadrado(x) = x*x;*

cuadrado*(5) = 5*5* : es la invocación de la función con parámetro *5*, también la evaluación remplazando en el cuerpo de la función el parámetro *5*.

Utilizando expresiones lambda se puede escribir:

$\lambda x.(*xx)$ es la abstracción de la función cuadrado.

Sea $M$ el cuerpo $*(xx)$ y $N$ la función constante *5*, entonces

$(\lambda x.(*xx))5 =_\beta 25$ según la igualdad beta.

***B. Variables libres y Acotadas***

La abstracción $\lambda x.M$ también restringe el valor de $x$ en $\lambda x.M$. Entonces, se dice que $x$ está acotada en $\lambda x.M$.

---

***3. La formulación esencial de algunos ejemplos de esta sección, se encuentra en ejemplos del capitulo 12 del texto de Sethi[4].***

***4. En el ejemplo se utilizó (x*x), que no es correcta en expresiones lambda. La notación prefija *( xx) es la que se debe utilizar.***

***5. Un lambda cálculo con tipos polimorfos se ha usado para estudiar los tipos en el lenguaje funcional ML.***

El conjunto *libre(M)* está formado por todas las variables libres de *M*, es decir, las variables que no están acotadas en *M*, dadas por las reglas:

***i)*** *libre(x)* $=\{x\}$ "x es libre en el término x"

***ii)*** *libre(MN)* $=libre(M) \cup libre(N)$ "Una variable es libre en *MN* si es libre en ***M*** o en *N*".

***iii)*** $libre(\lambda x.M) = libre(M) - \{x\}$ "Excepto *x*, todas las variable son libres en $\lambda x.M$"

Nota: *x* es libre en *M* si *M* tiene la forma $\lambda y.N$, donde *y* es diferente de *x* y la ocurrencia de *x* en el subtérmino *N* es libre.

La variable *x* en $\lambda x.M$ se denomina aparición, o simplemente asociación de *x*.

***Definición***: todas las apariciones de *x* en $\lambda x.M$ son acotadas dentro del ámbito de esta abstracción. Todas las apariciones no acotadas de una variable en un término son libres. Además, toda aparición de una variable debe ser libre o acotada, pero no simultáneamente.

***Ejemplo***

En $(\lambda y.z)(\lambda z.z)$, la variable *z* es libre, porque es libre en $\lambda y.z$. (regla ***ii*** ).

***C. Sustitución***

Aplicar una abstracción $\lambda x.M$ a un parámetro *N* se realiza *sustituyendo x* por *N* en el cuerpo *M*. Es decir, *N* reemplaza todas las apariciones libres de *x* en *M*.

La sustitución de un término *N* por una variable *x* en *M* se nota $\{N/x\}M$ y se define así:

1. Suponga que las variables libres de *N* no tienen apariciones acotadas en *M*. Entonces, el término $\{N/x\}M$ se forma remplazando con *N* todas las apariciones libres de *x* en *M*.
2. En otro caso, suponga que la variable *y* es libre en *N* y acotada en *M*. La asociación y las apariciones acotadas correspondientes de *y* en *M* se reemplazan de manera consistente por alguna variable nueva *z* o se utilizan índices posicionales. Se siguen nombrando las variables acotadas en *M* hasta que se pueda aplicar el caso 1. Posteriormente se aplica el caso 2.

***Ejemplo***

***(i)*** *M* no tiene apariciones acotadas (caso 1).

$\{u/x\}x = u$ "*u* reemplaza a *x* en el cuerpo *x*".
$\{u/x\}(xx) = (uu)$ "*u* remplaza a *x* en el cuerpo *xx*, entonces queda *uu*".
$\{(\lambda x.x)/x\}x = (\lambda x.x)$ "$\lambda x.x$ reemplaza a *x* en el cuerpo *x*, entonces queda $(\lambda x.x)$".

En estos ejemplos, como *M* no tiene apariciones acotadas, es decir no hay restricciones de ámbito en los cuerpos *x*, *(xx)* y *x*, se sigue como en el caso1.

***(ii)*** *M* no tiene apariciones libres de *x*.

$\{u/x\}y = y$ "*u* reemplaza a *x* en el cuerpo *y*. Como en *y* no hay apariciones libres de *x*, entonces queda *y*

$\{(\lambda y.y)/x\}(\lambda x.x) = \lambda x.x$
"$\lambda x.x$ reemplaza *a* las *x* libres en el cuerpo $\lambda x.x$, como en $\lambda x.x$, *x* es acotada en $\lambda x.x$, entonces queda $\lambda x.x$".

En general, cuando en *M* no hay variables libres *x* para la sustitución $\{N/x\}$ entonces queda el mismo *M*, es decir, $\{N/x\}M$ es *M*.

***(iii)*** *u* en *N* tiene apariciones acotadas en *M*

(caso 2).

$\{u/x\}(\lambda u.x) = \{u/x\}(\lambda z.x) = (\lambda z.u)$
$\{u/x\}(\lambda u.u) = \{u/x\}(\lambda z.z) = (\lambda z.z)$

Cuando en una sustitución variables acotadas se encuentran notadas con el mismo nombre de una variable para sustituir, es conveniente realizar un cambio de variable o colocación de índices. El

*axioma a* permite asignar sistemáticamente nuevo nombre a las variables acotadas. El axioma es el siguiente:

$$(\lambda x.M) =_\beta \lambda z.\{z/x\}M$$

Suponiendo que *z* no es libre en *M*.

***Ejemplo***

Sea el término $(\lambda xy.*xy)yx$.

Si no se renombran las variables y se reemplazan las apariciones libres, se tendría incorrectamente lo siguiente:

$$(\lambda y.*yy)x \Rightarrow *xx$$

Mientras que renombrando las variables se obtiene:

$$(\lambda uv.*uv)yx \Rightarrow (\lambda v.*yv)x$$
$$\Rightarrow *yx$$
$$(\lambda uv.*uv)yx =_\beta *yx$$

***Ejemplo***

$$\lambda x.x =_\beta \lambda y.y$$

$$\lambda xy.x =_\beta \lambda uy.u$$

ya que *x* no es libre.

***Reglas Dirigidas por Sintaxis***

Sean *P* y *Q* subtérminos del cuerpo *M* (*M*=*PQ*), en donde se hace la sustitución *N* por *x*. Entonces:

*(i)* $\{N/x\}x = N$

*(ii)* $\{N/x\}y = y$ si $y \neq x$.

*(iii)* $\{N/x\}(PQ) = \{N/x\}P\{N/x\}Q$

*(iv)* $\{N/x\}(\lambda x.P) = \lambda x.P$ "si x no es libre".

*(v)* $\{N/x\}(\lambda y.P) = \lambda y.\{N/x\}P$

si $y \neq x, y \notin$ libre(N).

*(vi)* $\{N/x\}(\lambda y.P) = \lambda z.\{N/x\}\{z/y\}P$

si $y \neq x, z \notin$ libre(N), $z \notin$ libre(P).

***Ejemplo***

$$\{u/x\}(\lambda u.x) = \{u/x\}(\lambda z.x)$$
$$= \lambda z.u$$

"u se renombra por z ya que no es libre (regla *vi*)" "regla *v*".

***Ejemplo***

$$\{u/x\}(\lambda y.u) = \lambda y.u$$

***D. Igualdad beta***

Axioma fundamental de la igualdad beta:

$(\lambda x.M)N =_\beta \{N/x\}M$

de forma que $(\lambda x.y)u =_\beta u$ y $(\lambda x.y)u =_\beta y$

pues $(\lambda x.x)u =_\beta \{u/x\}x = u$ y $(\lambda x.y)u = \{u/x\}y = y$

La igualdad Beta cumple las siguientes propiedades:

- *Reflexividad:* un término *M* es igual a sí mismo.
- *Conmutatividad:* si *M* es igual a *N*, entonces *N* es igual a *M*.
- *Transitividad:* si *M* es igual a *N* y *N* es igual a *P*, entonces *M* es igual a *P*.

De las tres propiedades anteriores se concluye que la igualdad beta define una relación de equivalencia.

Las propiedades anteriores se notan mediante las siguientes reglas:

*(i)* $M =_\beta M$

*(regla de reflexividad)*

*(ii)*

$$\frac{M =_\beta N}{N =_\beta M}$$

*(regla de conmutatividad)*

*" Si $M =_\beta N$, entonces $N =_\beta M$ "*

*(iii)*

$$\frac{M =_\beta N \qquad N =_\beta P}{M =_\beta P}$$

*(regla de transitividad)*

*"Si $M =_\beta N$ y $N =_\beta P$, entonces $M =_\beta P$"*

Otras reglas son:

*(iv)*

$$\frac{M =_\beta M' \qquad N =_\beta N'}{MN =_\beta M'N'}$$

*(regla 1 de congruencia)*

*(v)*

$$\frac{M =_\beta M'}{\lambda x.\, M =_\beta \lambda x.\, M'}$$

*(regla 2 de congruencia)*

***Ejemplo***

Sea $I = \lambda x.x$ y $S = \lambda xyz.(xz)(yz)$

o en forma completa:

$$S = (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$$

Verificar la siguiente igualdad: $SIII =_\beta I$

$(\lambda xyz.(xz)(yz))III = \underline{(\lambda xyz.(xz)(yz))(\lambda x.x)}(\lambda x.x)(\lambda x.x)$

$=_\beta (\lambda yz.\{(\lambda x.x)/x\}((xz)(yz)))(\lambda x.x)$
$(\lambda x.x)$

$=_\beta (\lambda yz.((\underline{\lambda x.x})z)(yz))(\lambda x.x)(\lambda x.x)$

$=_\beta (\underline{\lambda yz.z(yz)})(\underline{\lambda x.x})(\lambda x.x)$

$=_\beta (\lambda z.\ z(\underline{(\lambda x.x)z}))(\lambda x.x)$

$=_\beta (\underline{\lambda z.\ zz})(\lambda x.x)$

$=_\beta (\lambda x.x)(\lambda x.x)$

$=_\beta \lambda x.x$

$=_\beta I$

***E. Reducciones***

El cálculo con términos lambda hace uso del axioma fundamental de la igualdad $\beta$ y del axioma $\propto$, los cuales son llamados reducción *beta* y conversión *alfa* respectivamente, los cuales se denotan así:

$$(\lambda x.\, M)N \underset{\beta}{\Rightarrow} \{N/x\}M$$

*(reducción β)*

$$\lambda x.\, M \underset{\alpha}{\Rightarrow} \lambda z\{z/x\}M$$

*z no es libre en M (conversión α)*

Una *reducción* es cualquier secuencia de reducciones $\beta$ y *conversiones* $\alpha$. Donde el resultado de un cálculo no depende del orden en el cual se apliquen las reducciones[6]. Se dice que un término que no puede tener reducciones $\beta$ se encuentra en *forma normal*; el término $\lambda z.z$ se halla en forma normal porque ninguno de sus subtérminos es de la forma $(\lambda x.M)N$. La forma normal es única si existe.

El término $(\lambda x.M)N$ se denomina exred (expresión a reducir). De esta forma, si, $P \underset{\beta}{\Rightarrow} Q$ entonces $P$ tiene alguna exred $(\lambda x.M)N$ que se reemplaza con $\{N/x\}M$ para crear $Q$.

***6. Ver teorema de Church-Rosser. En SETHI "Lenguajes de Programación -Funciones y Contructores-" Sethi. Pág. 444-445.***

***Ejemplo***

Verificar $LII \Rightarrow_{\beta} I$. Donde $L = \lambda xy.x$ e $I = \lambda x.x$.

$$LII = \underline{(\lambda xy.x)(\lambda x.x)}(\lambda x.x) \underset{\beta}{\Rightarrow} (\lambda y.(\lambda x.x))(\lambda x.x) \underset{\beta}{\Rightarrow} \lambda x.x$$

NOTA: Se dice que una reducción es sin final cuando la reducción continúa de manera indefinida sin encontrar una forma normal. Por ejemplo, las reducciones que comienzan con $(\lambda x.xx)(\lambda x.xx)$.

***Estrategia de Reducción***

Existen dos estrategias de reducción, la *invocación por valor* y la *invocación por nombre.* En el primer caso se elige la exred del extremo izquierdo más interna de un término. En cambio, para las invocaciones *por nombre* se elige la exred del extremo izquierdo más externa. En este contexto, "interna" y "externa" se refieren a la forma como están anidados los términos.

***Ejemplo***

Reducir la siguiente expresión.

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z))$$

En el caso de la invocación por valor, se tiene:

$$\begin{aligned}(\lambda x.xx)((\lambda y.y)(\lambda z.z)) &\underset{\beta}{\Rightarrow} (\lambda x.xx)(\lambda z.z) \\ &\underset{\beta}{\Rightarrow} (\lambda z.z)(\lambda z.z) \\ &\underset{\beta}{\Rightarrow} (\lambda z.z)\end{aligned}$$

En el caso de invocación por nombre.

$$\begin{aligned}(\lambda x.xx)((\lambda y.y)(\lambda z.z)) &\underset{\beta}{\Rightarrow} ((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z)) \\ &\underset{\beta}{\Rightarrow} (\lambda z.z)((\lambda y.y)(\lambda z.z)) \\ &\underset{\beta}{\Rightarrow} (\lambda y.y)(\lambda z.z) \\ &\underset{\beta}{\Rightarrow} (\lambda z.z)\end{aligned}$$

Como se observa, la invocación por valor requiere menor número de reducciones $\beta$ para alcanzar una forma normal, pero existe la posibilidad de que en algunas expresiones se quede en un ciclo infinito. La reducción por nombre o reducción en *orden normal* garantiza alcanzar una forma normal, si existe. Comúnmente, los lenguajes funcionales utilizan la invocación por valor, debido a que es posible implantarla eficientemente.

**BIBLIOGRAFÍA**


1. BARENDREGT, H. P. "Studies in logic. and the foundations of mathematics - The Lambda Calculus ". Vol. 103. Amsterdam; North-Holland, 1984.

2. GUNTER, Carl A. "Semantic of Programming Languages - Structures and Techniques -", MIT Press, 1992.

3. GLASER, Edward L. "Dictionary of Computing", Oxford University Press. Oxford. 2a. Edición. 1986.

4. SETHI, Ravi. " Lenguajes de Programación. - Conceptos y Constructores -". Adisson Wesley. 1992.

5. WIKSTRÖM, Åke. "Functional Programming Using Standard ML". Prentice Hall International, Cambridge, 1987.

6. <http://www.cs.not.ac.uk/DepartmentStaff/mpj/faq.html>