Cálculo Lambda (*λ Calculus*)

En la década de 1930, varios matemáticos estaban interesados en la pregunta: ¿cuándo una función f: ℕ → ℕ es *computable*? Una definición informal de *computabilidad* es que debe haber un método que permita calcular **f(n)**, para cualquier **n** dado, usando lápiz y papel.

Por ejemplo, la función *sucesor* se puede definir de la siguiente manera:

```
succ (x) = x + 1
```

cuerpo de la función

parámetro formal

nombre de la función

Donde la aplicación de la función **succ** al número 4 sería:

```
succ (4) = 4 + 1 = 5
```

argumento efectivo

En la notación de flecha (*arrow notation*) se deja de lado el nombre de la función: $x \mapsto x + 1$

El inglés Alan Turing, el austríaco Kurt Gödel y estadounidense Alonzo Church definieron la computabilidad de funciones mediante tres modelos que son equivalentes entre sí, es decir, cada modelo define la misma clase de funciones computables. Esta equivalencia se conoce como la *tesis de Church-Turing*.

1. Turing (cuya tesis doctoral fue dirigida por Church) definió una computadora idealizada que ahora llamamos *máquina de Turing*, y postuló que una función es computable si y solo si puede ser calculada por dicha máquina. Conceptualmente, las actuales computadoras que siguen el modelo de Von Neumann no son más que *máquinas de Turing* con registros de acceso aleatorio. Los lenguajes de programación imperativos como C, Pascal, Fortran, etc., así como también todos los lenguajes de ensamblador, se basan en la forma en que se instruye a una *máquina de Turing*: mediante una secuencia de *sentencias* o *proposiciones*.

2. Gödel definió la clase de *funciones recursivas generales* como el más pequeño conjunto de funciones parciales con cualquier número de argumentos de los naturales en los naturales, que contiene todas las funciones constantes y la función sucesor, y tal que las funciones de proyección, como la composición y la recursión, son operaciones cerradas en este conjunto. Postuló que una función es computable si y solo si es una función recursiva general.

3. Church definió un lenguaje de programación idealizado llamado *cálculo lambda*, y postuló que una función es computable si y solo si puede escribirse como una expresión lambda. Podría decirse que el *cálculo lambda* es el lenguaje de programación más simple posible que es *Turing completo*. Muchos lenguajes de programación actuales, como Lisp, Scheme, Haskell, Clojure y Scala se basan en el *cálculo lambda*, al que le agregan características adicionales, como tipos de datos, E/S, etc. Los dispositivos construidos específicamente para la ejecución de programas escritos en estos lenguajes funcionales se denominan *máquinas de reducción*.

1. PROGRAMAS COMO EXPRESIONES

Un *programa funcional* consiste en una expresión **E** que representa tanto el algoritmo como los datos de entrada. Para su ejecución, se le aplican a **E** ciertas reglas de conversión.

La *reducción* consiste en reemplazar una parte P de **E** por otra expresión **P'** de acuerdo con las reglas de conversión dadas. En notación esquemática:

E[P] → E[P'], siempre y cuando **P → P'** esté de acuerdo con las reglas.

Este proceso de reducción se repetirá hasta que la expresión resultante no tenga más partes que puedan convertirse. Esta denominada *forma normal* **E*** de la expresión **E** consiste en la salida del programa funcional dado.

Por ejemplo:

```
(7 + 4) × (8 + 5 × 3)
→ 11 × (8 + 5 × 3)
→ 11 × (8 + 15)
→ 11 × 23
→ 253
```

En este caso, las reglas de reducción consisten en las *tablas* de sumar y multiplicar valores numéricos.

Mediante reglas de reducción también pueden convertirse datos simbólicos (no numéricos).

Por ejemplo:

```
primero (ordenar (unir (["perro", "conejo"], ordenar (["ratón", "gato"]))))
→ primero (ordenar (unir (["perro", "conejo"], ["gato", "ratón"])))
→ primero (ordenar (["perro", "conejo", "gato", "ratón"]))
→ primero (["conejo", "gato", "perro", "ratón"])
→ "conejo"
```

Las funciones como **primero**, **ordenar** y **unir** se pueden programar fácilmente combinando algunas reglas de conversión, por lo que se las denomina *combinadores*.

Los sistemas de reducción generalmente satisfacen la *propiedad de Church-Rosser*, que establece que la forma normal obtenida es independiente del orden de reducción de los subtérminos. De hecho, el primer ejemplo también puede reducirse de las siguientes maneras:

Cambiando el orden de reducción:

```
(7 + 4) × (8 + 5 × 3)
→ (7 + 4) × (8 + 15)
→ 11 × (8 + 15)
→ 11 × 23
→ 253
```

Reduciendo varias expresiones al mismo tiempo:

```
(7 + 4) × (8 + 5 × 3)
→ 11 × (8 + 15)
→ 11 × 23
→ 253
```

2. SINTAXIS DEL CÁLCULO LAMBDA

La sintaxis de una *expresión lambda* (también conocida como *término lambda*) se puede expresar de la siguiente manera en BNF (Backus-Naur-Form):

<expresión λ> ::= <variable> | (λ <variable>.< expresión λ>) | (<expresión λ> < expresión λ>)

Es decir, una expresión lambda puede ser de cualquiera de las siguientes tres clases:

1) una *variable*;
2) una *abstracción*: con una variable (o parámetro formal) y un *cuerpo* que también es una expresión lambda;
3) una *aplicación*: que tiene un operador (una función) y un operando (o argumento efectivo) que son también ambos expresiones lambda.

Los siguientes son ejemplos de expresiones lambda:

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

En **`http://www.biwascheme.org`** es posible evaluar en línea la siguiente *aplicación* escrita en Lisp / Scheme. Como se puede ver, la sintaxis de estos lenguajes (en especial el uso de los paréntesis) es bastante similar a la sintaxis del Cálculo Lambda:

```
((lambda (x) (+ x 1)) 4)
```

2.1. Convenciones

Para evitar tener que usar un número excesivo de paréntesis, se establecen ciertas convenciones al escribir expresiones lambda.

Por ejemplo, para reducir el número de paréntesis de la expresión **`(((λx.(λy.(y x))) a) b)`**:

1) Se omiten los paréntesis externos:
   `((λx.(λy.(y x))) a) b`
2) Se asume que las aplicaciones se asocian a la izquierda:
   `(λx.(λy.(y x))) a b`
3) Se asume que el cuerpo de las abstracciones se extiende hasta que se cierra un paréntesis o se alcanza el final de la expresión:
   `(λx.λy.y x) a b`
4) Opcionalmente (porque esto no disminuye el número de paréntesis), se pueden contraer múltiples abstracciones lambda:
   `(λx y.y x) a b`

Se obtiene de esta forma una expresión lambda que es equivalente a la original.

3. VARIABLES LIBRES Y LIGADAS

Sean ***x***, ***y***, ***z*** variables y ***M***, ***N***, ***P*** expresiones lambda cualesquiera:

La variable ***x*** ocurre *libre* en la expresión ***N*** si y solo si:

1) $N \equiv x$
2) $N \equiv \lambda z.M$ siendo $x \neq z$ y donde x ocurre libre en M
3) $N \equiv M\,P$ donde x ocurre libre en M y en P

La variable ***x*** ocurre *ligada* en la expresión ***N*** si y solo si:

1) $N \equiv \lambda z.M$ siendo $x \equiv z$ o cuando x ocurre ligada en M
2) $N \equiv M\,P$ donde x ocurre ligada en M y/o en P

Ejemplos:

- **λz.x y** — **x** libre, **y** libre
- **λx.x y** — **x** ligada, **y** libre
- **λy.x y** — **x** libre, **y** ligada
- **λx y.x y** — **x** ligada, **y** ligada
- **(λz.z x y) (λx.x)** — **x** libre (operador), **x** ligada (operando), **y** libre, **z** ligada

4. REGLAS DE CONVERSIÓN

En el cálculo lambda existen tres reglas de conversión que permiten transformar una expresión en otra. Como resultado, ambas expresiones denotan lo mismo y son, por lo tanto, equivalentes.

4.1. Regla de conversión Alfa (α)

Consiste en hacer un renombramiento de variable en una abstracción que tiene la forma **λx.M**. Si la variable **y** no ocurre libre en **M**, es posible sustituir por **y** todas las ocurrencias libres de **x** en **M**:

$$\lambda x.M =_\alpha \lambda y.M[y/x]$$

Ejemplos:

1) **λx.x** sustituyendo queda **λy.y**
2) **λx.y x** no se puede aplicar la regla α porque se ligaría la variable **y** que es libre. Con cualquier otra variable que no ocurra libre en **M**, sí se podría usar la regla α: **λz.y z**
3) **λx.z x (λu x.x u) v x** sustituyendo queda **λy.z y (λu x.x u) v y**
4) **λx y.x z y** debe convertirse primero en: **λx u.x z u** y luego en: **λy u.y z u**
5) De la expresión **x (λx.x y) (λy.z y)** pueden obtenerse:
   - **x (λt.t y) (λu.z u)** Obs: Solo esta primera sigue la convención de Barendregt
   - **x (λt.t y) (λy.z y)**
   - **x (λu.u y) (λu.z u)**
   - **x (λx.x y) (λx.z x)**

> Convención de Barendregt:
> 1) En una expresión, ninguna variable debería aparecer libre y ligada a la vez.
> 2) En diferentes términos no debería haber variables ligadas homónimas.

Pero no puede obtenerse:

- **z (λz.z y) (λy.z y)**

4.2. Regla de conversión Beta (β)

Consiste en realizar la reducción de una β-*redex* (expresión reducible β), que es una aplicación cuyo operador es una abstracción. Es decir, una β-*redex* es una aplicación que tiene la forma **(λx.M) N**.

Ejemplos:

- **(λx.(λu.u) (λv.x v)) ((λt.t t) w)** Una β-*redex* que contiene otras dos
- **(λx.x x) (λy.y y)** Una β-*redex*
- **(λx.x) (λy.y y) z** Una β-*redex*
- **x (λy.y y)** Ninguna β-*redex*

Una expresión lambda que no contiene ninguna β-*redex* está en *forma normal*. Mientras no se llegue a la forma normal, puede seguir aplicándose la regla de conversión Beta, que consiste en sustituir por **N** todas las ocurrencias libres de **x** en **M**:

$$(\lambda x.M)\ N =_\beta M[N/x]$$

Ejemplos:

- **(λu.u z u) a**
  =β a z a
- **(λx y.y x) y a**
  =α (λx z.z x) y a Obs: Es obligatorio para evitar que **y** se ligue al entrar
  =β (λz.z y) a
  =β a y
- **(λx.(λu.u) (λv.x v)) ((λt.t t) w)**
  =β (λu.u) (λv.(λt.t t) w v)
  =β λv.(λt.t t) w v
  =β λv.w w v
- **(λt.z) ((λx.x x) (λy.y y))**
  =β z
- **(λx.x x) (λy.y y)** Obs: No termina nunca. No tiene forma normal.

4.3. Regla de conversión Eta (η)

Consiste en realizar la reducción de una η-*redex* (expresión reducible η), que es una abstracción que tiene la forma **λv.M v** y en la cual **v** no ocurre libre en **M**.

La regla de conversión Eta establece que:

$$\lambda v.M\ v =_\eta M$$

Ejemplos:

- **(λx v.x v) ((λt.t t) w) =β λv.((λt.t t) w) v =β λv.w w v =η w w**
- **(λv.w x y v) z =η w x y z** Obs: También **(λv.w x y v) z =β w x y z**
- **λx.x t x** Obs: **λx.M x** no es una η-*redex* con **M ≡ x t** porque **x** es libre en **M**

5. ESTRATEGIAS DE REDUCCIÓN

La reducción de una expresión a su *forma normal* (si esta existe) puede hacerse de varias maneras. Hay disponibles herramientas que permiten seguir, paso a paso, diversas estrategias de reducción:

```
http://kdlcj.gitlab.io/lambda
http://projectultimatum.org/cgi-bin/lambda
http://www.cburch.com/dev/lambda/index.html
http://www.math.cmu.edu/~wgunther/lamred.html
```

A continuación se describen cuatro de las estrategias más conocidas.

5.1. *Call-by-name*

Consiste en ir reduciendo siempre la β-*redex* más externa desde la izquierda y que no esté ubicada dentro de una abstracción lambda, hasta llegar a una expresión en forma normal de cabecera (*head normal form*, una expresión sin β-*redex* en su inicio), que no siempre coincide con la forma normal. Ejemplo:

- **(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)**
  =β (λz.z) x (λt.t) ((λy.y) ((λz.z) x))
  =β x (λt.t) ((λy.y) ((λz.z) x))

5.2. Orden normal

Consiste en ir reduciendo siempre la β-*redex* más externa desde la izquierda. Ejemplo:

- **(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)**
  =β (λz.z) x (λt.t) ((λy.y) ((λz.z) x))
  =β x (λt.t) ((λy.y) ((λz.z) x))
  =β x (λt.t) ((λz.z) x)
  =β x (λt.t) x

Si existe la forma normal de una expresión lambda, esta estrategia siempre permite llegar a ella.

5.3. *Call-by-value*

Consiste en ir reduciendo siempre la β-*redex* más interna desde la izquierda y que no esté ubicada dentro de una abstracción lambda. Ejemplo:

- **(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)**
  =β (λu.u (λt.t) ((λy.y) u)) x
  =β x (λt.t) ((λy.y) x)
  =β x (λt.t) x

Aunque una expresión lambda tenga forma normal, esta estrategia no siempre permite llegar a ella.

5.4. Orden aplicativo

Consiste en ir reduciendo siempre la β-*redex* más interna desde la izquierda. Ejemplo:

- **(λu.u (λt.t) ((λy.y) u)) ((λz.z) x)**
  =β (λu.u (λt.t) u) ((λz.z) x)
  =β (λu.u (λt.t) u) x
  =β x (λt.t) x

Aunque una expresión lambda tenga forma normal, esta estrategia no siempre permite llegar a ella.

5.5. Comparación de estrategias

Mediante algunos ejemplos se muestran a continuación las características de las estrategias vistas.

La siguiente expresión lambda no tiene forma normal:

(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x))

Call-by-name	**Orden normal**
(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x)) =β (λt.t) ((λy.y) w) ((λz.z z) (λx.x x)) =β (λy.y) w ((λz.z z) (λx.x x)) =β w ((λz.z z) (λx.x x)) Se llega a la forma normal de cabecera	(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x)) =β (λt.t) ((λy.y) w) ((λz.z z) (λx.x x)) =β (λy.y) w ((λz.z z) (λx.x x)) =β w ((λz.z z) (λx.x x)) =β w ((λx.x x) (λx.x x)) ... Se produce un ciclo infinito
Call-by-value	**Orden aplicativo**
(λu.(λt.t) ((λy.y) w)) ((λv.v) r) ((λz.z z) (λx.x x)) =β (λu.(λt.t) ((λy.y) w)) r ((λz.z z) (λx.x x)) =β (λt.t) ((λy.y) w) ((λz.z z) (λx.x x)) =β (λt.t) w ((λz.z z) (λx.x x)) =β w ((λz.z z) (λx.x x)) =β w ((λx.x x) (λx.x x)) ... Se produce un ciclo infinito	(λu.(λt.t) ((λy.y) (w))) ((λv.v) r) ((λz.z z) (λx.x x)) =β (λu.(λt.t) w) ((λv.v) r) ((λz.z z) (λx.x x)) =β (λu.w) ((λv.v) r) ((λz.z z) (λx.x x)) =β (λu.w) r ((λz.z z) (λx.x x)) =β w ((λz.z z) (λx.x x)) =β w ((λx.x x) (λx.x x)) ... Se produce un ciclo infinito

La siguiente expresión lambda sí tiene forma normal:

(λu.w ((λy.y) t)) ((λz.z z) (λx.x x))

Call-by-name	**Orden normal**
(λu.w ((λy.y) t)) ((λz.z z) (λx.x x)) =β w ((λy.y) t) Se llega a la forma normal de cabecera	(λu.w ((λy.y) t)) ((λz.z z) (λx.x x)) =β w ((λy.y) t) =β w t Se llega a la forma normal
Call-by-value	**Orden aplicativo**
(λu.w ((λy.y) t)) ((λz.z z) (λx.x x)) =β (λu.w ((λy.y) t)) ((λx.x x) (λx.x x)) ... Se produce un ciclo infinito	(λu.w ((λy.y) t)) ((λz.z z) (λx.x x)) =β (λu.w t) ((λz.z z) (λx.x x)) =β (λu.w t) ((λx.x x) (λx.x x)) ... Se produce un ciclo infinito

Conclusiones: El orden normal es el más potente porque siempre encuentra la forma normal cuando esta existe. *Call-by-value* y el orden aplicativo son menos potentes, aunque a veces pueden ser más eficientes (por requerir menos pasos). *Call-by-name* solo llega a la forma normal de cabecera.

6. REPRESENTACIÓN DE VALORES DE VERDAD. FUNCIONES LÓGICAS

Mediante abstracciones es posible definir representaciones de los valores *verdadero* y *falso*, y funciones lógicas aplicables sobre ellos.

Una condición de la forma *si* **p** *entonces* **q**; *si no* **r** se representa en cálculo lambda de la siguiente manera:

If = λp.λq.λr.p q r

Al utilizar la regla de conversión Beta en una aplicación que tenga **If** como operador y tres operandos (por ejemplo, **a**, **b** y **c**), se obtendrá otra aplicación que tendrá **a** como operador y dos operandos (**b** y **c**). Si **a** representa el valor *verdadero*, al aplicar nuevamente la regla de conversión Beta, deberá obtenerse **b**. En cambio, si **a** representa el valor *falso*, deberá obtenerse **c**. Para ello, se adoptan las siguientes representaciones de los valores de verdad:

True = λx.λy.x (Selección del primero de dos operandos)

False = λx.λy.y (Selección del segundo de dos operandos)

En efecto, la reducción de **If True b c** da **b**, y la de **If False b c** da **c**:

(λp.λq.λr.p q r) (λx.λy.x) b c
=β (λq.λr.(λx.λy.x) q r) b c
=β (λr.(λx.λy.x) b r) c
=β (λx.λy.x) b c
=β (λy.b) c
=β b

(λp.λq.λr.p q r) (λx.λy.y) b c
=β (λq.λr.(λx.λy.y) q r) b c
=β (λr.(λx.λy.y) b r) c
=β (λx.λy.y) b c
=β (λy.y) c
=β c

La negación se representa mediante la siguiente abstracción:

Not = λp.p False True

Es sencillo verificar que **Not True** se reduce a **False** y **Not False** se reduce a **True**:

(λp.p (λx.λy.y) (λx.λy.x)) (λx.λy.x)
=β (λx.λy.x) (λx.λy.y) (λx.λy.x)
=β (λy.λx.λy.y) (λx.λy.x)
=β λx.λy.y

(λp.p (λx.λy.y) (λx.λy.x)) (λx.λy.y)
=β (λx.λy.y) (λx.λy.y) (λx.λy.x)
=β (λy.y) (λx.λy.x)
=β λx.λy.x

Las 16 posibles funciones lógicas diádicas (*conectivos*) se pueden definir mediante abstracciones. Por ejemplo, la conjunción, la disyunción y la disyunción exclusiva se definen así:

Conjunción:

And = λp.λq.p q False

Disyunción:

Or = λp.λq.p True q

Disyunción exclusiva:

Xor = λp.λq.p (q False True) q

7. REPRESENTACIÓN DE NÚMEROS. FUNCIONES NUMÉRICAS Y RELACIONALES

También mediante abstracciones se pueden definir *numerales* (representaciones de números) y funciones numéricas y relacionales aplicables sobre ellos.

El número **0** se representa así:

```
0 = λf.λx.x
```

La función **Succ** se define de la siguiente manera:

```
Succ = λn.λf.λx.f (n f x)
```

La representación del número **1** se obtiene reduciendo la aplicación **Succ 0**:

```
(λn.λf.λx.f (n f x)) (λf.λx.x)
=β λf.λx.f ((λf.λx.x) f x)
=β λf.λx.f ((λx.x) x)
=β λf.λx.f x
```

El numeral **2** se obtiene reduciendo la aplicación **Succ (Succ 0)** o **Succ 1**, y así sucesivamente:

```
0 = λf.λx.x
1 = λf.λx.f x
2 = λf.λx.f (f x)
3 = λf.λx.f (f (f x))
4 = λf.λx.f (f (f (f x)))
5 = λf.λx.f (f (f (f (f x))))
```

Puede verificarse que al reducir la aplicación **Succ 3** se obtiene el numeral **4**:

```
(λn.λf.λx.f (n f x)) (λf.λx.f (f (f x)))
=β λf.λx.f ((λf.λx.f (f (f x))) (f) x)
=β λf.λx.f ((λx.f (f (f x))) (x))
=β λf.λx.f (f (f (f x)))
```

Algunas funciones numéricas que se pueden utilizar con los numerales son las siguientes:

Predecesor de un número:

```
Pred = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
```

Suma de dos números:

```
Add = λm.λn.λf.λx.m f (n f x)
```

Resta de dos números:

```
Sub = λm.λn.n Pred m
```

Producto de dos números:

```
Mul = λm.λn.λf.λx.m (n f) x
```

Potencia de dos números (base y exponente):

```
Pow = λm.λn.λf.λx.n m f x
```

Enésimo término de la sucesión de Fibonacci:

```
Fibo = λn.n (λf.λa.λb.f b (Add a b)) (λx.λy.x) (λf.λx.x) (λf.λx.f x)
```

Las funciones relacionales que se pueden utilizar con los numerales son las siguientes:

Evaluar si un número es igual a cero:

```
IsZero = λn.n (λz.(λx.λy.y)) (λx.λy.x)
```

Por ejemplo, la reducción de **IsZero 4** da **False**:

```
(λn.n (λz.λx.λy.y) (λx.λy.x)) (λf.λx.f (f (f (f x))))
=β (λf.λx.f (f (f (f x)))) (λz.λx.λy.y) (λx.λy.x)
=β (λx.(λz.λx.λy.y) ((λz.λx.λy.y) ((λz.λx.λy.y) ((λz.λx.λy.y) x)))) (λx.λy.x)
=β (λz.λx.λy.y) ((λz.λx.λy.y) ((λz.λx.λy.y) ((λz.λx.λy.y) (λx.λy.x))))
=β λx.λy.y
```

En cambio, la reducción de **IsZero 0** da **True**:

```
(λn.n (λz.λx.λy.y) (λx.λy.x)) (λf.λx.x)
=β (λf.λx.x) (λz.λx.λy.y) (λx.λy.x)
=β (λx.x) (λx.λy.x)
=β λx.λy.x
```

Evaluar si un número es menor o igual que otro:

```
Lte = λx.λy.Iszero (Sub x y)
```

Evaluar si un número es mayor o igual que otro:

```
Gte = λx.λy.Lte y x
```

Evaluar si un número es menor que otro:

```
Lt = λx.λy.Not (Gte x y)
```

Evaluar si un número es mayor que otro:

```
Gt = λx.λy.Not (Lte x y)
```

Evaluar si dos números son iguales:

```
Eq = λx.λy.And (Lte x y) (Gte x y)
```

Evaluar si dos números son distintos:

```
Ne = λx.λy.Not (Eq x y)
```

8. COMBINADORES

Las expresiones lambda que no contienen ninguna variable libre se denominan *combinadores*. Algunos combinadores de uso frecuente tienen designaciones propias, por ejemplo los tres que dan nombre al *cálculo de combinadores* **SKI** (el cual es un tipo de *lógica combinatoria*):

S = λx.λy.λz.x z (y z)	Función de fusión (*Verschmelzungsfunktion*)
K = λx.λy.x	Función de constancia (*Konstanzfunktion*)
I = λx.x	Función de identidad *(Identitätsfunktion*)

Todo el cálculo lambda puede reformularse en el cálculo de combinadores **SKI**, que tiene una sola operación básica: la aplicación. Al ser equivalente al cálculo lambda, el cálculo de combinadores **SKI** es lo suficientemente potente como para codificar cualquier función computable. Los combinadores **S** y **K** se denominan *combinadores estándar*, ya que con ellos es posible representar todos los demás. Por ejemplo, **I = S K K**.

En efecto, al reducir **S K K** se obtiene **I:**

```
(λx.λy.λz.x z (y z)) (λx.λy.x) (λx.λy.x)
=β  (λy.λz.(λx.λy.x) z (y z)) (λx.λy.x)
=β  λz.(λx.λy.x) (z) ((λx.λy.x) z)
=β  λz.(λy.z) ((λx.λy.x) z)
=β  λz.z
```

Otros combinadores muy conocidos son los siguientes:

```
B = λx.λy.λz.x (y z)
C = λx.λy.λz.x z y
D = λx.λy.λz.λv.x y (z v)
J = λx.λy.λz.λv.x y (x v z)
M = λx.x x
O = λx.λy.y
Q = λx.λy.λz.y (x z)
R = λx.λy.λz.y z x
V = λx.λy.λz.z x y
W = λx.λy.x y y
Y = λf.(λx.f (x x)) (λx.f (x x))          Obs: Definido por Curry
Θ = (λx.λy.y (x x y)) (λx.λy.y (x x y))   Obs: Definido por Turing
Ω = (λx.x x) (λx.x x)
```

Las funciones recursivas se deben representar usando algún operador de punto fijo, como **Y** o **Θ**. Por ejemplo, las funciones *factorial* y *cociente* se definirían así:

```
Fact = Y (λf.λx.If (Iszero x) 1 (Mul x (f (Pred x))))

 Div = λn.(Y (λc.λn.λm.λf.λx.(λd.If (IsZero d) (0 f x)
       (f (c d m f x))) (Sub n m))) (Succ n)
```

9. PARES Y LISTAS

Las abstracciones permiten definir *pares ordenados* y *listas*. Un par ordenado está compuesto de dos elementos, denominados *primero* y *segundo*.

El par ordenado **(a . b)** se representa así:

(a . b) = λs.s a b

La función **Pair** utilizada para construir pares ordenados se define de la siguiente manera:

Pair = λx.λy.λs.s x y

Las funciones **First** y **Second** devuelven, respectivamente, los elementos *primero* y *segundo* de un par ordenado:

First = λp.p True

Second = λp.p False

Por ejemplo:

First (Pair p q)
(λp.p (λx.λy.x)) ((λx.λy.λs.s x y) p q)
=β (λp.p (λx.λy.x)) ((λy.λs.s p y) q)
=β (λp.p (λx.λy.x)) (λs.s p q)
=β (λs.s p q) (λx.λy.x)
=β (λx.λy.x) p q
=β (λy.p) q
=β p

Second (Pair p q)
(λp.p (λx.λy.y)) ((λx.λy.λs.s x y) p q)
=β (λp.p (λx.λy.y)) ((λy.λs.s p y) q)
=β (λp.p (λx.λy.y)) (λs.s p q)
=β (λs.s p q) (λx.λy.y)
=β (λx.λy.y) p q
=β (λy.y) q
=β q

Una lista puede estar vacía (es un par ordenado cuyo elemento *primero* tiene el valor **True**) o puede estar formada por un par ordenado cuyo elemento *primero* tiene el valor **False** y su elemento segundo es un par ordenado con dos valores: *cabeza* y *cola*.

La lista vacía se define así:

Nil = Pair True True Obs: Es más recomendable usar **Nil = λz.z**

La lista **(a b c)** se representa así:

(False . (a . (False . (b . (False . (c . (λz.z)))))))

La función para verificar si una lista está vacía es la siguiente:

Null = First

Una aplicación cuyo operador sea la función **Null** se reduce a **True** si el operando es **Nil** o a **False** si el operando es cualquier par ordenado cuyo elemento *primero* tenga el valor **False**.

Para construir un nodo de una lista, indicando la cabeza **x** y la cola **y**, se define la función **Cons**:

Cons = λx.λy.Pair False (Pair x y)

Las funciones **Head** y **Tail** devuelven, respectivamente, los elementos *cabeza* y *cola* de una lista:

Head = λz.First (Second z)

Tail = λz.Second (Second z)

Por ejemplo:

(Head (Tail (Tail (Cons a (Cons b (Cons c Nil)))))) da **c**.