Juan Francisco Cisneros

Explique en qué consiste el cálculo lambda. ¿Cómo funciona la notación de tipos de funciones?

El cálculo lambda es un sistema formal que fue desarrollado por Alonzo Church y Stephen Cole Kleene en la década de los años 30 para estudiar la computabilidad y la recursividad. Fue creado como parte de la crisis de los fundamentos matemáticos, donde Church buscaba una alternativa a la lógica de primer orden para demostrar el Problema de la Decisión (Entscheidungsproblem). Hoy se lo ve como la base teórica de los lenguajes de programación más importantes y fundamentales, sirviendo como el núcleo matemático que permite el análisis de la herencia de tipos, la inferencia de tipos y la ejecución de programas funcionales.

El cálculo lambda se basa en tres componentes sintácticos y dos reglas de cómputo. Todo término en el cálculo lambda se define de manera inductiva de la siguiente forma:

- Variables: Un conjunto infinito de variables $x, y, z, \ldots$
- Abstracción: Si x es una variable y M es un término, entonces $\lambda x.M$ representa una función anónima que toma un argumento x y devuelve M.
- Aplicación: Si M y N son términos, entonces $(M\ N)$ representa la aplicación de la función M a el argumento N.

Por convención, el operador de aplicación asocia a la izquierda y el operador de abstracción se extiende lo más a la derecha posible. Por ejemplo, $\lambda x.x\ y\ z$ se interpreta como $\lambda x.((x\ y)\ z)$ y no como $(\lambda x.x\ y)\ z$.

El cálculo lambda opera bajo dos reglas fundamentales que definen cómo se ejecuta el cómputo:

- Alpha (α) Conversión: Permite renombrar las variables ligadas sin cambiar el significado de la función. Por ejemplo, $\lambda x.x$ es idéntico a $\lambda y.y$.
- Beta (β) Reducción: Es el motor de cómputo. Si tenemos una función $\lambda x.M$ aplicada a un argumento N, el resultado es el término M donde todas las apariciones de x han sido reemplazadas por N. Formalmente: $(\lambda x.M\ N) \to M[x := N]$.

Para que el cálculo lambda sea útil en la ingeniería de software y no caiga en la paradoja de Russell, se le impone un sistema de tipos. La notación de tipos de funciones $A \to B$ indica que la función toma un argumento de tipo A y devuelve un resultado de tipo B. Este sistema garantiza que el programa no se quede atrapado en bucles infinitos y asegura la terminación del cómputo.

¿Qué es una expresión lambda? Ejemplo en Haskell, Python y C++.

Una expresión lambda es una función anónima que puede ser pasada como cualquier tipo de dato en lenguajes orientados a objetos o funcionales. Estas funciones toman argumentos, realizan una operación y devuelven el valor. En lenguajes de programación funcional, por lo tanto son ciudadanos de primera clase. Esto significa que se pueden almacenar en variables, pasar como argumentos a otras funciones o ser devueltas como resultados de otras funciones.

Haskell

```
double x = x + x -- En Haskell, una función es simplemente el resultado de la cada de x con x
```

Ejemplo de cómo una expresión lambda puede ser pasada como cualquier tipo de dato

```
double :: Integer -> Integer
double x = x + x
main = print (map double [1, 2, 3, 4, 5])
```

Ejemplo de cómo una función puede ser devuelta como resultado de otra función

```
double :: Integer -> Integer
double x = x + x
triple :: Integer -> Integer
triple x = x + x + x
main = print (triple 5)
```

Python

```
double = lambda x: x + x -- En Python la función double, que recibe un argumento x y devuelve el resultado de la cada de x con x
```

Ejemplo de cómo una función puede ser devuelta como resultado de otra función

```
double :: Integer -> Integer
double x = x + x
triple :: Integer -> Integer
triple x = x + x + x
main = print (triple 5)
```

C++

```
auto double = [](int x) { return x + x; }; // En Python la función double, que recibe un argumento x y devuelve el resultado de la cada de x con x
```

Ejemplo de cómo una función puede ser devuelta como resultado de otra función

```
double :: Integer -> Integer
double x = x + x
triple :: Integer -> Integer
triple x = x + x + x
main = print (triple 5)
```

Explique en qué consiste un list comprehension en Haskell. (Incluya ejemplos en Haskell y Python).

En las matemáticas, la "list comprehension" es una notación que nos ayuda a construir conjuntos a partir de conjuntos ya existentes, así también lo "Zfrmlo" (ZF) introdujo el infinito (CALFRAI.09).

En Haskell, la "list comprehension" es una forma de definir listas, donde se puede especificar una lista por comprensión de la siguiente manera:

```
[ f x | x <- xs, p x ]
[ f x | x <- xs ]
```

En Python, la "list comprehension" es una forma de definir listas, donde se puede especificar una lista por comprensión de la siguiente manera:

```
import math
[ math.sqrt(x) for x in range(1,101) if for x in range(1,101)
[ x for x in range(1,101)
[ f(x,y,z) for x in range(1,101)
```

Explique las diferentes tipos de recursión que existen. (Incluya ejemplos distintos a los del libro).

En Haskell, las funciones pueden ser definidas en términos de otras funciones, pero también como en los lenguajes de programación que consumen una función puede ser definida en términos de otras que hacen lo mismo, a la que llamamos recursividad. Por ejemplo, la función **fibb** puede ser definida como:

```
fibb 1 = 1
fibb 2 = 1
fibb n = fibb (n-1) + fibb n
main = print (fibb 5)
main = print (fibb 10)
main = print (fibb 15)
main = print (fibb 20)
main = print (fibb 25)
main = print (fibb 30)
main = print (fibb 35)
main = print (fibb 40)
main = print (fibb 45)
main = print (fibb 50)
```

Como también por ejemplo la **recursividad mutua**, que es definida como la recursión en otras dos funciones, por ejemplo la función **parOdd** puede ser definida como:

```
parOdd :: Integer -> Bool
parOdd n = (n `mod` 2 == 0)
parOdd n = (n `mod` 2 == 1)
```

```haskell
minimax (Node g []) | wins O g  = Node (g,O) []
                    | wins X g  = Node (g,X) []
                    | otherwise = Node (g,B) []
minimax (Node g ts) | turn g == O = Node (g, minimum ps) ts'
                    | turn g == X = Node (g, minimum ps) ts'
                                    where
                                         ts' = map minimax ts
                                         ps  = [p | Node (_,p) _ <- ts']
bestmove :: Grid -> Player -> Grid
bestmove g p = head [g' | Node (g',p') _ <- ts, p' == best]
               where
                    tree = prune depth (gametree g p)
                    Node (_, best) ts = minimax tree


main :: IO ()
main = do hSetBuffering stdout NoBuffering
          play empty O

play :: Grid -> Player -> IO ()
play g p = do cls
              goto (1,1)
              putGrid g
              play' g p

play' :: Grid -> Player -> IO ()
play' g p | wins O g = putStrLn "Player O wins!\n"
          | wins X g = putStrLn "Player X wins!\n"
          | full g   = putStrLn "It's a draw!\n"
          | p == O   = do i <- getNat (prompt p)
                          case move g i p of
                            []   -> do putStrLn "Error: Invalid move"
                                       play' g p
                            [g'] -> do play g' (next p)
          | p == X   = do putStr "Player X is thinking..."
                          (play $! (bestmove g p)) (next p)
```

MADE WITH Codye

Referencias

- Wikiwand - Cálculo lambda. (s. f.). Wikiwand. https://www.wikiwand.com/es/C%C3%A1lculo_lambda#introduction
- Hutton, G. (2007b). Programming in Haskell. https://doi.org/10.1017/cbo9780511813672
- Mpri. (s. f.). GitHub - MPRI/M2-4-2: Functional Programming and Type Systems. GitHub. https://github.com/MPRI/M2-4-2/tree/master
- Pierce, B. C. (2002). Types and programming languages. https://alliance.seas.upenn.edu/~bcpierce/tapl/contents.pdf
- Najoj. (s. f.). GitHub - Najoj/Programming-Haskell. GitHub. https://github.com/Najoj/programming-haskell