Introducción al cálculo lambda usando Racket

Technical Report · December 2019

DOI: 10.13140/RG.2.2.26552.29447

1 author:

Camilo Chacón Sartori
Pontificia Universidad Católica de Valparaíso
1 PUBLICATION **0** CITATIONS

SEE PROFILE

Introducción al cálculo lambda usando Racket*

Camilo Chacón Sartori
camilochs@gmail.com

Actualizado: January 6, 2020

El cálculo lambda es un notación formal que permite expresar funciones computables. El cual es el fundamento de la programación funcional. Se define con la letra Griega lambda (λ) y se expresa a través de expresiones lambda, y términos lambda que son usados para representar *binding variables*[1] dentro de una función. Este documento pretender ser una introducción básica a los aspectos teóricos y prácticos del cálculo lambda, y a la programación funcional. Para esto último usaremos el lenguaje de programación Racket.

1 Introducción

1.1 Historia

El cálculo lambda (cálculo-λ) fue introducido por Alonzo Church en su artículo: «An unsolvable problem of elementary number theory» (Church, 1936) en la década de 1930. Además fue supervisor de destacados alumnos como Alan Turing y Stephen Kleen en Princeton. El primero, propuso un modelo para realizar computación conocido como *a-machines* (*automatic machines*)[2] en su artículo «On Computable Numbers» (Turing, 1936). Y el segundo, mostraría la equivalencia de los dos modelos en su trabajo sobre la teoría de la recursividad. Dando paso a dos modelos distintos (pero equivalentes) de realizar la computación que actualmente se suele llamar: «tesis de Church-Turing», y fundando así lo que hoy en día se conoce como ciencia de la computación.

1.2 Aplicación

Años después, en la década de 1950 John McCarthy crearía Lisp que es uno de los primeros lenguajes de programación. Que a pesar de no haber sido basado en el cálculo-λ, el mismo John McCarthy dijo: «Lisp [...] no fue un intento de llevar a la práctica el cálculo lambda, aunque si alguien hubiera empezado con esa intención, podría haber terminado con algo como LISP.» (Szmulewicz; Wexelblat, 1978). Es así, como nace el paradigma de programación funcional que trabaja con *máquinas de reducción* (Barendregt and Barendsen (1984)). Por otro lado, el trabajo de Turing derivaría en el paradigma de programación imperativo.

Este documento pretende ser una breve introducción al cálculo lambda con un enfoque

[1] Esto significaría «variables vinculantes» o «variables enlazadas» pero prefiero mantener en este documento el concepto en su idioma original para evitar confusiones.

[2] Actualmente se le denomina máquina de Turing.

aplicado usando un lenguaje de programación funcional llamado Racket (un lenguaje que es derivado de Lisp y, a su vez de Scheme).

2 Descripción

2.1 Preliminar

Todo en cálculo-λ se basa en funciones. Por lo tanto, primero debemos definir que es una función matemática. Una función se puede definir de la siguiente manera: $f : a \to b$, donde la función f recibe un valor a y retorna un b. Además a y b pueden ser representados como conjuntos de elementos. Así, la función es una relación entre elementos de dos conjuntos. Para nuestro ejemplo el conjunto a se le llama **dominio** y el conjunto b **codomino**, que representan al conjunto de entrada y salida respectivamente. En notación conjuntista está misma función puede ser definida de la siguiente manera: $f(a) = b$.

Un ejemplo puede ser una función f que recibe un número entero que incrementa en 1. Entonces, $f(x) = x + 1$, donde a x se le asigna el valor 5, es decir: $f(5) = 5 + 1$ retornando 6.

2.2 Motivación

El cálculo lambda a diferencia con las funciones matemáticas trata con funciones *anónimas*, es decir, no existe la necesidad de asignarle un nombre explicito a una función, por ejemplo:

$$sumar\_numeros(x, y) \to x + y \tag{1}$$

Se puede escribir de manera anónima:

$$(x, y) \to x + y \tag{2}$$

Cada valor de entrada está **asociado** a su valor de salida.

Algo importante a mencionar es que el cálculo-λ solo admite funciones con un solo valor de entrada. Cuando queremos tratar con múltiples argumentos entonces se debe utilizar una propiedad conocida como *currying*, el cual transforma los argumentos de entrada en una cadena de funciones con un solo valor. En otras palabras, una función $f(x, y, z) = x + y + z$ que tiene tres argumentos puede ser reescrita de la siguiente forma: $(x \to (y \to (z \to (x + y + z)))$, donde el operador $\to$ asume una asociación a la derecha.

La función se puede evaluar usando *aplicación de funciones*. Hace referencia a **aplicar** un conjunto de argumentos desde el dominio para obtener el valor de retorno desde el codominio, en término de programación se diría **ejecutar** una función. Entonces la evaluación –usando reducción de expresiones– sería la siguiente:

$$\begin{aligned}
&((\underline{x}, \underline{y}, \underline{z}) \to x + y + z)(1, 2, 3) \\
&= ((1, 2, 3) \to x + y + z) \\
&= ((1, 2, 3) \to 1 + 2 + 3) \\
&= (1 + 2 + 3) \\
&= 6
\end{aligned}$$

Su equivalente en currying:

$$\begin{aligned}
&((x \to (y \to (z \to x + y + z)))(1)(2))(3) \\
&= (y \to (z \to 1 + y + z))(2))(3) \\
&= (z \to 1 + 2 + z)(3) \\
&= (1 + 2 + 3)(3) \\
&= (1 + 2 + 3) \\
&= 6
\end{aligned}$$

Intuitivamente nos podemos dar cuenta que se va aplicando una substitución (= reemplazo) de las variables a la izquierda (x, y, z) por los valores

a la derecha $(1, 2, 3)$, que están dentro de cada expresión[3].

2.3 El cálculo lambda

El cálculo-λ se puede expresar en un simple lenguaje usando la notación *Backus-Naur form* (que dicho sea de paso, cualquier lenguaje de programación se puede expresar con está notación):

⟨*Expr*⟩ ::= 'λ' ⟨*Var-list*⟩ '**.**' ⟨*Expr*⟩
  | ⟨*App-term*⟩

⟨*App-term*⟩ ::= ⟨*App-term*⟩ ⟨*Item*⟩
  | ⟨*Item*⟩

⟨*Item*⟩ ::= ⟨*var*⟩
  | (⟨*expr*⟩)

⟨*Var-list*⟩ := ⟨*var*⟩
  | ⟨*var*⟩ '**,**' ⟨*Var-list*⟩

⟨*var*⟩ := [a-z]+

Entonces según el lenguaje anteriormente definido, las siguientes **expresiones lambda** son validas:

$$\lambda x.x \tag{3}$$

$$\lambda x, y.x \tag{4}$$

$$\lambda x, y.y \tag{5}$$

$$\lambda x, y.x\ (y) \tag{6}$$

$$\lambda x, y, z.x\ (y\ (z)) \tag{7}$$

Un ejercicio para entender que son expresiones válidas radica en verlas como una secuencia de caracteres e ir clasificándolas con el lenguaje anteriormente descrito. Te darás cuenta que cumple con las reglas sintácticas.

Por el contrario, algunas expresiones invalidas pueden ser: $\lambda\lambda$, $x.x$, $\lambda..$, o $\lambda.x$. Ahora veremos como podemos implementar dichas expresiones en un lenguaje de programación.

Importante: La versión original de cálculo-λ propuesta por Alonzo Church es libre de tipos (*free-type*), esto significa que, no existe diferencia entre tipos de datos como en algunos lenguajes de programación estáticamente tipado como C++ que posee: int, double, char, string, etc. También es importante decir que, en Racket por defecto es *free-type* pero puedes usar una versión modificada del lenguaje que si admite tipos [4], lo mismo ocurre con el cálculo-λ que tiene una versión con tipos. En este documento usaremos la versión *free-type* en ambos casos.

Ejemplos en Racket. Las expresiones previas podemos probarlas usando el lenguaje de programación Racket. Para eso, primero debemos tener instalado el entorno de desarrollo Dr-Racket[5]. A continuación, podemos ver la lista de expresiones validas representada en código:

```
;; Expr. (3)
(define id
    (lambda (x)
        x))

;; Expr. (4)
(define k
    (lambda (x y)
        x))

;; Expr. (5)
(define k2
    (lambda (x y)
```

[3] Puedes probar el cálculo-λ en un interprete online: `https://jacksongl.github.io/files/demo/lambda/index.htm`

[4] `https://docs.racket-lang.org/ts-guide`

[5] Toda la información al respecto lo puedes encontrar en: `https://download.racket-lang.org/`.

```
        y))

;; Expr. (6)
(define f
    (lambda (x y)
        x y))

;; Expr. (7)
(define f
    (lambda (x y z)
        (x y z) ))
```

Una definición de una función en Racket comienza con la palabra reservada *define*, posteriormente se usa la palabra reservada *lambda* que es equivalente a λ. Por lo tanto, como se puede apreciar el código en Racket es similar al cálculo-λ.

Sin embargo, –si el lector es cuidadoso– se habrá dado cuenta que existen dos diferencias principales: (1) en nuestros ejemplos si debemos definir un nombre a la función (aunque eso no significa que Racket no acepte funciones anónimas); (2) la sintaxis en Racket se construye con la notación *expresión-S* que está basada en paréntesis (ver aquí[6] para más detalles).

Una vez definidas dichas funciones, podemos ejecutarlas (evaluarlas) en el entorno de programación:

```
> (id 1)
1

> (k 1 2)
1

> (k2 1 2)
2

> (f id 1)
1

> (f k2 1 2)
2
```

La ejecución de una función en Racket se realiza a través de paréntesis seguido por los argumentos separados por espacios, por ejemplo: (nombre_de_función a_1 a_2 ... a_n). Puede, a primera vista, parecer raro si vienes de lenguajes imperativos como C++ o Python, pero créeme, es algo muy divertido y tu visión de la programación podría cambiar (aunque no puedo asegurar que sea para bien, ¡no soy el culpable del resultado!).

2.3.1 Operadores

Toda expresión lambda válida es le suele llamar «término lambda». A través de estos términos podemos derivar algunos importantes operadores de cálculo-λ:

- Abstracción: Es un operador de abstracción, por ejemplo, si t es un término lambda y x es una variable, entonces la siguiente función anónima $(\lambda x.t)$ es un término lambda. Se le llama *binding variables* a la variable x en el término t. Notación formal: Se dice que un operador es de abstracción, si $T \equiv T[x]$ es un término que depende de x. Entonces $\lambda x.T[x]$ equivale a $x \to T[x]$. Por ejemplo: $(\lambda x.1 * x + 2)\mathbf{10} = 1 * \mathbf{10} + 2 = 12$.

- Aplicación: Es un operador de aplicación, si por ejemplo, t y k son términos lambdas, entonces $(t\ k)$ es un término lambda. Esto significa que tenemos una función t con argumento k. (Podemos ver aquí que el operador aplicación es similar a la manera que se ejecuta una función en Racket.) Notación formal: Si tenemos un término

[6] https://es.wikipedia.org/wiki/Expresión_S

$(\lambda x.T)V = T[x := V]$ donde $[x := V]$ es una *substitución* de V por x. Por ejemplo: $(\lambda x.x)1 = 1$.

Entonces, podemos concluir que cada operador también es un término lambda.

Importante: Hasta aquí el lector se pudo percatar de algo que es fundamental en la programación funcional: la inmutabilidad, la cual se refleja en los operadores de cálculo-λ que, por ejemplo, realizado una substitución esta es una operación atómica, es decir, posterior a realizarse no es posible modificar dicho valor.

Antes de continuar es importante aclarar el concepto de «variable» en el cálculo-λ, –el cual dicta mucho de ser equivalente a lo conocido en un lenguaje de programación imperativo–, por ejemplo el siguiente término: $\lambda x.x + y$, en el cálculo-λ trata a y como una variable que todavía no ha sido definida, y es totalmente válido. No así, sucede en lenguajes de programación imperativos donde si tenemos una variable z dentro de una función, y esta variable no fue declarada de manera global ni fue definida como argumento, no es valida (ya que de por sí, se asume que no existe).

Por otro lado, se puede utilizar los paréntesis para evitar ambigüedades al momento de definir términos. Por ejemplo, si tenemos un término (1) $\lambda x.((\lambda x.x)x)$ y (2) $(\lambda x.(\lambda x.x))x$. No son iguales. ¿Por qué? El primero, define dos funciones que a su vez define a otra interna que *aplica* y retorna, por ejemplo:

$$\begin{aligned}&\lambda x.((\lambda x.x)x)\\ &= \lambda x.((\lambda x.x)10)\\ &= (\lambda x.10)\end{aligned}$$

Ahora, podemos hacer uso de dicha función resultante y entregarle el valor 2 como argumento de entrada:

$$\begin{aligned}&(\lambda x.10)2\\ &= 10\end{aligned}$$

Este resultado (retorna 10 y descarta el 2) es curioso y es interesante mantenerlo en cuenta, por eso, vea con mucha atención el siguiente ejemplo. En el segundo caso, se define una función que *retorna* otra función y aplica, por ejemplo:

$$\begin{aligned}&(\lambda x.(\lambda x.x))x\\ &= (\lambda x.(\lambda x.x))10\\ &= ((\lambda x.(\lambda x.x))10)\\ &= (\lambda x.x)\end{aligned}$$

Vemos que retorna un función *identidad* (una función identidad es la que retorna siempre el mismo valor de entrada), o sea, ahora si (a diferencia del primer ejemplo) va retornar el mismo valor de entrada:

$$\begin{aligned}&(\lambda x.x)2\\ &= 2\end{aligned}$$

Con esto, nos damos cuenta que los términos lambda pueden operar de manera diferente si utilizamos los paréntesis en un orden distinto. Por eso de su importancia, y que nos hace pensar –de manera implícita– lo cuidadoso que debemos ser a la hora de definir términos. Esta precaución también es equivalente para cuando escribamos algoritmos con Racket.

Algunas consideraciones que cabe señalar:

- Si tenemos, por ejemplo $(\lambda x.x)y$ representa una función identidad *aplicada* a y. Además

una función de aplicación tiene **asociación a la izquierda**, es decir, si tenemos el término «$a\ b\ c$» equivale a: «$(a\ b)\ c$».

- Una función $(\lambda x.y)$ se le considera **constante** y es diferente a la anterior, dado que siempre retornara y independiente del valor a aplicarse a x. En consecuencia, x se descarta.

2.3.2 *Free* y *bound* variables

Son los dos tipos de variables en cálculo-λ; por ejemplo: $\lambda x.x\ y$, la variable x es *bound* y la y es *free*, la substitución $[x := V]$ se aplica a x donde V es un argumento de entrada. Es decir, las variables que no son *bound* (definidas en la función) son *free*; otro ejemplo: «$\lambda xy.x\ y\ z$», donde x e y son *bound* y z es *free*. (Además las variables puede ser un conjunto infinito $\lambda x_1, ..., x_n.(x_1, ..., x_n)$.)

Entonces decimos que una operación de abstracción realiza *bind* a una *bound* o *free* variables en un término lambda.

A continuación presentamos tres ejemplos de variables *bound* y *free* y después lo aplicaremos en Racket.

$$\begin{aligned}&(\lambda\underline{x}.x\ y)[\underline{x} := 1]\\ &= (1\ y)\end{aligned} \tag{8}$$

$$\begin{aligned}&(\lambda\underline{x}.x\ 1)[\underline{x} := (\lambda x.x)]\\ &= (\lambda(\lambda x.x).x\ 1)\\ &= (\lambda x.x)1\\ &= 1\end{aligned} \tag{9}$$

$$\begin{aligned}&(\lambda x.(\lambda y.x\ y)n)[x := (\lambda x.10)]\\ &= (\lambda x.((\lambda y.(x\ y))n))(\lambda x.10)\\ &= (\lambda(\lambda x.10).((\lambda y.(x\ y))n))\\ &= ((\lambda y.((\lambda x.10)\ y))n)\\ &= (\lambda x.10)[x := n]\\ &= 10\end{aligned} \tag{10}$$

Algunas aclaraciones, en (8) la variable x es *bound* y y es *free*; en (9) el argumento es una función de identidad que hace una substitución de x; y en (10) que, a su vez, es el ejemplo más interesante, el argumento es una función constante (cuando una función es argumento de otra, se le llama: *high-order function*[7]), donde n es *free* por tanto descarta a la variable anidada y. (Nuestros amigos programadores imperativos espero, solo espero, que no crean que esto sea magia. Espero.)

Ejemplos en Racket. Ahora utilicemos el conocimiento adquirido y llevémoslo a código. También debemos recordar que Racket esta influenciado por el cálculo-λ, lo que quiere decir que, no es lo mismo. Por tanto hay algunas diferencias que tenemos tener en cuenta.

```
;; Expr. (8)

(define y 1)
(define f
  (lambda (x)
     x y))

;; Aplicación (ejecución)
> (id 2)
1
```

En este ejemplo, ¿cuáles son las diferencias a cálculo-λ?

[7] https://es.wikipedia.org/wiki/Función_de_orden_superior

- En Racket debemos definir previamente las *free* variables, o sino no será posible compilar. Aunque como se ve, no esta definida en el cuerpo de la función *f*.
- El término final: $x\ y$, en Racket, al no estar entre paréntesis, retorna siempre la última variable (a la derecha), por eso en este caso descarta 2 y retorna 1. En cambio, si el término final hubiera sido: $(x\ y)$, se asume un intento de usar operador de aplicación (ejecución de una función), lo cual Racket interpretaría el valor 1 como una función (= procedimiento en Racket), pero como 1 no es una función, daría un error.

La expresión (9) es mucho más fácil de intuir dado que el comportamiento es similar, tanto en cálculo-λ como en Racket:

```
;; Expr. (9)
(define id
  (lambda (x)
    x))

(define c
  (lambda (x)
    (x 1)))

;; Aplicación (ejecución)
> (c id)
1
```

Para finalizar, la expresión (10) que tiene la función $f2$ tiene una función anidada; donde n es *free* y x al ser *bound* substituye la variable x dentro de la función anidada; retornando $(c1\ y)$; y dado que $c1$ es una función constante (no importa el argumento de entrada porque siempre retornara lo mismo) y se descarta y retorna 10.

```
;; Expr. (10)
(define n 0)

(define c1
  (lambda (x)
    10))

(define f2
  (lambda (x)
      ((lambda (y)
        (x y)) n)
    )
)

;; Aplicación (ejecución)
> (f2 c1)
10
```

2.3.3 Reducción

Conversión α. Ya hemos visto como funciona una substitución para hacer uso de variables *bound* y *free*. Pero debemos tener cuidado para no generar ambigüedades, por ejemplo en la siguiente expresión:

$$\begin{aligned}&(\lambda x.(\lambda y.y\ x))y\\&= (\lambda \underline{y}.(\lambda y.y\ \underline{y}))\\&= ERROR\end{aligned} \tag{11}$$

Esta substitución contiene un error porque el y que se aplica pasa a ser la variable *bound* a la izquierda, pero no es la misma y que se encuentra a la derecha dentro de la función interna.

¿Cómo podemos solucionar esto? Fácil. Renombrando las *bound* variables que crean ambigüedad. En este caso, se renombra la variable y de la función interna por z.

La expresión quedaría así:

$$
\begin{aligned}
&(\lambda x.(\lambda z.z\ x))y \\
&= (\lambda \underline{y}.(\lambda z.z\ \underline{y})) \\
&= (\lambda z.z\ y) \\
&= OK
\end{aligned}
\tag{12}
$$

Está confusión, en Racket no es un problema. Porque lo soluciona a través del uso de sus paréntesis de cada *scope* (Ver imagen 1).

(a) x externa. (b) x interna.

Figure 1: Tratamiento de ambigüedades en Racket.

Aunque eso no significa, en lo absoluto, que sea correcto. Siempre se debe procurar escribir nombre de variables idóneos, ya sea en Racket o en cualquier lenguaje de programación.

Reducción β. La reducción beta es similar a un paso de computación dentro de un algoritmo. Esto se puede ver en la expresión anterior (12), donde se va aplicando reducción hasta cuando ya no es posible continuar, pero, en el cálculo-λ *free-type* (que hemos estudiado en este documento) la reducción podría ser infinita (= no terminar). Por ejemplo:

$$
\begin{aligned}
&(\lambda x.x\ x)\ (\lambda x.x\ x) \\
&= (\lambda x.(\lambda x.x\ x)\ (\lambda x.x\ x)) \\
&= (\lambda x.(\lambda x.(\lambda x.x\ x)\ (\lambda x.x\ x))\ (\lambda x.(\lambda x.x\ x)\ (\lambda x.x\ x))) \\
&= \text{recursión infinita}
\end{aligned}
\tag{13}
$$

Conversión η. Es cuando dos funciones son equivalente, si y solo si, dado todos los argumentos retornan lo mismo. Es una forma de extender el uso de una función. Por ejemplo la función $\lambda x.(z\ x) = T$ es equivalente a la función $\lambda z.T = T$ si x es *bound*.

Ejemplo en Racket. Si tenemos dos funcione: f_1 y f_2, al aplicar conversión-η de f_1 a f_2:

```
(define f1
  (lambda (x)
    (f2 x)))

(define f2
  (lambda (x)
    (list x)))

;; Aplicación (ejecución)
> (equal? (f1 2) (f2 2))
#t ;; True
```

Algunas cosas interesantes de este ejemplo:

- La función *list* crea una lista en Racket.
- La función *equal?* compara dos expresiones; si son iguales retorna $\#t$ (= verdadero); en caso contrario retorna $\#f$ (= falso).
- Dado similares argumentos a la función f_1 y f_2 retorna el mismo resultado. En este caso retorna la misma lista: $'(2)$. En consecuencia, la función f_1 *extiende* de f_2.

2.3.4 Aritmética

Una de las características del cálculo-λ es permitir modelar desde operaciones lógicas pasando por estructura de datos a operaciones aritméticas. Para esto último, primero, debemos representar los números usando la *Church numerals*, que nos permite representar un número natural dentro del cálculo-λ en pos de operar con estos en nuestras funciones aritméticas. (Para los

siguientes ejemplos se recomienda utilizar el evaluador *online* para cálculo-λ[8].)

Church numerals. Cada número natural lo podemos representar como funciones anidadas de la siguiente forma:

$$\begin{aligned} 0 &= \lambda f.\lambda x.x \\ 1 &= \lambda f.\lambda x.f\ x \\ 2 &= \lambda f.\lambda x.f\ (f\ x) \\ 3 &= \lambda f.\lambda x.f\ (f(f\ x)) \\ n &= \lambda f.\lambda x.f_1\ (f_2\ (...(f_n\ x))) \end{aligned} \tag{14}$$

Como vimos en algunos ejemplos anteriores, aquí hacemos uso de *high-order functions* o sea de funciones que se utilizan como argumento. En (14) hay dos cuestiones relevantes: (1) el argumento f es una función y (2) siempre se retorna una única función (que internamente puede tener n funciones pero, siempre tiene una función de inicio de «cadena» de funciones que es exclusiva). Las siguientes operaciones han sido anteriormente definidas en artículo de Wikipedia de cálculo-λ en inglés y en (Rojas, 2015).

Suma. Con una composición podemos hacer la suma entre dos números naturales, por ejemplo la idea sería tener una función SUMA que al aplicar sea así: $(SUMA\ 1\ 2) \rightarrow 3$. Podemos definir dicha función así:

$$SUMA = \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f((n\ f)\ x)) \tag{15}$$

Si reemplazamos la *bound* variable m por su función correspondiente (numeral 1) sería: «$\lambda f.\lambda x.f\ x$», lo mismo ocurre con m para el numeral 2. Podemos ver que, el número total de funciones anidadas que se generan son la suma de los dos argumentos. Por otro lado, el término: $((n\ f)\ x)$ es equivalente a $(n\ f\ x)$ dado que se agrupa hacia la izquierda la aplicación.

Multiplicación. Es similar a la suma, dado que una multiplicación entre m y n significa la suma de m por n veces.

$$\begin{aligned} MULT &= \lambda m.\lambda n.\lambda f.m\ (n\ f) \\ MULT2 &= \lambda m.\lambda n.m\ (PLUS\ n)\ 0 \\ MULT &\equiv MULT2 \end{aligned} \tag{16}$$

Predecesor. Dado que los numerales definidos solo admiten valores enteros positivos, en caso de la *resta* debemos definir previamente la función *predecesora* ($PRED\ n \rightarrow n-1$) que se utiliza para cualquier valor superior a 0.

$$PRED = \lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u) \tag{17}$$

Esta función genera muchas derivaciones, por lo cual lo hace difícil entender cada paso (¡pero lo intentaremos!), así si queremos hacer: $(PRED\ 1 \rightarrow 0)$, la derivación debería ser la siguiente (están subrayado las substituciones para que sea más fácil de seguir):

[8] `https://jacksongl.github.io/files/demo/lambda/index.htm`

$$
\begin{aligned}
&(\lambda f.\lambda x.f\ x) = 1 \\
&(\lambda n.\lambda f.\lambda x.((n\ (\lambda g.\lambda h.h(gf)))\ (\lambda u.x))\ (\lambda u.u))\ \underline{(\lambda f.\lambda x.f\ x)} \\
&= ((\lambda n.(\lambda f.(\lambda x.(((\underline{n}\ (\lambda g.(\lambda h.(h\ (g\ f)))))\ (\lambda u.x))(\lambda x0.x0))))))\ \underline{(\lambda x1.(\lambda x2.(x1\ x2)))}) \\
&= (\lambda f.(\lambda x.((((\lambda x1.(\lambda x2.(\underline{x1}\ x2)))\ \underline{(\lambda g.(\lambda h.(h\ (g\ f)))))}\ (\lambda u.x))\ (\lambda x0.x0)))) \\
&= (\lambda f.(\lambda x.(((\lambda x2.((\lambda g.(\lambda h.(h\ (g\ f))))\ \underline{x2}))\ \underline{(\lambda u.x)})\ (\lambda x0.x0)))) \\
&= (\lambda f.(\lambda x.(((\lambda g.(\lambda h.(h\ (\underline{g}\ f))))\ \underline{(\lambda u.x)})\ (\lambda x0.x0)))) \\
&= (\lambda f.(\lambda x.((\lambda h.(\underline{h}\ ((\lambda u.x)\ f)))\ \underline{(\lambda x0.x0)}))) \\
&= (\lambda f.(\lambda x.((\lambda x0.\underline{x0})\ \underline{((\lambda u.x)\ f)}))) \\
&= (\lambda f.(\lambda x.((\lambda \underline{u}.x)\ \underline{f}))) \\
&= (\lambda f.(\lambda x.x)) = 0
\end{aligned}
\tag{18}
$$

(Esta derivación fue obtenida usando: `https://jacksongl.github.io/files/demo/lambda/index.htm`)

Resta. Con esto ya podemos definirla:

$$RESTA = \lambda m.\lambda n.n\ PRED\ m \qquad (19)$$

Entonces definidas las operaciones aritméticas ya podemos dar por entendido el mecanismo de derivación del cálculo-λ, al menos a un nivel inicial, cuestiones como la *división* queda a trabajo del curioso lector. Por ahora, veamos como estos operadores funcionan en Racket, que dicho sea de paso, es mucho más simple.

Ejemplos en Racket. Ahora veamos la facilidad de utilizar los mismos operadores ya implementados en Racket:

```
> (+ 1 2)
3
> (- 3 1)
2
> (* 4 (* 4 10))
160
> (+ 1 (/ 10 5))
3
```

Podemos ver que Racket usa *prefix notation* (= notación polaca) [9] junto a la *expresión-S*.

2.4 Racket

En esta sección final nos adentraremos –y relajaremos– con Racket, usando los conocimientos ya adquiridos de cálculo-λ.

2.4.1 Estructuras de datos

Listas. Las listas son una secuencia de elementos (sin acceso aleatorio ni índice) que aceptan cualquier tipo de dato y, a su vez, son inmutables (o sea no puede modificarse una vez creadas). Por lo demás, es la estructura de datos más popular en Racket.

```
(list) ; Lista vacías
(list 1 2 3) ; Tres números
(list "x" "y" "z") ; Tres caracteres
(list 1 2 (list (list) "x" "y"
   (list 1 0 1) "z")) ; Listas
   anidadas
```

Una notación más simple es remover la palabra *list* y anteponer una comilla simple «'» al paréntesis de la izquierda «(». A continuación se presentan las declaraciones que son equivalentes a las anteriores pero con la nueva notación:

```
'()
'(1 2 3)
'("x" "y" "z")
'(1 2 ('() "x" "y" '(1 0 1) "z"))
```

Funciones **car**[10] y **cdr**[11]. Son operaciones que se pueden aplicar sobre listas. Primero, *car* se utiliza para retorna el primer elemento de la lista; Segundo, *cdr* retorna la lista sin el primer elemento. Por ejemplo:

```
> (define x  '("x" "y" "z"))
> (car x)
"x"

> (cdr x)
'("y" "z")

> (cdr (cdr x))
'("z")

> (cdr (cdr (cdr x)))
'()
```

[9] https://es.wikipedia.org/wiki/notación_polaca

[10] car = «Contents of the Address part of Register number».

[11] cdr = «Contents of the Decrement part of Register number».

Función **cons**. Permite concatenar dos valores (independiente del tipo de valor).

```
> (cons 0 '(2 3))
'(0 2 3)

> (car (cons 5 2))
5

> (cdr (cons '(5) '()))
'()

> (equal? (cdr (cons '(5) '(1)))
    (cdr (cons '(5) 1)))
#f

> (equal? (cdr (cons '(5) '(1)))
    (cdr (cons '(5) '(1))))
#t
```

Tabla hash. Esta estructura de datos permite asociar una clave con un valor, por ejemplo $1 \rightarrow 10$, cuando queramos acceder a la clave 1 siempre retornara 10. En Racket existen varias variantes de tabla hash: una mutable y otra inmutable. Veremos ejemplos de cada una.

Mutable. Una vez creada está puede ser modificada: ya sea el valor de una clave o incluso borrar toda la tabla hash.

```
(define h (make-hash)) ; Se crea la
   tabla hash h.
(hash-set! h 1 10)
(hash-set! h 2 '(1 2 3))
(hash-set! h '(0 1) "binary")
> h
'#hash(((0 1) . "binary") (2 . (1 2
   3)) (1 . 10))
```

La función *hash-set* nos permite agregar una nueva clave-valor a la variable h, se agrega el «!» al final cuando se trata de una estructura mutable, en el caso opuesto, si omitimos el ! va retornar una nueva tabla hash con la nueva clave-valor sin haber *modificado* la original. Con la función *hash-ref* podemos acceder al valor de una clave en especifico: (hash-ref h 1) = 10.

Inmutable. Racket también soporta estructura de datos inmutables, lo que es muy útil en algunos casos.

```
(define hi (make-immutable-hash
    '([1 . "b"][2 . "a"]))) ; Se
    crea la tabla hash h con dos
    elementos.
> (hash-ref hi 1)
"b"
> (hash-set hi 3 2) ; crea una
    nueva tabla hash.
'#hash((1 . "b") (2 . "a") (3 . 2))
> (hash-set! hi 3 2)
. . hash-set!: contract violation
  expected: (and/c hash? (not/c
     immutable?))
  given: '#hash((1 . "b") (2 . "a"))
  argument position: 1st
  other arguments...:
```

Antes mencionamos que una estructura inmutable no puede ser modificada después de su creación, y como vemos en el ejemplo anterior cuando hacemos uso de la función *hash-set!* el interprete de Racket arroja un error.

Con *listas* y *hash* ya tenemos suficiente para comenzar a ver como construir algoritmos en Racket, en donde aplicaremos constantemente operaciones sobre listas utilizando recursividad. (También existen otras estructuras como: pair, vector y listas asociadas. Pero se lo dejamos al interés y curiosidad del lector.)

2.4.2 Algoritmos

Es momento de ver algunos algoritmos utilizando Racket, espero que, hasta el momento el lector allá encontrado interesante aprender Racket, pero aún falta lo mejor, y lo mejor, espero que

pueda ser visible en este apartado.

Fibonacci. Una manera para adentrarse en la recursividad es aprender a implementar *fibonacci*. Recordemos la funciones de como opera el algoritmo: (1) $f_0 = 0$; (2) $f_1 = 1$; y (3) $f_n = f_{n-1} + f_{n-2}$. En Racket la función *cond* es una condicional múltiples, es decir, *if-else if*.

```
(define fib
  (lambda (n)
    (cond
      ((= n 0) 0) ;if
      ((= n 1) 1) ;else if
      ((> n 0) (+ (fib (- n 1))
         (fib (- n 2)))) ;else if
     )
  )
)

> (fib 9)
34
```

La primera expresión en *cond* es $((= \ n \ 0) \ 0)$, de tal forma de que si se cumple $(= \ n \ 0)$ entonces retornara 0. De la misma manera ocurre para las otras dos expresiones.

Contar palabras. La idea es contar las veces que una palabra (entregada como argumento) se repite dentro de una lista.

```
(define word-count
  (lambda (list search count)
      (cond
        [(null? list) count]
        [(eq? (car list) search)
            (word-count (cdr list)
               search (+ count 1))]
        [(word-count (cdr list)
           search count)]
       )
  )
)

> (word-count '("aaa" "a" "bbb" "a"
   "aaa" "y") "aaa" 0)
2
> (word-count '("aaa" "a" "bbb" "a"
   "aaa" "y") "y" 0)
1
```

Lista inversa. ¿Qué tal si queremos revertir el orden de una lista? Claro, en Racket por defecto ya existe la función *reverse* que hace esto, pero, en este caso queremos implementarla por nosotros mismos. Una, de las tantas formas de hacerlo, es la siguiente:

```
(define last-element
  (lambda (list)
    (cond
      [(= (length list) 1) list]
      [(last-element (cdr list))]
)))

(define new-reverse
  (lambda (old-list new-list)
    (cond
      [(null? old-list) new-list]
      [(new-reverse
            (take old-list (-
               (length old-list)
               1)) (append new-list
               (last-element
               old-list)))]
    )
  )
)

> (new-reverse  '("aaa" "bbb" "a"
   "y") '())
'("y" "a" "bbb" "aaa")
```

Algunas apreciaciones con respecto a este último algoritmo:

- Aplicamos **descomposición** a nuestro algoritmo, o sea, creamos una segunda fun-

ción *last-element* que aplica una funcionalidad en especifico. Esto nos ayuda a reutilizar dicha función a futuro y que, no este ligada solamente a nuestra función *new-reverse* (además de hacerlo más legible).

- La función *take* toma los primero n elementos de una lista. Por ejemplo: (*take* (1 2 3) 2) = (1 2).
- La función *length* retorna el tamaño de la lista.
- La función *append* agrega un elemento a la lista.

3 Conclusión

En este documento hicimos una introducción al cálculo-λ libre de tipos (*free-types*) que, a su vez, se puede ver como una introducción al paradigma de programación funcional.

Revisamos los siguientes conceptos:

- Los operadores principales.
- Las *bound* y *free* variables.
- La operaciones de reducción.
- Modelado de operaciones aritméticas.

Por otro lado, cada operación en cálculo-λ lo explicamos con su equivalente en el lenguaje de programación Racket, además de dar una introducción a las estructuras de datos del lenguaje aplicadas a algunos tipos de algoritmos, donde prevalece la inmutabilidad y la recursividad, componentes principales en la programación funcional.

Solo me queda por decir: *De parvis grandis acervus erit.* («De las cosas pequeñas se nutren las cosas grandes».)

Referencias

H. H. Barendregt and E. Barendsen. Introduction to lambda calculus. *Nieuw archief voor wisenkunde*, 4:337–372, 01 1984.

A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):pp. 345–363, 1936. ISSN 00029327. URL `http://www.jstor.org/stable/2371045`.

R. Rojas. A tutorial introduction to the lambda calculus, 2015.

D. Szmulewicz. URL `https://danielsz.github.io/blog/2019-08-05T21_14.html`.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

R. L. Wexelblat, editor. *History of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 1978. ISBN 0127450408.

Wikipedia. Lambda calculus - wikipedia, the free encyclopedia.