Lenguajes del Programación

Unidad 2: Estilo Funcional

El Cálculo λ como núcleo para funciones

Manuel Soto Romero^{***}

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota exploraremos el Cálculo λ como uno de los modelos de cómputo más influyentes en la teoría de lenguajes de programación. A diferencia de las Máquinas de Turing, centradas en la manipulación secuencial de estados, el Cálculo λ pone el énfasis en la definición y aplicación de funciones, ofreciendo una visión más abstracta y elegante de la computación. Estudiaremos su sintaxis y semántica, las reglas de reducción y propiedades fundamentales como la confluencia, así como su capacidad para representar estructuras básicas de datos y operaciones aritméticas y lógicas. Finalmente, veremos cómo estas ideas dieron origen a lenguajes como ISWIM y sentaron las bases de muchos lenguajes funcionales modernos.

1. Modelos de Cómputo y Lenguajes de Programación

Un **modelo de cómputo** es una abstracción matemática que describe cómo se pueden realizar cálculos y resolver problemas utilizando un conjunto de reglas formales. Los modelos de cómputo proporcionan una manera de entender, definir y analizar lo que significa **computar** y lo que las computadoras pueden hacer. Algunos modelos de cómputo famosos son las **Máquinas de Turing**, el **Cálculo λ** y las **Máquinas de Estado Finito**.

En la teoría de lenguajes de programación, los modelos de cómputo son esenciales porque nos permiten:

- Proporcionan una base formal para definir cómo se comportan los lenguajes de programación y qué tipos de problemas pueden resolver.
- Nos ayudan a entender las limitaciones de los lenguajes de programación, identificando qué problemas son computables en un lenguaje dado y cuáles no.
- Diferentes modelos de cómputo permiten comparar lenguajes de programación en términos de **poder expresivo** y eficiencia.
- Sirven como base para la construcción de herramientas que ejecutan o traducen programas, como compiladores e intérpretes.

*Tomada del material desarrollado en colaboración para el curso de la Dra. Karla Ramírez Pulido con algunas adaptaciones para esta versión de la materia.

**Un agradecimiento a Juan Mario Sosa Romo por sus aportes durante la revisión de esta nota como parte de su servicio social.

Máquinas de Turing

Las **Máquina de Turing**, propuestas por Alan Turing en 1936, son uno de los modelos más influyentes y fundamentales. Este modelo consiste en una cinta infinita dividida en celdas, un cabezal de lectura/escritura que se mueve a lo largo de la cinta y un conjunto finito de estados con reglas que determinan el comportamiento del sistema.

Los lenguajes de programación imperativos, como C, JAVA y PYTHON, se basan en el modelo de las Máquinas de Turing porque reflejan su enfoque secuencial y controlado de los cómputos:

- Utilizan variables, asignaciones y estructuras de control para manipular el estado del programa de manera similar a cómo una Máquina de Turing manipula la cinta.
- En ambos casos, las instrucciones se ejecutan de manera secuencial, siguiendo un orden lógico que depende del estado actual.
- Las estructuras de control en lenguajes imperativos reflejan la capacidad de las Máquinas de Turing para cambiar de estado en funciones de la entrada y el estado actual.

No se profundizará del todo en este tema pues su estudio recae en el curso previo de **Autómatas y Lenguajes Formales**.

Cálculo λ

En esta nota introduciremos al Cálculo λ , desarrollado por Alonzo Church en la década de 1930. Es otro modelo de cómputo que se centra en la definición y aplicación de funciones. A diferencia de las máquinas de Turing, que operan sobre una cinta de estados, el Cálculo λ se basa en la idea de aplicar funciones a argumentos.

Algunas de sus principales características son:

- En el Cálculo λ , las funciones son tratadas como valores que pueden ser pasados como argumentos, devueltos como resultados y construidos en tiempo de ejecución.
- A pesar de su simplicidad, el Cálculo λ es capaz de expresar cualquier función computable, lo que lo hace equivalente en poder a las Máquinas de Turing.
- Es la base teórica de muchos lenguajes de programación funcionales como HASKELL y LISP, donde el enfoque está en la evaluación de funciones puras sin efectos secundarios.

El Cálculo λ es fundamental en la teoría de lenguajes de programación porque:

- Proporciona una forma simple y **elegante** de razonar sobre funciones y la computación en general, sirviendo como una base para la semántica de lenguajes de programación.
- Introduce conceptos como la sustitución, que como vimos en notas pasadas, es importante para entender cómo se evalúan las expresiones y se aplican las funciones en los lenguajes de programación.
- Ha influido en el diseño de lenguajes funcionales modernos, donde la manipulación de funciones y el evitar efectos secundarios son principios fundamentales.
- Permite la optimización de programas a través de técnicas como las β -reducciones y la aplicación parcial, que son esenciales en la optimización de código en compiladores.

2. Sintaxis

El Cálculo λ consiste de una notación para representar expresiones, llamados **términos λ** . Existen tres posibles términos: **variables**, **abstracciones λ** y **aplicaciones de función**.

Para construir dichos términos se tiene la siguiente gramática:

$$\begin{aligned} \langle \text{expr} \rangle & ::= \langle \text{var} \rangle \\ & \quad | (\lambda \langle \text{var} \rangle . \langle \text{expr} \rangle) \\ & \quad | (\langle \text{expr} \rangle \langle \text{expr} \rangle) \end{aligned}$$

$$\langle \text{var} \rangle ::= x \mid y \mid z \mid \dots$$

Variables

Las variables o identificadores son representadas mediante letras minúsculas. Algunos ejemplos de variables:

$$x \quad y \quad z \quad w$$

Abstracciones λ

Las funciones, abstracciones λ o simplemente abstracciones, representan la definición de funciones **anónimas**, esto es, no requieren de un nombre explícitamente para ser definidas o usadas. Tales abstracciones se componen de un encabezado y un cuerpo. El encabezado contiene el nombre del parámetro formal de la función y el cuerpo de ésta se compone otras subexpresiones que son por supuesto términos λ . Todo lo que se encuentra antes del punto es el encabezado y lo que se encuentra después es el cuerpo.

$$\underbrace{\lambda x}_{\text{Encabezado}} . \underbrace{x}_{\text{Cuerpo}}$$

Otros ejemplos de abstracciones son:

1. $\lambda x . \lambda y . xy$
2. $\lambda z . \lambda w . u$
3. $\lambda u . \lambda v . vvv$

Las expresiones anteriores, tienen como encabezado λx , λz y λu respectivamente, mientras que $\lambda y . xy$, $\lambda w . u$ y $\lambda v . vvv$ conforman el cuerpo de las mismas.

La primera abstracción de los ejemplos anteriores, es equivalente a la función de dos parámetros $\lambda xy . xy$ sin embargo, la especificación de términos λ del Cálculo λ no permite definir funciones con más de un parámetro. Una solución que se da a esta situación es modificar las definiciones de función de manera reciban un único parámetro (en este ejemplo x) que tendrán como cuerpo otra función de igualmente un parámetro (y), a esta técnica se le conoce como **currificación**. A continuación se exponen algunos ejemplos de abstracciones con su currificación:

1. $\lambda xy . xy \equiv_{\text{curry}} \lambda x . \lambda y . xy$
2. $\lambda zw . u \equiv_{\text{curry}} \lambda z . \lambda w . u$
3. $\lambda abc . cba \equiv_{\text{curry}} \lambda a . \lambda b . \lambda c . cba$

Aplicaciones de Función

La regla de producción $\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle$ de la gramática para términos λ representa la aplicación de funciones. Dada la expresión de una aplicación de función $e_1 e_2$, se dice que e_1 está en posición de función y que e_2 está en la posición de su argumento. De esta forma, una aplicación representa el proceso de evaluar una función con su respectivo argumento (también llamado parámetro real). Por ejemplo la siguiente función:

$$\lambda x.x \ \lambda y.y$$

representa la aplicación de la función $\lambda x.x$ con la función $\lambda y.y$, sin embargo, es posible incluir paréntesis que faciliten la lectura o ayuden a entender el orden de evaluación. Algunos ejemplos:

1. $(\lambda x.x) (\lambda y.y)$
2. xy
3. $(\lambda w.ws) (\lambda y.yy) u$

En caso de que no haya paréntesis explícitos, la aplicación de funciones asocia a la izquierda.

Alcance

El alcance de todo identificador se da en el cuerpo de las abstracciones.

En la expresión $\lambda x.x$ se dice que el identificador x está *ligado* o *acotado*, pues se encuentra en el cuerpo de la función que toma como parámetro el mismo nombre del identificador x (*de ligado*). Si un identificador no es precedido por una λ con el mismo parámetro, se dice que está *libre* en la expresión. Por ejemplo, en la siguiente expresión:

$$\lambda x.xy$$

El identificador x está ligado, mientras que el identificador y se encuentra libre. Otro ejemplo es:

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

Se observa que en la subexpresión $(\lambda x.x)$ el identificador o la variable x se encuentra ligada, mientras que la subexpresión $(\lambda y.x)$ la variable x está libre. Es importante observar que la x de la segunda subexpresión es independiente a la x de la primera.

Se define formalmente a los identificadores libres como sigue:

- x está libre en y , con y cualquier variable (incluyendo a x).
- x está libre en $(\lambda y.e)$ si y sólo si $x \neq y$ y x se encuentra libre en e , siendo e una expresión cualquiera.
- x está libre en $e_1 e_2$ si y sólo si x está libre en e_1 o en e_2 , con e_1 y e_2 expresiones cualquiera.

Los identificadores ligados se definen formalmente como sigue:

- x se encuentra ligado en $(\lambda x.e)$

- x se encuentra ligado en $(\lambda y.e)$ con $x \neq y$ si y sólo si x está ligado en e .
- x está ligado en $e_1 e_2$ si y sólo si, x está ligado en e_1 o e_2 .

Se observa que gracias a la tercera regla de ambas definiciones, estas no son excluyentes entre sí, esto es, un identificador puede estar libre y ligado al mismo tiempo. Una expresión e tal que no tiene variables libres, se dice que es una expresión **cerrada** también conocida como **combinador**.

α -equivalencias

Los nombres de los identificadores ligados no son importantes en el contexto del Cálculo λ . Por ejemplo $\lambda x.x$ y $\lambda y.y$ representan a la misma función y lo único que cambia es el nombre del identificador. A esto se le conoce como relación de α -*equivalencia*. Sin embargo, la regla tiene ciertas restricciones:

$$\lambda V.E \equiv_{\alpha} \lambda W.E [V := W] \text{ si } W \text{ no está en las variables libres de } E$$

Algunos ejemplos:

- $\lambda x.x \equiv_{\alpha} \lambda x.x$
- $\lambda x.x \equiv_{\alpha} \lambda y.y$
- $\lambda x.(\lambda x.x) x \equiv_{\alpha} \lambda y.(\lambda x.x) y$

3. Semántica

β -reducciones

La regla de β -*reducción* expresa la idea de aplicación de función y ésta es la única regla semántica del Cálculo λ , es decir, la forma de evaluar una expresión. Esta regla se define como sigue:

$$(\lambda x.e) s \rightarrow_{\beta} e [x := s]$$

Es decir, se deben sustituir todas las apariciones de x por s en e . Lo anterior expresa la idea de una función a la cual se le asigna el parámetro real a su parámetro formal. A la expresión $(\lambda x.e)$ se le llama **redex**¹, mientras que a la expresión $e [x := s]$ se le llama **reducto**.

Algunos ejemplos:

- $(\lambda x.x) x \rightarrow_{\beta} x$
- $(\lambda x.x) (\lambda x.x) \rightarrow_{\beta} \lambda x.x$
- $(\lambda x.x) (\lambda y.y) \rightarrow_{\beta} \lambda y.y$
- $(\lambda x.\lambda y.(xy)) (\lambda x.x) \rightarrow_{\beta} \lambda y.(\lambda x.x) y \rightarrow_{\beta} \lambda y.y$

¹Del inglés *reducible expression*.

Formas Normales

Dada una expresión e , si no existe e' tal que $e \rightarrow_{\beta} e'$, se dice que e se encuentra en Forma Normal. Es decir, dada una expresión, si ésta no puede reducirse más mediante β -reducciones, se dice entonces que esta expresión se encuentra en Forma Normal. Por ejemplo, todas las reducciones anteriores, están en Forma Normal.

Divergencia

También existen expresiones, que no tienen una Forma Normal pues sin importar cuantas reducciones se apliquen, siempre se obtiene una nueva expresión que puede ser reducida, en este caso se dice que la expresión *diverge* y por lo tanto no tiene Forma Normal. Por ejemplo, sean $\omega =_{def} \lambda x.xx$ y $\Omega =_{def} \omega\omega$, entonces:

$$\Omega =_{def} \omega\omega =_{def} (\lambda x.xx)\omega \rightarrow_{\beta} \omega\omega =_{def} \Omega$$

De manera que Ω se reduce a sí mismo, por lo tanto *diverge*:

$$\Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \Omega \rightarrow_{\beta} \dots$$

Confluencia

El Cálculo λ tiene una natural no determinista, por ejemplo la expresión $(\lambda x. (\lambda y.yx) z) v$ tiene dos reductos distintos:

- $(\lambda x. (\lambda y.yx) z) v \rightarrow_{\beta} (\lambda y.yv)$
- $(\lambda x. (\lambda y.yx) z) v \rightarrow_{\beta} (\lambda x.zx) v$

En este caso ambos caminos llevan a la misma Forma Normal (zv) sin embargo, debemos preguntarnos si esto siempre sucede. Si queremos usar el Cálculo λ como un sistema de cómputo, el resultado final debe ser el mismo independientemente del camino utilizado. Si esta propiedad llamada *confluencia* fuera falsa para nuestra regla de β -reducción, cualquier intento de usar el Cálculo λ como lenguaje de programación sería fallido.

Teorema 1: Confluencia o Propiedad de Church-Rosser

Si $e \rightarrow_{\beta}^ r$ y $e \rightarrow_{\beta}^* s$ entonces existe un término t tal que $r \rightarrow_{\beta}^* t$ y $s \rightarrow_{\beta}^* t$.*

Corolario 1: Unicidad de las Formas Normales

Si e tiene una Forma Normal entonces es única. Es decir si $e \rightarrow_{\beta}^ e_f$ y $e \rightarrow_{\beta}^* e'_f$ entonces $e_f = e'_f$ (salvo α -equivalencia).*

La demostración de estos resultados queda fuera de los alcances de este curso. El corolario garantiza que el Cálculo λ puede usarse como un sistema de cómputo. Sin embargo el uso de β -reducciones sin restricciones o estrategias adicionales podría prohibir la obtención de una Forma Normal. Por lo anterior, si se usara directamente como lenguaje de programación habría algunos programas que, teniendo un valor bien definido, podrían ciclarse infinitamente.

4. Representación de datos

Aritmética

El Cálculo λ permite modelar operaciones aritméticas. Para poder usar estas operaciones, añadiremos la representación de números.

Numerales de Church

Para representar números naturales, se tiene una definición para el cero y el resto de naturales, se construyen a partir de la función sucesor. De esta forma, podemos representar números con funciones que reciben una función sucesor (s) y una función cero (z), se definen cada uno de estos números aplicando s tantas veces como sea necesario a z . Se conoce a esta representación como **Numerales de Church**.

- $0 =_{def} \lambda s. \lambda z. z$
- $1 =_{def} \lambda s. \lambda z. s z$
- $2 =_{def} \lambda s. \lambda z. s (s z)$
- ...
- $n =_{def} \lambda s. \lambda z. \underbrace{s(\dots(s z)\dots)}_{n \text{ veces}}$

Por supuesto, todo depende de la definición de sucesor que demos. Una definición de esta función se muestra a continuación:

$$S =_{def} \lambda n. \lambda a. \lambda b. a (n a b)$$

Por ejemplo, para calcular el sucesor de cero, tenemos:

$$\begin{aligned} S 0 & \\ &=_{def} (\lambda n. \lambda a. \lambda b. a (n a b)) 0 \\ &\rightarrow_{\beta} \lambda a. \lambda b. a (0 a b) \\ &=_{def} \lambda a. \lambda b. a ((\lambda s. \lambda z. z) a b) \\ &\rightarrow_{\beta} \lambda a. \lambda b. a ((\lambda z. z) b) \\ &\rightarrow_{\beta} \lambda a. \lambda b. a b \equiv_{\alpha} \lambda s. \lambda z. s z =_{def} 1 \end{aligned}$$

Funciones Aritméticas

Una vez definidas estas funciones, podemos definir operaciones entre números naturales.

Suma

Una suma en realidad es una aplicación de la función sucesor, por ejemplo, si se desea sumar los números $n + m$, se debe aplicar la función sucesor n veces a m . Por ejemplo, para sumar $2 + 3$ tenemos:

$$\begin{aligned} 2 S 3 & \\ &=_{def} (\lambda s. \lambda z. s (s z)) S 3 \\ &\rightarrow_{\beta} (\lambda z. S (S z)) 3 \\ &\rightarrow_{\beta} S (S 3) \rightarrow_{\beta} \dots \rightarrow_{\beta} 5 \end{aligned}$$

De esta forma, podemos definir la función suma que recibe dos números n y m como sigue:

$$A =_{def} \lambda n. \lambda m. nSm$$

Debido a la conmutatividad de la suma, tenemos también la definición:

$$A' =_{def} \lambda n. \lambda m. mSn$$

Se deja como ejercicio al lector, demostrar la equivalencia entre estas dos funciones.

Multiplicación

Similarmente se define la operación multiplicación como sigue:

$$\lambda x. \lambda y. \lambda a. x (ya)$$

Ejercicio 1

Se deja como ejercicio al lector el cálculo del producto 3×3 , es decir:

$$(\lambda x. \lambda y. \lambda a. x (ya)) 33$$

Álgebra Booleana

Al igual que con los números, es posible representar expresiones booleanas y operaciones entre éstas. Las constantes lógicas *verdadero* y *falso* se representan mediante:

$$\begin{aligned} T &=_{def} \lambda x. \lambda y. x \\ F &=_{def} \lambda x. \lambda y. y \end{aligned}$$

La función T , dados dos valores para verdadero y falso (x y y respectivamente), regresa x , mientras que la función F regresa y .

Operaciones con Booleanos

Con estas definiciones es posible definir algunas operaciones del Álgebra Booleano, a continuación se muestran las definiciones de las funciones *not*, *or* y *and* junto con su tabla de verdad.

Negación

$$\neg =_{def} \lambda x. xFT$$

$$\begin{aligned}
& \neg F \\
& =_{def} (\lambda x. xFT) F \\
& \rightarrow_{\beta} FFT \\
& =_{def} (\lambda x. \lambda y. y) FT \\
& \rightarrow_{\beta} (\lambda y. y) T \\
& \rightarrow_{\beta} T
\end{aligned}$$

$$\begin{aligned}
& \neg T \\
& =_{def} (\lambda x. xFT) T \\
& \rightarrow_{\beta} TFT \\
& =_{def} (\lambda x. \lambda y. x) FT \\
& \rightarrow_{\beta} (\lambda y. F) T \\
& \rightarrow_{\beta} F
\end{aligned}$$

Disyunción

$$\vee =_{def} \lambda x. \lambda y. xTy$$

$$\begin{aligned}
& \vee FF \\
& =_{def} (\lambda x. \lambda y. xTy) FF \\
& \rightarrow_{\beta} (\lambda y. FTy) F \\
& \rightarrow_{\beta} FTF \\
& =_{def} (\lambda x. \lambda y. y) TF \\
& \rightarrow_{\beta} (\lambda y. y) F \\
& \rightarrow_{\beta} F
\end{aligned}$$

$$\begin{aligned}
& \vee FT \\
& =_{def} (\lambda x. \lambda y. xTy) FT \\
& \rightarrow_{\beta} (\lambda y. FTy) T \\
& \rightarrow_{\beta} FTT \\
& =_{def} (\lambda x. \lambda y. y) TT \\
& \rightarrow_{\beta} (\lambda y. y) T \\
& \rightarrow_{\beta} T
\end{aligned}$$

$$\begin{aligned}
& \vee TF \\
& =_{def} (\lambda x. \lambda y. xTy) TF \\
& \rightarrow_{\beta} (\lambda y. TTy) F \\
& \rightarrow_{\beta} TTF \\
& =_{def} (\lambda x. \lambda y. x) TF \\
& \rightarrow_{\beta} (\lambda y. T) F \\
& \rightarrow_{\beta} T
\end{aligned}$$

$$\begin{aligned}
& \vee TT \\
& =_{def} (\lambda x. \lambda y. xTy) TT \\
& \rightarrow_{\beta} (\lambda y. TTy) T \\
& \rightarrow_{\beta} TTT \\
& =_{def} (\lambda x. \lambda y. x) TT \\
& \rightarrow_{\beta} (\lambda y. T) T \\
& \rightarrow_{\beta} T
\end{aligned}$$

Conjunción

$$\wedge =_{def} \lambda x. \lambda y. xyF$$

Ejercicio 2

Se deja como ejercicio al lector desarrollar la tabla de verdad de la conjunción.

Condicional *if0*

También es posible definir condicionales. Por ejemplo, se define el condicional *if0* (Z). Si se aplica a un número n y éste resulta ser cero, se obtiene T ; en caso contrario, se obtiene F . La definición de esta función se muestra a continuación:

$$Z =_{def} \lambda x. xF\neg F$$

Veamos la reducción de $Z0$ y $Z1$:

$$\begin{aligned}
& Z0 \\
& =_{def} (\lambda x. xF\neg F) 0 \\
& \rightarrow_{\beta} 0F\neg F \\
& =_{def} (\lambda s. \lambda z. z) F\neg F \\
& \rightarrow_{\beta} (\lambda z. z) \neg F \\
& \rightarrow_{\beta} \neg F \\
& \rightarrow_{\beta} T
\end{aligned}$$

$$\begin{aligned}
& Z1 \\
& =_{def} (\lambda x. xF\neg F) 1 \\
& \rightarrow_{\beta} 1F\neg F \\
& =_{def} (\lambda s. \lambda z. sz) F\neg F \\
& \rightarrow_{\beta} (\lambda z. Fz) \neg F \\
& \rightarrow_{\beta} F\neg F \\
& =_{def} (\lambda x. \lambda y. y) \neg F
\end{aligned}$$

$\rightarrow_{\beta} (\lambda y.y) F$
 $\rightarrow_{\beta} F$

Ejercicio 3

Se deja como ejercicio al lector la definición de la función *if*.

Integrando Números con Booleanos

Con los naturales y booleanos definidos podemos definir otras funciones de utilidad.

Función Predecesor

Para definir esta función construiremos un par de la forma $(n, n - 1)$ y tomaremos el segundo elemento del par como resultado. Un par puede representarse mediante la función:

$$\lambda z.zab$$

La z permite realizar operaciones con el par. Por ejemplo, podemos obtener el primer y segundo elemento auxiliandonos de las funciones T (verdadero) y F (falso).

$$(\lambda z.zab) T \rightarrow_{\beta} Tab \rightarrow_{\beta} a$$

$$(\lambda z.zab) F \rightarrow_{\beta} Fab \rightarrow_{\beta} b$$

Podemos definir el par $(n + 1, n)$ a partir del par $(n, n - 1)$, representado por el parámetro p y extrayendo el primer elemento y aplicando la función sucesor. De esta forma definimos la función Φ como sigue:

$$\Phi =_{def} \lambda p.\lambda z.z (S (pT)) (pT)$$

De esta forma el predecesor de un número se obtiene aplicando n veces la función Φ al par $\lambda z.00$ y seleccionando el segundo elemento del par usando la función F .

$$P =_{def} \lambda n.(n\Phi (\lambda z.z00))F$$

Es importante notar que bajo esta definición, el predecesor de cero es cero, con lo cual se evita la generación de números negativos. Se deja como ejercicio al lector obtener el predecesor de 1.

Operadores Relacionales

Una vez definida la función predecesor, podemos definir una función que verifique si un número es mayor o igual que otro. Por ejemplo, para verificar $2 \geq 1$, basta con aplicar la función predecesor 2 veces a 1.

$$2P1 \rightarrow_{\beta} \dots \rightarrow_{\beta} 0$$

Con la función *if0* se verifica si esto se cumple. De esta forma, definimos la función $\geq$ como sigue:

$$\geq =_{def} \lambda x. \lambda y. Z (xPy)$$

De la misma forma, podemos definir la función igualdad $x = y$. Si se cumple $x \geq y$ y $y \geq x$, por lo tanto $x = y$. De esta forma, definimos la función $=$ que usa la operación $\wedge$ definida anteriormente.

$$= =_{def} (\lambda x. \lambda y. \wedge (\geq xy) (\geq yx))$$

De forma similar se pueden definir otras operaciones relacionales.

ISWIM

El Cálculo λ fue diseñado por Church con el fin de dar un sistema formal que permitiera modelar las matemáticas. En los años 50 y 60 se descubrió la conexión entre los lenguajes de programación y varios aspectos del Cálculo λ gracias a un intento de dar la especificación formal del lenguaje de programación ALGOL 60. Sin embargo, el Cálculo λ resulta ser demasiado primitivo para considerarlo un lenguaje de programación, al igual que las Máquinas de Turing son demasiado primitivas para considerarlas computadoras reales.

Peter J. Landin definió el lenguaje de programación ISWIM². Los fundamentos de su diseño fueron tomados del Cálculo λ como un punto de partida. ISWIM se define mediante la siguiente gramática similar a la del Cálculo λ :

```
<expr> ::= <var>
         | <cte>
         | <op> (<expr>+)
         | ( $\lambda$ <var>.<expr>)
         | (<expr> <expr>)
```

Por ejemplo, para nuestra primera versión de MINILISP:

```
<cte> ::= 1 | 2 | 3 | ...
```

```
<op> ::= + | -
```

²Del inglés *If you See What I mean*.

Aunque son similares sintácticamente, la principal diferencia entre ISWIM y el Cálculo λ es la forma en que evalúan funciones. Por ejemplo, la siguiente expresión del Cálculo λ :

$$(\lambda x.1)(\text{sub1 } \lambda y.y)$$

se reduce a 1. Esto ocurre debido a que las β -reducciones ignoran el parámetro real y únicamente realizan la sustitución correspondiente. Esta forma de evaluar es llamada **paso de parámetros por nombre**³ principalmente asociado con la evaluación perezosa. Sin embargo, muchos lenguajes de programación en la actualidad no hacen este tipo de evaluación y en su lugar usan una estrategia de evaluación llamada **paso de parámetros por valor**⁴ asociada con la evaluación glotona, es decir, evalúan el parámetro real antes de pasarlo a la función correspondiente. ISWIM implementa paso por valor. Más adelante, en este curso estudiaremos a profundidad estos conceptos.

En conclusión, aunque ISWIM nunca fue implementado como tal, éste ha servido como base para estudiar otras aplicaciones mediante la definición de **máquinas abstractas**. La filosofía de este lenguaje de programación es la que siguen algunos lenguaje de programación hoy en día como son LISP, SCHEME y ML.

7. Conclusión

El Cálculo λ se consolida como un núcleo teórico y práctico para entender la computación desde el paradigma funcional. Hemos visto que, pese a su simplicidad, es tan expresivo como las Máquinas de Turing, pero al mismo tiempo más adecuado para razonar sobre funciones, sustitución y evaluación. Su estudio nos permitió comprender conceptos como las β -reducciones, la noción de forma normal y la confluencia, además de observar cómo a partir de él se construyen representaciones de datos y operaciones que fundamentan lenguajes reales. Lejos de ser una curiosidad histórica, el Cálculo λ sigue vigente como herramienta para diseñar lenguajes, especificar semánticas y analizar programas, constituyéndose en el verdadero “núcleo” del estilo funcional en la programación contemporánea.

Referencias

- [1] M. Felleisen y M. Flatt, *Programming Languages and Lambda Calculi*. 2003.
- [2] M. Gabbrielli y S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2023.
- [3] S. Krishnamurthi, *Programming Languages: Application and Interpretation*. 2007.
- [4] K. D. Lee, *Foundations of Programming Languages*. Springer, 2017.
- [5] F. Miranda y L. González, *Notas de Clase de Lenguajes de Programación*, Facultad de Ciencias, UNAM, 2017.
- [6] H. R. Nielson y F. Nielson, *Semantics with Applications: An Appetizer*. Springer, 2007.
- [7] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.
- [8] K. Ramírez, M. Soto y J. Enríquez, *Notas de Clase de Lenguajes de Programación*, Facultad de Ciencias, UNAM, 2022.
- [9] R. Rojas, *A Tutorial Introduction to the Lambda Calculus*. 2019.
- [10] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

³call-by-name.

⁴call-by-value.