Unraveling Lambda Calculus: Developing a Tool for Parsing and Reducing Lambda Expressions

López Jorge[1], Bernal Rubén[2], Irrazabal Emanuel[3]

[1] Facultad de Cs. Ex. y Naturales y Agrimensura. Universidad Nacional del Nordeste, Corrientes (3400), Argentina
jodaniel.lopez213@gmail.com

[2] Facultad de Cs. Ex. y Naturales y Agrimensura. Universidad Nacional del Nordeste, Corrientes (3400), Argentina
rbernal@exa.unne.edu.ar

[3] Facultad de Cs. Ex. y Naturales y Agrimensura. Universidad Nacional del Nordeste, Corrientes (3400), Argentina
eirrazabal@exa.unne.edu.ar

Abstract. This article presents the development of a tool for the syntactic analysis and evaluation of lambda calculus expressions. This formalism, introduced by Alonzo Church, constitutes one of the fundamental models of theoretical computation and has served as a conceptual basis for the design of numerous functional programming languages.

The main objective of this work is to implement a parser capable of recognizing and processing lambda expressions, allowing their structured representation using abstract syntax trees (ASTs) and their subsequent evaluation through the β-reduction mechanism. The tool was developed in Python using the Parsy library, a system of parser combinators that allows for the modular and declarative construction of parsers.

In addition to the parser, a lambda expression evaluator is implemented that applies successive reductions until a normal form is reached whenever possible. To validate the system's operation, several examples of well-formed expressions and cases of syntactic error are presented. Finally, the reflective capacity of lambda calculus is explored by constructing a self-interpreter, demonstrating its expressive power and its equivalence with other universal models of computation.

This work illustrates both the conceptual simplicity and the computational depth of lambda calculus, highlighting its relevance in the study of computability and the semantics of programming languages.

Keywords: Lambda calculus · Syntax analyzer · Self-interpreter

Desentrañando el Cálculo Lambda: desarrollo de una herramienta para el parsing y reducción de expresiones lambda

López Jorge[1], Bernal Rubén[2], Irrazabal Emanuel[3]

[1] Facultad de Cs. Ex. y Naturales y Agrimensura. Universidad Nacional del Nordeste, Corrientes (3400), Argentina
jodaniel.lopez213@gmail.com

[2] Facultad de Cs. Ex. y Naturales y Agrimensura. Universidad Nacional del Nordeste, Corrientes (3400), Argentina
rbernal@exa.unne.edu.ar

[3] Facultad de Cs. Ex. y Naturales y Agrimensura. Universidad Nacional del Nordeste, Corrientes (3400), Argentina
eirrazabal@exa.unne.edu.ar

Abstracto. En este artículo se presenta el desarrollo de una herramienta para el análisis sintáctico y la evaluación de expresiones del cálculo lambda. Este formalismo, introducido por Alonzo Church, constituye uno de los modelos fundamentales de la computación teórica y ha servido como base conceptual para el diseño de numerosos lenguajes de programación funcionales.

El objetivo principal del trabajo es implementar un parser capaz de reconocer y procesar expresiones lambda, permitiendo su representación estructurada mediante árboles sintácticos abstractos (AST) y su posterior evaluación a través del mecanismo de reducción β. La herramienta fue desarrollada en Python utilizando la biblioteca Parsy, un sistema de combinadores de parsers que permite construir analizadores sintácticos de forma modular y declarativa.

Además del parser, se implementa un evaluador de expresiones lambda que aplica reducciones sucesivas hasta alcanzar una forma normal cuando es posible. Para validar el funcionamiento del sistema se presentan diversos ejemplos de expresiones bien formadas y casos de error sintáctico. Finalmente, se explora la capacidad reflexiva del cálculo lambda mediante la construcción de un auto-intérprete, demostrando su potencia expresiva y su equivalencia con otros modelos universales de computación.

Este trabajo ilustra tanto la simplicidad conceptual como la profundidad computacional del cálculo lambda, destacando su relevancia en el estudio de la computabilidad y la semántica de lenguajes de programación.

Palabras clave: Cálculo lambda · Analizador sintáctico · Auto-intérprete

1 Introducción

El cálculo lambda fue presentado en 1936 por Alonzo Church como una forma de formalizar el concepto de computabilidad efectiva y demostrar la irresolubilidad del *Entscheidungsproblem* (Church, 1936), se puede considerar como el lenguaje de

programación universal más pequeño del mundo. Es universal en el sentido de que cualquier función computable puede expresarse y evaluarse utilizando este formalismo, y por lo tanto, equivalente a las máquinas de Turing (Turing, 1937). Es por esto que resulta de interés de este trabajo como sistema formal de computación.

El objetivo principal de este proyecto es la construcción de un analizador sintáctico (parser) de expresiones lambda y demostrar la potencia y la universalidad del cálculo lambda mediante el desarrollo de un parser funcional que permita:

- Interpretar y evaluar expresiones lambda de forma estructurada.
- Construir un auto-intérprete dentro del propio sistema, como evidencia de la reflexividad del cálculo lambda.

Este trabajo se desarrolla en lo que sigue, de la siguiente manera: la Sección 2 presenta una breve definición del cálculo lambda, en la Sección 3 se codifica un analizador sintáctico (parser) y un evaluador de expresiones lambda, en la Sección 4 se hace uso del parser para verificar su funcionamiento con un conjunto seleccionado de expresiones lambda, en la Sección 5 se propone un auto-intérprete de expresiones lambda, es decir, una expresión lambda la cual toma otra expresión lambda y la reduce a su forma normal; y finalmente, en la Sección 6 se resume el trabajo con una reflexión final.

2 Definición

El cálculo lambda permite construir términos lambda y realizar operaciones de reducción sobre ellos.

El conjunto de λ-términos (notación Λ) se construye a partir de un conjunto infinito de variables $V = \{v_0, v_1, v_2, ...\}$ usando aplicación y abstracción de funciones (Barendregt, 1984).

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda && \text{(variable)} \\ M, N \in \Lambda &\Rightarrow (MN) \in \Lambda && \text{(aplicación)} \\ M \in \Lambda, x \in V &\Rightarrow (\lambda x.M) \in \Lambda && \text{(abstracción)} \end{aligned}$$

O también expresándolo como una gramática libre de contexto mediante notación BNF:

$$\begin{aligned} <expresión> &:= <abstracción> \\ &\mid <aplicación> \\ &\mid <variable> \\ <abstracción> &:= \lambda\ <variable>\ .\ <expresión> \\ <aplicación> &:= (<expresión>\ <expresión>) \\ <variable> &:= a \mid b \mid c \mid ... \end{aligned}$$

Una expresión puede estar entre paréntesis para mayor claridad, es decir, si E es una expresión, (E) es la misma expresión. Las únicas palabras clave utilizadas en el

lenguaje son λ y el punto. Para evitar saturar las expresiones con paréntesis, adoptamos la convención de que la función de aplicación asocia desde la izquierda, es decir, la expresión:

$$E_1 E_2 E_3 ... E_n$$

se evalúa como si fuera la expresión:

$$(...((E_1 E_2) E_3)...E_n)$$

Como se observó en la definición de expresiones anterior, una variable $(a, b, c, ...)$ es una expresión. Un ejemplo de función es el siguiente:

$$\lambda x.x$$

Esta expresión define la función identidad. El nombre después del λ es el identificador del argumento de esta función. La expresión después del punto (en este caso, una x) se denomina "cuerpo" de la definición.

Las funciones se pueden aplicar a expresiones. Un ejemplo de aplicación es:

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

Esta es la función identidad aplicada a y. Se utilizan paréntesis para mayor claridad y evitar ambigüedades. Las aplicaciones de la función se evalúan sustituyendo el valor del argumento x (en este caso y) en el cuerpo de la definición de la función, es decir,

$$(\lambda x.x)y = [y/x]x = y$$

En esta transformación, se utiliza la notación $[y/x]$ para indicar que todas las ocurrencias de x se sustituyen por y en la expresión de la derecha. A la sustitución de términos en otros se llama "reducción β" y representa el principal mecanismo para realizar cómputos en el cálculo lambda.

Los nombres de los argumentos en las definiciones de funciones no tienen ningún significado por sí mismos. Son simplemente "marcadores de posición", es decir, se utilizan para indicar cómo reorganizar los argumentos de la función al evaluarla. Por lo tanto:

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

y así sucesivamente. Usamos el símbolo "$\equiv$" para indicar que cuando $A \equiv B$, A es solo un sinónimo de B. A esta transformación de variables se la denomina "conversión α". Esto es útil para evitar colisiones de nombres cuando se hacen sustituciones pero no cambia el comportamiento de la función.

3 Parser

El parser se realizó con el lenguaje de programación Python haciendo uso de la librería Parsy (Parsy Developers, 2025), la cual es un *parser combinator* monádico para gramáticas LL(k) (*Left to right, Leftmost derivation*) y el proyecto se codificó en Google Colab, el entorno colaborativo basado en la nube de Google para escribir y ejecutar

código en Python, por lo que el proyecto se encuentra disponible para acceder y ejecutar desde cualquier navegador (López, 2025).

El código está estructurado en cuatro partes:

1. Primero se definen las clases que se usarán para conformar el árbol sintáctico abstracto (AST). Éstas son cuatro, una clase abstracta la cual implementa los métodos comunes a todos los términos λ, y luego una clase por cada construcción gramatical del cálculo lambda, es decir, una clase para las variables, una clase para las abstracciones y otra para las aplicaciones. Éstas clases heredarán e implementarán los métodos definidos en la clase base entre ellos, los métodos necesarios para la reducción-β.
2. Se construyen distintos parsers individuales para aceptar partes de la gramática que luego combinándolos y utilizando las clases definidas previamente se completa el parser que se usará para aceptar strings. Los parsers individuales realizan tareas muy básicas como por ejemplo aceptar un paréntesis izquierdo, otro acepta un caractér "λ", otro una letra y otro un paréntesis derecho, pareciera que no son muy útiles pero si se los ejecuta en secuencia se obtiene un parser de abstracciones lambdas y de esta manera, utilizando y reutilizando parsers sencillos se construyen parsers para expresiones más complejas. Una observación que se puede hacer es que no hay necesidad de desarrollar un lexer y un parser por separado, cada parser individual es autocontenido y no necesita de estructuras externas para aceptar o rechazar strings.
3. Se elabora un evaluador que se encarga de realizar reducciones-β a expresiones que construye el parser. El mismo está implementado a través de tres funciones, una que verifica si el término actual (la raíz del AST en principio) se puede reducir, esta función es llamada cada que vez que la función encargada de la reducción reduce el AST de manera recursiva, y finalmente la función de reducción es llamada dentro de otra que mediante un bucle infinito reduce el árbol hasta llegar a una forma normal, es decir, hasta llegar a una expresión irreducible, además de llevar un contador para cortar el bucle en caso de que se exceda un límite, saliendo así de bucles infinitos generados por las expresiones lambdas irreducibles. El evaluador de expresiones λ es necesario para implementar el auto-intérprete que se propone como caso de estudio en la Sección 5.
4. Como último, se formulan ejemplos de strings los cuales el parser debe aceptar o rechazar debidamente. Los detalles de la implementación se encuentran en la Sección 4.

4 Pruebas

En esta sección se mostrará el resultado de la ejecución de ocho ejemplos, cinco de los cuales son strings bien formados por los que el parser deberá armar el AST correspondiente y además, deberá evaluar la expresión y reducirla a una forma normal. Los otros tres casos son de strings mal formados por los que el parser deberá rechazarlos junto con un mensaje de error de sintaxis.

En la Figura 1 se puede ver el caso más simple posible el cual es la una sola variable que como no es una aplicación no se puede reducir, es decir, ya está en forma normal.

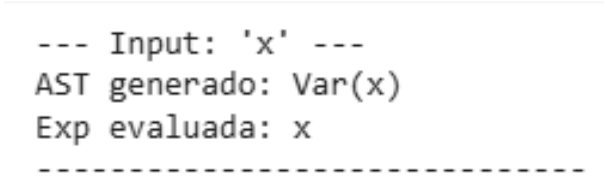

Figura 1. string compuesto por una variable

En la Figura 2 se puede ver que el string corresponde a una abstracción, más concretamente de la función identidad (para hacer fácil de tipear se usa la barra invertida como reemplazo de λ, pero el string "\x.x" debe interpretarse como la expresión $\lambda x.x$).

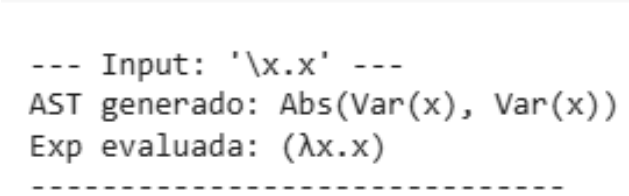

Figura 2. string compuesto por una abstracción

De manera similar al caso anterior la expresión se encuentra en su forma normal por ser una abstracción.

En la Figura 3 se aprecia una aplicación sencilla, la expresión es $(\lambda x.x)y$, la cual no está en forma normal, es decir, es reducible.

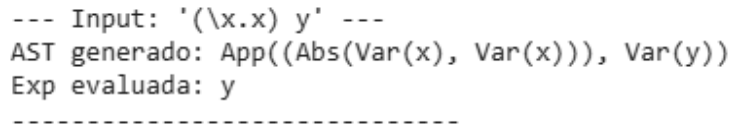

Figura 3. Aplicación de variable a abstracción

Como se vio en la Sección 2 la expresión se reduce de la siguiente manera:

$$(\lambda x.x)y = [y/x]x = y$$

En la Figura 4 se puede observar una expresión más compleja $(\lambda x.x)(\lambda y.y)$ pero el proceso de reducción es exactamente el mismo:

$$(\lambda x.x)(\lambda y.y) = [(\lambda y.y)/x] = (\lambda y.y)$$

como $(\lambda y.y)$ no es una aplicación, no se puede seguir reduciendo.

```
--- Input: '(\x.x) (\y.y)' ---
AST generado: App((Abs(Var(x), Var(x))), (Abs(Var(y), Var(y))))
Exp evaluada: (λy.y)
------------------------------
```

Figura 4. Aplicación de abstracción a abstracción

En la Figura 5 se aprecia un caso más complejo del tipo $E_1E_2E_3$, este es un buen caso para resaltar la capacidad del parser para asociar correctamente los términos hacia la izquierda $((E_1E_2)E_3)$ por lo que la evaluación de la expresión resulta:

$$\begin{gathered}(\lambda f.\lambda x.f(f\,x))\,(\lambda y.y)\,z = \\ (\lambda x.(\lambda y.y)\,((\lambda y.y)\,x))\,z = \\ (\lambda y.y)\,((\lambda y.y)\,z) = \\ (\lambda y.y)\,z = \\ z\end{gathered}$$

```
--- Input: '(\f.\x.f (f x)) (\y.y) z' ---
AST generado: App(App((Abs(Var(f), Abs(Var(x), App(Var(f), App(Var(f), Var(x))))), (Abs(Var(y), Var(y)))), Var(z))
Exp evaluada: z
------------------------------
```

Figura 5. Caso de múltiple abstracción

Ahora se verán casos en donde los strings están mal formados y el parser deberá rechazarlos.

De manera acertada el parser rechaza el string vacío y en el error se ve que el parser esperaba alguno de los siguientes símbolos: "(" (para leer una abstracción o aplicación), "[a-z]" (es una expresión regular compuesta por cualquier letra entre *a* y *z*), "\\" y "λ" (ambos denotan el primer caracter de un string que contenga una abstraccíón al principio). Además el parser indica en que posición del string esperaba encontrar alguno de dichos caracters, en este caso en la posición cero, es decir, al inicio del string.

```
--- Input: '' ---
---------------------------------------------------------------------------
ParseError                                Traceback (most recent call last)
/tmp/ipython-input-13-4104509950.py in <cell line: 0>()
     17 expr_str = ''
     18 print(f"--- Input: '{expr_str}' ---")
---> 19 parsed_ast = lambda_parser.parse(expr_str)
     20 print(f"AST generado: {parsed_ast}")
     21 evaluated_term = pprint(evaluate(parsed_ast))

                              1 frames
/usr/local/lib/python3.11/dist-packages/parsy/__init__.py in parse_partial(self, stream)
    110             return (result.value, stream[result.index :])
    111         else:
--> 112             raise ParseError(result.expected, stream, result.furthest)
    113
    114     def bind(self, bind_fn):

ParseError: expected one of '(', '[a-z]', '\\', 'λ' at 0:0
```

Figura 6. Caso de string vacío

En la Figura 7 se ve que el parser esperaba una letra o un espacio en la posición del punto porque el único símbolo válido luego de "\" es una letra denotando una abstracción. En el caso del espacio es porque podría consumirlo y descartarlo para avanzar buscando algún símbolo significativo.

```
--- Input: '\.x' ---
---------------------------------------------------------------------------
ParseError                                Traceback (most recent call last)
/tmp/ipython-input-14-2793965790.py in <cell line: 0>()
     17 expr_str = '\\.x'
     18 print(f"--- Input: '{expr_str}' ---")
---> 19 parsed_ast = lambda_parser.parse(expr_str)
     20 print(f"AST generado: {parsed_ast}")
     21 evaluated_term = pprint(evaluate(parsed_ast))

                              1 frames
/usr/local/lib/python3.11/dist-packages/parsy/__init__.py in parse_partial(self, stream)
    110             return (result.value, stream[result.index :])
    111         else:
--> 112             raise ParseError(result.expected, stream, result.furthest)
    113
    114     def bind(self, bind_fn):

ParseError: expected one of '[a-z]', '\\s+' at 0:1
```

Figura 7. Caso de argumento faltante

En el caso de la Figura 8 hay más posibilidades de caracteres válidos: el "(" porque podría esperar una expresión de la forma $(\lambda x.(\lambda y)...)$, en el caso de "[a-z]" porque esperaba una expresión de la forma $(\lambda x.v)$, el caso "\\" y λ por $\lambda x.\lambda y...$ y finalmente, el caso del espacio en blanco "\\s+" porque de igual manera al ejemplo anterior, un espacio en blanco es un símbolo válido que puede consumir el parser pero solo para descartarlo y seguir buscando.

```
--- Input: '(\x.) y' ---
---------------------------------------------------------------------------
ParseError                                Traceback (most recent call last)
/tmp/ipython-input-15-1259046945.py in <cell line: 0>()
     17 expr_str = '(\\x.) y'
     18 print(f"--- Input: '{expr_str}' ---")
---> 19 parsed_ast = lambda_parser.parse(expr_str)
     20 print(f"AST generado: {parsed_ast}")
     21 evaluated_term = pprint(evaluate(parsed_ast))

                                  1 frames
/usr/local/lib/python3.11/dist-packages/parsy/__init__.py in parse_partial(self, stream)
    110             return (result.value, stream[result.index :])
    111         else:
--> 112             raise ParseError(result.expected, stream, result.furthest)
    113
    114     def bind(self, bind_fn):

ParseError: expected one of '(', '[a-z]', '\\', '\\s+', 'λ' at 0:4
```

Figura 8. Caso de cuerpo de función faltante

5 Caso de estudio

En este caso de estudio se propone una codificación interna para términos lambdas que asigna a $M \in \Lambda$ una forma normal $\lceil M \rceil$ (Mogensen, 1992). Para ello primero se definirán una serie de notaciones y convenciones, por ejemplo, la expresión $\lambda x_1 x_2 ... x_n.M$ como abreviación de $\lambda x_1 \lambda x_2 ... \lambda x_n.M$ y $M_1 M_2 M_n$ como abreviación de $(...(M_1 M_2)...M_n)$.

Dos términos-λ son considerados *idénticos* si solamente difieren en los nombres de las variables ligadas (es decir, son α-convertibles) e *iguales* si pueden ser β-reducidos a términos-λ idénticos. Se usará el símbolo $\equiv$ para identidad y $\overset{\beta}{=}$ para igualdad. $M \to N$ significa que M se reduce a N en una reducción-β. $M \twoheadrightarrow N$ significa que M se reduce a N en cero o más reducciones-β.

Cuando un término-λ M tiene forma normal, usamos la notación NF_M para significar la forma normal de M.

Un *esquema de representación* para el lambda cálculo es mapeo inyectivo $\lceil \cdot \rceil : \Lambda \to NF_M$. Es decir, $\lceil \cdot \rceil$ representará cualquier término-λ por un término-λ en forma normal.

Un *autointérprete* es un término-λ E, tal que

$$E\ulcorner M\urcorner \overset{\beta}{=} M$$

para cualquier término-λ M. Esto es, E aplicado a la representación de M es igual a M misma.

El esquema de representación para el lambda cálculo es el siguiente

$$\begin{aligned}\ulcorner x\urcorner &\equiv \lambda abc.ax\\ \ulcorner MN\urcorner &\equiv \lambda abc.b\ulcorner M\urcorner\ulcorner N\urcorner\\ \ulcorner \lambda x.M\urcorner &\equiv \lambda abc.c(\lambda x.\ulcorner M\urcorner)\end{aligned}$$

Como un ejemplo, la codificación de $\lambda x.xx$ es

$$\lambda abc.c(\lambda x.(\lambda abc.b(\lambda abc.ax)(\lambda abc.ax)))$$

Presentaremos inicialmente (para facilitar la lectura) el autointérprete. Luego usaremos la codificación anterior para convertir esto al cálculo lambda puro.

$$\begin{aligned}E\ulcorner x\urcorner &\equiv x\\ E\ulcorner (MN)\urcorner &\equiv E\ulcorner M\urcorner E\ulcorner N\urcorner\\ E\ulcorner \lambda v.M\urcorner &\equiv \lambda v.E\ulcorner (Mv)\urcorner\end{aligned}$$

El intérprete E tiene una definición *recursiva* y en cálculo lambda la recursión se logra mediante el uso de un *combinador de punto fijo*. El *punto fijo* de un operador o función es un valor que no cambia bajo una transformación dada. Por ejemplo, la operación del cuadrado de un número tiene dos puntos fijos 0 y 1, dado que $0^2 = 0$ y $1^2 = 1$; y la función sucesor no tiene ningún punto fijo porque $n + 1 \neq n$ para todo n.

En el cálculo lambda decimos que un término-λ M es un combinador de punto fijo de F si:

$$MF \equiv F(MF)$$

entonces MF es un punto fijo de F. (Hindley & Seldin, 2008)

Ahora codificamos la sintaxis como términos-λ, reemplazando el *pattern matching* por aplicación, y usando el combinador de punto fijo Y para eliminar la recursión explícita. Esto produce el auto-intérprete completo.

$$\begin{aligned}E \equiv Y\lambda e.\lambda m.m(\lambda x.x)\\ (\lambda mn.(e\,m)(e\,m))\\ (\lambda mv.e(m\,v))\end{aligned}$$

donde

$$Y \equiv \lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x))$$

El siguiente cómputo verifica que Yg es efectivamente un punto fijo de la función g:

$$
\begin{aligned}
Y\,g &\equiv \lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x))\,g && \text{por la definición de } Y.\\
&\equiv (\lambda x.g(x\,x))(\lambda x.g(x\,x)) && \text{por reducción-}\beta\\
&\equiv g\,((\lambda x.g(x\,x))(\lambda x.g(x\,x))) && \text{por reducción-}\beta\\
&\equiv g\,(Y\,g) && \text{por el segundo resultado arriba}
\end{aligned}
$$

Mediante un razonamiento similar al anterior, se puede demostrar que es posible reducir $E\lceil N\rceil$ a un término-λ idéntico a N.

6 Conclusión

A lo largo de este trabajo, se presentó una introducción accesible pero rigurosa al cálculo lambda, destacando su rol fundamental como modelo de computación y base teórica para los lenguajes funcionales. Se logró implementar un parser funcional utilizando la librería Parsy en Python, capaz de interpretar y evaluar expresiones lambda correctamente, incluyendo casos válidos e inválidos.

Además, se exploró la capacidad reflexiva del cálculo lambda mediante la implementación de un auto intérprete, demostrando que este formalismo posee la potencia suficiente para simular su propia evaluación. Este ejercicio no solo evidencia la expresividad del sistema, sino que también refuerza su equivalencia con otros modelos universales como la máquina de Turing.

En conjunto, este proyecto pone de manifiesto tanto la simplicidad como la profundidad del cálculo lambda, consolidándolo como una herramienta fundamental para el estudio de la computabilidad y la semántica de los lenguajes de programación.

Referencias

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (Vol. 103). North-Holland Publishing Co.

Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, *58*(2), 345-363. https://doi.org/10.2307/2371045

Hindley, J. R., & Seldin, J. P. (2008). *Lambda-Calculus and Combinators: An Introduction* (2.ª ed.). Cambridge University Press.

López, J. D. (2025, junio 21). *Cálculo Lambda Parser y Evaluador*. https://colab.research.google.com/drive/1rjj2qrVJVXgbSiSSSNCCSx9io9ia-0AS?usp=sharing

Mogensen, T. Æ. (1992). Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, *2*(3), 345-364. https://doi.org/10.1017/S0956796800000423

Parsy Developers. (2025). *Parsy* (Versión 2.2) [Software]. https://github.com/python-parsy/parsy

Turing, A. M. (1937). Computability and λ-Definability. *The Journal of Symbolic Logic*, *2*(4), 153-163. https://doi.org/10.2307/2268280