Lenguajes del Programación
Unidad 2: Especificación Formal de los Lenguajes de Programación

Expresiones `let`

Manuel Soto Romero*

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota estudiaremos las expresiones `let`, un mecanismo esencial en los lenguajes de programación que permite declarar variables locales con un alcance bien definido. Partiremos de ejemplos en Haskell y Racket para motivar su funcionamiento, y después extenderemos nuestro lenguaje MiniLisp incorporando este nuevo constructor. A lo largo de la nota analizaremos su sintaxis concreta y abstracta, y precisaremos su semántica operacional, lo que nos conducirá al estudio formal de la sustitución y su relación con el Cálculo λ. Discutiremos también cómo estas expresiones sirven de base para comprender estrategias de evaluación glotonas y perezosas, y finalmente introduciremos los índices de Bruijn como una solución elegante a los problemas derivados del uso de identificadores, preparando el terreno para un razonamiento más robusto en teoría de lenguajes y compiladores.

1. Definición Informal

Las expresiones `let` son una construcción común en muchos lenguajes de programación que permite la creación de variables locales con un alcance limitado. Estas expresiones son fundamentales en la teoría de lenguajes de programación porque facilitan el entendimiento y la implementación de conceptos importantes como el alcance de las variables, la evaluación de expresiones y la optimización del código.

Funcionamiento de las Expresiones `let`

En general, una expresiones `let` permite definir variables que son locales a un bloque de código. Estas variables pueden ser utilizadas dentro del cuerpo de la expresión `let` pero no fuera de ella. El alcance de las variable definidas por `let` es, por tanto, limitado al bloque donde son declaradas.

Intuitivamente (y no formalmente), una expresión `let` consta de tres elementos básicos:

- **Identificadores:** El nombre de la variable a declarar.
- **Valores:** El valor de la variable que se está declarando.
- **Cuerpo:** El cuerpo donde serán usadas dichas declaraciones.

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

Ejemplo en Haskell

En Haskell, las expresiones `let` son utilizadas para introducir variables locales en una expresión:

```
-- Ejemplo de expresiï¿œn let en Haskell
calcularArea :: Double -> Double -> Double
calcularArea base altura =
   let area = base * altura / 2 in
      area
```

- En Haskell, la sintaxis `let <id> = <valor> in <cuerpo>` permite declarar `area` como una variable local dentro de `calcularArea`.
- `area` es calculada como `base * altura / 2` y se utiliza dentro de la expresión `in` para devolver el resultado.
- Las variables definidas en `let` sólo son accesibles dentro del alcance de `in`.

Ejemplo en Racket

Racket es una implementación moderna de Lisp por lo que nos servirá para entender muchas de las construcciones que añadiremos a nuestro pequeño MiniLisp.

En Racket, una expresión `let` también se utiliza para definir variables locales:

```
;; Versiï¿œn de Racket para calcular el ï¿œrea.
;; Usamos tambiï¿œn expresiones let.

(define (calcular-area base altura)
   (let ([area (/ (* base altura) 2)])
      area))
```

- La sintaxis `(let ([<id> <valor>]) <cuerpo>)` define `area` como una variable local.
- La variable `area` se calcula usando la expresión `(/ (* base altura) 2)`.
- `area` es accesible dentro del cuerpo `let`, devolviendo el resultado calculado.

Las expresiones `let` son importantes en la teoría de lenguajes de programación y las estudiaremos por motivos didácticos pues:

- Permiten definir variables con un alcance local, lo cual es fundamental para la organización y comprensión del código. Este enfoque local evita colisiones de nombres y errores de referencia.
- Las expresiones `let` permiten dividir cálculos complejos en pasos más pequeño y manejables, facilitando la legibilidad y el mantenimiento del código.
- Las optimizaciones en los compiladores pueden aprovechar las expresiones `let` para mejorar el rendimiento del código, mediante la eliminación de subexpresiones comunes y la propagación de constantes.

- En la semántica formal, las expresiones `let` se alinean bien con el Cálculo λ, facilitando el razonamiento matemático sobre el comportamiento de los programas.
- Facilitan la reutilización de cálculos y la definición de subexpresiones dentro de funciones, promoviendo la modularidad y la claridad.
- Estudian cómo se evalúan las expresiones y el orden en que se procesan las declaraciones, proporcionando una base sólida para comprender técnicas avanzadas como el uso de estrategias de evaluación (glotona vs. perezosa).

Las expresiones `let` son una herramienta poderosa en la programación y en el estudio de lenguajes de programación, ayudando a estructurar y organizar el código de manera efectiva y a comprender mejor el funcionamiento interno de los lenguajes y sus compiladores.

2. Sintaxis Concreta

Ahora que hemos ganado intuición en el funcionamiento de este tipo de expresiones, las añadiremos a nuestro lenguaje para estudiar todas las decisiones de diseño involucradas. Sin embargo, con fines didácticos comenzaremos con una versión simplificada de estas expresiones. La sintaxis concreta de nuestra nueva versión de MiniLisp se muestra a continuación[1]:

```
<Expr> ::= <Id>
         | <Int>
         | <Bool>
         | (+ <Expr> <Expr>)
         | (- <Expr> <Expr>)
         | (not <Expr>)
         | (let (<Id> <Expr>) <Expr>)

<Int> := ... | -1 | 0 | 2 | ...

<Bool> := #f | #t

<Id> := a | b | c | foo | goo | ...
```

Ejemplos de Expresiones

Algunos ejemplos de nuevas expresiones que podemos representar con este lenguaje:

- `(let (a 2) (+ a a))`
- `(let (a #t) (let (b (not a)) (not b)))`

[1] Abusamos de la notación de las gramáticas en EBNF para mostrar nuestras gramáticas a partir de esta nota con el fin de hacer más claras la adiciones que realicemos. El significado de los símbolos "..." puede deducirse de los ejemplos o preguntando directamente en clase.

Ejercicio 1

Antes de estudiar la semántica... ¿cuál sería el resultado de las dos expresiones anteriores? Si no te es claro, intenta construir su equivalente en Haskell o Racket y analiza los resultados.

3. Sintaxis Abstracta

Para definir la sintaxis abstracta de nuestros dos nuevos constructores quizá valga la pena pensar en la implementación de ellos. ¿Cómo podemos representar los identificadores en un lenguaje de programación anfitrión? Mediante cadenas.

- $Id(s)$ es un ASA si s : String.

$$\frac{s : \mathsf{String}}{Id(s)\ \mathsf{ASA}}$$

Para el caso de nuestra estructura `let`, recordemos sus tres componentes: (1) identificador, (2) valor y (3) cuerpo. Por lo tanto, necesitamos un árbol con 3 ramas. El identificador, como dijimos puede ser una cadena mientras que el valor y el cuerpo son otros árboles al poder ser llenados con otras expresiones del lenguaje.

- $Let(i, v, c)$ es un ASA si i : String, v ASA y c ASA.

$$\frac{i : \mathsf{String} \quad v\ \mathsf{ASA} \quad c\ \mathsf{ASA}}{Let(i, v, c)\ \mathsf{ASA}}$$

Ejercicio 2

¿Cuáles son los ASA correspondientes a las expresiones en sintaxis concreta que definimos anteriormente?

4. Semántica Operacional

Para entender la semántica dinámica de nuestro lenguaje, analicemos nuestras expresiones de ejemplo. ¿Cómo debería evaluarse esta expresión?

```
(let (a 2)
   (+ a a))
```

- La expresión que guía la evaluación es el cuerpo `(+ a a)`.
- Dado que `a` se declaró con el valor 2, debemos reemplazar este valor con un 2: `(+ 2 2)`
- Evaluamos y obtenemos 4 como resultado.

Ejercicio 3

Repite este análisis con nuestra otra expresión.

Con estos dos análisis se hace explícita la necesitad de contar con un algoritmo de sustición mismo que definiremos en la siguiente nota.

Sustitución

En el contexto de la Teoría de la Computación, la sustitución es un concepto fundamental que se refiere al proceso de reemplazar una variable por un valor o una expresión en una estructura específica (por ejemplo una fórmula proposicional o un programa). La sustitución además es un mecanismo esencial en modelos de cómputo como el Cálculo λ, por ende en lenguajes de programación funcionales, y en la semántica formal de los lenguajes de programación.

Podemos definir a la sustitución formalmente como la operación de tomar una expresión y reemplazar todas las ocurrencias de una variable en esa expresión por otra expresión o valor. En notación matemática, la sustitución de la variable x por la expresión e en una expresión f se denota como $f[x := e]$.

Por ejemplo, en la Lógica Proposicional:

- Si tenemos una fórmula $(p \land q \to r)\,[q := (s \lor t)]$, la sustitución indica reemplazar q por $(s \lor t)$ en $(p \land q \to r)$, resultando en $(p \land (s \lor t) \to r)$.

Ejemplo de Sustitución

Retomando el nuevo constructor `let` que añadimos a MiniLisp. Supongamos que tenemos la siguiente expresión:

```
(let (a 5)
   (+ a 3))
```

En este caso, la sustitución de `a` por `5` en `(+ a 3)` resulta en `(+ 5 3)`.

Importancia de la Sustitución en el Diseño de Lenguajes de Programación

- La sustitución es un paso esencial en la evaluación de expresiones, especialmente en lenguajes funcionales donde las funciones son tratadas como valores de primera clase. Permite aplicar funciones a argumentos al sustituir parámetros formales por parámetros reales, evaluando el cuerpo de la función con estos valores.

- Facilita el manejo del alcance de las variables, asegurando que las variables locales no interfieran con otras partes del código. En expresiones `let` y funciones anidadas (como veremos más adelante), la sustitución garantiza que las variables se resuelvan correctamente dentro de su alcance.

- Los compiladores utilizan sustitución para optimizar el código mediante técnicas como la propagación de constantes, eliminando la necesidad de volver a calcular valores conocidos. Permite identificar y reemplazar subexpresiones comunes, reduciendo el cálculo redundante y mejorando el rendimiento.

- La sustitución es central en la semántica formal de lenguajes de programación, proporcionando un marco para razonar sobre el comportamiento de los programas de manera precisa y matemática. En el Cálculo λ, la sustitución es la operación fundamental que define cómo se evalúan las funciones.
- Ayuda a prevenir errores relacionados con la declaración de variables al asegurarse de que las sustituciones no introduzcan variables libres inadvertidamente en un alcance donde no deberían estar.
- Es importante en la implementación de características como cerraduras[2] y funciones de orden superior, donde las funciones pueden ser devueltas y aplicadas con diferentes argumentos.

Un Algoritmo de Sustitución para MiniLisp

Retomemos ahora el análisis de los nuevos constructores *Id* y *Let* en MiniLisp. El comportamiento de nuestros nuevos constructores es el siguiente:

Identificadores Los identificadores de variables no tienen un comportamiento por sí mismos. Esto ocurre pues nuestros identificadores sólo pueden tener un valor asociado dentro de una expresión `let`. De esta forma al tratar de evaluar un identificador, deberíamos obtener un error. En esta versión de nuestra semántica simplemente bloquearemos la evaluación al no definir una regla para los identificadores.

Expresiones let Dada una expresión `let` de la forma `(let (i v) c)`, la evaluación de la misma consiste en cambiar todas las apariciones de `i` por en `c` por `v` y luego evaluar la expresión resultante. Por ejemplo en `(let (a 2) (+ a a))`, primero cambiamos en `(+ a a)` las apariciones de `a` por `2` resultando `(+ 2 2)` que al ser evaluada genera un valor de `4`.

La semántica de las expresiones `let` hace explícita la necesidad de contar con un *algoritmo de sustitución* para las expresiones del lenguaje. Definiremos esta algoritmo antes de dar la especificación de la semántica.

Algoritmo de Sustitución para MiniLisp[3]

Se denota a la sustitución por $e_1[x := e_2]$ que se lee como *sustituir en* e_1 *las apariciones de* x *por* e_2. Se define la sustitución sobre la sintaxis abstracta del lenguaje.

- **Identificadores**

  La sustitución de identificadores dependerá del identificador que captura la etiqueta *Id*. Por ejemplo si queremos cambiar x por 4 y tenemos *Id(x)* el resultado será *Id(4)* mientras que si tenemos *Id(y)* no hay nada que cambiar.

$$\begin{array}{llcll} Id(i) & [x := e] & = & Id(e) & Si\ x{=}i \\ Id(i) & [x := e] & = & Id(i) & Si\ x{\neq}\ i \end{array}$$

- **Números**

  Al no haber variables en este tipo de expresiones, la sustitución no tiene efecto alguno.

$$Num(n) \quad [x := e] \quad = \quad Num(n)$$

[2] *Closures*.

[3] Por facilitad a partir de esta sección y en futuras notas, omitiremos las comillas en los nombres de identificadores que definimos en la nota de clase anterior de forma tal que *Id("x")* se representa como $Id(x)$.

- **Booleanos**

  Igual que el caso anterior. Al no haber variables en este tipo de expresiones, la sustitución no tiene efecto alguno.

$$\mathit{Boolean(b)} \quad [x := e] \quad = \quad \mathit{Boolean(b)}$$

- **Operaciones**

  Basta con realizar recursivamente la sustitución en las subexpresiones respectivas.

$$\begin{array}{lllll} \mathit{Add(i,d)} & [x := e] & = & \mathit{Add(i[x:=e],d[x:=e])} \\ \mathit{Sub(i,d)} & [x := e] & = & \mathit{Sub(i[x:=e],d[x:=e])} \\ \mathit{Not(e)} & [x := e] & = & \mathit{Not(e[x:=e])} \end{array}$$

- **Expresiones `let`**

  En este tipo de expresiones se debe tener cuidado, pues la sustitución podría cambiar la semántica de nuestro programa, por ejemplo, si tenemos la expresión `(let (a 2) (+ a a))` y cambiamos $[a := 8]$ obtendríamos `(let (a 2) (+ 8 8))` donde el *ligado* de la variable se pierde. ¿Cómo definimos entonces este caso? Veamos primero algunas definiciones.

Definición 1: Alcance

*El **alcance** de una variable es la región de un programa en la cuál dichas variables alcanzan su valor. En MiniLisp el alcance de toda variable se da en el **cuerpo** de las expresiones* `let`.

Ejemplo 1

En la siguiente expresión el alcance de la variable `x` definida es el cuerpo `(+ x y)`.

```
(let (x y)
   (+ x y))
```

Definición 2: Variable de ligado

*La instancia de **ligado** de una variable es la instancia de una variable que da a ésta su valor. En MiniLisp la variable de ligado siempre será el **identificador** en las expresiones* `let`.

Ejemplo 2

En la siguiente expresión la única variable de ligado es `x`.

```
(let (x y)
   (+ x y))
```

Definición 3: Variable ligada

*Una variable está **ligada** si se encuentra contenida en el alcance de una variable de ligado por su nombre. En MiniLisp todas las variables que sean iguales al **identificador** son de ligado.*

Ejemplo 3

En la siguiente expresión la única variable ligada es la `x` que se encuentra en el alcance de nuestro identificador de ligado.

```
(let (x y)
   (+ x y))
```

Definición 4: Variable libre

*Una variable está **libre** si no se encuentra contenida en el alcance de una variable de ligado por su nombre. En MiniLisp todas las variable que sean distintas al **identificador** son libres.*

Ejemplo 4

En la siguiente expresión la única variable libre es `y` pues no se encuentra ligada a ninguna variable de ligado.

```
(let (x y)
   (+ x y))
```

De esta forma, para poder sustituir en una expresión `let` debemos preocuparnos únicamente por sustituir sólo aquellas variables que se encuentren libres. Es decir, aquellas que sean diferentes al identificador que trae con sigo un `let`.

Si el identificador del `let` a sustituir es igual al de la sustitución, únicamente podemos sustituir la expresión del valor y dejamos el cuerpo intacto.

$$Let(i,v,c)\quad [x := e]\quad =\quad Let(i,v[x:=e],c)\qquad Si\ i{=}x$$

En otro caso podemos sustituir también el cuerpo sin cambiar el alcance.

$$Let(i,v,c)\quad [x := e]\quad =\quad Let(i,v[x:=e],c[x:=e])\qquad Si\ i{\neq}x$$

Con esto en mente, podemos definir las reglas semánticas del lenguaje. Definimos únicamente la regla para nuestro nuevo constructor.

$$\frac{v \Rightarrow v_v \qquad c[i := v_v] \Rightarrow c_v}{Let(i,v,c) \Rightarrow c_v}$$

Ejemplo 5

Derivación de algunas expresiones.

- Sintaxis concreta: `foo`
  Sintaxis abstracta: *Id(foo)*
  Evaluación: Al no contar con reglas para los identificadores este tipo de programas se bloquean.
  Cambiaremos este comportamiento en otra nota.

- Sintaxis concreta: `(let (a 2) (+ a a))`
  Sintaxis abstracta: *Let(a,Num(2),Add(Id(a),Id(a)))*
  Evaluación semántica natural:

$$\dfrac{\dfrac{}{Num(2) \Rightarrow Num(2)} \quad \dfrac{\dfrac{}{Num(2) \Rightarrow Num(2)} \quad \dfrac{}{Num(2) \Rightarrow Num(2)}}{Add(Num(2),Num(2)) \Rightarrow Num(4)}}{Let(a,Num(2),Add(Id(a),Id(a))) \Rightarrow Num(4)}$$

- Sintaxis concreta: `(let (a #t) (let (b (not a)) (not b)))`
  Sintaxis abstracta:
  $Let(a, Boolean(True), Let(b, Not(Id(a)), Not(Id(b))))$
  Evaluación semántica natural:

$$\dfrac{\dfrac{}{Boolean(True) \Rightarrow Boolean(True)} \quad \dfrac{\dfrac{\dfrac{}{Boolean(True) \Rightarrow Boolean(True)}}{Not(Boolean(True)) \Rightarrow Boolean(False)} \quad \dfrac{\dfrac{}{Boolean(False) \Rightarrow Boolean(False)}}{Not(Boolean(False)) \Rightarrow Boolean(True)}}{Let(b,Not(Boolean(True)),Not(Id(b))) \Rightarrow Boolean(True)}}{Let(a,Boolean(True),Let(b,Not(Id(a)),Not(Id(b)))) \Rightarrow Boolean(True)}$$

Ejercicio 4

Define las reglas de semántica estructural correspondientes para *Id* y *Let*.

Un Primer Vistazo a las Estrategias de Evaluación

Dada la siguiente expresión, se tienen dos formas de evaluarla, usaremos sintaxis concreta por simpleza, sin embargo puedes comprobar los pasos usando sintaxis abstracta y nuestras reglas semánticas.

```
(let (x (+ 5 5))
   (let (y (- x 3))
      (+ x y)))
```

La primera consiste en evaluar el valor asociado a cada variable y luego sustituir en el cuerpo de la expresión correspondiente:

```
(let (x (+ 5 5)) (let (y (- x 3)) (+ x y)))
= (let (x 10) (let (y (- x 3)) (+ x y)))
= (let (y (- 10 3)) (+ 10 y))
= (let (y 7) (+ 10 y))
= (+ 10 7) = 17
```

La segunda consiste en sustituir en el cuerpo de la expresión y luego evaluar cada identificador asociado a la expresión correspondiente:

```
(let (x (+ 5 5)) (let (y (- x 3)) (+ x y)))
= (let (y (- (+ 5 5) 3)) (+ (+ 5 5) y))
= (+ (+ 5 5) (- (+ 5 5) 3))
= (+ 10 (- (+ 5 5) 3))
= (+ 10 (- 10 3))
= (+ 10 7)
= 17
```

La primera forma es conocido como estrategia de evaluación ansiosa o glotona y consiste en evaluar cada valor conforme aparece, mientras que la segunda forma, conocida como estrategia de evaluación perezosa evalúa cada valor hasta que sea necesario. Nuestras reglas semánticas definen una estrategia de evaluación glotona, ya que realiza la sustitución correspondiente evaluando la subexpresión del valor contenida en las expresiones `let`.

Más adelante, en otra nota, se profundiza en ambas estrategias de evaluación.

6. Índices de Bruijn

En esta sección, exploraremos los índices de Bruijn, una técnica que simplifica y fortalece la representación de variables en lenguajes de programación. A lo largo del tema, analizaremos los conflictos que pueden surgir al usar identificadores tradicionales, como la captura de variables y la necesidad de renombrarlas, y cómo los índices de Bruijn ofrecen una solución eficaz al identificar las variables por su posición en la estructura de anidamiento en lugar de por nombres. Este enfoque no solo previene la captura de variables sino que también facilita la manipulación formal y las pruebas de expresiones, haciendo que el código sea más robusto y menos propenso a errores.

Conflictos con Identificadores en el Diseño de Lenguajes de Programación

En el diseño de lenguajes de programación, como hemos visto, los identificadores son nombres que se asignan a variables pero también a otros elementos como funciones, por ejemplo. Sin embargo, el uso de identificadores puede presentar varios conflictos:

Captura de Variables Ocurre cuando una variable dentro de un alcance específico es accidentalmente reemplazada por una variable externa a dicho alcance con el mismo nombre. Por ejemplo al diseñar el algoritmo de sustitución para nuestras expresiones `let`, notamos que una variable interna podría ser capturada por una variable externa con el mismo nombre, cambiando el comportamiento esperado del programa.

```
(let (x 2)
   (let (y 3)
      (+ x y)))
```

Si intentamos sustituir `x` por otra expresión en una situación más compleja, podemos alterar el significado original de `x`. Esta noción aunque es capturada por nuestro algoritmo de sustitución es difícil de abstraer.

Renombrado de Variables Otra forma de evitar conflictos de nombres, es renombrando variables para asegurarse de que cada identificador se refiere al valor correcto, por ejemplo usando relaciones de α-equivalencia. Este proceso puede ser tedioso y propenso a errores, especialmente en lenguajes con un manejo manual del alcance de las variables.

Los Índices de Bruijn como Solución

Para evitar estos problemas, los indices de Bruijn ofrecen una alternativa a los identificadores tradicionales. Los índices de Bruijn son un método de representación de variables en distintas expresiones de un lenguaje de programación, donde las variables no se identifican por un nombre, sino por su posición relativa en el programa.

Fueron propuestos por el matemático holandés Nicolaas Govert de Bruijn en 1972 como una manera de simplificar y hacer más robusta la representación de variables en sistemas formales como el Cálculo λ.

Funcionamiento

Los índices de Bruijn son una notación que permite reemplazar el nombre de las variables por números. Este número indica la profundidad del alcance a la que se encuentra cada una de las variables a partir del cuerpo de una expresión. Por ejemplo, dada la siguiente expresión:

```
(let (x 5)
   (+ x x))
```

puede reescribirse usando índices de Bruijn como sigue:

```
(let 5
   (+ <0> <0>))
```

Las variables ligadas son reemplazadas por índices que indican su profundidad de alcance. Al tener una única variable en esta expresión `let`, la profundidad de éste es cero. De la misma forma, el nombre de las variables de ligado se elimina y únicamente se indica el valor asociado dentro del alcance de su definición. Otro ejemplo se tiene en la expresión:

```
(let (x 5)
   (let (y 6)
      (+ x y)))
```

que se reescribe usando índices de Bruijn como sigue:

```
(let 5
   (let 6
      (+ <1> <0>)))
```

La expresión anterior muestra que la suma, definida en la última línea, i.e. `{+ <1> <0>}` contiene en el lado izquierdo una referencia a la variable con profundidad 1 (comenzando desde cero) de alcance y el lado derecho una referencia a la variable con profundidad 0 de alcance. La profundidad se toma a partir del cuerpo actual y hacia arriba. De la misma forma, si el valor asociado a una variable de ligado incluye identificadores de un `let` anterior, este también debe reescribirse mediante índices de Bruijn tomando la posición actual como el índice de profundidad 0. Por ejemplo,

```
(let (x 5)
   (let (y (+ x 7))
      (+ x y)))
```

se reescribe usando índices de Bruijn como sigue:

```
(let 5
   (let (+ <:0> 7)
      (+ <:1> <:0>)))
```

Ventajas

Prevención de Captura de Variables Los índices de Bruijn eliminan el problema de captura porque las variables se identifican por su posición en la estructura de anidamiento, por por un nombre que pueda colisionar.

Simplificación del Renombrado No es necesario realizar renombrados complicados cuando se modifican expresiones. Los índices de Bruijn manejan automáticamente los cambios en el alcance.

Mayor Robustez en la Manipulación Formal En la semántica formal y los lenguajes que requieren manipulación automática de expresiones, los índices de Bruijn facilitan las pruebas y transformaciones de código, ya que reducen la complejidad de las sustituciones y otras operaciones manipulación.

Ejercicio 5

Define una función que, dado un Árbol de Sintaxis Abstracta (ASA) de una expresión en nuestro MiniLisp, lo transforme a una versión equivalente que utilice índices de Bruijn en lugar de identificadores explícitos. Considera cuidadosamente cómo se actualizan los niveles de profundidad cuando aparecen expresiones anidadas. Una vez implementada la función, reflexiona y explica por qué el uso de índices de Bruijn permite verificar estáticamente si una variable ha sido declarada con anterioridad, evitando así errores de referencia a identificadores inexistentes.

7. Conclusión

En conclusión, el estudio de las expresiones `let` nos permitió recorrer un trayecto que va desde la práctica de declarar variables locales hasta las implicaciones formales en la semántica de los lenguajes de programación. Observamos cómo la sustitución se vuelve el eje central de su interpretación, conectando de manera natural con el Cálculo λ y revelando su importancia para la optimización y el diseño de compiladores. Además,

el análisis de estrategias de evaluación mostró cómo distintas decisiones influyen en el comportamiento de los programas, y la introducción de los índices de Bruijn nos ofreció una herramienta poderosa para evitar conflictos de nombres y fortalecer la manipulación formal de expresiones. Con estas ideas, contamos con una base sólida para seguir explorando la semántica formal y los mecanismos que sustentan el diseño riguroso de lenguajes de programación.

Referencias

[1] M. Gabbrielli y S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2023.
[2] S. Krishnamurthi, *Programming Languages: Application and Interpretation*. 2007.
[3] K. D. Lee, *Foundations of Programming Languages*. Springer, 2017.
[4] H. R. Nielson y F. Nielson, *Semantics with Applications: An Appetizer*. Springer, 2007.
[5] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.
[6] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.