Universidad Nacional Autónoma de México

Facultad de Ciencias

Ciencias de la Computación

3 de Noviembre de 2025

Brenda Rodríguez Jiménez: 424143273
Cristian Ignacio Reyna Méndez: 320149579
José Eleazar López Montúfar: 320207219
Mariana López Pérez: 321244763

Introducción

El presente proyecto consiste en formalizar la sintaxis y la semántica de un lenguaje de programación de estilo Lisp, para posteriormente ser implementado en Haskell y Happy. Se busca definir exhaustivamente la sintaxis léxica y libre de contexto, así como su sintaxis abstracta y la eliminación de azúcar sintáctico. Además, se desea establecer la semántica operacional en paso pequeño, modelar ambientes y *bindings*, e ntegrar el régimen de evaluación ansioso con la estrategia de *call by value*.

El objetivo de este trabajo es aprender a analizar los lenguajes de programación como entidades matemáticas para entender cómo se ejecutan y funcionan, qué significan, y cómo corregir los sistemas creados a partir de éstos para hacerlos operar correctamente.

Formalización

1. Sintaxis léxica

A continuación, se definen las clases de *tokens* para el lenguaje:

- **Var** (identificadores de variables)
  Sea $A = [A - Za - z]$. Entonces, $Var = A^+$ *o* $Var = AA^*$
- **Int** (números enteros)
  Sea $D = [0 - 9]$ y $Z = [1 - 9]$. Entonces, $Int = 0 + ZD^* + -ZD^*$
- **Bool** (literales booleanas)
  $Bool = \#t + \#f$
- **Signos de puntuación**
  $(\; + \;) \; + \; [\; + \;] \; + \; ,$
- **Operadores**
  $+ + - + * + / + <= + >= + \; + \; + = + != +$
- **Palabras reservadas**
  $not + let + let^* + letrec + if0 + if + lambda + add1 + sub1 + sqrt + expt + head + tail +$

$cond + else + fst + snd + pair$

- **Espacios en blanco**
  $[\ \ \backslash t \backslash n \backslash r]^+$ (identifica propiamente los espacios, además de los tabuladores, saltos de línea y retornos de carro).
- **Comentarios**
  $--[\hat{}\backslash n]^*$ (reconoce todas las cadenas que empiezan con $--$ y que contienen cualquier secuencia de caracteres a excepción de un salto de línea).

2. Sintaxis libre de contexto (EBNF)

Se presenta ahora la gramática libre de contexto para el lenguaje. Los metasímbolos propios de la notación EBNF y los símbolos no terminales aparecen con el color negro habitual, mientras que los símbolos terminales de la gramática aparecen en este color.

$$
\begin{aligned}
&< Expr > ::= \ < Var > \ | \ < Bool > \ | \ < Int > \\
&\quad | \ (\ (expt \mid pair)\ < Expr > < Expr > \ ) \\
&\quad | \ (\ < Operador > < Expr > < Expr > \ \{< Expr >\}\ ) \\
&\quad | \ (\ < Unario > < Expr > \ ) \mid \ < Lista > \ | \ < Par > \\
&\quad | \ (\ letrec \ < Asig > < Expr > \ ) \\
&\quad | \ (\ < Lets > (\ < Asig > \mid (\ < Asig > \ \{< Asig >\}\ )\ )\ < Expr > \ ) \\
&\quad | \ (\ < Ifs > < Expr > < Expr > < Expr > \ ) \\
&\quad | \ (\ lambda\ (\ < Var > \ \{< Var >\}\ )\ < Expr > \ ) \\
&\quad | \ (\ < Expr > < Expr > \ \{< Expr >\}\ ) \\
&\quad | \ (\ cond < Claus > \ \{< Claus >\} \ < Else > \ ) \\
&< Operador > ::= \ + \mid - \mid * \mid / \mid = \mid != \mid <= \mid >= \mid \ \mid \\
&< Unario > ::= \ not \mid add1 \mid sub1 \mid sqrt \mid head \mid tail \mid fst \mid snd \mid - \\
&< Lista > ::= \ [\ [\ < Expr > \ \{\ ,\ < Expr >\}\ ]\ ] \\
&< Par > ::= \ (\ < Expr > \ ,\ < Expr > \ ) \\
&< Lets > ::= \ let \mid let^* \\
&< Ifs > ::= \ if0 \mid if \\
&< Claus > ::= \ [\ < Expr > < Expr > \ ] \\
&< Else > ::= \ [\ else \ < Expr > \ ] \\
&< Asig > ::= \ (\ < Var > < Expr > \ )
\end{aligned}
$$

Esta gramática define las construcciones siguientes:

- **Identificadores.** Ejemplo: `x` (en el núcleo: `IdC("x")`).
- **Números.** Ejemplo: `54` (en el núcleo: `NumC(54)`).
- **Booleanos.** Ejemplo: `#t` (en el núcleo: `BoolC(True)`).
- **Operaciones n-arias.** Ejemplo: `(+ 2 5 8)`

(en el núcleo: `AddC(NumC(2), AddC(NumC(5), NumC(8)))`).

- **Operaciones unarias.** Ejemplo: `(sub1 z)` (en el núcleo: `Sub1C(IdC("z"))`)
- **Potencia y pares ordenados.** Ejemplo: `(expt 3 2)`
  (en el núcleo: `ExptC(NumC(3), NumC(2))`) y `(pair x y)`
  (en el núcleo: `PairC(IdC("x"), IdC("z"))`)
- **Lets.** Ejemplo: `(let ((x 3) (y 4)) (+ x z))`
  (en el núcleo: `LetC "x"(NumC(3)) (LetC "z"(NumC(4)) (AddC (IdC("x")) (IdC("z"))))`
- **Ifs.** Ejemplo: `(if (< x 0) -1 1)`
  (en el núcleo: `IfC(LtC(IdC("x"), NumC(0)), NumC(-1), NumC(1))`
- **Condicional.** Ejemplo: `(cond [(< x 0) -1] [(> x 0) 1] [else 0])`
  (en el núcleo: `IfC(LtC(IdC("x"), NumC(0)), NumC(-1), IfC(GtC(IdC("x"), NumC(0)), NumC(1), NumC(0)))`
  En el caso del condicional por clausulado, las restricciones de uso del *else* son las siguientes:
  - Solo puede aparecer una vez en la expresión
  - Debe ser la última cláusula
  - No tiene guardia (siempre es verdadera, es el valor por defecto).
- **Funciones.** Ejemplo: `(lambda (x z) (+ x z))`
  (en el núcleo: `FunC "x"(FunC "z"(AddC (IdC("x")) (IdC("z")))))`
- **Aplicaciones.** Ejemplo: `((lambda (x z) (+ x z)) 2 6)`
  (en el núcleo: `AppC(AppC(FunC "x"(FunC "z"(AddC (IdC("x")) (IdC("z"))))) (NumC(2))) (NumC(6))`)

3. Sintaxis Abstracta (ASA)

Representamos a los ASA mediante reglas de inferencia como sigue:

- **Identificadores:**
  $$\frac{s : String}{Id(s)\ ASA}$$
- **Números:**
  $$\frac{n \in \mathbb{Z}}{Num(n)\ ASA}$$
- **Booleanos:**
  $$\frac{b \in \mathbb{B}}{Bool(b)\ ASA}$$
- **Operaciones n-arias:**
  $$\frac{e_i\ \ ASA\ \ \forall i \in \{1, 2,\ ...,\ n\}}{OpN(e_1, e_2,\ ...,\ e_n)\ ASA}$$

En donde $OpN \in \{Add, Sub, Mul, Div, Eq, Neq, Lt, Gt, Lte, Gte, List\}$. Los elementos de este conjunto corresponden, respectivamente, a los operadores de suma, resta, multiplicación, división, igual que, distinto de, menor que, mayor que, menor o igual que, mayor o igual que, y constructor de una lista.
Todas estas construcciones yacen en la superficie del lenguaje, puesto que en el núcleo, estas operaciones solo son necesarias en su versión binaria, como se verá más adelante.

- **Operaciones unarias:**
$$\frac{e\ ASA}{OpU(e)\ ASA}$$
En donde $OpU \in \{Add1, Sub1, Sqrt, Not, Fst, Snd, Head, Tail\}$. Los elementos de este conjunto corresponden, respectivamente, a los operadores de incremento, decremento, raíz cuadrada, negación, proyección del primer elemento, proyección del segundo elemento, cabeza de una lista y cola de una lista.

- **Potencia y pares ordenados:**
$$\frac{base\ ASA \quad pot\ ASA}{Expt(base, pot)\ ASA} \quad \text{y} \quad \frac{i\ ASA \quad d\ ASA}{Pair(i, d)\ ASA}$$

- **Lets:**
$$\frac{i_k : String \quad v_k\ ASA \quad e\ ASA \quad \forall k \in \{1, 2, \ ..., \ n\}}{OpLet([(i_1, v_1), (i_2, v_2), \ ..., \ (i_n, v_n)],\ e)\ ASA}$$
En donde $OpLet \in \{Let, Let*, LetRec\}$.
Estas tres construcciones pertenecen al superficie del lenguaje. Se obtienen a partir de la construcción *App* que se encuentra en el núcleo, como se especificará posteriormente.

- **Ifs:**
$$\frac{e_1\ ASA \quad e_2\ ASA \quad e_3\ ASA}{OpIf(e_1,\ e_2,\ e_3)\ ASA}$$
En donde $OpIf \in \{If, If0\}$.
Más adelante, observaremos cómo es que $if0$ resulta ser azúcar sintáctica de if (pues se trata únicamente del caso especial en el que $e_1 = 0$).

- **Condicional por clausulado:**
$$\frac{guard_i\ ASA \quad v_i\ ASA \quad else\ ASA \quad \forall i \in \{1, 2, \ ..., \ n\}}{Cond([(guard_1, v_1), (guard_2, v_2), \ ..., \ (guard_n, v_n)],\ else)\ ASA}$$
Esta construcción también es azúcar sintáctica de if, pues en realidad consiste solamente en una serie de ifs anidados.

- **Funciones y aplicaciones:**
  $\frac{x_i\ ASA \quad b\ ASA \quad \forall i \in \{1, 2, ..., n\}}{Fun((x_1, x_2, ..., x_n), b)\ ASA}$ y $\frac{f\ ASA \quad v_i\ ASA \quad \forall i \in \{1, 2, ..., n\}}{App(f, (v_1, v_2, ..., v_n))\ ASA}$
  Estas construcciones están en la superficie del lenguaje. En el núcleo, solo son necesarias las funciones que reciben un solo parámetro, y las cuales son aplicadas a un solo argumento. Este tipo de funciones se aplican de manera anidada, simulando así la presencia de múltiples parámetros. Este proceso es conocido como *currificación*.

4. Eliminación de azúcar sintáctica (Desugaring)

En esta sección formalizamos el proceso de eliminación de azúcar sintáctica, cuya finalidad es traducir expresiones de la sintaxis de superficie (`ExprS`) a la sintaxis del núcleo (`ExprC`) del lenguaje.

Este proceso garantiza que toda construcción sintáctica compleja se reescriba únicamente en términos de los constructores fundamentales, preservando su significado operacional. Para ello, empleamos reglas de derivación del estilo SOS, las cuales definen la relación:

$$\mathsf{desugar} : ASA \to ExprC$$

- **Identificadores:** Los identificadores en la superficie del del lenguaje se corresponden directamente al mismo identificador en el núcleo. No necesita simplificarse:

$$\frac{}{\mathsf{desugar}(\texttt{Id(s)}) = \texttt{IdC(s)}}(\mathsf{IdDesugar})$$

- **Números y booleanos:** Para los valores numéricos y booleanos, tenemos casos análogos:

$$\frac{}{\mathsf{desugar}(\texttt{Num(n)}) = \texttt{NumC(n)}}(\mathsf{NumDesugar})$$

$$\frac{}{\mathsf{desugar}(\texttt{Bool(b)}) = \texttt{BoolC(b)}}(\mathsf{BoolDesugar})$$

- **Operaciones n-arias:** Como se mencionó anteriormente, los operadores n-arios pertenecen a la superficie, y dentro del núcleo solo se cuentan con operadores binarias. En particular, el azúcar sintáctica de los operadores artiméticos y relacionales se elimina como sigue:

$$\frac{e_i \in ASA \ \ \forall i \in \{1, 2, ..., n\}}{\mathsf{desugar}(\texttt{OpS}(e_1, e_2, ..., e_n)) = \texttt{OpC}(\mathsf{desugar}(e_1), \mathsf{desugar}(\texttt{OpS}(e_2, ..., e_n)))}(\mathsf{OpAritRelDesugar})$$

  En donde $\texttt{OpS} \in \{Add, Sub, Mul, Div, Eq, Neq, Lt, Gt, Lte, Gte\}$ y `OpC` corresponde a los constructores binarios correspondientes a estos operadores:

$\texttt{OpC} \in \{AddC, SubC, MulC, DivC, EqC, NeqC, LtC, GtC, LteC, GteC\}$.

- **Operaciones unarias:**

$$\frac{}{\textsf{desugar}(\texttt{OpUS(e)}) = \texttt{OpUC}(\textsf{desugar}(\texttt{e}))}(\textsf{OpUDesugar})$$

En donde $\texttt{OpUS} \in \{Add1, Sub1, Sqrt, Not, Fst, Snd\}$ y
$\texttt{OpUC} \in \{Add1C, Sub1C, SqrtC, NotC, FstC, SndC\}$. Cada operador unario de la superficie, se corresponde directamente con un operador unario en el núcleo. El proceso solo aplica desazucaración recursivamente al operando.
Como se verá dentro de poco, las listas pertenecen solo a la superficie del lenguaje: en el núcleo, son desazucaradas haciendo uso de pares. Por lo tanto, para las operaciones unarias de $Head$ y $Tail$ tenemos las siguientes reglas:

$$\frac{}{\textsf{desugar}(\texttt{Head(e)}) = \texttt{FstC}(\textsf{desugar}(\texttt{e}))}(\textsf{HeadDesugar})$$

$$\frac{}{\textsf{desugar}(\texttt{Tail(e)}) = \texttt{SndC}(\textsf{desugar}(\texttt{e}))}(\textsf{TailDesugar})$$

- **Potencia y pares ordenados:**

$$\frac{}{\textsf{desugar}(\texttt{Expt}(b, p)) = \texttt{ExptC}(\textsf{desugar}(e), \textsf{desugar}(p))}(\textsf{ExptDesugar})$$

$$\frac{}{\textsf{desugar}(\texttt{Pair}(e_1, e_2)) = \texttt{PairC}(\textsf{desugar}(e_1), \textsf{desugar}(e_2))}(\textsf{PairDesugar})$$

- **Condicional if:** El proceso de desazucaración para esta operación simplemente consiste en desazucarar cada uno de los tres argumentos (condición, if y else):

$$\frac{}{\textsf{desugar}(\texttt{If}(c,\ t,\ e)) = \texttt{IfC}(\textsf{desugar}(c), \textsf{desugar}(t), \textsf{desugar}(e))}(\textsf{IfDesugar})$$

- **Condicional if0:** El constructor `if0` se transforma en un condicional binario que compara con cero:

$$\frac{}{\textsf{desugar}(\texttt{If0}(e_1,\ e_2,\ e_3)) = \texttt{IfC}(\texttt{EqC}(\textsf{desugar}(e_1), \texttt{NumC}(0)), \textsf{desugar}(e_2), \textsf{desugar}(e_3))}(\textsf{If0Desugar})$$

- **Condicional cond:** Las expresiones `Cond` se desazucaran en expresiones `If` anidadas, en donde se revisa cada guardia hasta que una se cumpla, y la expresión final `else` es la rama por defecto, que siempre es verdadera. (CondDesugar)

$$\frac{[g_i, e_i], \texttt{else} \in ASA \quad \forall i \in \{1, 2, ..., n\}}{\mathsf{desugar}(\texttt{Cond}([(g_1, e_1) \ldots (g_n, e_n)], \texttt{else})) = \texttt{IfC}(g_1, e_1, \texttt{IfC}(g_2, e_2, \ldots, \texttt{IfC}(\texttt{true, else})))}$$

Ligaduras let y let*: Ambas formas se descomponen en secuencias anidadas de `LetC`. La diferencia radica únicamente en el modo de evaluación de las ligaduras. (Let/Let*Desugar)

$$\frac{(x_i, e_i) \in ASA \quad b \in ASA \quad \forall i \in \{1, 2, ..., n\}}{\mathsf{desugar}(\texttt{OpLet}([(x_1, e_1), \ldots, (x_n, e_n)],\ b)) = \begin{array}{l}\texttt{LetC}(x_1, \mathsf{desugar}(e_1), (\texttt{LetC}(x_2, \mathsf{desugar}(e_2), \\ (...(\texttt{LetC}(x_n, \mathsf{desugar}(e_n), \mathsf{desugar}(b)) \ldots)\end{array}}$$

En donde $\texttt{OpLet} \in \{Let, Let*\}$.

Funciones y aplicaciones: Las funciones múltiples se representan mediante currificación (FunDesugar) y las aplicaciones se asocian a la izquierda (AppDesugar). Además, el caso base de las aplicaciones es cuando la lista de argumentos es vacía.

$$\frac{(x_1, \ldots, x_n),\ b \in ASA}{\mathsf{desugar}(\texttt{Fun}(x_1, \ldots, x_n),\ b) = \texttt{LamC}(x_1, (\texttt{LamC}(x_2, (\ldots (\texttt{LamC}(x_n, \mathsf{desugar}(b)) \ldots))}$$

$$\frac{}{\mathsf{desugar}(\texttt{App}(f, [])) = \mathsf{desugar}(f)} (\mathsf{AppDesugarBase})$$

$$\frac{f, (a_1, \ldots, a_n) \in ASA}{\mathsf{desugar}(\texttt{App}(f,\ (a_1, \ldots, a_n)) = \texttt{AppC}(\texttt{AppC}(\ldots \texttt{AppC}(\mathsf{desugar}(f), \mathsf{desugar}(a_1)...), \mathsf{desugar}(a_n))}$$

Recursión letrec: Tenemos dos casos. En el caso base, no hay *bindings*:

$$\frac{}{\mathsf{desugar}(\texttt{LetRec}([], b)) = \mathsf{desugar}(b)} (\mathsf{LetRecDesugarBase})$$

Para el caso recursivo, sea (f, e) el *binding* a la cabeza, y *tail* el resto. La recursión se modela mediante el combinador de punto fijo "Y", por lo que tenemos $\texttt{AppC}(Y, \texttt{LamC}(f, \mathsf{desugar}(e)))$, y luego continuamos recursivamente sobre *tail* (LetRecDesugar):

$$\frac{(f, e), b, tail \in ASA}{\mathsf{desugar}(\texttt{LetRec}((f, e) : tail, b)) = \texttt{LetC}(f,\ \texttt{AppC}(Y, \texttt{LamC}(f, \mathsf{desugar}(e))), \mathsf{desugar}(\texttt{LetRec}(tail, b)))}$$

Pares y listas: Para los pares, se desazucaran ambos componentes, mientras que las listas se construyen con pares anidados terminados en `NilC`.

$$\frac{}{\mathsf{desugar}(\texttt{Pair}(e_1, e_2)) = \texttt{PairC}(\mathsf{desugar}(e_1), \mathsf{desugar}(e_2))}$$

$$\frac{(e_1, \ldots, e_n) \in ASA}{\mathsf{desugar}(\texttt{List}(e_1, \ldots, e_n)) = \texttt{PairC}(\mathsf{desugar}(e_1), \texttt{PairC}(\ldots, \texttt{PairC}(\mathsf{desugar}(e_n), \texttt{NilC})))}$$

En conjunto, estas reglas especifican la traducción completa del lenguaje de superficie al núcleo, eliminando el azúcar sintáctica y unificando la semántica operacional de MiniLisp.

5. Semántica operacional

A continuación, se presenta una formalización de la semántica operacional de MiniLisp.

- **Valores y variables:** Tienen una única regla:
  Sea el conjunto de valores del lenguaje:

$$v \in Val ::= NumV(n) \mid BoolV(b) \mid NilV \mid PairV(v_1, v_2) \mid ClosureV(x, e, \epsilon_0).$$

$$\frac{}{(ValC(x), \epsilon) \to (ValC(x), \epsilon)}(\mathsf{ValStep})$$

  En donde $ValC \in \{NumC, BoolC, NilC\}$

$$\frac{\epsilon(x) = v}{(IdC(x), \epsilon) \to (ValC(v), \epsilon)}(\mathsf{IdStep})$$

- **Operaciones artiméticas y relacionales:** Estas operaciones, al desazucararse, se convierten en binarias. Por tanto, tenemos tres reglas:

$$\frac{(e_1, \epsilon) \to (e_1', \epsilon')}{(OpC(e_1, e_2), \epsilon) \to (OpC(e_1', e_2), \epsilon')}(\mathsf{OpCStep1})$$

$$\frac{(e_2, \epsilon) \to (e_2', \epsilon')}{(OpC(NumC(n), e_2), \epsilon) \to (OpC(NumC(n), e_2'), \epsilon')}(\mathsf{OpCStep2})$$

  En donde $OpC \in \{AddC, SubC, MulC, DivC, EqC, NeqC, LtC, GtC, LteC, GteC, Expt\}$.
  Para la regla 3, no tenemos premisas, puesto que ambos operandos ya son valores. Las conclusiones dependen de la operación en particular:
  - $(AddC(NumC(n), NumC(m)), \epsilon) \to (NumC(n+m), \epsilon)$ (AddCStep3)
  - $(SubC(NumC(n), NumC(m)), \epsilon) \to (NumC(n-m), \epsilon)$ (SubCStep3)
  - $(MulC(NumC(n), NumC(m)), \epsilon) \to (NumC(n*m), \epsilon)$ (MulCStep3)

- $(DivC(NumC(n), NumC(m)), \epsilon) \to (NumC(n/m), \epsilon)$ (DivCStep3)
- $(EqC(NumC(n), NumC(m)), \epsilon) \to (BoolC(n = m), \epsilon)$ (EqCStep3)
- $(NeqC(NumC(n), NumC(m)), \epsilon) \to (BoolC(n \neq m), \epsilon)$ (NeqCStep3)
- $(LtC(NumC(n), NumC(m)), \epsilon) \to (BoolC(n < m), \epsilon)$ (LtCStep3)
- $(GtC(NumC(n), NumC(m)), \epsilon) \to (BoolC(n > m), \epsilon)$ (GtCStep3)
- $(LteC(NumC(n), NumC(m)), \epsilon) \to (BoolC(n \leq m), \epsilon)$ (LteCStep3)
- $(GteC(NumC(n), NumC(m)), \epsilon) \to (BoolC(n \geq m), \epsilon)$ (GteCStep3)
- $(ExptC(NumC(n), NumC(m)), \epsilon) \to (NumC(n^m), \epsilon)$ (ExptCStep3)

- **Operaciones unarias:** Para este tipo de operaciones, tenemos dos reglas:

$$\frac{(e_1, \epsilon) \to (e_1', \epsilon')}{(OpUC(e_1), \epsilon) \to (OpUC(e_1'), \epsilon)} (\text{OpUCStep1})$$

En donde $OpUC \in \{Add1C, Sub1C, SqrtC, NotC, FstC, SndC\}$.
Para la segunda regla, las conclusiones dependen de la operación en particular:

- $(Add1C(NumC(n)), \epsilon) \to (NumC(n+1), \epsilon)$ (Add1CStep2)
- $(Sub1C(NumC(n)), \epsilon) \to (NumC(n-1), \epsilon)$ (Sub1CStep2)
- $(SqrtC(NumC(n)), \epsilon) \to (NumC(\sqrt{n}), \epsilon)$ (SqrtCStep2)
- $(NotC(Bool(b)), \epsilon) \to (BoolC(\neg b), \epsilon)$ (NotCStep2)
- $(FstC(PairC(x, y)), \epsilon) \to (x, \epsilon)$ (FstCStep2)
- $(SndC(PairC(x, y)), \epsilon) \to (y, \epsilon)$ (SndCStep2)

- **Pares ordenados:** Tenemos tres reglas. En las primeras dos, se evalúa el lado izquierdo y el derecho del par, respectivamente; en la tercera, cuando ambos lados ya son expresiones que representan valores, entonces el par y los elementos que la componen se transforman en valores:

$$\frac{(e_1, \epsilon) \to (e_1', \epsilon')}{(PairC(e_1, e_2), \epsilon) \to (PairC(e_1', e_2), \epsilon')} (\text{PairCStep1})$$

$$\frac{v_1 \in ValC \qquad (e_2, \epsilon) \to (e_2', \epsilon')}{(PairC(v_1, e_2), \epsilon) \to (PairC(v_1, e_2'), \epsilon')} (\text{PairCStep2})$$

$$\frac{v_1,\ v_2 \in ValC, \qquad v_1',\ v_2' \in ValV}{(PairC(v_1, v_2), \epsilon) \to (PairV(v_1', v_2'), \epsilon)} (\text{PairCStep3})$$

En donde $ValC = \{NumC, BoolC, NilC\}$ y En donde $ValV = \{NumV, BoolV, NilV\}$.

- **Condicional:** Nuevamente, hay tres casos: en el primero, se evalúa la condición; en el segundo consideramos que la condición es $True$ y se evalúa la rama *then*; de otro

modo, en el tercero, se evalúa la rame *else*:

$$\frac{(c, \epsilon) \to (c', \epsilon')}{(IfC(c, t, e), \epsilon) \to (IfC(c', t, e), \epsilon')}(\textsf{IfCStep1})$$

$$\frac{}{(IfC(BoolC(True), t, e), \epsilon) \to (t, \epsilon)}(\textsf{IfCStep2})$$

$$\frac{}{(IfC(BoolC(False), t, e), \epsilon) \to (e, \epsilon)}(\textsf{IfCStep3})$$

- **Let:** Para el *let*, tenemos tres casos: en el primero, se evalúa el valor y se propaga el ambiente. Después, si el valor no es un *closure*, se liga la variable al valor ya evaluado, y se evalúa el cuerpo con el ambiente resultante. Por último, si el valor es un *closure* que captura el ambiente $clEnv$, entonces se construye un *closure* autorreferente *rec* que tiene a $clEnv$ como ambiente capturado, extendido con la ligadura de x al mismo *rec*:

$$\frac{(val, \epsilon) \to (val', \epsilon')}{(LetC(x, val, body), \epsilon) \to (LetC(x, val', body), \epsilon')}(\textsf{LetCStep1})$$

$$\frac{val \in ValC \qquad val\ no\ es\ un\ closure}{(LetC(x, val, body), \epsilon) \to (body, \epsilon[x \leftarrow val])}(\textsf{LetCStep2})$$

$$\frac{val = ClosureV(param, clBody, clEnv)}{(LetC(x, val, body), \epsilon) \to (body, \epsilon[x \leftarrow rec])}(\textsf{IfCStep3})$$

  En donde $rec = ClosureV(param,\ clBody,\ (x \leftarrow rec) : clEnv)$

- **Funciones:** Las funciones anónimas `LamC` son valores del lenguaje; estas no se evalúan más allá de encapsular el cuerpo de la función junto con el ambiente en que fueron definidas.

$$\frac{}{(LamC(x, e), \epsilon) \to (ValC(ClosureV(x, e, \epsilon)), \epsilon)}\ (\textsf{LamCStep})$$

- **Aplicaciones:** Las aplicaciones de funciones, representadas como `AppC(f,a)`, siguen el régimen de evalución ansiosa, ya que primero se evalúa la función, luego su argumento y finalmente se aplica el valor del argumento dentro del cuerpo de la función.

$$\frac{(f, \epsilon) \to (f', \epsilon')}{(AppC(f, a), \epsilon) \to (AppC(f', a), \epsilon')}\ (\textsf{AppCStep1})$$

$$\frac{(a, \epsilon) \to (a', \epsilon')}{(AppC(ValC(ClosureV(x, e, \epsilon_0)), a), \epsilon) \to (AppC(ValC(ClosureV(x, e, \epsilon_0)), a'), \epsilon')}\ (\textsf{AppCStep2})$$

$$\frac{}{AppC(ValC(ClosureV(x, e, \epsilon_0)), v, \epsilon) \rightarrow RestoreC(e, \epsilon), \epsilon_0[x := v])}\,(\mathsf{AppCStep3})$$

Conclusiones

El desarrollo de un intérprete de MiniLisp en Haskell logró la formalización e implementación exhaustivas de un subconjunto de Lisp, estableciendo una correspondencia coherente entre su modelo teórico y su materializacion práctica. Mediante la construcción del analizador léxico (lexer), el analizador sintáctico (parser), el desazucarador (desugar) y el intérprete, se logró implementar todo el proceso de traducción y evaluación, con el cual los programas fuente en lenguaje MiniLisp, son transformados en resultados evaluados de acuerdo con la semántica operacional estructural definida.

La definición formal de las estructuras léxicas y sintácticas, junto con el diseño de la sintaxis abstracta, permitió lograr hacer una clara distinción entre la superficie del lenguaje, la cual es muy expresiva; y su núcleo desazucarado, el cual es mínimo. Este proceso de eliminación del azúcar sintáctica ayudó a reforzar la comprensión de la expresividad del lenguaje, y demostró cómo es que se puede reducir sistemáticamente desde construcciones de alto nivel hacia una base más pequeña, pero semánticamente equivalente.

El intérprete fue implementado con el régimen de evaluación ansiosa con estrategia de paso de parámetros por valor ($call-by-value$), y modeló correctamente los ambientes, *bindings*, recursividad y las funciones de primera clase, dejando clara así la coherencia entre la máquina abstracta y su implementación ejecutable en Haskell.

Desde un punto de vista crítico, el proceso del diseño e implementación trajo consigo varios retos teóricos y prácticos. La naturaleza recursiva del combinador Y y la extensión correcta de los ambientes nos desafiaban a desarrollar un buen razonamiento sobre el orden de evaluación y los *closures*. Además, la depuración del intérprete debía hacerse cuidadosamente para mantener el equilibrio entre la teoría y la implementación.

Entre las limitaciones encontradas se tiene la ausencia de estructuras de datos avanzadas, herramientas para depuración y estrategias de optimización para las evaluaciones recursivas. Posibles extensione futuras del lenguaje podrían incluir la introducción de la evaluación perezosa, definición de listas por comprensión, abstracciones de datos definidas por el usuario y herramientas de depuración. Estas adiciones ampliarían la expresividad de MiniLisp, manteniendo la relación entre su base teórica y su implementación práctica.

En conclusión, este proyecto proporcionó una comprensión mas clara y profunda sobre la relación entre la sintaxis, la semántica y la implementación en el diseño de lenguajes de programación.

Referencias

[Hutton(2016)] Hutton, G. (2016). *Programming in Haskell (2nd ed.)*. Cambridge University Press.

[Pierce(2002)] Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

[Material de apoyo(2025)] Material de apoyo de ayudantía (2024). *Notas de Lenguajes de Programación I: Formalización de la Sintaxis y la Semántica*. Facultad de Ciencias, UNAM.