Proyecto1

Aguirre Cruz Casandra
Carrillo Benitez Valentina
Vázquez Rincón Oscar

November 5, 2025

1 Introducción

1.1 Motivación

Desde la entrada a la carrera dábamos por hecho el funcionamiento de un lenguaje de programación incluso al saber el como opera un a computadora por la materia Organización y Arquitectura de Computadoras que mediante compuertas lógicas podíamos generar instrucciones básicas y de ahí daba a cosas mucho mas complejas, de ahí daban un salto donde hacías instrucciones mas complejas en ensamblador y de ahí lo pasa a lenguaje de maquina(en binario) pero no sabíamos el como un lenguaje de programación entiende lo que programamos por eso este proyecto nos va ayudar en el proceso del detrás de todo,

1.2 Objetivos

- Comprender mejor el como funciona paso a paso un lenguaje de programación desde su sintaxis y semántica
- Formalizar un lenguaje de programación en cada una de sus formalizaciones

1.3 Delimitación de Proyecto

La delimitación de proyecto abarca la extension de una gramatica base para incluir la formalizacion de la sintaxis ademas de la gramatica, donde da via a un interprete funcional. Ahora delimitaremos el proyecto en sus diferentes partes:

- **Nivel Formal:**
  El proyecto debe realizar una especificacion matematica completa del lenguaje extendido, donde este abarca:
  - Sintaxis Lexica
  - Sintaxis Libre de Contexto

  - Sintaxis Abstacta
  - Sintaxis Operacional

- **Nivel Practico:**
  Se debe contruir un interprete funcional en Haskell que pueda implementar directamente las formalizaciones teoricas. Este debe cumplir las siguientes partes:

  - Lexer
  - Parser
  - Desugar
  - Interp

Ahora bien por la situacion y el contexto del proyecto este unicamente se limita a la creacion de un interprete, no de un compilador, de la misma forma este se regira del uso de Semantica Operacional Estructural. Por ultimo este tendra funcionalidades especiales para demostrar el funcionamiento del interprete que serian las pruebas solicitadas: Factorial, Fibonacci, map y filter.

2 Formalización

2.1 Sintaxis Concreta

2.1.1 Sintaxis léxica

Notación:
- $+$ representa la operación *suma* para expresiones regulares.
- A^* representa la operación *cerradura de Kleene* sobre el conjunto A para expresiones regulares.
- A^+ representa la operación *cerradura positiva* sobre el conjunto A para expresiones regulares.

Paréntesis

$$(+)$$

Operadores sobre enteros:

$$+ + - + * + / + = + < + > + >= + <= + != $$

Números:

Definimos los conjuntos:

$$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\},\ Z = D - \{0\}$$

Enteros:

$$0 + ZD^* + -ZD^*$$

Booleanos:

$$\#\ (\ f + t\ )$$

Identificadores:

Definimos el conjunto:[1]

$$A = \{a, b, c, ..., z\}$$

Definimos la expresión regular para generar cadenas que sirvan de identificadores a la variables:[2]

$$A^+$$

Palabras reservadas:

$$not + and + or + if + fst + snd + head + tail$$

$$+ let + let^* + letrec + lambda + cond + null + else$$

2.1.2 Sintaxis libre de contexto

Sintaxis libre de contexto (EBNF). Proveer una gramática completa en EBNF del lenguaje extendido, coherente con el léxico definido. Usar una notación uniforme (metas símbolos agrupadores y operadores de repetición) y señalar ambigüedades si las hubiera.

Usamos notación EBNF

[1] donde $a, b, c, ..., z$ son las letras del alfabeto a 26 caracteres.

[2] Usamos la cerradura positiva dado que el identificador de una variable no puede ser vacío.

Definimos la sintaxis libre de contexto:[3]

$$
\begin{aligned}
Exp ::= \; & < Id > \\
| \; & < Int > \\
| \; & < Bool > \\
| \; & (- < Exp >) \\
| \; & (\; + \; < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; - \; < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; * \; < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; / \; < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; = < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; < < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; > < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; <= < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; >= < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; ! = < Param > < Param > \{ < Param > \}\;) \\
| \; & (\; not \; < Exp > \;) \\
| \; & (\; and \; < Exp > < Exp > \;) \\
| \; & (\; or \; < Exp > < Exp > \;) \\
| \; & (\; if \; < Exp > < Exp > < Exp > \;) \\
| \; & (\; cond \; < Claus > \{ < Claus > \}\; [else \; < Exp >]) \\
| \; & (\; let \; (< Sust > \{ < Sust > \})(< Exp >)) \\
| \; & (\; let * \; (< Sust > \{ < Sust > \})(< Exp >)) \\
| \; & (\; letrec \; (< Sust > \{ < Sust > \})(< Exp >)) \\
| \; & (\; lambda(< Id > \{ < Id > \}) \; < Exp >) \\
| \; & (\; < Exp > < Param > \{ < Param > \}\;) \\
| \; & (\; < Exp >, < Exp > \;) \\
| \; & (\; fst \; < Exp > \;) \\
| \; & (\; snd \; < Exp > \;) \\
| \; & [\;] \\
| \; & [\; < List > \;] \\
| \; & (\; head \; < Exp > \;) \\
| \; & (\; tail \; < Exp > \;) \\
| \; & Null
\end{aligned}
$$

[3]donde $a \mid b \mid c \mid ... \mid z$ son las letras del alfabeto a 26 caracteres.

$$
\begin{aligned}
Id &::= < Char > \\
&\quad | < Id >< Char > \\
Char &::= a \mid b \mid c \mid ... \mid z \\
Int &::= 0 \\
&\quad | < Digit > \\
&\quad | < Int >< Digit > \\
Digit &::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
Param &::= < Int > \mid < Bool > \mid < Id > \mid < Exp > \\
Claus &::= [(< Exp >)\ (< Exp >)] \\
Bool &::= \#t \mid \#f \\
List &::= < Exp > \\
&\quad | < List >, < Exp > \\
Sust &::= (< Id > < Exp >)
\end{aligned}
$$

Los operadores variádicos en su definición se refieren a operaciones que puedan aceptar un número de variable de argumentos o parámetros. Para efectos de las funciones en este proyecto, nuestras operaciones variádicas se puede acuñar a la azúcar sintáctica que permitirá su funcionamiento sobre las múltiples entradas que un usuario del lenguaje pueda ingresar.

Se aplica *Param* a las operaciones que requieren un soporte variádico. En el caso de enteros, se aplicaría afectivamente a las operaciones +, -, *, /, así como a las comparaciones =, ¡, ¿, ¡=, ¿=, !=. En el caso de los booleanos, se aplica a la operacion *cond*, que recibe una o más cláusulas. Finalmente, la función lambda.

2.2 Sintaxis Abstracta

2.2.1 Azúcar Sintáctica

Para definir posteriormente la semántica de los operadores variádicos necesitamos, desde la sintaxis, formalizar una función de traducción que los convierta a expresiones binarias del núcleo, es decir, la eliminación de azúcar sintáctica. En este proceso se transforma recursivamente una expresión de superficie a una del núcleo. Así, usaremos la eliminación del azúcar sintáctica mediante la siguiente regla:

Definimos una función γ que transforma la sintaxis concreta a sintaxis abstracta usando la sustitución, en notación:

$$\gamma(op\ expr_1\ expr_2 .... \ expr_x)$$

Haremos recursión sobre la cantidad de expresiones a recibir:

- **Caso base (n=2):** Si la operación recibe exactamente dos argumentos, se envía directamente al ASA binario:

$$\gamma(op\ expr_1\ expr_2)) = op(\gamma(expr_1),\ \gamma(expr_2))$$

- **Caso recursivo (n>2):** En caso de recibir más expresiones, podemos reducirla a una operación binaria que encierra los parámetros desde $expr_1$ hasta $expr_{n-1}$ y los opera con $expr_n$:

$$\gamma(op\ expr_1\ ...\ expr_n)) = op(\gamma(op(\ expr_1\ ...\ expr_{n-1})), \gamma(expr_n))$$

  De esta manera los operadores $=, +, -, *, /$ y las comparaciones pueden recibir un total de n expresiones, con $n \geq 2$.

De igual forma, la aplicación de una función a múltiples argumentos es azúcar sintáctica para una serie de aplicaciones anidadas, cada una pasando un solo argumento.

Definimos las siguientes reglas en nuestra función de traducción γ:

- **Traducción de `lambda` variádica:** Una `lambda` con n parámetros se traduce en n nodos `Lambda` anidados.

  $\gamma(\text{ (lambda } (x_1\ x_2\ ...\ x_n) \text{ cuerpo) }) = \text{Lambda}(x_1, \gamma(\text{ (lambda } (x_2\ ...\ x_n) \text{ cuerpo) }))$

  El caso base es una `lambda` con un solo parámetro, que se traduce directamente al constructor del núcleo:

$$\gamma(\text{ (lambda } (x_1) \text{ cuerpo) }) = \text{Lambda}(x_1, \gamma(\text{cuerpo}))$$

- **Traducción de aplicación variádica:** La aplicación de una función a n argumentos se traduce en n nodos `App` anidados, con asociatividad por la izquierda.

$$\gamma(\ (f\ a_1\ a_2\ ...\ a_n)\ ) = \text{App}(\gamma(\ (f\ a_1\ a_2\ ...\ a_{n-1})\ ), \gamma(a_n))$$

  El caso base es la aplicación a un solo argumento, que se traduce directamente al constructor del núcleo:

$$\gamma(\ (f\ a_1)\ ) = \text{App}(\gamma(f), \gamma(a_1))$$

- **Traducción de Negación (-):** La negación unaria se traduce a una resta partiendo de 0.

$$\gamma(\ (-\ e)\ ) = \text{Sub}(\text{Num}(0), \gamma(e))$$

- **Traducción de `cond`:** Se traduce en expresiones `If` anidadas.

$$\gamma(\text{ (cond } (c_1\ e_1)\ ...\ (c_n\ e_n) \text{ [else } e_{else}])) = \text{If}(\gamma(c_1), \gamma(e_1), \gamma(\text{ (cond ...) }))$$

$$\gamma(\text{ (cond [else } e_{else}])) = \gamma(e_{else})$$

- **Traducción de `let`, `let*` y `letrec`:** `let` es azúcar para una aplicación de `lambda`.

  $\gamma(\text{ (let } ((x_1\ v_1)...(x_n\ v_n))\ e)\ ) = \gamma(\text{ ((lambda } (x_1...x_n)\ e)\ v_1\ ...\ v_n)\ )$

  `let*` es azúcar para `App` y `Lambda` anidados.

  $$\gamma(\text{ (let* } ((x\ v))\ e)\ ) = \text{App}(\text{Lambda}(x, \gamma(e)), \gamma(v))$$

  $\gamma(\text{ (let* } ((x_1\ v_1)\ (x_2\ v_2)...)\ e)\ ) = \text{App}(\text{Lambda}(x_1, \gamma(\text{ (let* } ((x_2\ v_2)...)\ e)\ )), \gamma(v_1))$

  `letrec` se desazucara usando el combinador de punto fijo `Fix`.

  $\gamma(\text{ (letrec } ((x\ v))\ e)\ ) = \text{App}(\text{Lambda}(x, \gamma(e)), \text{Fix}(\text{Lambda}(x, \gamma(v))))$

- **Traducción de Listas, `head` y `tail`:** Las listas se traducen a `Pair` anidados que terminan en `Null`.

  $$\gamma(\text{ [}e_1\ e_2\ ...\ e_n\text{] }) = \text{Pair}(\gamma(e_1), \gamma(\text{ [}e_2\ ...\ e_n\text{] }))$$

  $$\gamma(\text{ [] }) = \text{Null}$$

  `head` y `tail` se traducen a `Fst` y `Snd` (según `Desugar.hs`).

  $$\gamma(\text{ (head } e)\ ) = \text{Fst}(\gamma(e))$$

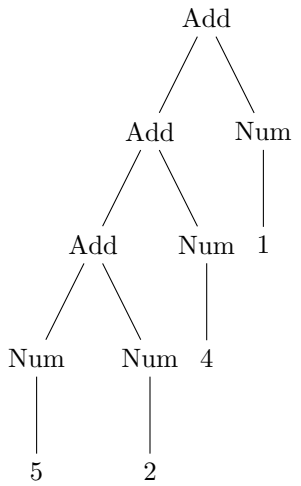
  $$\gamma(\text{ (tail } e)\ ) = \text{Snd}(\gamma(e))$$

Con base a nuestra función γ, definimos entonces la regla de inferencia para el *desugaring* sobre operaciones como la suma de la siguiente manera:

$$\frac{s_1, s_2\ ASA}{Add(s_1, s_2)\ ASA} \qquad \frac{s_n\ ASA \quad Add(s_1, s_2, ..., s_{n-1})\ ASA}{Add(Add(s_1, s_2, ..., s_{n-1}), s_n))\ ASA}$$

de tal forma que con éstas reglas es posible construir el siguiente árbol de sintaxis abstracta para la expresión:

+(5 2 4 1)



En base a la regla de *desugaring* para la suma, podemos obtener el resto de reglas de inferencia para operaciones variádicas sobre enteros de forma análoga, reglas que llamaremos:

- $Subs(s_1, s_2)$ para la resta.
- $Mult(s_1, s_2)$ para la multiplicación.
- $Div(s_1, s_2)$ para la división.
  En el caso que s_2 sea 0:

$$\frac{e_1 \Rightarrow \text{Num}(n_1) \qquad e_2 \Rightarrow \text{Num}(0)}{\text{Div}(e_1, e_2) \Rightarrow \text{Error}(\text{"División por cero"})}$$

- $Equals(s_1, s_2)$ para la comparación de igualdad entre enteros.
- $LessE(s_1, s_2)$ para la comparación de *menor estricto que* entre enteros.
- $GreatE(s_1, s_2)$ para la comparación de *mayor estricto que* entre enteros.
- $Less(s_1, s_2)$ para la comparación de *menor o igual que* entre enteros.
- $Great(s_1, s_2)$ para la comparación de *mayor o igual que* entre enteros.
- $Diff(s_1, s_2)$ para la comparación de diferencia entre enteros.

2.3 Otras reglas de inferencia

Reglas para elementos básicos:

$$\frac{n \in \mathbb{N}}{Num(n)\ ASA} \qquad \frac{b \in \mathfrak{B}}{Bool(b)\ ASA} \qquad \frac{i \text{ es una cadena}}{Id(i)\ ASA}$$

Reglas para definir valores finales:

$$\frac{n \in \mathbb{N}}{Final(Num(n))\ ASA} \qquad \frac{b \in \mathfrak{B}}{Final(Bool(b))\ ASA} \qquad \frac{i \text{ es una cadena}}{Final(Id(i))}$$

Operaciones sobre booleanos:

$$\frac{p\ ASA}{Not(p)\ ASA} \qquad \frac{p\ ASA \quad q\ ASA}{And(p,q)\ ASA} \qquad \frac{p\ ASA \quad q\ ASA}{Or(p,q)\ ASA}$$

Condicionales:

azúcar sintáctica

2.4 Constructores del Núcleo (ASA)

Expresiones `let` y `lambda` (Núcleo):

$$\frac{i : \text{String} \quad c : \text{ASA}}{\text{Lambda}(i, c) : \text{ASA}} \qquad \frac{f : \text{ASA} \quad a : \text{ASA}}{\text{App}(f, a) : \text{ASA}} \qquad \frac{e : \text{ASA}}{\text{Fix}(e) : \text{ASA}}$$

Pares ordenados y Listas (Núcleo):

$$\frac{a\ ASA \quad b\ ASA}{Pair(a, b)\ ASA} \qquad \frac{t\ ASA}{Fst(t)\ ASA} \qquad \frac{t\ ASA}{Snd(t)\ ASA}$$

$$\frac{}{\text{Null}\ ASA} \qquad \frac{l\ ASA}{HeadL(l)\ ASA} \qquad \frac{l\ ASA}{TailL(l)\ ASA}$$

(Nota: `HeadL` y `TailL` están definidos en `DesuExp` y `Interp.hs`, pero `Desugar.hs` actualmente los traduce a `Fst` y `Snd`.

2.5 Semántica Operacional

2.5.1 Reglas de inferencia (Small-Step)

Definimos la reglas de inferencia en paso pequeño para nuestras funciones de la forma siguiente:

Reglas de reducción para elementos básicos:

$$\frac{}{\langle Num(n),\ \varepsilon\rangle \to \langle Num(n),\ \varepsilon\rangle} \quad \text{(Num-Reduction)}$$

$$\frac{}{\langle Bool(n),\ \varepsilon\rangle \to \langle Bool(n),\ \varepsilon\rangle} \quad \text{(Bool-Reduction)}$$

$$\frac{}{\langle Id(n),\ \varepsilon\rangle \to\ error} \quad \text{(Free-Variable)}$$

$$\frac{}{\langle Null,\ \varepsilon\rangle \to \langle Null,\ \varepsilon\rangle} \quad \text{(Null-Reduction)}$$

Y en general:

$$\frac{}{\langle Final(n), \varepsilon\rangle \to \langle Final(n), \varepsilon\rangle} \quad \text{(Fin-reduction)}$$

Reglas para operaciones con enteros:

$$\frac{n \to n'}{\langle Add(n, m),\ \varepsilon\rangle \to \langle Add(n', m),\ \varepsilon\rangle} \quad \text{(Add-Step1)}$$

$$\frac{m \to m'}{\langle Add(Num(n), m),\ \varepsilon\rangle \to \langle Add(Num(n), m'),\ \varepsilon\rangle} \quad \text{(Add-Step2)}$$

$$\frac{}{\langle Add(Num(n), Num(m)),\ \varepsilon\rangle \to \langle (n+m),\ \varepsilon\rangle}\ ^{4} \qquad \text{(Add-Reduction)}$$

Análogamente, podemos obtener:

$$\frac{n \to n'}{\langle Subs(n,m),\ \varepsilon\rangle \to \langle Subs(n',m),\ \varepsilon\rangle} \qquad \text{(Subs-Step1)}$$

$$\frac{m \to m'}{\langle Subs(Num(n),m),\ \varepsilon\rangle \to \langle Subs(Num(n),m'),\ \varepsilon\rangle} \qquad \text{(Subs-Step2)}$$

$$\frac{}{\langle Subs(Num(n), Num(m)),\ \varepsilon\rangle \to \langle (n-m),\ \varepsilon\rangle}\ ^{5} \qquad \text{(Subs-Reduction)}$$

Utilizamos el mismo sistema de tres reglas para el resto de operaciones (multiplicación y división) llamando a las reglas de forma similar a las reglas de suma y resta.

$$\frac{n \to n'}{\langle Equals(n,m),\ \varepsilon\rangle \to \langle Equals(n',m),\ \varepsilon\rangle} \qquad \text{(Equals-Step1)}$$

$$\frac{m \to m'}{\langle Equals(Num(n),m),\ \varepsilon\rangle \to \langle Equals(Num(n),m'),\ \varepsilon\rangle} \qquad \text{(Equals-Step2)}$$

$$\frac{}{\langle Equals(Num(n), Num(m)),\ \varepsilon\rangle \to \langle (n == m),\ \varepsilon\rangle}\ ^{6} \qquad \text{(Equals-Reduction)}$$

$$\frac{n \to n'}{\langle LessE(n,m),\ \varepsilon\rangle \to \langle LessE(n',m),\ \varepsilon\rangle} \qquad \text{(LessE-Step1)}$$

$$\frac{m \to m'}{\langle LessE(Num(n),m),\ \varepsilon\rangle \to \langle LessE(Num(n),m'),\ \varepsilon\rangle} \qquad \text{(LessE-Step2)}$$

$$\frac{}{\langle LessE(Num(n), Num(m)),\ \varepsilon\rangle \to \langle (n < m),\ \varepsilon\rangle}\ ^{7} \qquad \text{(LessE-Reduction)}$$

Al igual que con los operadores suma y resta, de manera análoga a *Equals* y *LessE* podemos obtener el resto de comparaciones entre enteros.

[4] + se refiere a la operación *suma* de haskell sobre el tipo de dato Num
[5] - se refiere a la operación *resta* de haskell sobre el tipo de dato Num
[6] == se refiere al predicado *es igual a* de haskell sobre el tipo de dato Eq
[7] < se refiere al predicado *es estrictamente menor que* de haskell sobre el tipo de dato Eq

Reglas para operaciones con booleanos: [8]

$$\frac{b \to b'}{\langle Not(b), \varepsilon\rangle \to \langle Not(b'), \varepsilon\rangle} \quad \text{(Not-Step)}$$

$$\frac{}{\langle Not(Bool(b)), \varepsilon\rangle \to \langle not\ b, \varepsilon\rangle}^{9} \quad \text{(Not-Reduction)}$$

$$\frac{b \to b'}{\langle And(b, c), \varepsilon\rangle \to \langle And(b', c), \varepsilon\rangle} \quad \text{(And-Step1)}$$

$$\frac{c \to c'}{\langle And(Bool(b), c), \varepsilon\rangle \to \langle And(Bool(b), c'), \varepsilon\rangle} \quad \text{(And-Step2)}$$

$$\frac{}{\langle And(Bool(b), Bool(c)) \to \langle b\&\&c, \varepsilon\rangle}^{10} \quad \text{(And-Reduction)}$$

$$\frac{b \to b'}{\langle Or(b, c), \varepsilon\rangle \to \langle Or(b', c), \varepsilon\rangle} \quad \text{(Or-Step1)}$$

$$\frac{c \to c'}{\langle Or(Bool(b), c), \varepsilon\rangle \to \langle Or(Bool(b), c'), \varepsilon\rangle} \quad \text{(Or-Step2)}$$

$$\frac{}{\langle Or(Bool(b), Bool(c)), \varepsilon\rangle \to \langle b \parallel c, \varepsilon\rangle} \quad \text{(Or-Reduction)}$$

Condicionales:

$$\frac{c \to c'}{If(c, t, e) \to If(c', t, e)} \quad \text{(If-Step)}$$

$$\frac{}{If(Bool(True), t, e) \to t} \quad \text{(If-True)}$$

$$\frac{}{If(Bool(False), t, e) \to e} \quad \text{(If-False)}$$

[8] || se refiere a la operación *or* de haskell sobre el tipo de dato Bool
[9] *not b* se refiere a la operación *not* de haskell sobre el tipo de dato Bool
[10] `&&` se refiere a la operación *and* de haskell sobre el tipo de dato Bool

Expresiones lambda y aplicación:
Para let

$$\frac{v \to v^{'}}{Let(Id(i), v, c) \to Let(Id(i), v', c)} \quad \text{(Let-Step)}$$

$$\frac{(sust\ c\ i\ Final(e)) \to c^{'}}{Let(Id(i), Final(e), c) \to c'} \quad \text{(Let-Sust)}$$

Para lambda

$$\frac{}{\langle Lambda(Id(i), c),\ \varepsilon \langle Lambda(Id(i), c),\ \varepsilon \rangle} \quad \text{(Lambda-Value)}$$

$$\frac{f \to f'}{\langle App(f, a), \varepsilon \rangle \to \langle App(f', a), \varepsilon \rangle} \quad \text{(App-Step 1)}$$

$$\frac{a \to a'}{\langle App(Lambda(i, c), a), \varepsilon \rangle \to \langle App(Lambda(i, c), a'), \varepsilon \rangle} \quad \text{(App-Step2)}$$

$$\frac{c[i := v] \to c'}{\langle App(Lambda(i, c), Final(v)), \varepsilon \rangle \to \langle c', \varepsilon \rangle} \quad \text{(App-Sust)}$$

$$\frac{f \to f'}{\langle App(f, a), \varepsilon \rangle \to \langle App(f', a), \varepsilon \rangle} \quad \text{(App-Step 1)}$$

$$\frac{a \to a'}{\langle App(Lambda(i, c), a), \varepsilon \rangle \to \langle App(Lambda(i, c), a'), \varepsilon \rangle} \quad \text{(App-Step2)}$$

$$\frac{}{\langle App(Lambda(i, c), v), \varepsilon \rangle \to \langle sust(c, i, v), \varepsilon \rangle} \quad \text{(App-Sust)}$$

*(Asumiendo que v es un valor, como en **`Interp.hs`**)*

Reglas para `Fix` (recursión):

$$\frac{e \to e'}{\langle Fix(e), \varepsilon \rangle \to \langle Fix(e'), \varepsilon \rangle} \quad \text{(Fix-Step)}$$

$$\frac{}{\langle Fix(Lambda(x, e)), \varepsilon \rangle \to \langle sust(e, x, Fix(Lambda(x, e))), \varepsilon \rangle} \quad \text{(Fix-Reduction)}$$

Para la sustitución de las apariciones de la variable de ligado, definiremos el siguiente método en Haskell:[11]

```haskell
--Recibe una expresion, un identificador, una expresion y devuelve
    una expresion
--Entonces sustituye en este arbol llamado tal por este valor
sust :: DesuExp -> String -> DesuExp -> DesuExp
sust(Num n) _  _= Num n
sust(Bool b) _  _= Bool b
```

[11] Para mayores referencias, revisar la implementación del intérprete en el archivo Interp.hs

```
sust Null _  _ = Null
sust(Id s) var val = if s == var then val else Id s
sust(Add e1 e2) var val= Add (sust e1 var val ) (sust e2 var val)
sust(Sub e1 e2) var val = Sub(sust e1 var val) (sust e2 var val)
sust(Mult e1 e2) var val =Mult(sust e1 var val) (sust e2 var val)
sust(Div e1 e2) var val = Div(sust e1 var val) (sust e2 var val)
sust(Equals e1 e2) var val = Equals(sust e1 var val)(sust e2 var
    val)
sust(LessE e1 e2) var val = LessE(sust e1 var val)(sust e2 var val)
sust(GreatE e1 e2) var val = GreatE (sust e1 var val) (sust e2 var
    val)
sust(Less e1 e2) var val = Less (sust e1 var val) (sust e2 var val)
sust(Great e1 e2) var val =  Great (sust e1 var val)( sust e2 var
    val)
sust(DiffD e1 e2) var val = DiffD(sust e1 var val)(sust e2 var val)
sust(Not e1) var val = Not( sust e1 var val)
sust(And e1 e2) var val= And(sust e1 var val)(sust e2 var val)
sust(Or e1 e2) var val= Or(sust e1 var val)(sust e2 var val)

sust(If c t e) var val= If (sust c var val) (sust t var val) (sust
    e var val)
sust(App f a) var val= App (sust f var val) (sust a var val)
sust(Pair e1 e2 ) var val = Pair(sust e1 var val) (sust e2 var val)
sust(Fst e) var val = Fst (sust e var val)
sust(Snd e) var val = Snd (sust e var val)
sust(HeadL e) var val = HeadL(sust e var val)
sust(TailL e) var val = TailL(sust e var val)

sust(Fix e )var val = Fix (sust e var val)
sust(Lambda p c) i v =
        if i == p
        then Lambda p c
        else Lambda p (sust c i v)
```

Listing 1: Función sustitución Haskell (Interp.hs)

Operaciones con pares ordenados:

$$\frac{t \to t'}{\langle Fst(t), \varepsilon\rangle \to \langle Fst(t'), \varepsilon\rangle} \quad \text{(Fst-Step)}$$

$$\frac{}{\langle Fst(Pair(v_1, v_2)), \varepsilon\rangle \to \langle v_1, \varepsilon\rangle} \quad \text{(Fst-Reduction)}$$

$$\frac{t \to t'}{\langle Snd(t), \varepsilon\rangle \to \langle Snd(t'), \varepsilon\rangle} \quad \text{(Snd-Step)}$$

$$\frac{}{\langle Snd(Pair(v_1, v_2)), \varepsilon\rangle \to \langle v_2, \varepsilon\rangle} \quad \text{(Snd-Reduction)}$$

$$\frac{a \to a^{'}}{\langle Pair(a, b), \varepsilon\rangle \to \langle Pair(a', b), \varepsilon\rangle} \quad \text{(Pair-Step1)}$$

$$\frac{b \to b^{'}}{\langle Pair(v, b), \varepsilon\rangle \to \langle Pair(v, b'), \varepsilon\rangle} \quad \text{(Pair-Step2)}$$

(Asumiendo que v es un valor)

$$\frac{}{\langle Pair(v_1, v_2), \varepsilon\rangle \to \langle Pair(v_1, v_2), \varepsilon\rangle} \quad \text{(Pair-Reduction)}$$

(Asumiendo que v_1 y v_2 son valores)

Operaciones con listas (Constructores `HeadL` y `TailL`):

$$\frac{l \to l'}{\langle HeadL(l), \varepsilon\rangle \to \langle HeadL(l'), \varepsilon\rangle} \quad \text{(HeadL-Step)}$$

$$\frac{}{\langle HeadL(Pair(v_1, v_2)), \varepsilon\rangle \to \langle v_1, \varepsilon\rangle} \quad \text{(HeadL-Reduction)}$$

$$\frac{l \to l'}{\langle TailL(l), \varepsilon\rangle \to \langle TailL(l'), \varepsilon\rangle} \quad \text{(TailL-Step)}$$

$$\frac{}{\langle TailL(Pair(v_1, v_2)), \varepsilon\rangle \to \langle v_2, \varepsilon\rangle} \quad \text{(TailL-Reduction)}$$

Consideramos listas heterogéneas que agrupan valores finales.

3 Justificaciones

La forma en el que se implemento el interprete se baso en el diseño para obtener un núcleo de lenguaje minimo mientras se tiene una sintaxis superficial conveniente para su uso. Principalmente nos decidimos en separa el lenguajes superficial y lenguaje de nucleo , esto se refleja en el Parser(Lo que escribe el usuario) y con el Desugar(Donde los ASA pasan a estar definidos en DesuExp). De misma manera la separación dada es por la dificultad afuera del interprete porque este mismo no necesita hacer todo el trabajo de entender las construcciones complejas, solo opera sobre los conjuntos minimo de expresiones del nucleo. Por ejemplo en el caso de let, let* y letrec no existen en el núcleo del interprete, dado que pasan como azúcar sintáctica para la aplicación de funciones y el combinador del punto fijo. Primero pasaremos de lo mas general a lo más especifico, con el objetivo de abarcar lo más general.

3.1 Interprete

El diseño del interprete se justifica como una implementación de small-step, donde la semantica se implementa con la funcion llamada **bstep** que es llamada por eval, que hace alusión a una regla de la semántica operacional. Tomemos un ejemplo donde **bstep (Add (Num n1) e2) = Add (Num n1) (bstep e2)** se utiliza la regla que evalua para dar así el lado izquierdo(provocando un valor). La forma en que se va a aplicar las funciones usamos a **App** donde se implementa usando la función **sust**. El por que de usarla sale de la implementación directa de la beta reducción del calculo lambda, solo que en usar entornos, el valor

del argumento se sustituye por el parámetro en el cuerpo de la función y por ultimo la implementación de Fix, el motivo de su implementación es porque es forma directa de sustituir la función recursiva dentro su propio cuerpo dando así que ejecute un paso en su propia recursivo. Para aprovechar, el condicional se trata como azúcar sintáctica para if anidado, donde el interprete implementa la semántica de if y en el desugar genera la lógica necesaria

3.2 Operadores Variadicos

La forma en que implementamos estos operadores variadicos, nos permite tener (+ 12 3) dado que se hace la traducción de las expresiones variadicas a expresiones binarias anidadas con asociatividad a la izquierda.

3.3 Listas y Pares

Dado que en el nucleo no contiene ningun tipo de list, se trata traduciendo la listas a pares anidados que terminan en Null.Donde nos apoyamos de la implementacion de Lisp,dando asi una consecuencia que seria que la operacion de lista(head y tail) pasa a ser Fst y Snd que se justifica por la azúcar sintáctica.

3.4 Lets

Lo primero que se pude notar es que no existen los lets en el núcleo del interprete dado que es como la azucar sintáctica para la aplicación de funciones combinado con el combinador de punto fijo.

- **let(FunP):** De la forma que si tuviéramos $(let(x\ y)x$ se parece semanticamente a la aplicación de función porque al traducirlo el interprete este solo necesita tener la semántica de la aplicación.
- **let*(FunPE):** Este mismo requiere un poco mas de precaución dado que se necesita una evaluacion secuencial pero podemos aprovechar la estructura anidada de App y Lambda que nos ayuda a reducir la logica de los entornos secuenciales en el interprete.
- **letrec(FunRecP):** Donde la recursion puede con el combinador de punto fijo.El nucleo solo necesita la regla para Fix y esta se encarga de desarrollar la recursion. Esto permite una implementacion de la recursion que es a la vez poderosa y semántica simple.

4 Resultados

4.1 Implementación

MiniLisp fue implementado con el pipeline que nos mencionaron: *lexer* → *parser* → *desugar* → *interprete*. Ocupamos **Alex** para el análisis léxico y

Happy para el análisis sintáctico.
Módulos que ocupamos:

- **Lexer.x:** Donde está la definición de tokens mediante expresiones regulares.
- **Parser.y:** Toma los tokens generados por el Lexer e implementa la grámatica EBNF con deficiones formales.
- **Desugar.hs:** Transforma la sintáxis aplicando las reglas de eliminación de azúcar sintáctica.
- **Interp.hs:** Donde implementamos la semántica operacional con paso pequeño, gestionando ambientes de evaluación y sustitución de variables.
- **Menu.hs:** Da una interfaz donde el usuario puede escribir expresiones para saber su resultado y pruebas ya predefinidas.
- **Main.hs:** Coordina la ejecución del pipeline, llama al Menú interactivo y maneja la entrada y salida del sistema.
- **MiniLisp.hs:** Tiene definiciones de tipos de datos para la sintaxis abstracta, funciones auxiliares y la implementación de la semántica operacional.

4.2 Pruebas

Este MiniLisp se probó con varios casos que ocupan las construcciones del lenguaje, como funciones recursivas, operaciones aritméticas y listas (se intentó probar con funciones de orden superior como **map** y **filter**, pero no lo logramos). Estas pruebas se realizaron desde el menú que creamos, en el cual solicita al usuario que, en el caso de una expresión "personalizada", la escriba y en el caso de las demás pruebas, sólo pide el valor de n y ya teniendo esto, la consola sólo muestra el resultado.
Por lo mismo a continuación veremos unos ejemplos de los casos que hicimos, junto con su resultado.
El menú de MiniLisp únicamente muestra el resultado final, no muestra el procedimiento paso a paso.

- **Operaciones con listas:**
  (head [1, 2, 3, 4, 5])
  Resultado: 1
  (tail [1, 2, 3, 4, 5])
  Resultado: (2, (3, (4, (5, []))))
- **Operaciones variádicas:**
  (+ 1 2 3 4 5)
  Resultado: 15
  (* 1 2 3 4 5)
  Resultado: 120

- **Expresiones condicionales y booleanas:**
  (if (¡ 3 5) 10 20)
  Resultado: 10
  (and t f)
  Resultado: f
  (or f f)
  Resultado: f
  (not f)
  Resultado: t

- **Suma de los primeros n naturales:**
  (letrec ((suma (lambda(n)(if (== n 0) 0 (+ n (suma (- n 1))))))) (suma 5))
  Resultado: 15

- **Factorial:**
  (letrec ((fact (lambda(n) (if (== n 0) 1 (* n (fact (- n 1))))))) (fact 5))
  Resultado: 15

- **Fibonacci:**
  (letrec ((fib (lambda(n) (if (¡= n 1) n (+ (fib (-n 1)) (fib (- n 2))))))) (fib 6))
  Resultado: 8

Los resultados obtenidos muestran que la implementación de la semántica operacional con paso pequeño es correcta y el intérprete cumple con las reglas de inferencia formalizadas. Aunque no logramos la ejecución de funciones de orden superior como **filter** y **map**, el sistema funciona correctamente para listas, recursión, operaciones aritméticas, condicionales y booleanos.

5 Conclusiones

Al desarrollar este MiniLisp, nos permitió comprender más sobre el funcionamiento interno de un lenguaje de programación. Entendimos desde la definición sintáctica a la implementación de la semántica operacional.

A lo largo de la creación de este MiniLip, enfrentamos varios retos, en la parte del parser y el intérprete tuvimos que aprender a cómo traducir las formalizaciones a código, aparte tuvimos que analizar detalladamente el manejo de sustituciones y la recursión de las expresiones, también aprendimos a implementar la semántica operacional con paso pequeño, el cual se puede romper si olvidamos o ponemos mal alguna regla.

Algunas **limitaciones** que tuvimos fueron: la implementación de las funciones de orden superior como **map** y **filter**, que no pudimos integrar completamente. Además el menú que implementamos sólo muestra el resultado, no

muestra el paso a paso de la expresión.

Como posibles **extensiones futuras**, cosideramos agregar funciones de orden superior totalmente funcionales y mejorar el sistema de salida para mostrar las etapas de evaluación.
Optimizar la recursión y aumentar la cantidad de operadores.

En conclusión, este proyecto nos permitió conectar la teoría formal con la práctica, nos ayudó a entender más sobre la semántica operacional y comprender el diseño de los lenguajes que en la base de la materia a su vez nos da el trasfondo de lo que necesita un lenguaje de programacion.

6 Bibliografía

Sipser, M. (1996). Introduction to the Theory of Computation. ACM SIGACT News, 27(1), 27-29. https://doi.org/10.1145/230514.571645

Soto, M. S. (2025, 1 noviembre). LDP. LDP. Recuperado 5 de noviembre de 2025, de https://lambdasspace.github.io/LDP/?authuser=1

Soto, S. R. (s. f.). GitHub - lambdasspace/MiniLisp: Intérpretes del curso de Lenguajes de Programación. GitHub. https://github.com/lambdasspace/MiniLisp