



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Lenguajes de Programación

Proyecto 1: MiniLisp

PROFESOR

Manuel Soto Romero

AYUDANTES

Diego Méndez Medina
José Alejandro Pérez Márquez
Erick Daniel Arroyo Martínez
Mauro Emiliano Chávez Zamora

ALUMNOS

González Téllez Lorena - 321288952
López Rodríguez Leslie Renée - 321171915
Marentes Mosqueda Bruno Sebastián - 321260606

5 de noviembre de 2025

Índice general

1. Introducción	4
1.1. Objetivos	4
1.1.1. Objetivo general	4
1.1.2. Objetivos específicos	4
1.2. Referencias históricas	5
2. Formalización	6
2.1. Sintaxis Concreta	6
2.1.1. Sintaxis Léxica	7
2.1.2. Sintaxis Libre de Contexto	8
2.2. Sintaxis Abstracta	11
2.2.1. Definiciones	11
2.2.2. Árboles de Sintaxis Abstracta	11
2.3. Eliminación de azúcar sintáctica	18
2.3.1. Principios Fundamentales	18
2.3.2. Ventajas	19
2.3.3. Expresiones básicas	20
2.3.4. Operaciones aritméticas, booleanas y predicados	20
2.3.5. Pares ordenados	23
2.3.6. Listas como pares ordenados	23
2.3.7. Condicionales	24
2.3.8. Funciones Lambda y Aplicaciones	25
2.3.9. Asignaciones locales: Let, Let* y Letrec	26
2.4. Semántica Operacional Estructural	29
2.4.1. Definiciones	29
2.4.2. Ambiente Inicial	30
2.4.3. Reglas de transición	30
3. Justificaciones	40
3.1. Elecciones de notación	40
3.1.1. Sistema de notación para la sintaxis abstracta	40
3.1.2. Convenciones para semántica operacional	40
3.1.3. Decisiones en la gramática concreta	40
3.1.4. Representación de estructuras de datos	41
3.1.5. Enfoque educativo	41
4. Resultados	42
4.1. Descripción de la implementación	42
4.1.1. Arquitectura general	42

4.1.2.	Decisiones de diseño	43
4.2.	Casos de prueba	44
4.2.1.	Test 1: Suma de primeros N naturales	44
4.2.2.	Test 2: Factorial	44
4.2.3.	Test 3: Fibonacci	45
4.2.4.	Test 4: Map (Aplicar función a elementos de lista)	45
4.2.5.	Test 5: Filter (Filtrar elementos de lista)	46
4.3.	Ejemplos de ejecución	47
4.3.1.	REPL	47
4.4.	Análisis de resultados	50
4.4.1.	Tests	50
4.4.2.	Desazucarización	50
4.4.3.	Alcance estático	50
5.	Conclusiones	51

Capítulo 1

Introducción

1.1. Objetivos

1.1.1. Objetivo general

Los lenguajes de programación son las herramientas que necesitamos para resolver problemas, son el puente entre el pensamiento humano y la ejecución computacional. Estos sistemas formales permiten capturar y estructurar procesos computacionales, por lo que su estudio y análisis es necesario para garantizar su corrección.

El enfoque del presente proyecto consiste en plantear la extensión del lenguaje MiniLISP formalizando la sintaxis y semántica correspondiente para implementarlo en HASKELL mediante la herramienta HAPPY. MiniLISP representa un núcleo esencial de la familia LISP, proporciona una sintaxis simple (S-expressions) y una semántica funcional básica, este es nuestro caso de estudio.

Mediante este documento presentamos la formalización de la modificación del lenguaje, iniciando con la definición de la sintaxis léxica y libre de contexto que después (mediante la eliminación de la azúcar sintáctica) se transforma en la sintaxis abstracta correspondiente para así especificar la semántica operacional estructural (paso pequeño) que concreta las reglas de reducción de las expresiones que nos proporciona la sintaxis abstracta.

Con esto en mente, se plantea la coherencia entre el modelo teórico propuesto y su materialización práctica.

1.1.2. Objetivos específicos

1. **Extender el sistema de tipos** de MiniLISP para incorporar:
 - Pares ordenados como tipos producto
 - Listas heterogéneas como tipos recursivos
 - Funciones curricadas de múltiples parámetros
2. **Definir reglas de tipado estático** que garanticen la seguridad de tipos en las construcciones extendidas, previniendo errores de ejecución.
3. **Establecer las reglas de reducción semántica** para cada nueva construcción del lenguaje, demostrando propiedades de preservación y progreso.
4. **Implementar el mecanismo de evaluación** que respete la semántica operacional especificada, manejando adecuadamente los ambientes léxicos y la sustitución.

5. **Verificar la completitud del sistema** demostrando que toda expresión bien tipada en la sintaxis extendida puede ser reducida a una forma normal.

1.2. Referencias históricas

El lenguaje Lisp representa uno de los pilares fundamentales de la historia en los lenguajes de programación, influenciado fuertemente por el cálculo Lambda de Alonzo Church (1930's), proporcionando los fundamentos matemáticos para las funciones anónimas y la aplicación funcional.

MiniLISP emerge como núcleo pedagógico que captura la esencia de la familia LISP, rescatando el espíritu de Lisp 1.5 donde un lenguaje poderoso puede definirse con pocas funciones primitivas [1]. Su diseño minimalista permite el estudio de conceptos fundamental de programación funcional, la implementación accesible para fines educativos y la comprensión de mecanismo de evaluación y ambientes. Su innovador artículo "Recursive Functions of Symbolic Expressions and Their Computation by Machine" de 1960 no solo estableció los principios teóricos de Lisp, sino que también introdujo conceptos revolucionarios como la recursividad, el recolector de basura y la distinción entre código y datos a través de la notación S-expression, ideas que décadas después seguirían siendo fundamentales en el diseño de lenguajes de programación.

El dialecto principal de Lisp entre 1960 y 1965 fue Lisp 1.5 y con esto se introdujo el modelo de evaluación de funciones, el manejo explícito de ambientes léxicos y el sistema de macros que permitía a los programadores extender el lenguaje mismo. La publicación del manual "LISP 1.5 Programmer's Manual" en 1962 estableció un estándar que influyó en todas las implementaciones posteriores, desde MACLISP e Interlisp hasta los dialectos modernos como Scheme y Common Lisp.[1]

Con Lisp 1.5 también llegó la creación de MiniLISP décadas después como núcleo pedagógico que captura la esencia de la familia LISP. MiniLISP rescata el espíritu del Lisp 1.5 de 1962, donde un lenguaje poderoso puede definirse con solo algunas pocas funciones primitivas.

Lisp no solo definió las bases de la programación funcional, sino que también influyó profundamente en lenguajes posteriores como Scheme, Clojure, y Emacs Lisp. Su filosofía de tratar el código como datos inspiró a paradigmas modernos de metaprogramación y reflexión en lenguajes como Python y Julia[1]. A más de seis décadas de su creación, Lisp sigue siendo un referente en la investigación sobre inteligencia artificial, lenguajes de propósito simbólico y diseño de compiladores[1].

Capítulo 2

Formalización

Para poder extender el lenguaje MiniLisp con las funcionalidades ya mencionadas necesitamos formalizar las estructuras que usaremos, cómo se escribirán las operaciones y los elementos del lenguaje, la gramática, las reglas de evaluación para cada función implementada y como se ve la diferencia entre la superficie y el núcleo del lenguaje.

Para la formalización necesitamos definir cada componente del lenguaje:

- **Sintaxis:** Son las reglas y estructuras que especifican cómo deben escribirse los símbolos y palabras reservadas de un programa en nuestra extensión de MiniLisp. Definiremos la sintaxis concreta (reglas y gramática) y la sintaxis abstracta (Árboles de Sintaxis Abstracta). Además, especificaremos la eliminación de azúcar sintáctica para cada componente, para así identificar cuales elementos tiene el núcleo de nuestro lenguaje.
- **Semántica:** Para poder evaluar expresiones en el lenguaje necesitamos indicar el significado y la lógica llevada a cabo por cada elemento. Esta parte es fundamental para nuestro intérprete, para que ejecute de manera adecuada de acuerdo a cada expresión.

Algunos conceptos a los que haremos referencia a lo largo de este capítulo son los siguientes:

- *Lenguaje anfitrión:* Nos referimos como lenguaje anfitrión al lenguaje de programación en el cual está implementado o escrito nuestro intérprete. Como mencionamos en los objetivos, para este proyecto el lenguaje anfitrión es Haskell.
- *Superficie:* La superficie es la sintaxis del lenguaje tal como lo ve y lo escribe el programador. Contiene azúcar sintáctica, que como veremos adelante, facilita el uso del lenguaje para los humanos pero no para nuestra máquina.
- *Núcleo:* Llamamos núcleo al conjunto mínimo y más simple de constructores necesarios para expresar la semántica completa del lenguaje, consiste en la forma primitiva del lenguaje.

2.1. Sintaxis Concreta

Para definir la estructura de las expresiones, sentencias y programas de nuestro lenguaje de programación necesitamos formalizar la **sintaxis concreta**, la cual consiste en un conjunto de reglas que determinan cómo se visualizan los programas para el usuario (el programador). Esta parte de la formalización está diseñada para ser leída por humanos, es decir, son las formas de las cadenas con las que hará uso del lenguaje MINILISP. La especificación formal incluye dos tipos de reglas:

- **Reglas léxicas:** Definen los tokens básicos del lenguaje.
- **Reglas sintácticas:** Definen cómo se combinan los tokens para formar expresiones válidas.

2.1.1. Sintaxis Léxica

El primer paso en la formalización es representar las cadenas aceptadas por nuestro lenguaje por medio de *expresiones regulares*. Estas son una forma muy compacta de definir un lenguaje, en la formalización se proporcionó una expresión regular por cada componente léxico.

Expresión Regular

Considerando el conjunto de símbolos Σ que le llamamos **alfabeto**, definimos una *expresión regular* como:

- Si $a \in \Sigma$, entonces a es una expresión regular.
- ϵ es una expresión regular, representa la cadena vacía.
- Si α y β son dos expresiones regulares, estas pueden generar las siguientes expresiones regulares:
 - $\alpha + \beta$ representa la unión.
 - $\alpha\beta$ representa la concatenación de dos cadenas.
 - α^* representa la Estrella de Kleene, que son cero o más repeticiones de la cadena α .

Mediante la sintaxis léxica describimos formalmente la estructura de los componentes léxicos mediante **tokens** y **lexemas**. Para definir estos tokens hacemos uso de *expresiones regulares*, por lo que definimos el siguiente conjunto Σ como el alfabeto de las cadenas generadas por las expresiones.

Token: Un **token**[2] es la unidad mínima con significado en un lenguaje de programación. Representa categorías léxicas como identificadores, palabras reservadas, operadores, literales y delimitadores.

Lexema: Un **lexema**[2] es la secuencia específica de caracteres que forma un token particular. Mientras el token representa la categoría, el lexema es la instancia concreta.

- **Alfabeto:**

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, =, <, >, \leq, \geq, !, (,), [,], \#, \text{ } , a, \dots, z, A, \dots, Z\}$$

Ahora definimos las expresiones regulares que describen cada componente léxico del lenguaje. Estos tokens (representados por expresiones regulares) se forman a partir del alfabeto ya definido.

- **Espacios en blanco:** Separan tokens pero no son significativos.

$$_ + \backslash t + \backslash n$$

$_$: Espacio normal.

$\backslash t$: Tabular.

$\backslash n$: Salto de línea.

- **Operadores aritméticos:** Utilizaremos la suma, resta, división y multiplicación para operar sobre expresiones numéricas. Es importante hacer distinción entre los signos de suma para esta expresión, utilizamos $+$ como la representación de suma en nuestro lenguaje. Por otro lado, $+$ es notación de la expresión regular, que indica la unión de dos expresiones regulares.

$$+ + - + / + *$$

- **Operadores relacionales:** Dentro de estos operadores de orden nombramos los siguientes: igualdad, mayor que, mayor o igual que, menor que, menor o igual que, y distinto de. Estos nos permiten comparar expresiones numéricas.

$$= + > + \geq + < + \leq + !=$$

- **Paréntesis y comas:** Los paréntesis son tokens que representan las aperturas y cierre de las expresiones. Además nos auxiliamos de los paréntesis combinados con comas para representar los pares ordenados.

$$(+) + ,$$

- **Números enteros:** Números positivos, negativos o cero. Definimos nuestro conjunto de dígitos mediante $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, y además nos auxiliamos del conjunto $Z = D - \{0\}$ para evitar formar números como 01 y similares.

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

- **Identificadores:** Nombres de variables o funciones. Tenemos una letra inicial seguida de una combinación de letras. Esta expresión nos ayuda a representar variables dentro de expresiones, las cuales serán evaluadas de acuerdo a la expresión.

$$[a - zA - Z][a - zA - Z0 - 9]^*$$

- **Booleanos:** Valores lógicos. Verdadero y Falso respectivamente.

$$\#t + \#f$$

- **Listas:** Como se define cuando se quiere escribir una lista, usamos la representación con corchetes y los elementos de la lista separados por comas.

$$[+ , +]$$

- **Palabras reservadas:** Tokens que representan las palabras clave del lenguaje y no pueden usarse como identificadores, además de los combinadores predefinidos.

$$\text{add1} + \text{sub1} + \text{sqrt} + \text{expt} + \text{not} + \text{and} + \text{or} +$$

$$\text{let} + \text{let}^* + \text{letrec} + \text{if0} + \text{if} + \text{cond} + \text{else} + \text{lambda} + \text{fst} + \text{snd} + \text{head} + \text{tail} + \text{Z}$$

2.1.2. Sintaxis Libre de Contexto

La otra parte de la formalización de la Sintaxis Concreta es la Sintaxis Libre de Contexto, para esto hacemos uso de *gramáticas libres de contexto* que definen las reglas para la formación de las sentencias que acepta nuestro lenguaje MINILISP.

Gramática Libre de Contexto

Una **gramática libre de contexto** se define como una 4-tupla

$$G = (N, \Sigma, P, S),$$

donde:

- N es un conjunto finito de *símbolos no terminales* (o variables). Estos símbolos representan categorías sintácticas y pueden ser descompuestos en otras categorías o en símbolos terminales.
- Σ es un conjunto finito de *símbolos terminales* (o constantes). Estos son los símbolos básicos del lenguaje, que aparecen en las sentencias del lenguaje (por ejemplo, palabras reservadas, operadores, identificadores, etc.).
- P es un conjunto finito de *reglas de producción*. Cada regla tiene la forma

$$A \rightarrow \alpha,$$

donde $A \in N$ y $\alpha \in (N \cup \Sigma)^*$.

- S es el *símbolo inicial*, tal que $S \in N$, desde el cual se empieza la derivación de las cadenas del lenguaje.

A continuación definimos formalmente estas reglas para combinar de forma concreta los componentes léxicos. La representación de esta gramática es mediante la notación EBNF (Backus-Naur Extendido).

BNF, del inglés "Backus-Naur Form" fue introducida por John Backus y Peter Naur, quienes fueron los precursores en la utilización de una notación formal para describir la sintaxis de un lenguaje de programación. Este método formal expresa gramáticas libres de contexto. Por otro lado, **EBNF**.^{Extended} "Backus-Naur", definido en el estándar ISO/IEC 14977:1996[3] extiende BNF facilitando los elementos repetitivos en las reglas de producción. A diferencia de su antecesor, se implementan operadores como $\{\}$ y $[]$ para elementos opcionales, permitiendo una descripción más concisa de la sintaxis. Para nuestra gramática usaremos **EBNF**, definida en el estándar ISO/IEC 14977:1996[3]. Los símbolos utilizados tienen los siguientes significados:

- $\langle A \rangle ::= B$ (Sección 4.3): Define que el símbolo no terminal A está compuesto por la expresión B
- $A \mid B$ (Sección 4.4): Representa alternativas - A o B
- **'terminal'** (Sección 4.16): Símbolos terminales del lenguaje entre comillas simples
- $\{A\}$ (Sección 4.12): Repetición de cero o más veces de A
- A^+ : Repetición de una o más veces de A (extensión común)
- $[a - zA - Z]$ (Sección 4.16): Rango de caracteres terminales
- $\langle A \rangle$: Símbolos no terminales (meta-identificadores)

Gramática extendida de MiniLisp en EBNF[3]:

$$\begin{aligned}
 \langle S \rangle &::= \langle Expr \rangle \\
 \langle Expr \rangle &::= \langle Var \rangle \\
 &| \langle Int \rangle \\
 &| \langle Bool \rangle \\
 &| \langle List \rangle \\
 &| '(\langle Expr \rangle ' \langle Expr \rangle)' \\
 &| '(+ \langle Expr \rangle^+)' \\
 &| '(- \langle Expr \rangle^+)' \\
 &| '(* \langle Expr \rangle^+)' \\
 &| '(/ \langle Expr \rangle^+)' \\
 &| 'add1' \langle Expr \rangle \\
 &| 'sub1' \langle Expr \rangle \\
 &| 'sqrt' \langle Expr \rangle' \\
 &| 'expt' \langle Expr \rangle \langle Expr \rangle' \\
 &| 'not' \langle Expr \rangle' \\
 &| '(and' \langle Expr \rangle^+)' \\
 &| '(or' \langle Expr \rangle^+)' \\
 &| 'let' '({ (\langle Var \rangle \langle Expr \rangle^+) } \langle Expr \rangle)' \\
 &| 'let*' '({ (\langle Var \rangle \langle Expr \rangle^+) } \langle Expr \rangle)' \\
 &| 'letrec' '({ \langle Var \rangle \langle Expr \rangle^+ } \langle Expr \rangle)' \\
 &| 'if0' \langle Expr \rangle \langle Expr \rangle \langle Expr \rangle' \\
 &| 'if' \langle Expr \rangle \langle Expr \rangle \langle Expr \rangle' \\
 &| 'cond' '{ (\langle Expr \rangle \langle Expr \rangle^+) }^+ (else' \langle Expr \rangle^+)' \\
 &| 'lambda' '({ \langle Var \rangle }^+) \langle Expr \rangle' \\
 &| '(\langle Expr \rangle \langle Expr \rangle^+)' \\
 &| '(= \langle Expr \rangle^+)' \\
 &| '(> \langle Expr \rangle^+)' \\
 &| '(< \langle Expr \rangle^+)' \\
 &| '(>= \langle Expr \rangle^+)' \\
 &| '(<= \langle Expr \rangle^+)' \\
 &| '(! = \langle Expr \rangle^+)' \\
 &| '(fst' \langle Expr \rangle^+)' \\
 &| '(snd' \langle Expr \rangle^+)' \\
 &| '(head' \langle Expr \rangle^+)' \\
 &| '(tail' \langle Expr \rangle^+)' \\
 \langle Var \rangle &::= \langle ID \rangle \\
 \langle ID \rangle &::= [a-zA-Z][a-zA-Z0-9_] \\
 \langle Int \rangle &::= \langle N \rangle | '-' \langle M \rangle \\
 \langle D \rangle &::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' \\
 \langle N \rangle &::= '0' | \langle D \rangle \langle N \rangle^* \\
 \langle M \rangle &::= \langle D \rangle \langle N \rangle^* \\
 \langle Bool \rangle &::= '#t' | '#f' \\
 \langle List \rangle &::= '[]' | '[' \langle Expr \rangle \{ ' \langle Expr \rangle \}'
 \end{aligned}$$

2.2. Sintaxis Abstracta

2.2.1. Definiciones

Después de definir las reglas de producción para nuestro lenguaje MINILISP, corresponde formalizar la **sintaxis abstracta**, la cual define cómo se visualizan los programas para el intérprete o compilador. En esta formalización representamos la esencia del *input*, ignorando los detalles sintácticos superficiales.

Para esto, es conveniente presentar un tipo de dato abstracto como representación de esta sintaxis. Usaremos los **Árboles de Sintaxis Abstracta**, para abstraer los programas que la máquina recibe haciéndolos más entendibles para la misma.

Árbol de Sintaxis Abstracta - ASA

Un **árbol de sintaxis abstracta** (**ASA**) es un árbol ordenado $A = (N, A, R)$ donde:

- N es un conjunto finito de nodos que representan las construcciones del lenguaje mediante etiquetas y las hojas representan a sus respectivos valores.
- $A \subseteq N \times N$ es un conjunto de aristas dirigidas que conectan a los nodos.
- $R \in N$ es la raíz del nodo.

Para representar estas reglas consideramos las consecuencia como el nodo del árbol dado que se cumplen las premisas, que que serán los hijos de dicho nodo:

$$\frac{\text{premisas}}{\text{consecuencia}}$$

2.2.2. Árboles de Sintaxis Abstracta

Para representar la estructura jerárquica y lógica de los programas consideraremos los siguientes elementos del lenguaje como las hojas de los Árboles de Sintaxis Abstracta, es decir, los nodos finales que no tienen hijos.

Números: Num(n) es ASA si n es un número entero.

$$\frac{n \in \mathbb{Z}}{\text{Num}(n) \text{ ASA}}$$

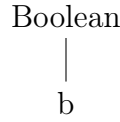
Con su representación de árbol:

$$\begin{array}{c} \text{Num} \\ | \\ n \end{array}$$

Booleanos: Boolean(b) es un ASA si b es un valor booleano, es decir, su valor está en $\mathbb{B} = \{True, False\}$.

$$\frac{b \in \mathbb{B}}{\text{Boolean}(b) \text{ ASA}}$$

Con su representación de árbol:



Variables: $Var(i)$ es un ASA si i es una cadena del conjunto $\mathbb{S} = \{s \mid s \in [a - zA - Z]^*\}$

$$\frac{i \in \mathbb{S}}{Var(i) \text{ ASA}}$$

Con su representación de árbol:

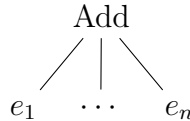


Ahora que definimos los valores que pueden tomar las hojas del árbol, debemos definir las operaciones sobre dichos valores que tomarán la forma de los nodos de nuestro árbol.

Operaciones aritméticas: Para que la suma de uno o más elementos sea un ASA, todos las expresiones que reciba en los parámetros también deben serlo. Identificamos a la suma como *Add*.

$$\frac{e_1 \text{ ASA} \quad e_2 \text{ ASA} \quad \cdots \quad e_n \text{ ASA} \quad n \geq 2}{Add(e_1, \dots, e_n) \text{ ASA}}$$

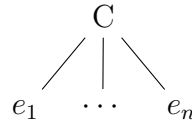
Con su representación como árbol:



Además de la suma, tendremos las representaciones análogas para la resta, la multiplicación y la división. Consideremos el operador $C \in \{Add, Sub, Mul, Div\}$ donde: *Add* corresponde a la suma (+), *Sub* a la resta (−), *Mul* a la multiplicación (*) y *Div* a la división (/). Entonces la operación indicada será un ASA si todas las expresiones recibidas como parámetros son ASA también. Debido a la aridad mayor o igual a 2, los árboles tendrán n hijos.

$$\frac{e_1 \text{ ASA} \quad e_2 \text{ ASA} \quad \cdots \quad e_n \text{ ASA} \quad n \geq 2}{C(e_1, \dots, e_n) \text{ ASA}}$$

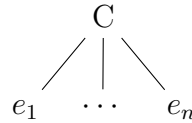
Con su representación como árbol:



Operaciones booleanas: Para las operaciones booleanas binarias denotamos la conjunción (\wedge) como *And* y la disyunción lógica (\vee) como *Or*. Sea el operador lógico $C \in \{And, Or\}$, si todos los parámetros que recibe son ASA, entonces la operación indicada también lo es. Al igual que en las operaciones aritméticas, permitimos operar sobre más de dos expresiones, por lo que el operador correspondiente tendrá n hijos para n expresiones recibidas.

$$\frac{b_1 \text{ ASA} \quad b_2 \text{ ASA} \quad \cdots \quad b_n \text{ ASA} \quad n \geq 2}{C(b_1, \dots, b_n) \text{ ASA}}$$

Con su representación como árbol:



Por otro lado, el operador lógico unario de negación (\neg) se denota en la sintaxis abstracta como *Not*. De modo que $Not(b)$ será un ASA si el parámetro que recibe b es un ASA.

$$\frac{b \text{ ASA}}{Not(b) \text{ ASA}}$$

Con su representación como árbol:



Otras operaciones aritméticas: Además de las operaciones aritméticas ya mencionadas, se implementan las operaciones *add1* (suma 1 al número recibido), *sub1* (resta 1 al número recibido), *sqrt* (calcula la raíz cuadrada) y *expt* (calcula el resultado del primer número recibido elevado al segundo número recibido). Las operaciones mencionadas las denotamos como *Add1*, *Sub1*, *Sqrt* y *Expt* respectivamente. De este modo, $Add1(e)$, $Sub(e)$, $Sqrt(e)$ y $Expt(e_1, e_2)$ son ASA si se cumple que las expresiones recibidas en los parámetros también son ASA (e , e_1 y e_2). Las primeras tres operaciones tienen un hijo y a la última le corresponden dos hijos.

$$\frac{e \text{ ASA}}{Add1(e) \text{ ASA}} \quad \frac{e \text{ ASA}}{Sub1(e) \text{ ASA}} \quad \frac{e \text{ ASA}}{Sqrt(e) \text{ ASA}} \quad \frac{e_1 \text{ ASA} \quad e_2 \text{ ASA}}{Expt(e_1, e_2) \text{ ASA}}$$

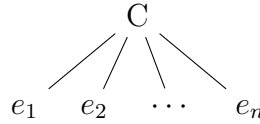
Con sus respectivas representaciones como árboles:



Predicados sobre enteros: Para comparaciones entre números enteros denotamos al operador "=" (igual que) como Eq , a "<" (menor que) como Lt , a ">" (mayor que) como Gt , a "<=" (menor o igual que) como Lte , a ">=" (mayor o igual que) como Gte , y a "!=" (distinto de) como Ne . De este modo, para $C \in \{Eq, Lt, Gt, Lte, Gte, Ne\}$, si todos los elementos recibidos en los parámetros son ASA, también lo será el operador correspondiente. Estos nodos tendrán n hijos (al menos dos) debido a que son predicados variádicos.

$$\frac{e_1 \text{ ASA} \quad \cdots \quad e_n \text{ ASA}}{C(e_1, \dots, e_n) \text{ ASA}}, \quad n \geq 2$$

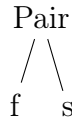
Con su representación como árbol:



Pares ordenados: En el lenguaje podemos formar pares ordenados de la forma (f, s) . Denotamos $Pair(f, s)$ a estos pares ordenados, donde f y s deben ser ASA para que el par $Pair(f, s)$ sea también un ASA. Debido a que recibe tiene dos parámetros, este nodo tendrá dos hijos.

$$\frac{f \text{ ASA} \quad s \text{ ASA}}{Pair(f, s) \text{ ASA}}$$

Con su representación como árbol:



Proyecciones: Una vez que definidos los pares ordenados, contamos con la operación $fst(f, s)$ que devuelve el primer elemento del par y $snd(f, s)$ que devuelve el segundo. Denotamos estas operaciones en sintaxis abstracta como Fst y Snd respectivamente. De este modo, si p es un ASA, entonces $Fst(p)$ y $Snd(p)$ son ASA también. El árbol tiene un único hijo.

$$\frac{p \text{ ASA}}{Fst(p) \text{ ASA}} \quad \frac{p \text{ ASA}}{Snd(p) \text{ ASA}}$$

Con su representación como árbol:



Asignaciones locales: Let y Let*

Identificadores: Un Id es un ASA si s es una cadena válida.

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

Con su representación como árbol:

$$\begin{array}{c} \text{Id} \\ | \\ s \end{array}$$

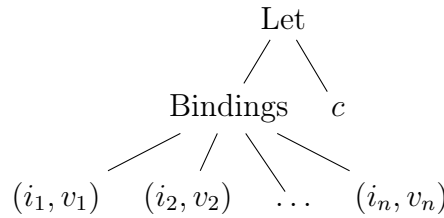
Binding: Un binding es un par (i, v) donde i es un identificador (String) y v es un ASA que representa el valor a asignar.

$$\frac{i : \text{String} \quad v \text{ ASA}}{(i, v) \text{ Binding}}$$

Let : El constructor Let recibe una lista no vacía de bindings y un cuerpo. Todos los bindings se evalúan en paralelo, por lo que por ahora ninguna variable puede depender de las demás dentro de la misma expresión let.

$$\frac{i_1, \dots, i_n : \text{String} \quad v_1, \dots, v_n \text{ ASA} \quad c \text{ ASA} \quad \text{con}(n \geq 1)}{\text{Let}([(i_1, v_1), \dots, (i_n, v_n)], c) \text{ ASA}}$$

Con su representación como árbol:

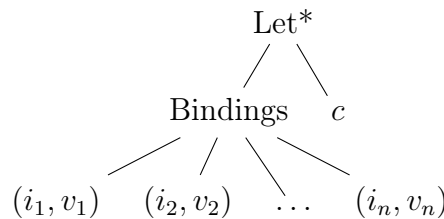


Tenemos que en Let, todas las asignaciones se consideran en paralelo, antes de extender el ambiente con los bindings. Por lo tanto, expresiones como `(let ((x 17) (y x)) y)` nos darían error, ya que x no está definida al momento de evaluar el binding de y .

Let*: El constructor Let* recibe una lista no vacía de bindings y un cuerpo. Los bindings se evalúan secuencialmente de izquierda a derecha, permitiendo ahora sí que ya cada variable pueda depender de otras variables evaluadas anteriormente.

$$\frac{i_1, \dots, i_n : \text{String} \quad v_1, \dots, v_n \text{ ASA} \quad c \text{ ASA} \quad \text{con}(n \geq 1)}{\text{Let}^*([(i_1, v_1), \dots, (i_n, v_n)], c) \text{ ASA}}$$

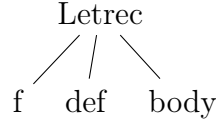
Con su representación como árbol:



Letrec : Para funciones recursivas, se usa un único binding que se resuelve mediante el combinador Z.

$$\frac{f : \text{String} \quad \text{def ASA} \quad \text{body ASA}}{\text{Letrec}(f, \text{def}, \text{body}) \text{ ASA}}$$

Con su representación como árbol:

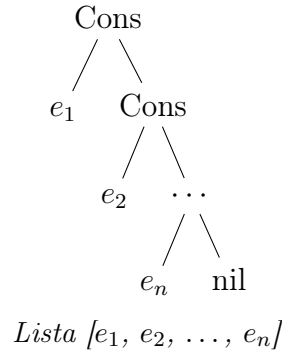
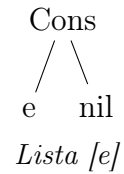


Listas: La lista vacía que representamos con *nil* es un ASA. Para la construcción de listas, si *e* es un ASA y *l* es un ASA que representa otra lista, entonces la construcción *Cons(e, l)* también es un ASA, teniendo como cabeza *e*.

$$\frac{}{\text{nil ASA}} \quad \frac{e \text{ ASA} \quad l \text{ ASA}}{\text{Cons}(e, l) \text{ ASA}}$$

Con su representación como árbol:

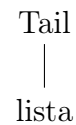
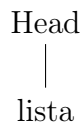
nil
Lista vacía



Operaciones de listas: Si la expresión que representa una lista es un ASA, entonces para dicha expresión *l*, Head(*l*) y Tail(*l*) son también ASA, un árbol con un único hijo. Considere $C \in \{Head, Tail\}$.

$$\frac{l \text{ ASA}}{C(l) \text{ ASA}}$$

Con su representación como árbol:



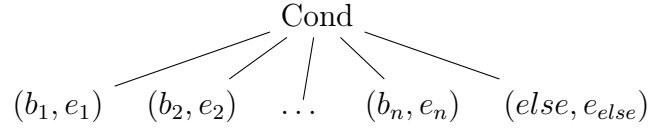
Condicional por cláusulas: Sea

$$\text{Cond}(\{(b_1, e_1), \dots, (b_n, e_n), (else, e_{else})\})$$

Una expresión condicional, donde cada b_i es un booleano ASA y cada e_i es un ASA. La evaluación se realiza en orden y selecciona la primera cláusula cuya guarda sea verdadera; la cláusula **else** actúa como caso por defecto.

$$\frac{b_1 \text{ ASA} \quad e_1 \text{ ASA} \quad \cdots \quad b_n \text{ ASA} \quad e_n \text{ ASA} \quad e_{\text{else}} \text{ ASA}}{\text{Cond}(((b_1, e_1), \dots, (b_n, e_n), (\text{else}, e_{\text{else}})))) \text{ ASA}}$$

Con su representación como árbol:

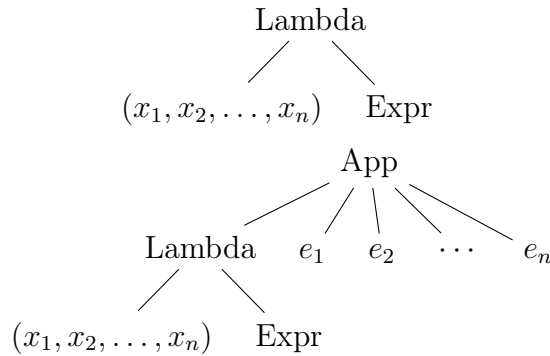


Funciones anónimas (Lambda): Una función anónima con múltiples parámetros es un ASA si su cuerpo es un ASA. La aplicación de la función a argumentos también es un ASA si todos los argumentos son ASA. La aplicación se evalúa de izquierda a derecha, es decir, $\text{App}(\text{App}(f, x), y)$ representa $((fx)y)$.

$$\frac{\text{Expr ASA}}{\text{Lambda}((x_1, x_2, \dots, x_n), \text{Expr}) \text{ ASA}}$$

$$\frac{\text{Lambda}((x_1, x_2, \dots, x_n), \text{Expr}) \text{ ASA} \quad e_1 \text{ ASA} \quad e_2 \text{ ASA} \quad \cdots \quad e_n \text{ ASA}}{\text{App}(\text{Lambda}((x_1, x_2, \dots, x_n), \text{Expr}), e_1, e_2, \dots, e_n) \text{ ASA}}$$

Con su representación como árbol:



Combinadores Predefinidos

El lenguaje incluye combinadores predefinidos en el ambiente inicial que no requieren definición explícita por el usuario.

Combinador Z:

El combinador Z permite la implementación de recursión. En el ambiente inicial ε_0 , se define:

$$Z \in \varepsilon_0$$

Donde Z está ligado a la siguiente expresión cerrada del núcleo:

$$Z = \text{Fun}(f, \text{App}(\text{Fun}(x, \text{App}(\text{Id}(f), \text{Fun}(y, \text{App}(\text{App}(\text{Id}(x), \text{Id}(x)), \text{Id}(y))))), \text{Fun}(x, \text{App}(\text{Id}(f), \text{Fun}(y, \text{App}(\text{App}(\text{Id}(x), (x)), \text{Id}(y))))))))))$$

Tenemos que para cualquier función g , se cumple:

$$\text{App}(Z, g) \equiv g(\text{App}(Z, g))$$

Esta propiedad permite que g reciba su propia definición como argumento, permitiéndonos la recursión.

2.3. Eliminación de azúcar sintáctica

El concepto de **azúcar sintáctica** fue acuñado por Peter Landin para referirse a construcciones en un lenguaje de programación que no añaden poder expresivo pero mejoran la legibilidad y conveniencia [4]. Según Reynolds:

El **azúcar sintáctica** [4] se refiere a construcciones en un lenguaje de programación que pueden definirse traduciéndolas a construcciones más fundamentales del lenguaje. El proceso de **desazucarización** (o también **desugaring**) consiste en reemplazar ocurrencias del lado izquierdo de una ecuación definitoria por el lado derecho correspondiente.

Para definir como sucede este proceso durante la evaluación, daremos las reglas de desazucarización para cada elemento del lenguaje. Estas reglas de traducción van de la superficie del lenguaje (lo que ve el usuario o programador) al núcleo mínimo (como lo procesa nuestra máquina). Usaremos la siguiente estructura para poder representarlos de manera visual. Nombramos como *Desugar* a este proceso dentro de las reglas de traducción, básicamente, recibe cualquier expresión que haya ingresado el usuario y aplica las reglas para transformarse al núcleo.

$$\frac{\text{premisas}}{\text{Desugar}(\text{superficie}) \rightarrow \text{núcleo}}$$

2.3.1. Principios Fundamentales

1. **Minimalismo:** Reducir el lenguaje a un conjunto mínimo de construcciones primitivas que preserven su poder expresivo completo.
2. **Composicionalidad:** Garantizar que cada transformación preserve la estructura composicional del lenguaje.
3. **Preservación de la Semántica:** Mantener el significado computacional intacto durante la transformación

Las reglas de eliminación de azúcar sintáctica presentadas se basan en principios establecidos en el diseño de lenguajes de programación.[4, 5].

Operadores Variádicos: La transformación de operadores con múltiples argumentos en aplicaciones binarias anidadas sigue el principio de **currificación** establecido en la semántica lenguajes funcionales [6]. Por ejemplo:

$$(+ \ 1 \ 2 \ 3) \rightarrow (+ \ (+ \ 1 \ 2)3)$$

Una transformación necesaria ya que simplifica la semántica ya que en el núcleo solo necesitamos definir operaciones binarias, reduce la complejidad pues el evaluador maneja un solo caso en lugar de n casos y permite la composición, donde las operaciones se construyen a partir de primitivas simples.

Listas: La transformación de listas a estructuras **Cons** anidadas sigue el enfoque fundamental de LISP:[7]

$$[1, 2, 3] \rightarrow \text{Cons}(1, \text{Cons}(2, \text{Cons}(3, \text{Nill})))$$

Así permitimos en esta representación que sea canónica en la familia de lenguajes LISP, el tratamiento recursivo uniforme de las listas y facilita la implementación de operaciones de listas.

En esta sección ocuparemos este enfoque pero mediante los pares ordenados, que son los que nos ayudarán a implementar estas definiciones como desazucarización. **Funciones Lambda Currificadas:** La transformación de funciones multiparámetro a aplicaciones sucesivas sigue el **principio de currificación** [6]:

$$(\text{lambda}(x\ y\ e) \rightarrow ((\text{lambda}x) (\text{lambda}y) e))$$

Con esta aproximación reducimos todas las funciones a un único parámetro, permitimos la aplicación parcial de funciones y simplificamos el sistema de tipos y la semántica.

Condicional por cláusulas: La transformación de **cond** a estructuras condicionales básicas sigue el principio de **descomposición de formas parciales** [5]:

$$(\text{cond}(b_1\ e_1)(b_2\ e_2)(\text{else}\ e_3)) \rightarrow (\text{if}\ b_1\ e_1\ (\text{if}\ b_2\ e_2\ e_3))$$

Reducimos las múltiples construcciones de control a una primitiva básica que permite una implementación más simple del evaluador y facilitamos el razonamiento formal sobre el flujo de control.

2.3.2. Ventajas

EL proceso de desazucarización proporciona múltiples beneficios [4]:

- **Implementación Simplificada:** El núcleo del lenguaje es más pequeño y más fácil de implementar correctamente.
- **Verificación Formal:** Un conjunto reducido de construcciones facilita la verificación formal y pruebas de corrección.
- **Extensibilidad:** Nuevas formas de azúcar sintáctica pueden añadirse sin modificar el núcleo del lenguaje.
- **Portabilidad:** La semántica del núcleo puede implementarse en diferentes entornos manteniendo la misma interfaz de superficie.

Corrección de las Transformaciones

Cada regla de transformación preserva la semántica operacional del lenguaje. La corrección se asegura mediante:

1. **Preservación de Tipos:** Si una expresión en la superficie está bien tipada, su traducción al núcleo también lo está.
2. **Equivalencia Operacional:** Para cualquier estado y ambiente, la evaluación de una expresión superficie produce el mismo resultado que su traducción al núcleo.
3. **Terminación:** La terminación de programas se preserva bajo transformación.

Asignaciones locales: **Let** y **Let***

1. **Simplicidad del núcleo:** El intérprete solo necesita implementar **Id**, **Fun** y **App**, no casos especiales para **let**.
2. **Alcance estático garantizado:** Las cerraduras capturan el ambiente, asegurando alcance léxico.
3. **Currificación automática:** Múltiples bindings se traducen naturalmente a funciones de un parámetro.
4. **Equivalencia semántica:** La evaluación de la forma desazucarizada produce exactamente el mismo resultado que una implementación directa de **let**.

2.3.3. Expresiones básicas

Números: Un número en la superficie se traduce directamente al nodo **Num** en el núcleo.

$$\frac{n \in \mathbb{Z}}{Desugar(n) \mapsto Num(n)}$$

■ Ejemplos:

$$Desugar(7) \mapsto Num(7) \quad Desugar(-3) \mapsto Num(-3)$$

Booleanos: Los valores booleanos de la superficie (**#t** y **#f**) se traducen a nodos **Boolean(True)** y **Boolean(False)** respectivamente en el núcleo.

$$\frac{}{Desugar(\#t) \mapsto Boolean(True)} \quad \frac{}{Desugar(\#f) \mapsto Boolean(False)}$$

■ Ejemplos:

$$Desugar(\#t) \mapsto Boolean(True) \quad Desugar(\#f) \mapsto Boolean(False)$$

Variables: Una variable en la superficie se traduce a un nodo **Var** en el núcleo.

$$\frac{i \in \mathbb{S}}{Desugar(i) \mapsto Var(i)} \quad \mathbb{S} = \{s \mid s \in [a - zA - Z]^*\}$$

■ Ejemplos:

$$Desugar(x) \mapsto Var(x) \quad Desugar(y) \mapsto Var(y)$$

2.3.4. Operaciones aritméticas, booleanas y predicados

Operaciones aritméticas: Las operaciones aritméticas en la superficie son manejados como operadores variádicos (que reciben hasta más de dos expresiones en los argumentos), sin embargo, para el núcleo es diferente. En esta formalización pasamos de operadores variádicos a operadores binarios mediante la anidación de las expresiones recibidas por el operador, de modo que se van asociando poco a poco hasta solo manejar operadores binarios, para los cuales definiremos las reglas en la sección correspondiente.

Un concepto importante a considerar es el uso de la estructura *LASA*, que nos ayudará con las siguientes reglas de transformación. Un *LASA* es una estructura que representa una lista de *ASA*.

$$\frac{x \text{ ASA} \quad xs \text{ ASA}}{(x : xs) \text{ LASA}}$$

De esta manera iremos representando como vamos transformando elemento por elemento de nuestra expresión. Si tenemos un operador variádico y recibe n elementos, entonces almacenamos cada uno de ellos en una lista *LASA*.

Para los siguientes operadores se utiliza la asociatividad por la izquierda que veremos reflejada en las reglas de transformación para las operaciones aritméticas. Las expresiones aritméticas en la superficie se traducen a nodos **Add**, **Sub**, **Mul** o **Div** en el núcleo. Las operaciones con más de dos argumentos se descomponen en aplicaciones binarias anidadas y curricadas.

En este punto haremos una distinción importante, tomaremos $\odot \in \{+, -, *, /\}$ y $Op \in \{Add, Sub, Mul, Div\}$. Mencionamos que en el núcleo únicamente se manejan operadores binarios, por lo que \odot representa la operación variádica en la superficie, mientras que Op representa la operación binaria correspondiente, es decir, para $+$ denotamos su operación binaria *Add*, para $-$ tenemos *Sub*, para $*$ tenemos *Mul* y para $/$ tenemos *Div*. De esta manera iremos analizando los distintos casos. Haremos una generalización para estas cuatro operaciones debido a que no estamos manejando su evaluación y nos interesa formalizar como se elimina la azúcar sintáctica y dado que todas estas operaciones tendrán asociatividad por la izquierda podemos generalizarlo con el uso de \odot y Op , cuyos valores ya definimos.

$$\begin{array}{ll} \frac{(e : Nil) \text{ LASA}}{Desugar(\odot e) \mapsto Desugar(e)} & \text{Solo tenemos una expresión} \\ \frac{(e_1 : e_2) \text{ LASA}}{Desugar(\odot e_1 e_2) \mapsto Op(Desugar(e_1), Desugar(e_2))} & \text{Usamos operador binario} \\ \frac{(e_1 : (e_2 : resto)) \text{ LASA} \quad \text{resto va hasta la expresión } e_n}{Desugar(\odot e_1 e_2 \cdots e_n) \mapsto Desugar(\odot(\odot(e_1, e_2) : resto))} & \text{Hasta } n \text{ expresiones} \end{array}$$

■ Ejemplos:

$$\begin{aligned} Desugar(+ 1 2 3) &\mapsto Add(Add(Num(1), Num(2)), Num(3)) \\ Desugar(- 10 3 2) &\mapsto Sub(Sub(Num(10), Num(3)), Num(2)) \\ Desugar(* 2 3 4) &\mapsto Mul(Mul(Num(2), Num(3)), Num(4)) \\ Desugar(/ (+ 2 3) 5 8) &\mapsto Div(Div(Add(Num(2), Num(3)), Num(5)), Num(8)) \end{aligned}$$

Operaciones booleanas: En el caso de las operaciones booleanas, realizamos la desazucarización para los operadores booleanos *And* y *Or* (conjunción y disyunción respectivamente) de forma análoga a como explicamos en las operaciones aritméticas (la asociatividad es la misma, asociatividad por la izquierda). Consideramos la misma definición de la estructura *LASA* que corresponde a una lista de Árboles de Sintaxis Abstracta (*ASA*).

A diferencia de las operaciones aritméticas, consideramos $\odot \in \{and, or\}$ los operadores variádicos que se manejan en la superficie (la vista al usuario), por otro lado tenemos $Op \in \{And, Or\}$ los operadores binarios en el lenguaje, que se verán únicamente en el núcleo del lenguaje a la hora de evaluar. De esta manera, también generalizamos las reglas para la conjunción y disyunción sin afectar la semántica, pues

únicamente estamos transformando como se ve el lenguaje para que lo interprete la máquina. Con esto, el operador variádico *and* corresponde con el operador binario *And*, y *or* con *Or*.

$$\begin{array}{c}
\frac{(e : Nil) \text{ LASA}}{Desugar(\odot e) \mapsto Desugar(e)} \quad \text{Solo tenemos una expresión} \\
\\
\frac{(e_1 : e_2) \text{ LASA}}{Desugar(\odot e_1 e_2) \mapsto Op(Desugar(e_1), Desugar(e_2))} \quad \text{Usamos operador binario} \\
\\
\frac{(e_1 : (e_2 : resto)) \text{ LASA} \quad \text{resto va hasta la expresión } e_n}{Desugar(\odot e_1 e_2 \cdots e_n) \mapsto Desugar(\odot(\odot(e_1, e_2) : resto))} \quad \text{Hasta } n \text{ expresiones}
\end{array}$$

■ **Ejemplos:**

$$\begin{aligned}
Desugar(or \#f \#f \#t) &\mapsto Or(Or(Boolean(False), Boolean(False)), Boolean(True)) \\
\\
Desugar(and \#t (or \#f \#f) \#f) & \\
\mapsto And(And(Boolean(True), Or(Boolean(False), Boolean(False))), Boolean(False))
\end{aligned}$$

Por otro lado, tenemos el operador unario de la negación lógica. Este operador no es variádico como la conjunción y la disyunción, por lo que su estructura después de la eliminación de azúcar sintáctica deja igual la parte del operador en núcleo *Not* que representa al operador en superficie *not* y son la negación de un valor booleano.

$$\frac{e \text{ ASA}}{Desugar(Not(e)) \mapsto Not(Desugar(e))}$$

■ **Ejemplos:**

$$\begin{aligned}
Desugar(Not(Boolean(True))) &\mapsto Not(Boolean(True)) \\
\\
Desugar(not(and \#t \#f \#f)) & \\
\mapsto Not(And(And(Boolean(True), Boolean(False)), Boolean(False)))
\end{aligned}$$

Predicados sobre enteros: Similar a como manejamos la eliminación de azúcar sintáctica en los operadores variádicos anteriores y considerando la misma estructura planteada sobre *LASA*, se suman los predicados sobre enteros a los operadores variádicos del lenguaje. Como predicados sobre enteros tenemos en la superficie $\odot \in \{=, <, <=, >, >=, ! =\}$ como operadores variádicos y $Op \in \{Eq, Lt, Lte, Gt, Gte, Neq\}$, donde la *igualdad* corresponde a $=$ con *Eq*, *menor que* corresponde a $<$ con *Lt*, *menor o igual que* corresponde a $<=$ con *Lte*, *mayor que* corresponde a $>$ con *Gt*, *mayor o igual que* corresponde a $>=$ con *Gte*, y por último *distinto de* Para los siguientes operadores de comparación se utiliza la asociatividad por la izquierda. Para expresiones con más de dos argumentos, se desazucarizan en aplicaciones binarias anidadas. Cada predicado en la superficie se traduce a su nodo correspondiente en el núcleo.

$$\begin{array}{c}
\frac{(e : Nil) \text{ LASA}}{Desugar(\odot e) \mapsto Desugar(e)} \quad \text{Solo tenemos una expresión} \\
\\
\frac{(e_1 : e_2) \text{ LASA}}{Desugar(\odot e_1 e_2) \mapsto Op(Desugar(e_1), Desugar(e_2))} \quad \text{Usamos operador binario} \\
\\
\frac{(e_1 : (e_2 : resto)) \text{ LASA} \quad \text{resto va hasta la expresión } e_n}{Desugar(\odot e_1 e_2 \cdots e_n) \mapsto Desugar(\odot(\odot(e_1, e_2) : resto))} \quad \text{Hasta } n \text{ expresiones}
\end{array}$$

Ejemplos:

$$\begin{aligned}
Desugar(= 2 2 3 2) &\mapsto Eq(Eq(Eq(Num(2), Num(2)), Num(3)), Num(2)) \\
Desugar(< 1 2 3) &\mapsto Lt(Lt(Num(1), Num(2)), Num(3)) \\
Desugar(> 4 3 2) &\mapsto Gt(Gt(Num(4), Num(3)), Num(2)) \\
Desugar(<= 5 5 6) &\mapsto Le(Le(Num(5), Num(5)), Num(6)) \\
Desugar(>= 7 6 6) &\mapsto Ge(Ge(Num(7), Num(6)), Num(6)) \\
Desugar(! = 1 2 1) &\mapsto Ne(Ne(Num(1), Num(2)), Num(1))
\end{aligned}$$

2.3.5. Pares ordenados

Pares ordenados: Un par ordenado en la superficie se traduce a `Pair(f, s)` en el núcleo.

$$\frac{f \text{ ASA} \quad s \text{ ASA}}{Desugar((f, s)) \mapsto Pair(Desugar(f), Desugar(s))}$$

■ Ejemplos:

$$\begin{aligned}
Desugar(1, 2) &\mapsto Pair(Num(1), Num(2)) \\
Desugar(x, y) &\mapsto Pair(Var(x), Var(y))
\end{aligned}$$

Proyecciones: Las operaciones `fst` y `snd` operan sobre pares ordenados, estas se traducen a `Fst` y `Snd` respectivamente en el núcleo.

$$\begin{array}{ll}
\frac{p \text{ ASA}}{Desugar(fst \ p) \mapsto Fst(Desugar(p))} & \text{Corresponde a First} \\
\frac{p \text{ ASA}}{Desugar(snd \ p) \mapsto Snd(Desugar(p))} & \text{Corresponde a Second}
\end{array}$$

■ Ejemplos:

$$\begin{aligned}
Desugar(fst \ (1, 2)) &\mapsto Fst(Pair(Num(1), Num(2))) \\
Desugar(snd \ (x, y)) &\mapsto Snd(Pair(Var(x), Var(y)))
\end{aligned}$$

2.3.6. Listas como pares ordenados

En la superficie podemos ingresar listas de la forma $[e_1, \dots, e_n]$ donde cada e_n es un ASA. Sin embargo, en el núcleo del lenguaje no tenemos un dato específico para las listas, sino que mediante la eliminación de azúcar sintáctica se transforma a pares anidados. De esta manera los elementos del lenguaje *head* y *tail* son ambos azúcar sintáctica, pues después del desugaring simplemente estamos operando con pares ordenados y ya no definimos dichas operaciones.

Listas:

La lista vacía se traduce a *Nil*, que es el elemento del núcleo del lenguaje que usaremos para representar dicha estructura, de esta manera ya no debemos ocupar `[]`, que es azúcar sintáctica.

$$\overline{Desugar([])} \mapsto Nil$$

Para listas con un elemento usaremos el par ordenado del mismo elemento con *Nil*, nuestra variable para la lista vacía.

$$\frac{e \text{ ASA}}{\text{Desugar}([e]) \rightarrow \text{Pair}(e, \text{Nil})}$$

Si consideramos la estructura que antes definimos como lista de ASA, podemos dar otra formalización donde almacenamos en una lista de ASA todos los elementos de la lista proporcionada en la superficie.

$$\frac{(x : xs) \text{ LASA}}{\text{Desugar}((x : xs)) \mapsto \text{Pair}(\text{Desugar}(x), \text{Desugar}(xs))}$$

■ **Ejemplo:**

$$\text{Desugar}([1, 2, 3]) \mapsto \text{Pair}(\text{Num}(1), \text{Pair}(\text{Num}(2), \text{Pair}(\text{Num}(3), \text{Nil})))$$

Operaciones de listas:

Como mencionamos antes, las operaciones de listas en el núcleo son representadas por las operaciones de los pares ordenados.

$$\frac{e \text{ ASA}}{\text{Desugar}(\text{head } e) \mapsto \text{Fst}(\text{Desugar}(e))} \quad \text{Head pasa a First}$$

$$\frac{e \text{ ASA}}{\text{Desugar}(\text{tail } e) \mapsto \text{Snd}(\text{Desugar}(e))} \quad \text{Tail pasa a Second}$$

■ **Ejemplos:**

$$\text{Desugar}(\text{head } [1, 2, 3]) \mapsto \text{First}(\text{Pair}(\text{Num}(1), \text{Pair}(\text{Num}(2), \text{Pair}(\text{Num}(3), \text{Nil}))))$$

$$\text{Desugar}(\text{tail } [1, 2, 3]) \mapsto \text{Tail}(\text{Pair}(\text{Num}(1), \text{Pair}(\text{Num}(2), \text{Pair}(\text{Num}(3), \text{Nil}))))$$

2.3.7. Condicionales

Condicional: Para los condicionales recibimos tres expresiones, la primera es la condición que define el camino a seguir, la segunda define que se va a devolver si la condición es verdadera y la tercera se devuelve cuando es falsa la condición. En el núcleo (denotamos *If*), cuando tenemos esta operación que indica el condicional, mediante la desazucarización pasa igual, pero debe desazucarizar las expresiones que recibe.

$$\frac{c \text{ ASA} \quad t \text{ ASA} \quad e \text{ ASA}}{\text{Desugar}(\text{if } c \text{ } t \text{ } e) \mapsto \text{If}(\text{Desugar}(c), \text{Desugar}(t), \text{Desugar}(e))}$$

■ **Ejemplo:**

$$\text{Desugar}(\text{if}(\#t) \ 1 \ 2) \mapsto \text{If}(\text{Boolean}(\text{True}), \text{Num}(1), \text{Num}(2))$$

If0: El condicional (en la superficie) *if0* recibe tres expresiones al igual que el condicional *if*, el orden y significado de dichas expresiones es el mismo (condición, then, else), sin embargo, *if0* es azúcar sintáctica de *if*, pues simplemente podemos comparar la expresión que recibamos en *if0* como condición (la representamos como *c*) y ver si es igual a 0 con los predicados que ya hemos definido en esta sección. Las demás expresiones también las desazucarizamos, pero después de eliminar la azúcar sintáctica podemos ver que hacemos uso del condicional *if* anterior. Tomando en cuenta esto, cuando definamos las

reglas semánticas únicamente debemos desarrollar únicamente las que corresponden al condicional que definimos antes, pues después de esta transformación no necesitamos otras adicionales.

$$\frac{c \text{ ASA} \quad t \text{ ASA} \quad e \text{ ASA}}{\text{Desugar}(if0 \ c \ t \ e) \mapsto If(Eq(\text{Desugar}(c), \text{Num}(0), \text{Desugar}(t), \text{Desugar}(e)))}$$

■ **Ejemplo:**

$$\text{Desugar}(if0(0) \ 42 \ 7) \mapsto If(Eq(\text{Num}(0), \text{Num}(0)), \text{Num}(42), \text{Num}(7))$$

Condicional por clausulado: El condicional por clausulado nos ofrece otra funcionalidad al lenguaje. A diferencia de los condicionales ya definidos, con el clausulado podemos ingresar más de una condición de manera ordenada. En esta estructura vamos a utilizar guardas booleanas que contienen expresiones asociadas, en la evaluación se irán evaluando una a una hasta encontrar la primera de ellas que sea verdadera. Al final de todas las guardas, es imperativo almacenar la correspondiente a **else** con valor verdadero, para que de este modo actúe como caso por defecto (para cuando ninguna de las clausulas fue verdadera).

Esta operación es una aplicación del condicional que definimos primero ($If(c, t, e)$ en sintaxis abstracta), pues su función es aplicar varios condicionales anidados, de esta manera se irá evaluando uno por uno. Con esto en mente, veremos que este condicional por clausulado es nuevamente azúcar sintáctica del If inicial, así en las reglas correspondientes a la semántica solo tendremos que definir las reglas para el If que definimos al inicio de la sección.

Las **restricciones de uso de la cláusula else** son las siguientes:

- Actúa como caso por defecto. Se evalúa si y solo si todas las guardas booleanas anteriores han fallado. Si **else** apareciera antes, cualquier cláusula que le siguiera sería código inalcanzable debido a que siempre la asociamos con valor *True*.
- La clausula **else** debe aparecer como máximo una vez. De lo contrario estaríamos contradiciendo la definición de que está definida como opción por defecto, y al igual que en el punto anterior, no evaluamos todo.

$$\frac{c \text{ ASA}, e \text{ ASA}, \text{resto ASA}}{\text{Desugar}(\text{cond } (c \ e) \ \text{resto}) \mapsto If(\text{Desugar}(c), \text{Desugar}(e), \text{Desugar}(\text{cond } \text{resto}))}$$

$$\frac{e \text{ ASA}}{\text{Desugar}(\text{cond } (\text{True } e)) \rightarrow \text{Desugar}(e)}$$

■ **Ejemplo:**

$$\begin{aligned} &\text{Desugar}(\text{cond } ((>x \ 0) \ 1) \ ((= \ x \ 0) \ 0) \ (\text{True } -1)) \\ &\mapsto If(\text{Gt}(\text{Var}(x), \text{Num}(0)), \text{Num}(1), If(Eq(\text{Var}(x), \text{Num}(0)), \text{Num}(0), \text{Num}(-1))) \end{aligned}$$

2.3.8. Funciones Lambda y Aplicaciones

De acuerdo con la estructura **LASA** previamente definida, podemos representar las listas de parámetros y argumentos en las expresiones **Lambda** y **App**. En la superficie, las funciones anónimas pueden definirse con múltiples parámetros, mientras que en el núcleo se traducen mediante funciones currificadas **Lambda**, donde cada parámetro se aplica secuencialmente. El cuerpo de la función **Expr** debe ser un **ASA**.

Funciones anónimas (Lambda):

$$\frac{(x : \text{Nil}) \text{ LASA}, \text{ Expr ASA}}{\text{Desugar}(\text{lambda } (x) \text{ Expr}) \mapsto (\text{Lambda } x) \text{ Desugar}(\text{Expr})} \quad \text{Solo un parámetro}$$

$$\frac{(x_1 : (x_2 : \text{resto})) \text{ LASA}, \text{ Expr ASA}}{\text{Desugar}(\text{lambda } (x_1 \ x_2 \ \dots \ x_n) \text{ Expr}) \mapsto (\text{Lambda } x_1) \text{ Desugar}(\text{lambda } (x_2 \ \dots \ x_n) \text{ Expr})} \quad \text{Varios parámetros}$$

De esta forma, una función anónima con múltiples parámetros se transforma en una secuencia de funciones curricadas anidadas, aplicadas una tras otra, siguiendo la estructura de la lista LASA.

■ **Ejemplo:**

$$\text{Desugar}(\text{lambda } (x \ y) (+ \ x \ y)) \mapsto ((\text{Lambda } x) (\text{Lambda } y) ((\text{Add } x) y))$$

Aplicación de funciones (App):

$\text{App}(f, e_1, \dots, e_n)$ es una expresión del núcleo que representa la aplicación de la función f a los argumentos e_1, \dots, e_n . Toda llamada a una función anónima en la superficie se traduce a una secuencia explícita de aplicaciones **App**, separando la función de sus argumentos. Para representar la lista de argumentos usamos nuevamente la estructura **LASA**.

$$\frac{(f : \text{Nil}) \text{ LASA}}{\text{Desugar}((f)) \mapsto \text{Desugar}(f)} \quad \text{Solo la función}$$

$$\frac{(f : (a_1 : \text{Nil})) \text{ LASA}}{\text{Desugar}((f \ a_1)) \mapsto \text{App}(\text{Desugar}(f), \text{Desugar}(a_1))} \quad \text{Un argumento}$$

$$\frac{(f : (a_1 : (a_2 : \text{resto}))) \text{ LASA}}{\text{Desugar}((f \ a_1 \ a_2 \ \dots \ a_n)) \mapsto \text{App}(\text{Desugar}((f \ a_1)), \text{Desugar}((a_2 \ \dots \ a_n)))} \quad \text{Varios argumentos}$$

■ **Ejemplo:**

$$\begin{aligned} & \text{Desugar}(((\text{lambda } (x \ y) (+ \ x \ y)) \ 2 \ 3)) \\ & \mapsto \text{App}(\text{App}((\text{Lambda } \text{Var}(x)) (\text{Lambda } \text{Var}(y)) ((\text{Add } \text{Var}(x)) \text{Var}(y))), \text{Num}(2)), \text{Num}(3)) \end{aligned}$$

2.3.9. Asignaciones locales: Let, Let* y Letrec**Identificadores:**

$$\text{Desugar}(i) = \text{Id}(i) \quad \text{donde } i : \text{String}$$

Funciones curricadas: Una función de un solo parámetro.

$$\text{Desugar}((\text{lambda } (p) \ c)) = \text{Fun}(p, \text{Desugar}(c))$$

Aplicación:

$$\text{Desugar}((f \ a)) = \text{App}(\text{Desugar}(f), \text{Desugar}(a))$$

Desazucarización de Let

Let evalúa todos los valores de los bindings en el ambiente actual (sin acceso a bindings previos), luego extiende el ambiente con todas las asignaciones simultáneamente.

Caso base:

$$\text{Desugar}((\text{let } [] \ c)) = \text{Desugar}(c)$$

Caso con un binding:

$$\frac{i : \text{String} \quad v \text{ ASA} \quad c \text{ ASA}}{\text{Desugar}((\text{let } [(i, v)] \ c)) = \text{App}(\text{Fun}(i, \text{Desugar}(c)), \text{Desugar}(v))}$$

Caso recursivo (múltiples bindings):

$$\frac{\begin{array}{c} i_1 : \text{String} \quad v_1 \text{ ASA} \\ \text{rest} = [(i_2, v_2), \dots, (i_n, v_n)] \\ c \text{ ASA} \end{array}}{\text{Desugar}((\text{let } [(i_1, v_1) \mid \text{rest}] \ c)) = \text{App}(\text{Fun}(i_1, \text{Desugar}((\text{let rest } c))), \text{Desugar}(v_1))}$$

Cada binding (i_k, v_k) se traduce a una aplicación (lambda $(i_k) \dots$) v_k donde el cuerpo contiene las aplicaciones anidadas restantes.

Ejemplo:

$$\begin{aligned} & \text{Desugar}((\text{let } [(x, 1), (y, 2), (z, 3)] \ (* \ x \ y \ z))) \\ &= ((\text{lambda } (x) \ ((\text{lambda } (y) \ ((\text{lambda } (z) \ (* \ x \ y \ z)) \ 3)) \ 2)) \ 1) \end{aligned}$$

Desazucarización de Let*

Let* evalúa cada binding secuencialmente, permitiendo que cada valor v_i pueda referirse a variables ligadas en bindings anteriores (i_1, \dots, i_{i-1}) .

Caso base:

$$\text{Desugar}((\text{let}^* [] \ c)) = \text{Desugar}(c)$$

Caso con un binding:

$$\frac{i : \text{String} \quad v \text{ ASA} \quad c \text{ ASA}}{\text{Desugar}((\text{let}^* [(i, v)] \ c)) = \text{App}(\text{Fun}(i, \text{Desugar}(c)), \text{Desugar}(v))}$$

Caso recursivo:

$$\frac{\begin{array}{c} i_1 : \text{String} \quad v_1 \text{ ASA} \\ \text{rest} = [(i_2, v_2), \dots, (i_n, v_n)] \\ c \text{ ASA} \end{array}}{\text{Desugar}((\text{let}^* [(i_1, v_1) \mid \text{rest}] \ c)) = \text{App}(\text{Fun}(i_1, \text{Desugar}((\text{let}^* \text{rest } c))), \text{Desugar}(v_1))}$$

Letrec

Letrec permite definiciones recursivas donde la función puede referirse a sí misma en su definición y para esto utilizamos el combinador **Z**.

Definición del Combinador **Z**:

En cálculo lambda:

$$Z = \lambda f. (\lambda x. f (\lambda y. (x x) y)) (\lambda x. f (\lambda y. (x x) y))$$

Regla de desazucarización de Letrec:

$$\frac{f : \text{String} \quad \text{def ASA} \quad \text{body ASA}}{\text{Desugar}((\text{letrec} ((f \text{ def})) \text{ body})) = \text{Desugar}((\text{let} [(f, \text{App}(Z, \text{Fun}(f, \text{def}))]) \text{ body}))}$$

Con ésta regla transformamos **letrec** en un **let** donde el valor asignado a **f** es la aplicación del combinador **Z** a una función que recibe 'f' como parámetro y así permitiendo que se auto-referencie (para la recursión).

Ejemplo: *Fibonacci*

```
(letrec ((fib (lambda (n)
  (if0 (- n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))))
(fib 6))
```

Desazucarización:

1. *Aplicar regla de letrec:*

$$\begin{aligned} & \text{Desugar}((\text{letrec} ((fib (\lambda (n) (\text{if0} (- n 1) n (+ (fib (- n 1)) (fib (- n 2))))))) \\ & \quad (fib 6))) \\ &= \text{Desugar}((\text{let} [(fib, (Z (\lambda (fib) (\lambda (n) \dots)))] (fib 6))) \end{aligned}$$

2. *Expandir el cuerpo completo:*

$$\begin{aligned} & \text{Desugar}((\text{let} [(fib, \\ & \quad (Z (\lambda (fib) \\ & \quad \quad (\lambda (n) \\ & \quad \quad \quad (\text{if0} (- n 1) \\ & \quad \quad \quad \quad n \\ & \quad \quad \quad \quad (+ (fib (- n 1)) (fib (- n 2))))))]) \\ & \quad (fib 6))) \end{aligned}$$

3. *Desazucarizar let a aplicación:*

$$\begin{aligned} &= ((\lambda (fib) (fib 6)) \\ & \quad (Z (\lambda (fib) \\ & \quad \quad (\lambda (n) \\ & \quad \quad \quad (\text{if0} (- n 1) n (+ (fib (- n 1)) (fib (- n 2))))))) \end{aligned}$$

2.4. Semántica Operacional Estructural

2.4.1. Definiciones

La **semántica** de un lenguaje de programación define el significado que se le da a las construcciones sintácticas que ya hemos formalizado para MiniLisp. Mediante esta formalización se busca especificar como funciona la ejecución de cada implementación en el lenguaje, para ello, definiremos las reglas que aplican para cada elemento de nuestro lenguaje.

Como herramientas para la formalización de la semántica contamos con el enfoque *operacional*, el *denotativo* y el *axiomático*. El enfoque para este proyecto es el operacional. La **semántica operacional** define el significado de los programas mediante la especificación de reglas de transición, estas reglas muestran como va cambiando de estados la máquina hasta llegar a un estado final de ser válido el programa que se procesa o llega a un error.

El uso de transiciones es fundamental para la definición de las reglas.

Sistema de Transición

Un sistema de transición se define como una tupla (E, δ, I) , donde:

- E es un conjunto finito de estados.
- $\delta \subseteq E \times E$ es una relación de transición entre estados, que describe cómo un estado puede evolucionar a otro.
- I es un conjunto de estados iniciales, que especifica los estados desde los cuales puede comenzar la ejecución del sistema.

De igual manera, contamos con dos maneras de formalizar la semántica operacional: la *estructural* (o paso pequeño) y la *natural* (o paso grande). Debido a que queremos especificar minuciosamente el cambio de estados en la ejecución del lenguaje, nos apoyaremos del enfoque estructural. En la **semántica operacional estructural**, damos un análisis detallado del comportamiento al descomponer los pasos de ejecución en cada transición. Es importante mencionar que la extensión de MiniLisp que estamos formalizando se ejecuta de izquierda a derecha, es decir, vamos procesando elemento por elemento iniciando por la izquierda y finalizando con el elemento a la derecha.

Dentro del lenguaje definimos estructuras que permiten asignar valores o expresiones a variables, estas asignaciones toman lugar en otras expresiones de acuerdo al alcance que definió dicha variable. Esto nos lleva a la **sustitución**, el proceso de reemplazar una variable por un valor o una expresión en una estructura específica, consiste en tomar una expresión y reemplazar todas las ocurrencias de una variable en esa expresión por otra expresión o valor.

Sin embargo, como técnica de evaluación es engorrosa al obligarnos a reescribir constantemente el texto del programa, por lo que para aplicar una compensación entre espacio y tiempo almacenaremos la sustitución en una estructura de datos llamada entorno (environment) [8].

Un **entorno** registra nombres y sus valores correspondientes, es decir, es una colección de pares clave-valor. El comportamiento de esta estructura la consideramos como una pila, esta forma de almacenamiento permite que las sustituciones sean llamadas conforme las requiera la ejecución del programa. De este modo, cada vez que encontramos una vinculación, recordamos su valor (binding, es decir, lo

almacenamos), y cuando encontramos una variable, buscamos su valor [8]. Estos entornos son nuestros **ambientes de evaluación**, para esta formalización denotamos estas estructuras como ε , y este puede ser vacío (\emptyset).

2.4.2. Ambiente Inicial

El intérprete inicia la evaluación con un ambiente inicial ε_0 que contiene los combinadores y funciones primitivas predefinidas:

$$\varepsilon_0 = \{("Z", \text{Closure}(Z, \emptyset))\}$$

Donde Z es la expresión cerrada del combinator de punto fijo definida en la sección 2.3.

Este ambiente inicial se extiende con cada nuevo binding durante la evaluación, pero el combinator Z siempre permanece accesible en todas las evaluaciones.

2.4.3. Reglas de transición

Definimos las reglas mediante la relación denotada como $e_1 \rightarrow e_2$, lo cual indica que e_1 se reduce a e_2 en un paso. Presentamos las reglas paso a paso de la ejecución, analizando el caso en el que se encuentra nuestra evaluación de la expresión.

Los números y los booleanos tenemos se reducen a sí mismos, pues ya son valores terminales y no necesitan procesarse. El ambiente de evaluación no cambia. Otro elemento en el lenguaje a considerar es la lista vacía, pues a partir de esta vamos generando la construcción de listas, está esta representada con *nil* y se reduce a sí misma.

Números

$$\frac{n \in \mathbb{N}}{\langle \text{Num}(n), \varepsilon \rangle \rightarrow \langle \text{Num}(n), \varepsilon \rangle}$$

Booleanos

$$\frac{b \in \{\text{True}, \text{False}\}}{\langle \text{Boolean}(b), \varepsilon \rangle \rightarrow \langle \text{Boolean}(b), \varepsilon \rangle}$$

Lista vacía

$$\overline{\langle \text{Nil}, \varepsilon \rangle \rightarrow \langle \text{Nil}, \varepsilon \rangle}$$

Ahora podemos empezar a especificar las operaciones sobre nuestro lenguaje. Como se mencionó anteriormente, tenemos que ubicar el estado en el que nos encontramos dentro de la ejecución.

Operaciones aritméticas

Al momento de definir la eliminación de azúcar sintáctica (desugaring), mostramos que los operadores variádicos (suma, resta, multiplicación, división) al tener más de dos expresiones a resolver, se anidan de forma que quedan en términos de los operadores binarios correspondientes. Por lo que para el núcleo del lenguaje solo se usan los operadores binarios de dichas operaciones. Con esto en mente, definimos las operaciones aritméticas binarias como sigue.

■ Suma

Cuando ninguno de los argumentos es un número (expresado como $Num(n)$ con $n \in \mathbb{Z}$), tenemos que reducir en un paso la primera expresión a la izquierda.

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle Add(i, d), \varepsilon \rangle \rightarrow \langle Add(i', d), \varepsilon \rangle}$$

Cuando la primera expresión es un número (expresado como $Num(n)$ donde $n \in \mathbb{Z}$) pero la segunda expresión no lo es, entonces reducimos en un paso la segunda expresión.

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle Add(Num(n), d), \varepsilon \rangle \rightarrow \langle Add(Num(n), d'), \varepsilon \rangle}$$

Cuando tenemos que las expresiones que recibe Add ya son números, digamos $Num(n)$ y $Num(m)$ donde $n, m \in \mathbb{Z}$, entonces recurrimos a la suma en el lenguaje anfitrión para resolver el operador.

$$\overline{\langle Add(Num(n), Num(m)), \varepsilon \rangle \rightarrow \langle Num(n + m), \varepsilon \rangle}$$

■ Resta

Tomamos el mismo análisis usado en la suma, es decir, si es necesario se reduce primero el lado izquierdo hasta obtener un número terminal, luego lo mismo para el lado derecho hasta tener la suma de dos números y por último se recurre a la suma en el lenguaje anfitrión.

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle Sub(i, d), \varepsilon \rangle \rightarrow \langle Sub(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle Sub(Num(n), d), \varepsilon \rangle \rightarrow \langle Sub(Num(n), d'), \varepsilon \rangle}$$

$$\overline{\langle Sub(Num(n), Num(m)), \varepsilon \rangle \rightarrow \langle Num(n - m), \varepsilon \rangle}$$

■ Multiplicación

De forma similar a como hemos manejado la suma y la resta hasta ahora, definimos las reglas para la multiplicación binaria. Usamos el análisis análogo a la suma y la resta para el desarrollo de las reglas de este operador.

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle Mul(i, d), \varepsilon \rangle \rightarrow \langle Mul(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle Mul(Num(n), d), \varepsilon \rangle \rightarrow \langle Mul(Num(n), d'), \varepsilon \rangle}$$

$$\overline{\langle Mul(Num(n), Num(m)), \varepsilon \rangle \rightarrow \langle Num(n * m), \varepsilon \rangle}$$

■ División

De nuevo usamos el mismo análisis para las operaciones aritméticas que hemos definido hasta ahora. A diferencia de ellas, aquí tenemos que hacer un paso extra de verificación sobre la división por cero, pues de ser cero el dividendo tendremos que arrojar un error indicando que esa operación no es válida.

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle Div(i, d), \varepsilon \rangle \rightarrow \langle Div(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle Div(Num(n), d), \varepsilon \rangle \rightarrow \langle Div(Num(n), d'), \varepsilon \rangle}$$

$$\frac{}{\langle Sub(Num(n), Num(0)), \varepsilon \rangle \rightarrow \langle Error, \varepsilon \rangle} \quad \text{Cuando el dividendo es cero, arrojam error}$$

$$\frac{m \neq 0}{\langle Sub(Num(n), Num(m)), \varepsilon \rangle \rightarrow \langle Num(n/m), \varepsilon \rangle} \quad \text{Cuando es distinto a cero, podemos operar}$$

Otras operaciones aritméticas

Además de las operaciones básicas ya implementadas, el lenguaje posee 4 operaciones aritméticas más: incremento en 1 (denotado como *add1*), decremento en 1 (denotado como *sub1*), raíz cuadrada (denotado como *sqr*) y potencias (denotado como *?*). A continuación se presentan las reglas de evaluación correspondientes a cada operación.

■ Incremento en 1

Cuando la expresión recibida no es un número (expresado como *Num*(*n*) con $n \in \mathbb{Z}$), entonces reducimos en un paso dicha expresión.

$$\frac{\langle e, \varepsilon \rangle \rightarrow \langle e', \varepsilon \rangle}{\langle Add1(e), \varepsilon \rangle \rightarrow \langle Add1(e'), \varepsilon \rangle}$$

Ya que la expresión es un número (expresado como *Num*(*n*) con $n \in \mathbb{Z}$) entonces recurrimos a la suma en el lenguaje anfitrión para poder aplicar $n + 1$ con $n \in \mathbb{Z}$ (el incremento en 1 de un número).

$$\frac{}{\langle Add1(Num(n)), \varepsilon \rangle \rightarrow \langle Num(n + 1), \varepsilon \rangle}$$

■ Decremento en 1

El análisis de sus reglas es análogo al caso del incremento en 1, únicamente cambia la operación realizada en el lenguaje anfitrión, en lugar de sumar 1, restamos 1 ($n - 1$ donde $n \in \mathbb{Z}$).

$$\frac{\langle e, \varepsilon \rangle \rightarrow \langle e', \varepsilon \rangle}{\langle Sub1(e), \varepsilon \rangle \rightarrow \langle Sub1(e'), \varepsilon \rangle}$$

$$\frac{}{\langle Sub1(Num(n)), \varepsilon \rangle \rightarrow \langle Num(n - 1), \varepsilon \rangle}$$

■ Raíz cuadrada

Cuando el argumento no es un número (expresado como $Num(n)$ con $n \in \mathbb{Z}$) debemos reducirlo en un paso.

$$\frac{\langle e, \varepsilon \rangle \rightarrow \langle e', \varepsilon \rangle}{\langle Sqrt(e), \varepsilon \rangle \rightarrow \langle Sqrt(e'), \varepsilon \rangle}$$

En el caso donde la expresión que se recibe es un número (expresado como $Num(n)$ con $n \in \mathbb{Z}$), podemos aplicar dicha operación en el lenguaje anfitrión.

$$\overline{\langle Sqrt(Num(n)), \varepsilon \rangle \rightarrow \langle Num(Sqrt(n)), \varepsilon \rangle}$$

■ Potencia

Cuando ninguno de los argumentos es un número (expresado como $Num(n)$ con $n \in \mathbb{Z}$), reducimos en un paso la primera expresión (el lado izquierdo).

$$\frac{\langle e_1, \varepsilon \rangle \rightarrow \langle e'_1, \varepsilon \rangle}{\langle Expt(e_1, e_2), \varepsilon \rangle \rightarrow \langle Expt(e'_1, e_2), \varepsilon \rangle}$$

Cuando la primera expresión ya es un número (expresado como $Num(n)$ con $n \in \mathbb{Z}$) y la segunda expresión no es un número, se reduce en un paso la segunda expresión como sigue.

$$\frac{\langle e_2, \varepsilon \rangle \rightarrow \langle e'_2, \varepsilon \rangle}{\langle Expt(Num(n), e_2), \varepsilon \rangle \rightarrow \langle Expt(Num(n), e'_2), \varepsilon \rangle}$$

Finalmente, cuando ambas expresiones que recibe $Expt$ ya son números, digamos $Num(n)$ y $Num(m)$ donde $n, m \in \mathbb{Z}$, entonces recurrimos a la operación correspondiente en el lenguaje anfitrión para obtener el resultado.

$$\overline{\langle Expt(Num(n), Num(m)), \varepsilon \rangle \rightarrow \langle Num(n^m), \varepsilon \rangle}$$

Predicados sobre enteros

De manera similar a las operaciones aritméticas (las definidas con enfoque variádico), en la sección de eliminación de azúcar sintáctica se presentó como en la superficie podemos ingresar todas las expresiones que aplicarán al operador correspondiente, sin embargo, para el núcleo del lenguaje recurrimos a las operaciones binarias respectivas de cada operador implementado. Con esto en mente, se definen las reglas de evaluación únicamente para los operadores binarios de cada predicado sobre los enteros que maneja el lenguaje.

■ Igualdad

Dados dos números enteros queremos identificar si son iguales, por lo que diremos que es verdadero si son exactamente el mismo número y falso en caso contrario. Cuando el primer argumento aún no es un número, se reduce en un paso dicha expresión.

$$\frac{e_1 \rightarrow e'_1}{\langle Eq(e_1, e_2), \varepsilon \rangle \rightarrow \langle Eq(e'_1, e_2), \varepsilon \rangle}$$

Cuando la primera expresión ya es un número (digamos $Num(n)$ donde $n \in \mathbb{Z}$) pero la segunda aún no lo es, se desarrolla en un paso la segunda expresión.

$$\frac{e_2 \rightarrow e'_2}{\langle Eq(Num(n), e_2), \varepsilon \rangle \rightarrow \langle Eq(Num(n), e'_2), \varepsilon \rangle}$$

Cuando ambos argumentos de la expresión Eq (que representa la igualdad $=$) ya son números (expresado como $Num(n), Num(m)$ con $n, m \in \mathbb{N}$), entonces se aplicamos la igualdad del lenguaje anfitrión con su resultado que es un *Booleano*, que indica si los dos números son iguales o no (true o false respectivamente). Es importante mencionar que la operación $e_1 = e_2$ mencionada dentro del elemento booleano ya corresponde a la operación del lenguaje anfitrión.

$$\overline{\langle Eq(Num(n), Num(m)), \varepsilon \rangle} \rightarrow \langle Boolean(e_1 = e_2), \varepsilon \rangle$$

■ Menor que

En esta operación se determina si la primera expresión es menor que la segunda expresión recibida. Cuando ninguno de los argumentos es un terminal correspondiente a un número, se reduce primero el argumento de la izquierda.

$$\frac{e_1 \rightarrow e'_1}{\langle Lt(e_1, e_2), \varepsilon \rangle \rightarrow \langle Lt(e'_1, e_2), \varepsilon \rangle}$$

Cuando la primera expresión ya es un número (digamos $Num(n)$ donde $n \in \mathbb{Z}$) pero la segunda no lo es, se reduce la segunda expresión en un paso.

$$\frac{e_2 \rightarrow e'_2}{\langle Lt(Num(n), e_2), \varepsilon \rangle \rightarrow \langle Lt(Num(n), e'_2), \varepsilon \rangle}$$

Finalmente, cuando ambos argumentos ya son números (digamos $Num(n)$ y $Num(m)$ donde $n, m \in \mathbb{Z}$), se realiza la comparación usando el operador del lenguaje anfitrión, con el resultado *Booleano* que indica si es verdadero que el primer número es menor que el segundo.

$$\overline{\langle Lt(Num(e_1), Num(e_2)), \varepsilon \rangle} \rightarrow \langle Boolean(e_1 < e_2), \varepsilon \rangle$$

■ Mayor que

Funciona análogamente al operador menor que. La única diferencia es la operación correspondiente al lenguaje anfitrión, sin embargo, las reglas de reducción siguen la misma lógica ya explicada.

$$\frac{e_1 \rightarrow e'_1}{\langle Gt(e_1, e_2), \varepsilon \rangle \rightarrow \langle Gt(e'_1, e_2), \varepsilon \rangle}$$

$$\frac{e_2 \rightarrow e'_2}{\langle Gt(Num(e_1), e_2), \varepsilon \rangle \rightarrow \langle Gt(Num(e_1), e'_2), \varepsilon \rangle}$$

$$\overline{\langle Gt(Num(e_1), Num(e_2)), \varepsilon \rangle} \rightarrow \langle Boolean(e_1 > e_2), \varepsilon \rangle$$

■ Menor o igual que

Se definen las reglas para este operador análogamente al segundo operador analizado, menor que. La diferencia es que no sólo lo considera verdadero cuando la primera expresión es menor que la

segunda, sino que también cuando son iguales. El análisis y desarrollo de las reglas de evaluación son los mismos.

$$\frac{e_1 \rightarrow e'_1}{\langle Lte(e_1, e_2), \varepsilon \rangle \rightarrow \langle Lte(e'_1, e_2), \varepsilon \rangle}$$

$$\frac{e_2 \rightarrow e'_2}{\langle Lte(Num(e_1), e_2), \varepsilon \rangle \rightarrow \langle Lte(Num(e_1), e'_2), \varepsilon \rangle}$$

$$\frac{}{\langle Lte(Num(e_1), Num(e_2)), \varepsilon \rangle \rightarrow \langle Boolean(e_1 \leq e_2), \varepsilon \rangle}$$

■ Mayor o igual que

Análogamente a las reglas sobre predicados ya definidas, tenemos las correspondientes al predicado mayor o igual que.

$$\frac{e_1 \rightarrow e'_1}{\langle Gte(e_1, e_2), \varepsilon \rangle \rightarrow \langle Gte(e'_1, e_2), \varepsilon \rangle}$$

$$\frac{e_2 \rightarrow e'_2}{\langle Gte(Num(e_1), e_2), \varepsilon \rangle \rightarrow \langle Gte(Num(e_1), e'_2), \varepsilon \rangle}$$

$$\frac{}{\langle Gte(Num(e_1), Num(e_2)), \varepsilon \rangle \rightarrow \langle Boolean(e_1 \geq e_2), \varepsilon \rangle}$$

■ Distinto de

Por último, las reglas para este predicado son análogas a las presentadas anteriormente (en los operadores de predicados sobre números). De igual manera, el único cambio (además de la sintaxis abstracta) es la operación correspondiente al lenguaje anfitrión.

$$\frac{e_1 \rightarrow e'_1}{\langle Neq(e_1, e_2), \varepsilon \rangle \rightarrow \langle Neq(e'_1, e_2), \varepsilon \rangle}$$

$$\frac{e_2 \rightarrow e'_2}{\langle Neq(Num(e_1), e_2), \varepsilon \rangle \rightarrow \langle Neq(Num(e_1), e'_2), \varepsilon \rangle}$$

$$\frac{}{\langle Neq(Num(e_1), Num(e_2)), \varepsilon \rangle \rightarrow \langle Boolean(e_1 \neq e_2), \varepsilon \rangle}$$

Operaciones booleanas

Al igual que los anteriores operadores variádicos, al pasar por la eliminación de azúcar sintáctica (que ya desarrollamos en la sección correspondiente) ya solo tenemos estos operadores anidados de forma que ya solo operan sobre dos expresiones, es decir, manejamos los operadores binarios correspondientes. De este modo, en el núcleo únicamente se opera sobre dos expresiones con la misma lógica que hemos desarrollado hasta ahora, es decir, de izquierda a derecha.

■ Conjunción

Buscamos desarrollar el lado izquierdo de la expresión hasta que sea un valor booleano, después hacemos lo mismo para el lado derecho y cuando en ambos lados tenemos valores booleanos, podemos usar la operación correspondiente en el lenguaje anfitrión.

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle And(i, d), \varepsilon \rangle \rightarrow \langle And(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle And(Boolean(b), d), \varepsilon \rangle \rightarrow \langle And(Boolean(b), d'), \varepsilon \rangle}$$

$$\frac{}{\langle And(Boolean(b_1), Boolean(b_2)), \varepsilon \rangle \rightarrow \langle Boolean(b_1 \wedge b_2), \varepsilon \rangle}$$

■ Disyunción

Para estas reglas tenemos el caso análogo a la conjunción, el único cambio es la operación dentro del lenguaje anfitrión.

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle Or(i, d), \varepsilon \rangle \rightarrow \langle Or(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle Or(Boolean(b), d), \varepsilon \rangle \rightarrow \langle Or(Boolean(b), d'), \varepsilon \rangle}$$

$$\frac{}{\langle Or(Boolean(b_1), Boolean(b_2)), \varepsilon \rangle \rightarrow \langle Boolean(b_1 \vee b_2), \varepsilon \rangle}$$

■ Negación

Contrario a los otros operadores (disyunción y conjunción), este operador es unario y únicamente tenemos que ir reduciendo en un paso la expresión que recibe hasta llegar a un valor booleano. De este modo, ya podemos operar con el lenguaje anfitrión.

$$\frac{\langle b, \varepsilon \rangle \rightarrow \langle b', \varepsilon \rangle}{\langle Not(b), \varepsilon \rangle \rightarrow \langle Not(b'), \varepsilon \rangle}$$

$$\frac{}{\langle Not(Boolean(b)), \varepsilon \rangle \rightarrow \langle Boolean(\neg b), \varepsilon \rangle}$$

Pares ordenados

La idea de trabajar con pares ordenados es reducirlos hasta que ambos elementos del par sean valores terminales y ya no puedan ser desarrollados. Se reduce en un paso el primer elemento del par hasta tener un valor terminal, luego hacemos lo mismo para el lado derecho y por último, un par con ambos elementos como valores terminales se reduce a sí mismo. De este modo, diremos que en las siguientes reglas v_f y v_s son números, booleanos, un par ordenado o un error.

$$\frac{\langle f, \varepsilon \rangle \rightarrow \langle f', \varepsilon \rangle}{\langle Pair(f, s), \varepsilon \rangle \rightarrow \langle Pair(f', s), \varepsilon \rangle}$$

$$\frac{\langle s, \varepsilon \rangle \rightarrow \langle s', \varepsilon \rangle}{\langle Pair(v_f, s), \varepsilon \rangle \rightarrow \langle Pair(v_f, s'), \varepsilon \rangle}$$

$$\frac{}{\langle Pair(v_f, v_s), \varepsilon \rangle \rightarrow \langle Pair(v_f, v_s), \varepsilon \rangle}$$

Proyecciones

Otra nueva funcionalidad del lenguaje son las proyecciones. Una proyección nos devuelve un elemento del par ordenado (el primero o el segundo según la operación que se quiera aplicar).

■ First

Esta operación, busca devolver el primer elemento del par ordenado, por lo que debe ir reduciendo en paso pequeño la expresión que recibe hasta tener un par de valores terminales y así devolver el primero de ellos, el cual vemos representado con v_1 .

$$\frac{\langle p, \varepsilon \rangle \rightarrow \langle p', \varepsilon \rangle}{\langle Fst(p), \varepsilon \rangle \rightarrow \langle Fst(p'), \varepsilon \rangle}$$

$$\frac{}{\langle Fst(Pair(v_1, v_2)), \varepsilon \rangle \rightarrow \langle v_1, \varepsilon \rangle}$$

■ Second

El otro caso para las proyecciones es donde devolvemos el segundo elemento del par ordenado. Igual que en *First* buscamos reducir la expresión hasta tener un par ordenado de valores terminales y ahora devolver el segundo de ellos, representado por v_2 .

$$\frac{\langle p, \varepsilon \rangle \rightarrow \langle p', \varepsilon \rangle}{\langle Snd(p), \varepsilon \rangle \rightarrow \langle Snd(p'), \varepsilon \rangle}$$

$$\overline{\langle Snd(Pair(v_1, v_2)), \varepsilon \rangle \rightarrow \langle v_2, \varepsilon \rangle}$$

Asignaciones Locales

Como nos dice la nota 10, **let** y **let*** son azúcar sintáctica de la aplicación de función, pero también **letrec**. En el núcleo, no existen reglas específicas para estos constructos, pues se desazucaran completamente a aplicaciones con lambdas curificadas.

Identificadores

Los identificadores buscan su valor en el ambiente de evaluación ε . Si el identificador i está en el ambiente, se sustituye por su valor; de lo contrario, es una variable libre y genera error.

$$\frac{i \in \varepsilon}{\langle Id(i), \varepsilon \rangle \rightarrow \langle \varepsilon(i), \varepsilon \rangle}$$

$$\frac{i \notin \varepsilon}{\langle Id(i), \varepsilon \rangle \rightarrow \langle Error, \varepsilon \rangle}$$

Funciones (generan cerraduras)

Una función se evalúa a una cerradura que captura el ambiente actual. Esto garantiza alcance estático.

$$\overline{\langle Fun(p, c), \varepsilon \rangle \rightarrow \langle Closure(p, c, \varepsilon), \varepsilon \rangle}$$

Aplicación de función

La aplicación evalúa primero la función, luego el argumento, y finalmente aplica la función usando el ambiente capturado en la cerradura (alcance estático).

$$\frac{\langle f, \varepsilon \rangle \rightarrow \langle f', \varepsilon \rangle}{\langle App(f, a), \varepsilon \rangle \rightarrow \langle App(f', a), \varepsilon \rangle}$$

$$\frac{\langle a, \varepsilon \rangle \rightarrow \langle a', \varepsilon \rangle}{\langle App(Closure(p, c, \varepsilon'), a), \varepsilon \rangle \rightarrow \langle App(Closure(p, c, \varepsilon'), a'), \varepsilon \rangle}$$

$$\frac{v_a \text{ es valor}}{\langle App(Closure(p, c, \varepsilon'), v_a), \varepsilon \rangle \rightarrow \langle c, \varepsilon'[p \leftarrow v_a] \rangle}$$

Letrec

$$\overline{\langle Letrec(f, def, body), \varepsilon \rangle \rightarrow \langle Let([(f, App(Z, Fun(f, def)))]), body), \varepsilon \rangle}$$

Condicional Booleano

Nuestro operador de condicional recibe 3 expresiones. La primera es la condición, si se cumple debemos devolver la segunda (la parte correspondiente a *then*), de lo contrario devolveremos la tercera expresión (la correspondiente a *else*). Para esta operación primero tenemos que tener un valor booleano en el primer argumento, pues de lo contrario no podemos determinar que camino tomar, por lo que reducimos la expresión en un paso hasta obtenerlo. Al conseguir dicho valor booleano, tenemos dos caminos, *Boolean(True)* y *Boolean(False)*, por lo que las otras dos reglas simplemente devuelven la expresión que corresponde a cada una.

$$\frac{\langle e_1, \varepsilon \rangle \rightarrow \langle e'_1, \varepsilon \rangle}{\langle If(e_1, e_2, e_3), \varepsilon \rangle \rightarrow \langle If(e'_1, e_2, e_3), \varepsilon \rangle}$$

$$\frac{}{\langle If(Boolean(True), e_2, e_3), \varepsilon \rangle \rightarrow \langle e_2, \varepsilon \rangle}$$

$$\frac{}{\langle If(Boolean(False), e_2, e_3), \varepsilon \rangle \rightarrow \langle e_3, \varepsilon \rangle}$$

Operaciones de listas

Como se vió en la eliminación de azúcar sintáctica (desugaring), las listas en el núcleo se ven como pares anidados, por lo que las operaciones como *Head* y *Tail* en realidad no son implementadas dentro del núcleo, más bien funcionan como *First* y *Second* respectivamente. Así las reglas que aplican para las listas son en realidad las reglas que definimos para los pares ordenados.

Condicional if0

Como se vio en la eliminación de azúcar sintáctica, el constructo *if0* no se encuentra definido directamente en el núcleo del lenguaje. En la superficie, *if0* recibe tres expresiones —una condición, una rama *then* y una rama *else*—, con el propósito de evaluar la primera y verificar si su valor es cero. Sin embargo, esto no requiere una nueva regla semántica, pues el comportamiento de *if0* se obtiene al traducirlo al condicional *if* ya definido.

Durante la desazucarización, el término *if0 c t e* se transforma en una comparación mediante el predicado de igualdad con cero, es decir, en *if (Eq c 0) t e*. Después de esta transformación, toda la evaluación recae en las reglas ya definidas para el condicional *if*, por lo que no es necesario incluir reglas adicionales en la semántica de paso pequeño.

Así, el constructo *if0* se reconoce como azúcar sintáctica del condicional base, y su comportamiento queda completamente cubierto por las reglas semánticas del *if*.

Condicional por clausulado

Como se vio en la eliminación de azúcar sintáctica, el constructo *cond* no existe directamente en el núcleo. En la superficie, permite definir múltiples guardas booleanas y una cláusula final, que comúnmente actúa como el caso por defecto. Sin embargo, en el núcleo todo se traduce a una serie de *if* anidados, evaluados en orden hasta que una condición resulta verdadera. De esta forma, un *cond* con varias cláusulas se desazucariza completamente a una cadena de condicionales *if*, donde cada condición y su expresión asociada se evalúan secuencialmente hasta encontrar la primera que se cumple.

Funciones anónimas (Lambda) y Aplicación

En el núcleo, las funciones anónimas (*Lambda*) son expresiones que capturan el ambiente de la evaluación en el momento en que se crean, generando una *cerradura*(*Closure*).

Esto garantiza un **alcance estático**, el cual garantiza que todas las variables libres dentro del cuerpo

c se resuelvan en el entorno donde se definió la función.

$$\frac{}{\langle \text{Lambda}(p, c), \varepsilon \rangle \rightarrow \langle \text{Closure}(\text{Lambda}(p, c), \varepsilon), \varepsilon \rangle}$$

En $\text{App}(f, a)$ la aplicación de una función a un argumento. La evaluación sigue:

Evaluamos primero la expresión f , lo cual asegura que la función sea una cerradura lista para aplicarse.

$$\frac{\langle f, \varepsilon \rangle \rightarrow \langle f', \varepsilon \rangle}{\langle \text{App}(f, a), \varepsilon \rangle \rightarrow \langle \text{App}(f', a), \varepsilon \rangle}$$

Luego evaluamos el argumento a para obtener un valor canónico a' , donde solo procede cuando el argumento es un valor, así respetando el orden de la evaluación

$$\frac{\langle a, \varepsilon \rangle \rightarrow \langle a', \varepsilon \rangle}{\langle \text{App}(\text{Closure}(\text{Lambda}(p, c), \varepsilon'), a), \varepsilon \rangle \rightarrow \langle \text{App}(\text{Closure}(\text{Lambda}(p, c), \varepsilon'), a'), \varepsilon \rangle}$$

Cuando el argumento y la función son valores, aplicamos la cerradura. El parámetro p se asocia con el argumento v_a dentro del entorno capturado ε' . Esto garantiza que cualquier referencia a variables libres en c se resuelva según el entorno donde se definió.

$$\frac{v_a \text{ es canónico}}{\langle \text{App}(\text{Closure}(\text{Lambda}(p, c), \varepsilon'), v_a), \varepsilon \rangle \rightarrow \langle c, \varepsilon'[p \leftarrow v_a] \rangle}$$

Asignaciones locales: **let**, **let*** y **letrec**

Como se vio en la eliminación de azúcar sintáctica, los constructos **let**, **let*** y **letrec** no existen directamente en el núcleo. En la superficie, permiten definir variables locales y establecer valores dentro de un bloque de código limitado en alcance. Sin embargo, en el núcleo, todos estos constructos se traducen a expresiones equivalentes formadas únicamente por funciones (**lambda**) y aplicaciones (**App**), que son las verdaderas estructuras primitivas del lenguaje.

En el caso de **let**, las asignaciones se evalúan de forma simultánea en el ambiente actual, sin que las variables dependan unas de otras. Su desazucarización genera una secuencia de funciones anidadas que reciben los valores asociados y los aplican en conjunto, representando el mismo comportamiento con las reglas semánticas básicas del núcleo.

Por su parte, **let*** evalúa las asignaciones de manera secuencial, permitiendo que cada variable pueda depender de las definidas en pasos anteriores. Su traducción al núcleo también se realiza mediante funciones anidadas, pero en un orden que refleja esta dependencia progresiva.

Finalmente, **letrec** permite la definición de funciones recursivas, es decir, funciones que pueden referirse a sí mismas dentro de su propio cuerpo. Para expresar este comportamiento en el núcleo, se emplea el combinador **Z**, que permite capturar la recursión sin necesidad de nombres explícitos. Así, un **letrec** se traduce internamente a un **let** donde el valor de la función se obtiene aplicando dicho combinador.

De esta manera, la semántica de estos tres constructos no requiere reglas adicionales en la semántica de paso pequeño, ya que su comportamiento se deriva completamente de las reglas ya definidas para funciones y aplicaciones en el núcleo del lenguaje.

Capítulo 3

Justificaciones

3.1. Elecciones de notación

La formalización de MiniLISP requiere de notaciones precisas para cada componente del lenguaje. Esta sección documenta las convenciones adoptadas y su justificación técnica, para mantener una precisión formal, claridad expositiva y consistencia con la tradición académica en diseño de lenguajes de programación.

3.1.1. Sistema de notación para la sintaxis abstracta

- **ASA con constructores explícitos:** Utilizamos notación funcional (por ejemplo: `Add(e_1, e_2)`) en lugar de notación infija (por ejemplo: `e_1 + e_2`) para reflejar fielmente la estructura arbórea y facilitar la implementación en Haskell, siguiendo convenciones establecidas en la implementación de lenguajes funcionales. [6]
- **Currificación sistemática:** Todas las funciones de múltiples parámetros se representan como aplicaciones sucesivas de funciones unarias, principio fundamental del cálculo lambda que influyó originalmente a LISP. [7]
- **Valores canónicos:** Distinguimos explícitamente entre expresiones arbitrarias y valores canónicos (`Num(n)`, `Boolean(b)`, `nil`) en las reglas semánticas para precisar los puntos donde la evaluación se detiene.

3.1.2. Convenciones para semántica operacional

- **Relación de transición:** Empleamos $\langle e, \varepsilon \rangle \rightarrow \langle e', \varepsilon' \rangle$ donde el ambiente ε se incluye explícitamente para manejo de estado, aproximación estándar en semántica operacional estructural. [4]
- **Reglas de inferencia:** Utilizamos el formato estándar de reglas de inferencia con premisas y conclusión, ampliamente aceptado en la comunidad de semántica formal. [5]
- **Manejo de errores explícito:** Incluimos reglas específicas para propagación de errores (`Error`) en el lugar de asumir terminación anómala silenciosa.

3.1.3. Decisiones en la gramática concreta

- **EBNF sobre BNF:** Seleccionamos la notación EBNF extendida [3] por su mayor expresividad para repeticiones y opcionalidad, manteniendo compatibilidad con herramientas modernas de procesamiento de lenguajes.

- **Tokens distinguibles:** Utilizamos `#t` y `#f` para booleanos, evitando ambigüedad con identificadores y manteniendo coherencia con la familia de lenguajes Lisp. [1]
- **Prefijación consistente:** Mantenemos la notación prefija `(+ 1 2)` para todas las operaciones, fiel al espíritu LISP original [7], aun cuando difiere de la notación matemática convencional. [7]

3.1.4. Representación de estructuras de datos

- **Listas como construcciones anidadas:** Representamos listas como `Cons(e, l)` en lugar de notación de puntos `(e . l)` para enfatizar la naturaleza recursiva y facilitar la implementación en Haskell.
- **Pares explícitos:** Utilizamos `Pair(f, s)` en lugar de tuplas implícitas para mantener la distinción entre pares ordenados y aplicaciones de funciones.
- **Clausuras con ambiente:** Representamos las clausuras como `Closure(Lambda(p, c), ε)` para hacer explícito el ambiente léxico capturado, esencial para la semántica de funciones de primera clase. [4]

3.1.5. Enfoque educativo

Estas elecciones buscan facilitar la transición entre la especificación formal y la implementación práctica en Haskell, manteniendo al mismo tiempo el rigor necesario para el análisis formal. La consistencia con la literatura establecida en teoría de lenguajes de programación [5, 4] asegura que el trabajo sea accesible para investigadores familiarizados con semántica formal, mientras que la claridad en la exposición lo hace adecuado para fines educativos en cursos de lenguajes de programación. [2]

Capítulo 4

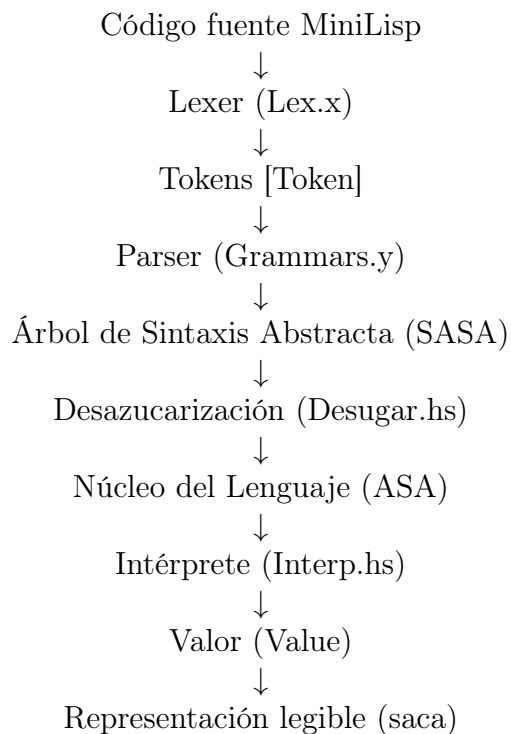
Resultados

4.1. Descripción de la implementación

4.1.1. Arquitectura general

Nuestro intérprete de MiniLisp implementa: `lexer` \rightarrow `parser` \rightarrow desazucarización \rightarrow intérprete.

Flujo de ejecución



Componentes principales

Lexer (Lex.x): El lexer se encarga del análisis léxico, transformando la cadena de entrada en una secuencia de tokens. Es ahí donde se analizan los identificadores, números, booleanos, operadores y palabras reservadas. Y lo generamos utilizando la herramienta Alex.

Parser (Grammars.y): Por otro lado, el parser se encarga de implementar el análisis sintáctico. Construye el árbol de sintaxis abstracta de superficie (SASA) a partir de los tokens. Este lo generamos usando la herramienta Happy.

Desazucarización (Desugar.hs): Transforma construcciones azucaradas de la sintaxis de superficie a primitivas del núcleo:

- Operadores variádicos se convierten a aplicaciones anidadas binarias
- `let` se desazucariza a lambdas currificadas
- `let*` se desazucariza de forma secuencial
- `letrec` se desazucariza usando el combinador `Z`
- Condicionales `if0` y `cond` se transforman a `if`
- Listas se representan como pares anidados

Intérprete (Interp.hs): Implementa la evaluación del lenguaje núcleo usando cerraduras (closures) para implementar funciones de orden superior con alcance estático. Mantiene un ambiente (env) que asocia identificadores con valores.

REPL (MiniLisp.hs): Proporciona una interfaz interactiva con menú principal, comandos de ayuda (`:help`, `:ejemplos`, `:test`), y evaluación de expresiones.

4.1.2. Decisiones de diseño

Combinador `Z`

Se implementó el combinador `Z` :

$$Z = \lambda f.(\lambda x.f(\lambda y.(x\ x)\ y))(\lambda x.f(\lambda y.(x\ x)\ y))$$

`(lambda y. ((x x) y))` retrasa la aplicación, evitando divergencia infinita. Esto era muy importante para que `letrec` pueda funcionar correctamente.

Operadores como variádicos en superficie, binarios en núcleo

La sintaxis de superficie permite operadores con aridad $n \geq 2$:

$$(+\ 1\ 2\ 3\ 4\ 5) \quad \text{y} \quad (+\ 1\ 2)$$

En el núcleo, se desazucarizan a aplicaciones binarias anidadas:

$$(+\ 1\ (+\ 2\ (+\ 3\ (+\ 4\ 5))))$$

Listas como pares anidados

Las listas en sintaxis de superficie:

$$[1, 2, 3, 4, 5]$$

Se desazucarizan a la representación de pares:

$$\text{Pair}(1, \text{Pair}(2, \text{Pair}(3, \text{Pair}(4, \text{Pair}(5, \text{Nil}))))))$$

Esto permite reutilizar la semántica de pares sin primitivas especiales de listas.

Alcance estático mediante cerraduras

El `interp` implementa cerraduras (closures) que capturan el ambiente léxico en el momento de la definición de la función.

Ejemplo:

```
(let ((x 10))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 3))))
```

Esto nos regresa 13 ya que usa $x = 10$ del ambiente de f , no $x = 5$.

4.2. Casos de prueba

El proyecto incluye 5 casos de prueba.

4.2.1. Test 1: Suma de primeros N naturales

Descripción

Calcula la suma de los primeros n números naturales usando recursión:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Expresión:

```
(letrec (sum (lambda (n)
               (if (= n 0)
                   0
                   (+ n (sum (- n 1))))))
  (sum 10))
```

Explicación

- **letrec:** Define la función recursiva `sum`
- **Caso base:** Si $n = 0$, retorna 0
- **Caso recursivo:** Suma n más `(sum (- n 1))`
- **Evaluación:** $(sum\ 10) = 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0 = 55$

Resultado esperado: 55

4.2.2. Test 2: Factorial

Descripción

Calcula el factorial: $n! = n \times (n-1) \times \dots \times 2 \times 1$.

Expresión:

```
(letrec (fact (lambda (n)
  (if (= n 0)
      1
      (* n (fact (- n 1))))))
(fact 5))
```

Explicación

- **Caso base:** $0! = 1$
- **Caso recursivo:** $n! = n \times (n - 1)!$
- **Evaluación:** $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Resultado esperado: 120**4.2.3. Test 3: Fibonacci****Descripción**

Calcula el n -ésimo número de Fibonacci: $F(n) = F(n - 1) + F(n - 2)$ con $F(0) = 0, F(1) = 1$.

Expresión:

```
(letrec (fib (lambda (n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2))))))
(fib 10))
```

Explicación

- **Caso base:** Si $n < 2$, retorna n
- **Caso recursivo:** Suma de los dos números Fibonacci anteriores
- **Evaluación:** $F(10) = 55$ (secuencia: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55)

Resultado esperado: 55**4.2.4. Test 4: Map (Aplicar función a elementos de lista)****Descripción**

Implementa la función `map` que aplica una función a cada elemento de una lista, retornando la lista transformada.

Expresión:

```
(letrec (map (lambda (f lst)
              (if (null? lst)
                  empty
                  (pair (f (head lst))
                        (map f (tail lst))))))
  (map (lambda (x) (* x 2)) [1, 2, 3, 4, 5]))
```

Explicación

- **Caso base:** Si la lista está vacía, retorna `empty`
- **Caso recursivo:**
 - Aplica la función f al primer elemento: `(f (head lst))`
 - Recursivamente mapea sobre el resto: `(map f (tail lst))`
 - Construye el par: `(pair)`
- **Evaluación:** Duplica cada elemento de la lista $[1, 2, 3, 4, 5] \rightarrow [2, 4, 6, 8, 10]$

Resultado esperado: `[2, 4, 6, 8, 10]`

4.2.5. Test 5: Filter (Filtrar elementos de lista)**Descripción**

Implementa la función `filter` que retiene solo los elementos que satisfacen un predicado.

Expresión:

```
(letrec (filter (lambda (pred lst)
                  (if (null? lst)
                      empty
                      (if (pred (head lst))
                          (pair (head lst)
                                (filter pred (tail lst)))
                          (filter pred (tail lst)))))
  (filter (lambda (x) (> x 3)) [1, 2, 3, 4, 5, 6]))
```

Explicación

- **Caso base:** Si la lista está vacía, retorna `empty`
- **Caso recursivo:**
 - Si el predicado es verdadero para `(head lst)`, incluye el elemento
 - Si no, descarta el elemento
 - Recursivamente filtra el resto de la lista
- **Evaluación:** Retiene solo elementos mayores que 3: $[1, 2, 3, 4, 5, 6] \rightarrow [4, 5, 6]$

Resultado esperado: [4, 5, 6]

4.3. Ejemplos de ejecución

4.3.1. REPL

Menú principal

Al compilar y ejecutar el proyecto, la terminal muestra:

```
=====
||      Proyecto 01 - MiniLisp      ||
||  Lenguajes de Programación 2026-1  ||
||  Universidad Nacional Autónoma de México  ||
||      Facultad de Ciencias      ||
||                                  ||
||      Desarrollado por:          ||
||  => Lorena González Téllez      ||
||  => Leslie Renée López Rodríguez ||
||  => Bruno Sebastián Marentes Mosqueda ||
=====
```

Escribe expresiones de MiniLisp o comandos:

```
(exit)    - Salir del intérprete
:help     - Mostrar ayuda
:ejemplos - Mostrar ejemplos
:test     - Ejecutar funciones de prueba
:clear    - Limpiar pantalla
```

===== MENÚ PRINCIPAL =====

1. Iniciar REPL interactivo
2. Ejecutar tests de las 5 funciones de prueba
3. Ver ayuda
4. Ver ejemplos
5. Salir

Selecciona una opción (1-5):

Ejecución de tests (opción 2)

Al seleccionar la opción 2, se ejecutan automáticamente los 5 casos de prueba:

EJECUTANDO TESTS

Test 1: Suma de primeros N naturales

```
Expresión: (letrec (sum (lambda (n) (if (= n 0) 0 (+ n (sum (- n 1)))))...)
CORRECTO: 55
```

Test 2: Factorial

Expresión: (letrec (fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))...
CORRECTO: 120

Test 3: Fibonacci

Expresión: (letrec (fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib...
CORRECTO: 55

Test 4: Map para listas

Expresión: (letrec (map (lambda (f lst) (if (null? lst) empty (pair (f ...
CORRECTO: [2, 4, 6, 8, 10]

Test 5: Filter para listas

Expresión: (letrec (filter (lambda (pred lst) (if (null? lst) empty (if...
CORRECTO: [4, 5, 6]

REPL interactivo (opción 1)**Ejemplo 1: Expresiones aritméticas**

MiniLisp> (+ 1 2 3 4 5)
=> 15

MiniLisp> (* 2 3 4)
=> 24

MiniLisp> (expt 2 10)
=> 1024

Ejemplo 2: Operadores lógicos

MiniLisp> (and #t #t #f)
=> #f

MiniLisp> (or #f #f #t)
=> #t

MiniLisp> (not #t)
=> #f

Ejemplo 3: Listas

MiniLisp> [1, 2, 3, 4, 5]
=> [1, 2, 3, 4, 5]

MiniLisp> (head [10, 20, 30])
=> 10

```
MiniLisp> (tail [10, 20, 30])  
=> [20, 30]
```

```
MiniLisp> (null? empty)  
=> #t
```

```
MiniLisp> (null? [1, 2])  
=> #f
```

Ejemplo 4: Funciones de orden superior

```
MiniLisp> ((lambda (x y) (+ x y)) 5 7)  
=> 12
```

```
MiniLisp> ((lambda (f) (f 10)) (lambda (x) (* x 2)))  
=> 20
```

Ejemplo 5: Let variádico con múltiples bindings

```
MiniLisp> (let ((x 10) (y 20) (z 30)) (+ x y z))  
=> 60
```

```
MiniLisp> (let ((f (lambda (x) (* x 2)))) (f 5))  
=> 10
```

Ejemplo 6: Let* secuencial

```
MiniLisp> (let* ((x 5) (y (+ x 3)) (z (+ y 2))) (+ x y z))  
=> 23
```

Explicación:

- $x = 5$
- $y = 5 + 3 = 8$
- $z = 8 + 2 = 10$
- Resultado: $5 + 8 + 10 = 23$

Ejemplo 7: Condicionales

```
MiniLisp> (if (> 5 3) (* 2 10) (* 3 5))  
=> 20
```

```
MiniLisp> (if0 (- 5 5) 100 200)  
=> 100
```

```
MiniLisp> (cond [(< 1 2) 10] [(= 2 2) 20] [else 30])  
=> 10
```

4.4. Análisis de resultados

4.4.1. Tests

Vimos que el proyecto pasa perfectamente los 5 test:

- **Recursión simple:** Suma de naturales y factorial funcionan correctamente
- **Recursión múltiple:** Fibonacci con múltiples llamadas recursivas se evalúa correctamente
- **Funciones de orden superior:** Map y filter demuestran la capacidad de pasar funciones como argumentos
- **Combinador Z:** Las funciones recursivas usando letrec se comportan correctamente con evaluación estricta

4.4.2. Desazucarización

La desazucarización que implementamos también cumple con lo que debería de hacer.

- **Operadores variádicos:** Se transforman correctamente a operaciones binarias anidadas
- **Let/Let*:** Se desazucarizan a lambdas currificadas manteniendo la semántica correcta
- **Letrec:** El combinador Z permite recursión sin divergencia infinita
- **Condicionales:** If0 y cond se transforman correctamente a if

4.4.3. Alcance estático

El intérprete implementa correctamente alcance estático mediante cerraduras:

```
(let ((x 10))
  (let ((f (lambda (y) (+ x y))))
    (let ((x 5))
      (f 3))))
```

=> 13

La función `f` nos regreso 13 y no 8, lo que confirma que utiliza $x = 10$ y no $x = 5$ del ambiente local.

Capítulo 5

Conclusiones

El desarrollo del presente proyecto ha culminado en la exitosa formalización e implementación de una extensión del lenguaje `MiniLISP`, cumpliendo así con los objetivos teóricos y prácticos planteados. El trabajo realizado no solo representa la materialización de un intérprete funcional, sino que también constituye una demostración rigurosa del puente que conecta la teoría de los lenguajes de programación con su aplicación concreta.

La primera fase del proyecto se centró en la formalización del lenguaje, estableciendo una base teórica sólida. Se definió de manera precisa la sintaxis concreta mediante reglas léxicas y una gramática libre de contexto en `EBNF`, lo que permitió especificar sin ambigüedades la estructura superficial del lenguaje. Posteriormente, se diseñó la sintaxis abstracta (`ASA`), que captura la esencia estructural de los programas y sirve como la representación interna para el intérprete. Un componente crucial de esta fase fue la especificación de las reglas de eliminación de azúcar sintáctica, las cuales permitieron reducir un lenguaje de superficie a un núcleo minimalista compuesto por un conjunto reducido de construcciones primitivas. Este proceso de desazucarización no solo simplificó drásticamente la implementación del evaluador, sino que también facilitó el razonamiento formal sobre la semántica del lenguaje.

La segunda fase se dedicó a la semántica operacional estructural (paso pequeño), donde se especificaron las reglas de reducción que definen el comportamiento computacional de cada construcción del núcleo. La adopción de un modelo de evaluación con alcance estático, implementado a través de cerraduras (closures) que capturan el ambiente léxico, garantizó un comportamiento predecible y consistente. La inclusión del combinador de punto fijo `Z` como un primitivo en el ambiente inicial fue una decisión de diseño clave que permitió la implementación de funciones recursivas de manera elegante y formalmente correcta, sin necesidad de extender el núcleo del lenguaje.

Finalmente, la implementación del intérprete en `Haskell`, utilizando las herramientas `Alex` y `Happy` para el análisis léxico y sintáctico, validó la coherencia y corrección de nuestro modelo teórico. Los resultados obtenidos en los casos de prueba, que abarcaron desde algoritmos recursivos clásicos (`factorial`, `Fibonacci`) hasta funciones de orden superior (`map`, `filter`), demostraron que el sistema se comporta exactamente como lo predicen las reglas semánticas. La correcta evaluación de expresiones con alcance estático y la gestión de asignaciones locales (`let`, `let*` y `letrec`) confirmaron la robustez del diseño.

Estas limitaciones, sin embargo, no son fallos, sino fronteras que delinean el camino a seguir. El trabajo futuro lógico incluiría la implementación de dicho verificador de tipos estático para garantizar la seguridad de tipos antes de la ejecución, así como la optimización del evaluador, quizás transformándolo en un compilador a una máquina virtual de *bytecode*. Como reflexión final, el proyecto también es una conclusión sobre la idoneidad de la herramienta: `Haskell` demostró ser un vehículo extraordinariamente apto para este dominio. Sus tipos de datos algebraicos se mapean directamente con la `ASA`, y su *pattern matching* refleja de forma casi isomórfica las reglas de la semántica operacional. El proyecto, por tanto, también concluye que un lenguaje funcional tipado estáticamente es una herramienta ideal para prototipar, implementar y verificar la semántica de otros lenguajes. En retrospectiva, este proyecto

ha permitido consolidar una comprensión profunda de los conceptos fundamentales que subyacen a los lenguajes de programación: la distinción entre sintaxis y semántica, el poder de la abstracción a través del desazucaramiento, la mecánica de la evaluación mediante reglas de transición y la importancia del manejo de ambientes para garantizar un alcance correcto. La extensión de `MiniLISP`, más que un mero ejercicio de programación, ha sido una inmersión en el arte y la ciencia del diseño de lenguajes, demostrando que un conjunto pequeño de primitivas bien definidas es suficiente para construir un sistema computacionalmente completo.

Bibliografía

- [1] LispWorks Ltd. (s. f.). *Common Lisp HyperSpec — Section 1.1.2 History*. Recuperado de https://www.lispworks.com/documentation/HyperSpec/Body/01_ab.htm
- [2] Sebesta, R. W. (2015). *Concepts of Programming Languages* (11th ed.). Pearson Education. (p. 135)
- [3] ISO/IEC 14977:1996. (1996). *Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization.
- [4] Reynolds, J. C. (2009). *Theories of Programming Languages*. Cambridge University Press. (pp. 45-46)
- [5] Mitchell, J. C. (1996). *Foundations for Programming Languages*. MIT Press.
- [6] Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall.
- [7] McCarthy, J. (1960). *Recursive Functions of Symbolic Expressions and Their Computation by Machine*. Communications of the ACM. (pp. 8-9)
- [8] S. Krishnamurthi, *Programming Languages: Application and Interpretation*, 2023.
- [9] Plotkin, G. D. (1975). *Call-by-name, call-by-value and the lambda-calculus*. Theoretical Computer Science.
- [10] Simon Marlow and the Alex developers, *Alex: A tool for generating lexical analysers in Haskell*. 2022. Disponible en: <https://haskell-alex.readthedocs.io/>. Accedido: 2025-10-23.
- [11] Simon Marlow and the Happy developers, *Happy: A parser generator for Haskell*. 2022. Disponible en: <https://haskell-happy.readthedocs.io/>. Accedido: 2025-10-23.
- [12] M. Soto Romero, “Semántica Estática”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Especificación Formal de los Lenguajes de Programación, Facultad de Ciencias, UNAM, Semestre 2026-1.
- [13] M. Soto Romero, “Expresiones let”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Especificación Formal de los Lenguajes de Programación, Facultad de Ciencias, UNAM, Semestre 2026-1.
- [14] M. Soto Romero, “El Cálculo λ como núcleo para funciones”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Estilo Funcional, Facultad de Ciencias, UNAM, Semestre 2026-1.
- [15] M. Soto Romero, “Expresiones lambda”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Estilo Funcional, Facultad de Ciencias, UNAM, Semestre 2026-1.
- [16] M. Soto Romero, “Máquinas abstractas”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Estilo Funcional, Facultad de Ciencias, UNAM, Semestre 2026-1.

- [17] M. Soto Romero, “Ambientes de evaluación”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Estilo Funcional, Facultad de Ciencias, UNAM, Semestre 2026-1.
- [18] M. Soto Romero, “Alcance estático y dinámico”. *Notas de Clase de Lenguajes de Programación*, Unidad 2: Estilo Funcional, Facultad de Ciencias, UNAM, Semestre 2026-1.
- [19] Erick Daniel Arroyo Martínez, *Cálculo de Compiladores Correctos: Del Régimen Estricto al Perezoso*. Tesis de licenciatura en Ciencias de la Computación, Facultad de Ciencias, Universidad Nacional Autónoma de México, Ciudad Universitaria, CDMX, México, 2025. Tutor: M.C.I.C. Manuel Soto Romero.