Universidad Nacional Autónoma de México

Facultad de Ciencias

Lenguajes de Programación

Profesor:

Manuel Soto Romero

Ayudantes de Laboratorio:

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

Proyecto 1:

MiniLisp

Integrantes:

Trejo Maya Diego Alexander
Hernández Islas Leonardo Daniel
Carrillo Sánchez Rafael Esteban
Morales Martínez Edgar Jesús

5 de noviembre de 2025

Introducción

Motivación

Como parte del curso de **Lenguajes de Programación**, tras haber revisado la teoría necesaria para comprender la construcción de intérpretes y lenguajes formales, este proyecto busca aplicar de manera práctica los conocimientos adquiridos hasta el momento. El objetivo principal es construir un lenguaje de programación denominado **MiniLisp**, partiendo de su diseño teórico hasta su implementación funcional.

A través de este desarrollo se pretende fortalecer la comprensión de los conceptos de *sintaxis*, *semántica*, *azúcar sintáctica*, *bindings*, *evaluación ansiosa (call-by-value)* y su relación directa con la ejecución práctica del lenguaje en **Haskell**. De esta forma, se integran los aspectos teóricos con la experiencia práctica en el diseño e implementación de lenguajes de programación.

Objetivos específicos

1. Desarrollar la sintaxis léxica y libre de contexto del lenguaje **MiniLisp**, así como su representación en una sintaxis abstracta, incorporando la noción de azúcar sintáctica y su posterior eliminación para obtener un núcleo más simplificado del lenguaje.

2. Definir la semántica operacional estructural de **MiniLisp**, especificando tanto los modelos conceptuales basados en máquinas abstractas como sus implementaciones ejecutables en **Haskell**.

3. Construir formalmente los ambientes de evaluación y los *bindings* como mecanismos esenciales para garantizar la consistencia semántica y el correcto manejo de los alcances y valores dentro del lenguaje.

4. Integrar el régimen de evaluación ansiosa (*call-by-value*) mediante una estrategia de paso de parámetros por valor, analizando su influencia en la ejecución de los programas y en el diseño del intérprete.

5. Implementar el lenguaje **MiniLisp** de forma funcional en **Haskell**, mostrando la correspondencia explícita entre las reglas de formalización teórica y su aplicación práctica en la ejecución del lenguaje.

6. Mostrar, a través del proceso de eliminación del azúcar sintáctica, el tránsito desde la expresividad inicial del lenguaje hacia un núcleo reducido, fortaleciendo así la comprensión del vínculo entre la teoría y la práctica en el diseño de lenguajes de programación.

Delimitación del proyecto

Formalización

Sintaxis Concreta

Para comenzar a diseñar nuestro lenguaje, primero debemos partir de definir su sintaxis concreta.

La sintaxis concreta de un lenguaje de programación se refiere a la estructura que tiene y que define exactamente cómo se deben escribir los programas en este [10].

La sintaxis concreta se puede definir como un par (L, G) donde [10]:

- L es la definición léxica representada por el conjunto de expresiones regulares R.
- G es la gramática libre de contexto (N, Σ, P, S).

Sintaxis Léxica

Para definir nuestra sintaxis léxica de manera formal nos vamos a apoyar en la teoría de autómatas y lenguajes formales, más específicamente en el uso de expresiones regulares. Para esto **recapitularemos** qué es formalmente un lenguaje y a qué se refiere el término *token*.

Un **lenguaje** se puede definir como un subconjunto de Σ^*, donde Σ es un alfabeto, mientras que un *token* es un símbolo abstracto que representa un tipo de unidad léxica. Esto a su vez es el lexema dentro de nuestro lenguaje de programación, es decir, secuencias de caracteres mínimas con algún significado dentro de este, que puede ser representado por expresiones regulares [10].

Una expresión regular es una forma de notación para definir un lenguaje. Podemos definir las expresiones regulares formalmente de la siguiente manera:

Constantes y variables que denotan los lenguajes y tres operadores de operaciones (unión $+$, punto $\cdot$ y estrella de Kleene *). Describimos las expresiones regulares recursivamente, pues con esta definición no sólo describimos qué son, sino también para cada una de las expresiones E el lenguaje que representa, denotado como $L(E)$ puesto que E, estrictamente hablando, es sólo una expresión y no un lenguaje, tenemos que denotarlo por aparte como $L(E)$.

Casos base: La base consiste en tres partes:

1. La constante ε y $\emptyset$ son expresiones regulares y denotan el lenguaje entre $\{\varepsilon\}$ y $\emptyset$, respectivamente. Esto es $L(\varepsilon) = \{\varepsilon\}$ y $L(\emptyset) = \emptyset$.

2. Si a es un símbolo, entonces a es una expresión regular. Esta expresión denota el lenguaje $L(a) = \{a\}$, donde a se refiere a sí misma.

3. Una variable, normalmente en mayúsculas y en itálica L, es una variable que representa cualquier lenguaje.

Inducción: Hay cuatro partes de pasos inductivos, uno por cada uno de los operadores y uno por la introducción a los paréntesis:

1. Si E y F son expresiones regulares, entonces $E+F$ es una expresión regular que denota la unión de los lenguajes $L(E)$ y $L(F)$. Esto es $L(E + F) = L(E) \cup L(F)$.

2. Si E y F son expresiones regulares, entonces EF es una expresión regular que denota la concatenación de los lenguajes $L(E)$ y $L(F)$. Esto es $L(EF) = L(E)L(F)$.

3. Si E es una expresión regular, entonces E^* es una expresión regular, denotando la cerradura del lenguaje $L(E)$. Que es $L(E^*) = (L(E))^*$.

4. Si E es una expresión regular, entonces (E), E entre paréntesis, es una expresión regular que denota el mismo lenguaje E. Formalmente: $L((E)) = L(E)$.

[2]

Sintaxis Léxica de MiniLisp

Una vez definida qué es la sintaxis léxica formalmente, vamos a utilizarla para definir nuestra sintaxis léxica dentro de nuestro lenguaje de programación **MiniLisp**. Empezaremos por definir nuestro alfabeto Σ, que se va a conformar por nuestros símbolos:

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, +, (,), *, a...z, A...Z\}$$

Donde $a...z$ se refiere al alfabeto castellano en minúsculas y $A...Z$ al alfabeto castellano en mayúsculas.

Con este alfabeto nos basta para poder representar la sintaxis léxica de la base otorgada para nuestro lenguaje **MiniLisp**. Así, pasaremos a representar nuestros *tokens* dentro del lenguaje con expresiones regulares, las cuales tendrán un identificador del lado derecho que nos servirá para darnos una idea de lo que representan, seguidos por una flecha y con la expresión regular correspondiente hasta el lado derecho.

letra → A | B | . . . | Z | a | b | . . . | z

dígito → 0 | 1 | . . . | 9

Z → 1 | . . . | 9

id → *letra* (*letra* | *dígito*)*

dígitos → Z *dígito** + - Z *dígito**

fracción → . *dígito** | ε

número → *dígitos fracción*

if → `if`

then → `then`

else → `else`

[3]

Implementación del lexer usando ALEX

Una vez formalizado en papel nuestro analizador léxico, podemos empezar a crear nuestro lexer en forma de código. Para esto se nos habilitó el uso de una herramienta llamada **ALEX**.

ALEX es una herramienta que ayuda a generar analizadores léxicos (*lexers* o *scanners*) en **Haskell**. El analizador léxico implementa una descripción de los tokens para ser reconocidos como expresiones regulares [5].

Podemos instalar ALEX como instalamos cualquier otro paquete en Haskell, esto es:

```
cabal install alex
```

o

```
stack install alex
```

mediante `cabal` y `stack` respectivamente.

Dentro de nuestro archivo `.x`, las llaves `{}` sirven para insertar **"scrap code"**, donde básicamente indicamos que queremos poner código en Haskell. El scrap que ponemos hasta el inicio del archivo por lo general se usa para indicar el módulo de nuestro archivo Haskell, por lo que lo vamos a estar usando mucho dentro de nuestro código.

El comando `%wrapper "basic"` controla cómo se llama la función principal del escáner, qué tipo de entrada consume y qué tipo de salida devuelve, esto es así porque cuando ALEX genera el analizador léxico, no solo produce la tabla de transiciones y las funciones de escaneo, sino que también necesita envolver el código desde una interfaz que lo haga usable desde Haskell.

El **"wrapper" básico** (el que estamos utilizando) genera un escáner que toma como entrada un *string* de Haskell, lo "tokeniza" y produce como salida una lista de tokens.
Usamos `$digito = 0-9` y `$alfabeto = [a-zA-Z]` para indicar los dígitos y el alfabeto que queremos manejar en nuestro lexer. Cabe mencionar que estas son llamadas "variables de patrón" y se basan en la notación de las expresiones regulares que definimos formalmente, de igual manera al poner un + al final de estas nos referimos que esto es válido si se reconoce una o más expresiones con la misma forma.

Con código hexadecimal definimos otra variable patrón para los espacios en blanco más comunes que queremos que se eviten cuando leamos nuestra cadena, esto es: `$white = [\x20\x09\x0A\x0D\x0C\x0B]` para `\x20 = ' '` (space), `\x09 = tab`, `\x0A = LF`, `\x0D = CR`, `\x0C = FF`, `\x0B = VT`.

La línea `tokens :-` finaliza la definición de las macros y empieza la definición del scanner.

Ahora pasamos a especificar las definiciones de nuestros tokens donde usamos el formato `regexp {code}` donde regexp indica que si `<regexp>` (patrones) coincide con la entrada entonces devuelve `<code>` (acción Haskell) que a su vez puede ser reemplazado por `;` indicando que el token de la entrada de caracteres se debe ignorar.

Como ya tenemos nuestras variables patrón definidas podemos usarlas para empezar a indicar nuestros tokens, los espacios en blanco los ignoramos y empezamos a definir el nombre de cada uno de nuestros tokens. En la parte `regexp` cuando utilizamos \seguido de un carácter queremos indicar que este no es un metacaracter especial sin embargo cuando lo usamos dentro del lado derecho `{code}` ya sea como `_ ->Token o \s ->Token` la primera forma indica una lambda dentro de Haskell que usa _ como comodín para el argumento indicando que no le importa el valor mientras que la segunda usa s para indicar un argumento llamado s que es una subcadena reconocida dentro del patrón, en nuestro caso al argumento le llamaremos `lexema`, esto nos servirá para ahorrarnos la definición de cada uno de nuestros símbolos pues ya tendrán su identificador y no tendremos que definir manualmente uno por uno.

Para agregar **comentarios** a nuestro lenguaje utilizaremos el símbolo `~~` pues la doble virgulilla no nos parece muy usada y queríamos darle algo de personalidad a nuestro lenguaje. Para indicar que todo lo que se encuentre al lado derecho de nuestro comentario sin importar el carácter o si tiene cero o más repeticiones lo ignore, ponemos el comando `"~.*"` ; donde `.` indica que no importa el carácter y `*` hace la función de la estrella de Kleene en las expresiones regulares.

Al final del documento creamos un apartado entre llaves `{...}` para declarar el tipo de los tokens y crear una función `main` para poder probarlos. Alex nos provee de una función integrada para invocar nuestro escáner: `alexScanTokens :: String ->[Token]`.

Para evitar extendernos innecesariamente en el documento, los tokens que extienden el lenguaje `MiniLisp` que se definen posteriormente en el documento omitiremos escribirlos textualmente, sin embargo, todos siguen la misma estructura que definimos anteriormente, por lo que no habría un gran cambio en el archivo más que la adición de estos con el mismo formato. Adicionalmente el archivo se encuentra comentado en cada una de sus partes.

Para ejecutar el archivo individualmente y comprobar que la tokenización funciona tenemos que escribir la siguiente línea:

```
alex Lexer.x
```

Si se generó el archivo `.hs` sin ningún error podemos pasar ahora sí a probarlo en nuestro intérprete con el siguiente comando:

```
ghci Lexer.hs
```

Para probarlo tenemos que escribir `lexer` y entre comillas la cadena que queramos tokenizar. Ejemplo:

```
lexer "1+2"
```

Que nos generará la siguiente salida:

[TokenNum 1, TokenSuma, TokenNum 2]

Sintaxis Libre de Contexto de *MiniLisp*

Esta se refiere a la estructura de un **LDP** en la que las reglas de formación de sus secuencias se pueden escribir mediante una gramática libre de contexto o **GLC** por sus siglas en inglés.

Gramática formal:

Una gramática formal se define formalmente como [9]:
Un 4-dúplo **G = (N, T, P, S)**, donde:

- N es un conjunto finito de símbolos *no terminales*.
- T es un conjunto de *símbolos terminales* ($\mathrm{N} \cap \mathrm{T} = \emptyset$)
- P es un conjunto finito de *producciones*.
- S es el símbolo inicial tal que $\mathrm{S} \notin (\mathrm{N} \cup \mathrm{T})$. La restricción no es de uso generalizado, depende de las formas de reglas que se utilicen.

Las producciones en P son parejas ordenadas (α, β) con $\alpha = \gamma A \delta$ en la cual β, γ y δ son posiblemente vacías en $(N \cup T)^*$ y $A \in N$ o bien $A = S$. Denotamos a la pareja (α, β) como $\alpha \to \beta$.

Gramática libre de contexto:

Ahora, una gramática **G = (N, T, P, S)** es libre de contexto si sus producciones son de alguna de las siguientes formas [9]:

- S $\to \varepsilon$
- A $\to$ a

Con S y $A \in N$, S es el símbolo inicial y $|A| \leq |\alpha|$, $\alpha \in (N \cup T)^+$ y si la producción $S \to \varepsilon$ está en la gramática, S no aparece del lado derecho de ninguna producción.

Gramática en forma de Backus–Naur (BNF):

Para otorgar una manera clara y precisa de definir la sintaxis de los lenguajes de programación, alrededor del año 1960 se introdujo la notación **BNF (Backus–Naur Form)** para gramáticas libres de contexto [10]. Sus características principales son [11]:

- **Símbolos no terminales:** Se escriben entre ⟨⟩ (ejemplo: ⟨Expr⟩ o ⟨Ide⟩).
- **Reglas de producción:** Se representan igual que en la notación clásica pero se cambia el símbolo $\to$ por `::=`.
- **Uso del operador "|":** El operador "|" se utiliza para separar las distintas alternativas en las reglas de producción.

Gramática en forma EBNF (Extended Backus Naur Form):

Debido a las limitaciones de la forma BNF para escribir repeticiones y agrupaciones de forma compacta, a lo largo de varios años se hicieron extensiones a la forma BNF, lo que se convirtió en la forma extendida de Backus Naur, forma que se estandarizó en 1996 por la ISO/IEC 14977 [10].

Los componentes básicos de una gramática en forma EBNF son [10]:

- **Repetición:** Las secuencias que pueden repetirse cero o más veces se indican usando llaves `{ }`.
- **Opcionalidad:** Las secuencias opcionales se indican usando corchetes `[ ]`.

- **Agrupaciones:** Las agrupaciones de elementos se indican usando paréntesis `( )`.
- **Alternativas:** Las alternativas se heredan de BNF mediante el operador `|`.

Gramática a emplear en EBNF:

Para nuestro proyecto, se utilizará una gramática libre de contexto en forma EBNF, debido a que otorga una manera clara y precisa para nuestra sintaxis y a la vez permite repeticiones y agrupaciones de forma compacta.

Consideramos las siguientes especificaciones dadas:

- Debe haber paréntesis al inicio y al final de cada expresión.
- Se tiene notación prefija.

Gramática base:

```
Expr ::= Var
       | Int
       | Bool
       | ( + Expr Expr )
       | ( - Expr Expr )
       | ( not Expr )
       | ( let (Var Expr) Expr )
       | ( letrec (Var Expr) Expr )
       | ( if0 Expr Expr Expr )
       | ( lambda (Var) Expr )
       | ( Expr Expr)
Var ::= Identificador de variable
Int ::= Constante entera
Bool ::= #t | #f
```

Implementación de las extensiones:

Extensiones solicitadas:

1. **Operadores variádicos.** Los operadores actualmente binarios deben aceptar múltiples argumentos (*aridad* ≥ 2).

2. **Operadores aritméticos:** Multiplicación y división. Incorporar `*` y `/` con la misma aridad *variádica* anterior así como los operadores `add1` (de incremento) y `sub1` (de decremento), `sqrt` (raíz cuadrada) y `expt` (potencia).

3. **Predicados sobre enteros:** Igualdad y comparaciones. Incluir `=`, `<`, `>`, `>=`, `<=`, `!=` con soporte variádico.

4. **Pares ordenados y proyecciones:** Añadir la formación de pares `(e1, e2)` y las proyecciones: `(fst (1, #))`, `(snd (3, 5))`.

5. **Asignaciones locales:** `let` y `let*` variádicos. Añadir las construcciones `let` y `let*` que permitan realizar asignaciones locales con múltiples variables (*aridad* ≥ 1). En el caso de `let`, todas las asignaciones deben considerarse en paralelo, mientras que en `let*` deben evaluarse de manera secuencial.

6. **Condicional booleano:** Incorporar la forma `(if e1 e2 e3)`.

7. **Listas y operaciones básicas:** Extender con sintaxis de listas por corchetes y operaciones `head`/`tail`: `[1, 2, 3, 4]`, `(head [1, 2, 3, 4])`, `(tail [3, 5, 6])`, `[]`.

8. **Condicional por clausulado:** `cond` con rama `else`. Añadir la construcción `cond`, que permita escribir múltiples condiciones de forma ordenada. Cada cláusula debe tener una guarda booleana y una expresión asociada, evaluándose en orden hasta encontrar la primera verdadera. Además, se debe incluir una cláusula final `else`, cuya guarda se considera siempre verdadera. Especificar en EBNF la notación de las cláusulas.

9. **Funciones anónimas y aplicaciones:** La superficie del lenguaje debe incluir lambdas variádicas de la forma `(lambda (x1 ... ) Expr)`, permitiendo definir funciones con múltiples parámetros y su aplicación a argumentos.

Gramática en EBNF con las extensiones solicitadas:

```
<Expr> ::= <Var>
        | <Int>
        | <Bool>
        | ( + <Expr> <Expr> { <Expr> } )      ... suma con aridad  2
        | ( - <Expr> <Expr> { <Expr> } )      ... resta con aridad  2
        | ( * <Expr> <Expr> { <Expr> } )      ... multiplicación con aridad  2
        | ( / <Expr> <Expr> { <Expr> } )      ... división con aridad  2
        | ( add1 <Expr> )                     ... incremento de 1
        | ( sub1 <Expr> )                     ... decremento de 1
        | ( sqrt <Expr> )                     ... raíz cuadrada
        | ( expt <Expr> <Expr> )              ... potencia
        | ( not <Expr> )                      ... operación de negación
        | ( = <Expr> <Expr> { <Expr> } )      ... comparación de igualdad
        | ( < <Expr> <Expr> { <Expr> } )      ... comparación de menor
        | ( > <Expr> <Expr> { <Expr> } )      ... comparación de mayor
        | ( >= <Expr> <Expr> { <Expr> } )     ... comparación de mayor igual
        | ( <= <Expr> <Expr> { <Expr> } )     ... comparación de menor igual
        | ( != <Expr> <Expr> { <Expr> } )     ... comparación de no igual
        | ( <Expr> , <Expr> )                 ... pares ordenados
        | ( fst <Expr> )                      ... proyección primer elemento
        | ( snd <Expr> )                      ... proyección segundo elemento
        | ( let <BindLet> { <BindLet> } <Expr> )  ... asign. locales (paralelo)
        | ( let* <BindLet> { <BindLet> } <Expr> ) ... asign. locales (sec.)
        | ( letrec <BindLet> <Expr> )         ... let recursivo
        | ( if <Expr> <Expr> <Expr> )         ... condicional si-verdadero
        | ( if0 <Expr> <Expr> <Expr> )        ... condicional si-0
        | ( lambda ( <Var> { <Var> } ) <Expr> ) ... funciones anónimas
        | ( <Expr> <Expr> { <Expr> } )        ... aplicación de función
        | ``[`` ``]''                         ... lista vacía
        | ``[`` <Expr> { , <Expr> } ``]''     ... lista con elementos
        | ( head <Expr> )                     ... cabeza de lista
        | ( tail <Expr> )                     ... cola de lista
        | ( cond <Clause> { <Clause> } ``[`` <ElseClause> ``]'' )
                                           ... condicional por clausulado

<BindLet> ::= ( <Var> <Expr> )                ... binding variable
<Clause> ::= ``[`` <Expr> <Expr> ``]''         ... cláusula guarda booleana
<ElseClause> ::= ``[`` else <Expr> ``]''
<Var> ::= Identificador de variable
<Int> ::= Constante entera
<Bool> ::= #t | #f
```

Sintaxis Abstracta (ASA)

Definición formal

La sintaxis abstracta representa los constructores de programación mediante árboles donde cada nodo interior representa un operador y sus hijos, a su vez, representan los operandos[3]. A diferencia de los árboles de parsing, en la sintaxis abstracta los nodos interiores representan construcciones de programación en lugar de no terminales. Esta representación se define mediante tipos de datos algebraicos en Haskell, que capturan la estructura esencial del lenguaje.

Definición Azúcar Sintáctica

Decimos que azúcar sintáctica se refiere a aquellas construcciones que se traducen manera sistemática a construcciones más básicas del lenguaje durante el procesamiento [12].

Expresiones del SASA

A las expresiones de la sintaxis abstracta que son azúcar sintáctica de otras expresiones, les diremos SASA (Sugar Árbol Sintaxis Abstracta).

Expresiones del LASA

Para las expresiones con aridad $>= 2$ o que reciban varios parámetros o argumentos, se define el tipo LSASA (List Sugar Árbol Sintaxis Abstracta), representando una lista de SASA.

LSASA de la lista vacía

$$\frac{}{\texttt{nil}\ \text{LSASA}}$$

LSASA de la lista con elementos

$$\frac{x\ \text{SASA} \quad xs\ \text{LSASA}}{\texttt{(x:xs)}\ \text{LSASA}}$$

SASA de $\langle Var \rangle$

$$\frac{i : \text{String}}{\text{IDS}(i)\ \text{SASA}}$$

SASA de $\langle Int \rangle$

$$\frac{n \in \mathbb{Z}}{\text{NumS}(n)\ \text{SASA}}$$

SASA de $\langle Bool \rangle$

$$\frac{b \in \{\text{True}, \text{False}\}}{\text{BooleanS}(b)\ \text{SASA}}$$

SASA del vacío

$$\frac{}{\text{NilS SASA}}$$

SASA de un *binding* de una variable con un valor

$$\frac{i : \text{String} \quad v\ \text{SASA}}{\text{BindLet}(i, v)\ \text{SASA}}$$

SASA de una cláusula de una guarda booleana con una expresión

$$\frac{i\ \text{SASA} \quad d\ \text{SASA}}{\text{Clause}(i, d)\ \text{SASA}}$$

SASA de una cláusula *else*

$$\frac{e\ \text{SASA}}{\text{ElseClause}(e)\ \text{SASA}}$$

SASA de la suma variádica

$$\frac{as\ \text{LSASA}}{\text{AddS}(as)\ \text{SASA}}$$

SASA de la resta variádica

$$\frac{as\ \text{LSASA}}{\text{SubS}(as)\ \text{SASA}}$$

SASA de la multiplicación variádica

$$\frac{as\ \text{LSASA}}{\text{MultS}(as)\ \text{SASA}}$$

SASA de la división variádica

$$\frac{as \text{ LSASA}}{\text{DivS}(as) \text{ SASA}}$$

SASA de la potencia

$$\frac{b \text{ SASA} \quad e \text{ SASA}}{\text{Expt}(b, e) \text{ SASA}}$$

SASA del incremento de 1

$$\frac{e \text{ SASA}}{\text{Add1}(e) \text{ SASA}}$$

SASA del decremento de 1

$$\frac{e \text{ SASA}}{\text{Sub1}(e) \text{ SASA}}$$

SASA de la raíz cuadrada

$$\frac{e \text{ SASA}}{\text{SqrtS}(e) \text{ SASA}}$$

SASA de la operación de negación

$$\frac{e \text{ SASA}}{\text{NotS}(e) \text{ SASA}}$$

SASA de la comparación de igualdad variádica

$$\frac{as \text{ LSASA}}{\text{Equals}(as) \text{ SASA}}$$

SASA de la comparación de "menor que" de enteros variádica

$$\frac{as \text{ LSASA}}{\text{SmallerS}(as) \text{ SASA}}$$

SASA de la comparación de "mayor que" de enteros variádica

$$\frac{as \text{ LSASA}}{\text{BiggerS}(as) \text{ SASA}}$$

SASA de la comparación de "mayor igual" de enteros

$$\frac{as \text{ LSASA}}{\text{BigEq}(as) \text{ SASA}}$$

SASA de la comparación de "menor igual" de enteros

$$\frac{as \text{ LSASA}}{\text{SmallEq}(as) \text{ SASA}}$$

SASA de la comparación de "no igual" de enteros

$$\frac{as \text{ LSASA}}{\text{NotEq}(as) \text{ SASA}}$$

SASA de los pares

$$\frac{l \text{ SASA} \quad r \text{ SASA}}{\text{PairS}(l, r) \text{ SASA}}$$

SASA de la proyección del primer elemento

$$\frac{e \text{ SASA}}{\text{FstS}(e) \text{ SASA}}$$

SASA de la proyección del segundo elemento

$$\frac{e \text{ SASA}}{\text{SndS}(e) \text{ SASA}}$$

SASA de la asignación local con múltiples variables en paralelo

$$\frac{as \text{ LSASA} \quad e \text{ SASA}}{\text{LetS}(as, e) \text{ SASA}}$$

SASA de la asignación local con múltiples variables secuencial

$$\frac{as \text{ LSASA} \quad e \text{ SASA}}{\text{LetSeq}(as, e) \text{ SASA}}$$

SASA de la asignación de variables recursiva

$$\frac{bdlet \text{ SASA} \quad e \text{ SASA}}{\text{LetRec}(bdlet, e) \text{ SASA}}$$

SASA de la condicional "si verdadero entonces"

$$\frac{c \text{ SASA} \quad t \text{ SASA} \quad e \text{ SASA}}{\text{IfS}(c, t, e) \text{ SASA}}$$

SASA de la condicional "si 0 entonces"

$$\frac{c \text{ SASA} \quad t \text{ SASA} \quad e \text{ SASA}}{\text{If0}(c, t, e) \text{ SASA}}$$

SASA de las funciones anónimas con varios parámetros

$$\frac{p_i : \text{String} \quad i \in \mathbb{N} \quad c \text{ SASA}}{\text{LambdaS}(p_0, p_1, \ldots, p_n, c) \text{ SASA}}$$

SASA de las aplicaciones de función con varios argumentos

$$\frac{f \text{ SASA} \quad as \text{ LSASA}}{\text{AppS}(f, as) \text{ SASA}}$$

SASA de la lista con elementos

$$\frac{as \text{ LSASA}}{\text{List}(as) \text{ SASA}}$$

SASA de la cabeza de una lista

$$\frac{e \text{ SASA}}{\text{Head}(e) \text{ SASA}}$$

SASA de la cola de una lista

$$\frac{e \text{ SASA}}{\text{Tail}(e) \text{ SASA}}$$

SASA de condición por clausulado

$$\frac{as \text{ LSASA} \quad e \text{ SASA}}{\text{Cond}(as, e) \text{ SASA}}$$

Expresiones del ASA:

A las expresiones de la sintaxis abstracta que no sean azúcar sintáctica de otras expresiones (i.e. que pertenecen al lenguaje core o núcleo) les diremos ASA (Árbol Sintaxis

Abstracta).

ASA de $\langle Var \rangle$

$$\frac{i : \text{String}}{\text{ID}(i)\ \text{ASA}}$$

ASA de $\langle Int \rangle$

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

ASA de $\langle Bool \rangle$

$$\frac{b \in \{\text{True}, \text{False}\}}{\text{Boolean}(b)\ \text{ASA}}$$

ASA de la suma binaria

$$\frac{i\ \text{ASA} \quad d\ \text{ASA}}{\text{Add}(i, d)\ \text{ASA}}$$

ASA de la resta binaria

$$\frac{i\ \text{ASA} \quad d\ \text{ASA}}{\text{Sub}(i, d)\ \text{ASA}}$$

ASA de la multiplicación binaria

$$\frac{i\ \text{ASA} \quad d\ \text{ASA}}{\text{Mult}(i, d)\ \text{ASA}}$$

ASA de la división binaria

$$\frac{i\ \text{ASA} \quad d\ \text{ASA}}{\text{Div}(i, d)\ \text{ASA}}$$

ASA de la raíz cuadrada

$$\frac{e\ \text{ASA}}{\text{Sqrt}(e)\ \text{ASA}}$$

ASA de la potencia

$$\frac{b \text{ ASA} \quad e \text{ ASA}}{\text{Expt}(b, e) \text{ ASA}}$$

ASA de la operación de negación

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

ASA del vacío

$$\frac{}{\text{Nil ASA}}$$

ASA de la condicional "si verdadero entonces"

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

ASA de la igual de enteros binaria

$$\frac{i \text{ ASA} \quad d \text{ ASA}}{\text{Eq}(i, d) \text{ ASA}}$$

ASA de la comparación de "menor que" de enteros variádica

$$\frac{i \text{ ASA} \quad d \text{ ASA}}{\text{MenorQue}(i, d) \text{ ASA}}$$

ASA de la comparación de "mayor que" de ent.. variádica

$$\frac{i \text{ ASA} \quad d \text{ ASA}}{\text{MayorQue}(i, d) \text{ ASA}}$$

ASA de los pares

$$\frac{l \text{ ASA} \quad r \text{ ASA}}{\text{Pair}(l, r) \text{ ASA}}$$

ASA de la proyección del primer elemento

$$\frac{e \text{ ASA}}{\text{Fst}(e) \text{ ASA}}$$

ASA de la proyección del segundo elemento

$$\frac{e \text{ ASA}}{\text{Snd}(e) \text{ ASA}}$$

ASA de la función anónima

$$\frac{p : \text{String} \qquad c \text{ ASA}}{\text{Lambda}(p, c) \text{ ASA}}$$

ASA de la aplicación de función

$$\frac{f \text{ ASA} \qquad a \text{ ASA}}{\text{App}(f, a) \text{ ASA}}$$

Implementación de la gramática usando HAPPY

Una vez formalizado en papel nuestra gramática podemos empezar a utilizar Happy, que es la herramienta que se nos habilitó para empezar a implementar la gramática que queremos pasar en nuestro analizador **gramático**.

Happy es una herramienta que ayuda a generar la gramática que queremos analizar en nuestro analizador **gramático** generando un módulo compilable de Haskell. El analizador **gramático**. Un analizador gramático (también conocido como parser) toma los tokens generados por nuestro analizador léxico y verifica que la cadena recibida sea una cadena que se reconozca dentro del lenguaje [3].

Podemos instalar Happy como instalamos cualquier otro paquete en Haskell, esto es:

```
cabal install alex
```

o

```
stack install alex
```

mediante cabal y stack respectivamente.

Dentro de nuestro archivo `.y` las llaves `{}` sirven para insertar *scrap code*, donde literalmente indicamos que queremos poner código en Haskell; el scrap que ponemos hasta el inicio del archivo por lo general se usa para indicar el módulo de nuestro archivo Haskell:

```
{
module Grammars where
import Lex (Token(..), lexer)
}
```

`module Grammars where` define el nombre de nuestro módulo y `where` indica que el cuerpo del módulo es lo que sigue. `import Lex (Token(..), lexer)` sirve para exportar el tipo de dato *token* y el tipo de dato *lexer*, como se encuentran dentro de los paréntesis respectivamente.

Otra parte importante de nuestro archivo son las declaraciones:

```
%name parse
%tokentype { Token }
%error { parseError }
```

La primera línea declara el nombre de la función `parser` que Happy va a generar, en este caso `parse`. La segunda línea declara el tipo de tokens que nuestro parser va a aceptar, tal que el tipo va a ser `[Token] ->T`, donde `T` es el tipo de dato que retorna nuestro parser y se determina por las reglas de producción que especificaremos después. La última línea literalmente indica el nombre de la función que nuestro parser va a llamar al momento de que ocurra un error.

La siguiente parte de nuestro archivo se conforma de declarar todos los tokens posibles y se indica su comienzo con la sentencia `%token`. Los tokens van a estar definidos del lado izquierdo, mientras que del lado derecho va a servir para indicar el tipo de patrón que sigue el token, por esto el parser espera recibir un flujo de tokens donde va a buscar que coincidan con el patrón dado. Además, el símbolo `$$` servirá para representar el valor de un token.

En la siguiente parte del archivo vamos a especificar nuestras siguientes reglas de producción para la gramática, donde cada regla de producción consiste de un símbolo no terminal en el lado izquierdo seguido por dos puntos más una o más expansiones a la derecha separadas por el símbolo pipe. Cabe recalcar que cada expansión tiene código Haskell asociado a ella. El parser reducirá la cadena de entrada usando las reglas gramaticales que definimos hasta que solo un símbolo permanezca; el valor de este símbolo será el valor de regreso de nuestro parser.

Para definir estas reglas vamos a basarnos en nuestra forma EBNF que definimos en el documento, por lo que tendremos que además definir las reglas auxiliares como `BindLet`, `Clause` y `ElseClause`. Además, para facilitar en nuestra forma EBNF utilizamos los símbolos `{}` con el fin de simplificar la notación para expresiones con aridad mayor o igual a 2. Sin embargo, nosotros tendremos que definir funciones auxiliares que representen este símbolo, como por ejemplo: `ExprList`, `BindList`, `BarList`, `ClauseList`, etc. También utilizaremos la misma estrategia para las listas de elementos, de tal forma que para cada una de éstas se puede definir como su expresión más simple seguida de otra lista, es decir, de alguna forma utilizaremos asociatividad en pares a la izquierda y se aceptarán recursivamente.

Finalmente, agregamos otro apartado con scrap code donde definimos nuestro árbol de

sintaxis abstracta (ASA) esto con el fin de definir la semántica que utilizaremos después, pues Happy solo se encarga de reconocer la sintaxis.

Para ejecutar el archivo y comprobar que funciona hay que primero haber generado nuestro analizador léxico con **Alex**, de tal forma que nos genere un archivo `Lexer.hs`. Así ya podemos importar a nuestro archivo de Happy el módulo con los tokens definidos que necesitamos. Para ejecutarlo de forma manual tenemos que generar el archivo Grammars.hs a través de happy con el siguiente comando:

```
happy Grammars.y
```

Si se generó el archivo `.hs` sin ningún error, podemos pasar ahora sí a probarlo en nuestro intérprete con el siguiente comando:

```
ghci Grammars.hs
```

Siempre y cuando se encuentre en la misma ruta nuestro archivo `Lexer.hs`. Finalmente, para probarlo tenemos que escribir `parse` y entre paréntesis `lexer` y entre comillas una expresión que acepte nuestro lenguaje, pues de lo contrario mandará error. Ejemplo:

```
parse (lexer "(+ 1 2)")
```

Que nos generará la siguiente salida:

AddS (NumS 1) (NumS 2)

Otro ejemplo sería:

```
parse (lexer "(lambda (x y z) (+ 1 5 8))")
```

Que nos generará la siguiente salida:

LambdaS ["x","y","z"] (AddS (AddS (NumS 1) (NumS 5)) (NumS 8))

Eliminación de Azúcar sintáctica:

La eliminación de azúcar sintáctica es un proceso fundamental en el diseño de lenguajes de programación que permite distinguir entre una sintaxis de superficie (rica y expresiva) y un núcleo mínimo (simple y fundamental) [7,1]. Según Felleisen et al. [2], este proceso establece una correspondencia sistemática entre construcciones convenientes y formas más primitivas, preservando el poder expresivo mientras se simplifica la semántica.

Convención notacional:

Utilizaremos la siguiente notación para las reglas de desazucarización:

$$\frac{SUPERFICIE}{NUCLEO}(nombre - regla)$$

Donde:
SUPERFICIE: Expresión en la sintaxis extendida
NÚCLEO: Traducción equivalente en el núcleo mínimo
NOMBRE-REGLA: Identificador de la regla

Formalmente, para la desazucarización queremos definir una función que preserve el significado de nuestras expresiones y que a su vez traduzca semánticamente la azúcar sintáctica a las expresiones más simples, es decir, las primitivas del núcleo del lenguaje.

Anteriormente definimos nuestro árbol de sintaxis azucarado (también conocido como de alto nivel), donde representamos el lenguaje tal como lo escribiría el usuario; por eso, en lugar de definir la estructura como ASA, lo definimos como *SASA*. Al implementar el desugar tenemos que pasar de nuestro árbol de sintaxis concreta a un árbol de sintaxis abstracta; para esto vamos a definir una función por inducción estructural

$$D : \text{SASA} \to \text{ASA}$$

tal que, para cualquier expresión $e \in$ SASA, al aplicar la función se preserva el significado. Para esto definimos el núcleo funcional puro con todas las estructuras sin azúcar sintáctica, de modo que:

- "$\to$" representa la regla de desazucarización, que va de la sintaxis azucarada al núcleo.
- x_i representan las subexpresiones de la lista; al pasar a x_i' se desazucariza.

Expresiones directas:

---------- (DES-NUM)
NumS(n) $\Rightarrow$ Num(n)

---------- (DES-BOOL)
BoolS(b) $\Rightarrow$ Boolean(b)

---------- (DES-ID)
IdS(x) $\Rightarrow$ Id(x)

---------- (DES-NIL)
NilS $\Rightarrow$ Nil

Reglas inductivas estructurales:

Usando una notación estándar de reglas inductivas:

Para $n \geq 2$ y $i = 1..n$, supongamos $e_i \Rightarrow e'_i$. Entonces:

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{AddS}([e_1, \dots, e_n]) \Rightarrow \text{foldl1 Add } [e'_1, \dots, e'_n]} \quad \text{(DES-ADD-n-left)}$$

y expandiendo el *foldl1*:

$$\text{AddS}([e_1, \dots, e_n]) \Rightarrow \text{Add}(\dots \text{ Add}(\text{Add}(e'_1, e'_2), e'_3) \ \dots, \ e'_n)$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{SubS}([e_1, \dots, e_n]) \Rightarrow \text{foldl1 Sub } [e'_1, \dots, e'_n]} \quad \text{(DES-SUB-n-left)}$$

y expandiendo el *foldl1*:

$$\text{SubS}([e_1, \dots, e_n]) \Rightarrow \text{Sub}(\dots \text{ Sub}(\text{Sub}(e'_1, e'_2), e'_3) \ \dots, \ e'_n)$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{MulS}([e_1, \dots, e_n]) \Rightarrow \text{foldl1 Mul } [e'_1, \dots, e'_n]} \quad \text{(DES-MUL-n-left)}$$

y expandiendo el *foldl1*:

$$\text{MulS}([e_1, \dots, e_n]) \Rightarrow \text{Mul}(\dots \text{ Mul}(\text{Mul}(e'_1, e'_2), e'_3) \ \dots, \ e'_n)$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{DivS}([e_1, \dots, e_n]) \Rightarrow \text{foldl1 Div } [e'_1, \dots, e'_n]} \quad \text{(DES-DIV-n-left)}$$

y expandiendo el *foldl1*:

$$\text{DivS}([e_1, \dots, e_n]) \Rightarrow \text{Div}(\dots \text{ Div}(\text{Div}(e'_1, e'_2), e'_3) \ \dots, \ e'_n)$$

Desugar comparaciones:

Reglas auxiliares:

Empty

$$\frac{}{} \quad \text{(DES-CHAIN-EMPTY)}$$

$\text{BiggerS}([]) \Rightarrow \text{Boolean True}$

$SmallerS([]) \Rightarrow \text{Boolean True}$

$BigEq([]) \Rightarrow \text{Boolean True}$

$SmallEq([]) \Rightarrow \text{Boolean True}$

$Eq([]) \Rightarrow$ Boolean True
$NotEq([]) \Rightarrow$ Boolean True

Singleton

$$\frac{}{\text{BiggerS}([e1]) \Rightarrow \text{Boolean True}} \text{(DES-CHAIN-SINGLE)}$$

$SmallerS([e1]) \Rightarrow$ Boolean True
$BigEq([e1]) \Rightarrow$ Boolean True
$SmallEq([e1]) \Rightarrow$ Boolean True
$Eq([e1]) \Rightarrow$ Boolean True
$NotEq([e1]) \Rightarrow$ Boolean True

Regla: desazucarización binaria con recursividad

Chain $n \geq 2$

$$\frac{e1 \Rightarrow e1' \quad e2 \Rightarrow e2' \quad \dots \quad en \Rightarrow en'}{\text{CtorS([e1,...,en])} \Rightarrow \text{If (ctor } e1' \ e2') \text{ (If (ctor } e2' \ e3') \text{ (... (If (ctor } e'_{n-1} \ e'_n) \text{ (Boolean True) (Boolean False)) ...) (Boolean False)) (Boolean False)} \text{(DES-CHAIN-n)}$$

auxBE a b = If (MayorQue a b) (Boolean True) (Eq a b)

auxSE a b = If (MenorQue a b) (Boolean True) (Eq a b)

auxNE a b = Not (Eq a b)

Reglas concretas:

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{BiggerS}([e_1, \dots, e_n]) \Rightarrow \text{If (MayorQue } e'_1 \ e'_2) \text{ (chain\_desugared}([e_2, \dots, e_n])) \text{ (Boolean False)}} \text{(DES-BIGGERS)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{SmallerS}([e_1, \dots, e_n]) \Rightarrow \text{If (MenorQue } e'_1 \ e'_2) \text{ (chain\_desugared}([e_2, \dots, e_n])) \text{ (Boolean False)}} \text{(DES-SMALLERS)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{BigEq}([e_1, \dots, e_n]) \Rightarrow \text{If (auxBE } e'_1 \ e'_2) \text{ (chain\_desugared}([e_2, \dots, e_n])) \text{ (Boolean False)}} \text{(DES-BIGEQ)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{SmallEq}([e_1, \dots, e_n]) \Rightarrow \text{If (auxSE } e'_1 \ e'_2) \text{ (chain\_desugared}([e_2, \dots, e_n])) \text{ (Boolean False)}} \text{(DES-SMALLEQ)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{EqS}([e_1, \dots, e_n]) \Rightarrow \text{If (Eq } e'_1 \ e'_2) \text{ (chain\_desugared}([e_2, \dots, e_n])) \text{ (Boolean False)}} \text{(DES-EQS)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \dots \quad e_n \Rightarrow e'_n}{\text{NotEq}([e_1, \dots, e_n]) \Rightarrow \text{If (auxNE } e'_1 \ e'_2) \text{ (chain\_desugared}([e_2, \dots, e_n])) \text{ (Boolean False)}} \text{(DES-NOTEQ)}$$

Desazucarizaciones unarias

$$\frac{e \Rightarrow e'}{\text{SmallEq}([e_1, \ldots, e_n]) \Rightarrow \text{If (auxSE } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-SMALLEQ)}$$

$$\frac{e \Rightarrow e'}{\text{EqS}([e_1, \ldots, e_n]) \Rightarrow \text{If (Eq } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-EQS)}$$

$$\frac{e \Rightarrow e'}{\text{NotEq}([e_1, \ldots, e_n]) \Rightarrow \text{If (auxNE } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-NOTEQ)}$$

Reglas concretas:

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \ldots \quad e_n \Rightarrow e'_n}{\text{BiggerS}([e_1, \ldots, e_n]) \Rightarrow \text{If (MayorQue } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-BIGGERS)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \ldots \quad e_n \Rightarrow e'_n}{\text{SmallerS}([e_1, \ldots, e_n]) \Rightarrow \text{If (MenorQue } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-SMALLERS)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \ldots \quad e_n \Rightarrow e'_n}{\text{BigEq}([e_1, \ldots, e_n]) \Rightarrow \text{If (auxBE } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-BIGEQ)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \ldots \quad e_n \Rightarrow e'_n}{\text{SmallEq}([e_1, \ldots, e_n]) \Rightarrow \text{If (auxSE } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-SMALLEQ)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \ldots \quad e_n \Rightarrow e'_n}{\text{EqS}([e_1, \ldots, e_n]) \Rightarrow \text{If (Eq } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-EQS)}$$

$$\frac{e_1 \Rightarrow e'_1 \quad e_2 \Rightarrow e'_2 \quad \ldots \quad e_n \Rightarrow e'_n}{\text{NotEq}([e_1, \ldots, e_n]) \Rightarrow \text{If (auxNE } e'_1\ e'_2) \text{ (chain\_desugared}([e_2, \ldots, e_n])) \text{ (Boolean False)}} \text{(DES-NOTEQ)}$$

Desazucarizaciones unarias

$$\frac{e \Rightarrow e'}{\text{Add1}(e) \Rightarrow \text{Add}(e',\ \text{Num 1})} \text{(DES-ADD1)}$$

$$\frac{e \Rightarrow e'}{\text{Sub1}(e) \Rightarrow \text{Sub}(e',\ \text{Num 1})} \text{(DES-SUB1)}$$

$$\frac{e \Rightarrow e'}{\text{SqrtS}(e) \Rightarrow \text{App (Id "sqrt) } e'} \text{(DES-SQRT)}$$

$$\frac{a \Rightarrow a' \quad b \Rightarrow b'}{\text{Expt}(a, b) \Rightarrow \text{App (App (Id .}^{\text{ex}}\text{pt) } a')\ b'} \text{(DES-EXPT)}$$

$$\frac{e \Rightarrow e'}{\text{NotS}(e) \Rightarrow \text{Not}(e')} \text{(DES-NOT)}$$

$$\frac{a \Rightarrow a' \quad b \Rightarrow b'}{\text{PairS}(a, b) \Rightarrow \text{Pair}(a',\ b')} \text{(DES-PAIR)}$$

$$\frac{e \Rightarrow e'}{\text{FstS}(e) \Rightarrow \text{Fst}(e')} \quad \text{(DES-FST)}$$

$$\frac{e \Rightarrow e'}{\text{SndS}(e) \Rightarrow \text{Snd}(e')} \quad \text{(DES-SND)}$$

— para lista con n elementos (n $\geq$ 0) —

$$\frac{e_1 \Rightarrow e_1' \quad e_2 \Rightarrow e_2' \quad \ldots \quad e_n \Rightarrow e_n'}{\text{List}([e_1, \ldots, e_n]) \Rightarrow \text{foldr Cons Nil } [e_1', \ldots, e_n']} \quad \text{(DES-LIST)}$$

$$\frac{e \Rightarrow e'}{\text{HeadS}(e) \Rightarrow \text{Head}(e')} \quad \text{(DES-HEAD)}$$

$$\frac{e \Rightarrow e'}{\text{TailS}(e) \Rightarrow \text{Tail}(e')} \quad \text{(DES-TAIL)}$$

Desazucarización de funciones, aplicación y condicionales

$$\frac{b \Rightarrow b'}{\text{LambdaS}(ps,\, b) \Rightarrow \text{Fun } ps\ b'} \quad \text{(DES-LAMBDA)}$$

$$\frac{f \Rightarrow f' \quad a \Rightarrow a'}{\text{AppS}(f,\, a) \Rightarrow \text{App}(f',\, a')} \quad \text{(DES-APP)}$$

$$\frac{c \Rightarrow c' \quad t \Rightarrow t' \quad e \Rightarrow e'}{\text{IfS}(c,\, t,\, e) \Rightarrow \text{If}(c',\, t',\, e')} \quad \text{(DES-IF)}$$

$$\frac{c \Rightarrow c' \quad t \Rightarrow t' \quad e \Rightarrow e'}{\text{If0}(c,\, t,\, e) \Rightarrow \text{If}(\text{Eq}(c',\, \text{Num } 0),\, t',\, e')} \quad \text{(DES-IF0)}$$

$$\frac{e_1 \Rightarrow e_1' \quad e_2 \Rightarrow e_2' \quad \ldots \quad e_n \Rightarrow e_n' \quad \text{body} \Rightarrow \text{body}'}{\text{LetS}([\text{Bind } x_1\ e_1, \ldots, \text{Bind } x_n\ e_n], \text{body}) \Rightarrow \text{App}\big(\ldots\big(\text{App}\big(\text{App}\big(\text{Fun } [x_1, \ldots, x_n]\ \text{body}'\big)\ e_1'\big)\ e_2'\big) \ldots\ e_n'\big)} \quad \text{(DES-LETS-N)}$$

$$\frac{\text{LetSeq}([], \text{body}) \Rightarrow \text{body}'}{\text{LetSeq}([], \text{body}) \Rightarrow \text{body}'} \quad \text{(DES-LETSEQ-EMPTY)}$$

$$\frac{e_1 \Rightarrow e_1' \quad \text{LetSeq}(bs, \text{body}) \Rightarrow \text{body}''}{\text{LetSeq}(\text{Bind } x\ e_1 : bs, \text{body}) \Rightarrow \text{App}\big(\text{Fun } [x]\ \text{body}''\big)\ e_1'} \quad \text{(DES-LETSEQ-CONS)}$$

$$\frac{e_1 \Rightarrow e_1' \quad e_2 \Rightarrow e_2'}{\text{LetRec}(\text{Bind } f\ e_1,\, e_2) \Rightarrow \text{App}\big(\text{Fun } [f]\ e_2'\big)\big(\text{App}(\text{Id "fix") (Fun } [f]\ e_1')\big)} \quad \text{(DES-LETREC)}$$

para n cláusulas n ≥ 1 *con Else e*

$$\frac{c_1 \Rightarrow c_1' \quad e_1 \Rightarrow e_1' \quad \ldots \quad c_n \Rightarrow c_n' \quad e_n \Rightarrow e_n' \quad e \Rightarrow e'}{\text{Cond}([\text{Clause } c_1\ e_1, \ldots, \text{Clause } c_n\ e_n], \text{Else } e) \Rightarrow \text{If}(c_1',\, e_1',\, \text{If}(c_2',\, e_2',\, \ldots\ \text{If}(c_n',\, e_n',\, e') \ldots))} \quad \text{(DES-COND-n)}$$

Implementación del Desugar usando Haskell

Una vez definidas nuestras funciones núcleo de nuestro ASA podemos implementarlas en haskell, para representar las que ya son mínimas, importamos el módulo donde definimos

nuestra SASA y creamos un *data structure* con las ASA núcleo tal que las demás expresiones se van a terminar descomponiendo en estas.

La función desugar se implementa como en nuestra formalización donde Num, Bool, Id y Nil se mantienen igual al ser mínimas, las funciones con aridad mayor o igual a 2 se desazucarizan al buscar el patrón de la operación a realizar y aplicando recursivamente la función de sugar sobre el resto de la cadena. En las expresiones que son estrictamente unarias o binarias sólo desazucarizamos las expresiones que tengan como parámetros.

Para la desazucarización de las comparaciones (=, >, <, >=, <=, y !=) vamos a definir una función auxiliar `chainComparison` basada en la formalización que describimos arriba, donde si la comparación no existe o se hace sobre un elemento te devuelve `true`, en caso de comparar más elementos, `chainComparison` va a tomar de dos en dos elementos y le va a aplicar la función que ingreses desazucarizando las expresiones, así recursivamente. Si la primera comparación te regresa un `false` pasa a la siguiente comparación, y sigue comparando hasta que se terminen los elementos, de modo que las comparaciones terminan en una condicional `if` sin azúcar sintáctica, pues `if` forma parte de nuestro núcleo.

De igual forma (>=, <= y !=) tienen su propia desazucarización, pues (>= a b) se descompone en `if` (>a b) (`boolean true`) (= a b), dado que si se cumple la primera condición se devuelve `true` y en caso contrario se checa la segunda condición, "`<=`" se implementa de manera análoga y finalmente `(!= a b)` pasa a la negación de la igualdad `(not (eq a b))`.

Las últimas funciones que vale la pena analizar individualmente de nuestra implementación es `let` y su variante `let*`, ya que la primera se evalúa en paralelo, mientras que la segunda secuencialmente. Al evaluar en paralelo las subexpresiones independientes pueden evaluarse en distintos procesos y semánticamente suele modelarse como una reducción paralela o multi-reducción, mientras que al evaluar en secuencial se establece una estrategia y se actúa conforme a esta. Por lo tanto `let` se desazucariza como una única función con n parámetros aplicada a n expresiones y `let*` se desazucariza como aplicaciones anidadas.

Semántica de *MiniLisp*:

Después de definir la sintaxis del lenguaje, es necesario precisar cómo los términos se evaluarán. Para eso, daremos una formalización de la semántica del lenguaje usando semántica operacional.

La semántica operacional se refiere al uso de una máquina abstracta, donde el estado de la máquina es un término del lenguaje y su comportamiento se define por una función de transición de estados [1].

Emplearemos el enfoque de semántica estructural o de paso pequeño para formalizar la semántica operacional del lenguaje. Esto, puesto que dicho enfoque permite descomponer la evaluación de las expresiones del lenguaje en transiciones elementales y así ver detalladamente

la dinámica de ejecución del mismo [12].

Para mantener una consistencia con las notaciones vistas en clase, la notación a emplear de los juicios de transición será

$$e \ \rightarrow \ e'$$

, lo que significa que la expresión e se reduce en un paso a una expresión e' [14].

De igual forma, la notación de las derivaciones será la misma que la vista en clase

$$\frac{x_1 \ \ldots \ x_n}{x}$$

, donde:

- Una regla sin premisas corresponde a un axioma [12].
- Una regla con premisas describe cómo, dado ciertos juicios, se infiere un nuevo juicio [12].

Evaluación y alcance del lenguaje:

La evaluación que tiene el lenguaje es ansiosa (por requerimientos del proyecto) y tiene alcance estático (por requerimientos del proyecto), es decir:

- **Evaluación Ansiosa:** Si una expresión del lenguaje tiene subexpresiones, siempre se evaluarán primero las subexpresiones y luego se aplicará la expresión original con los valores obtenidos [15].
- **Alcance Estático:** El valor de una variable se determina observando el bloque de código donde fue declarada [16].

Semántica en Paso Pequeño (Small Step):

Para nuestro lenguaje, emplearemos una máquina abstracta **CE** compuesta por dos componentes principales [15]:

- **Control:** La expresión que se evalúa.
- **Ambiente:** Asociaciones entre variables y sus valores.

El uso de esta máquina permite tener una complejidad lineal en la búsqueda de los valores de variables (si se implementa en una pila o una lista, como es nuestro caso), pues tiene un ambiente que guarda los valores de dichas variables y permite resolverlos dinámicamente [15].

Ahora, definimos el sistema de transición de paso pequeño con el uso de la máquina CE de la siguiente forma: la tupla $\big(C, \ \rightarrow, \ I, \ F\big)$, donde:

- $C = \{\, (e, \varepsilon) \mid e \in \text{ASA},\ \varepsilon \text{ un ambiente léxico} \,\}$ ..... configuraciones.
- $\rightarrow \subseteq\ C \times C$ .... la relación de reducción (entre configuraciones (e, ε)). En small step reducimos configuraciones.
- $I = \text{ASA}$ ... estados iniciales, pues en paso pequeño todas las expresiones pueden ser estados iniciales.
- $F = \{\, \text{Num}(n) \mid n \in \mathbb{Z} \,\} \cup \{\, \text{Bool}(b) \mid b \in \{\text{True}, \text{False}\} \,\} \cup \{\, \text{Pair}(v_1, v_2) \mid v_1, v_2 \in F \,\} \cup \{\, \text{Closure}(p, c, \varepsilon_0) \mid p : \text{String},\ c \in \text{ASA},\ \varepsilon_0 \text{ es un ambiente léxico} \,\}$ .... valores canónicos.

Expresiones del ASAV:

Dado que en la implementación del lenguaje se debe poder diferenciar entre los estados que son finales de los que no lo son, se construye el conjunto **ASAV** (Árbol de Sintaxis Abstracta Value) para dicho fin. Estas expresiones (valores canónicos) son las que se regresan en cada ejecución de un programa en el lenguaje.

ASAV de <Int>:

$$\frac{n \in \mathbb{Z}}{\text{Num}(n)\ \text{ASAV}} \qquad \text{(ASAV-INT)}$$

ASAV de <Bool>:

$$\frac{b \in \{\text{True},\ \text{False}\}}{\text{Bool}(b)\ \text{ASAV}} \qquad \text{(ASAV-BOOL)}$$

ASAV de los pares:

$$\frac{l \text{ ASAV} \qquad r \text{ ASAV}}{\text{Pair}(l, r) \text{ ASAV}} \tag{ASAV-PAIR}$$

ASAV de las cerraduras de función:

$$\frac{p : \text{String} \qquad c \in \text{ASA} \qquad \varepsilon \text{ es un ambiente léxico}}{\text{Closure}(p, c, \varepsilon) \text{ ASAV}} \tag{ASAV-CLOS}$$

Ambientes (ε) y búsqueda de variables (Lookup):

Usamos un ambiente *léxico* ε que asocia identificadores a *valores canónicos* (ASAV). El ambiente se modela como una lista (o pila) de enlaces con sombreados por derecha:

- $\varepsilon ::= [\,] \mid \varepsilon[x \mapsto v]$
- **Función de búsqueda** (toma el *enlace más reciente*): $(x, v) \in \varepsilon$ si $\varepsilon = \varepsilon'[x \mapsto v]$ o $(x, v) \in \varepsilon'$.

Reglas de Lookup:

$$\frac{(x, v) \in \varepsilon}{\langle \text{Id}(x),\, \varepsilon \rangle \;\to\; \langle v,\, \varepsilon \rangle} \tag{Lookup-Hit}$$

$$\frac{x \notin \text{dom}(\varepsilon)}{\langle \text{Id}(x),\, \varepsilon \rangle \;\to\; \text{error}} \tag{Lookup-Miss}$$

Régimen de evaluación que usaremos (para todo el small-step):

- **Estrategia: Call-by-Value (CBV)** con orden izquierda → derecha.

Primero se reduce el operando/guardia del lado izquierdo, luego el derecho, y *sólo* cuando ambos son valores se aplica la regla base del constructor (suma, comparación, aplicación, etc.).

- **Alcance: estático (léxico).**

Las funciones se evalúan como cerraduras $\text{Closure}\langle p, c, \varepsilon_0 \rangle$ que capturan el ambiente del punto de definición. La regla de βv sustituirá enlazando $p \mapsto v$ en ε_0, no en el ambiente de llamada.

- **Determinismo (esbozo):**

Con CBV y el orden $L \to R$, para toda configuración no final $\langle e, \varepsilon\rangle$ existe a lo más un contexto activo y, por tanto, un siguiente paso $\langle e', \varepsilon'\rangle$. Esto asegura que la relación $\to$ sea determinista a nivel de un paso.

Esquemas contextuales (patrones de evaluación):

Bajo **CBV con orden izquierda $\to$ derecha**, toda construcción con subexpresiones se evalúa primero a la *izquierda* y después a la *derecha*. Para evitar repetir reglas, usamos *familias* que luego se instancian por operador.

Binarios (familia general):

$$\frac{\langle e_1, \varepsilon\rangle \;\to\; \langle e_1', \varepsilon\rangle}{\langle \mathrm{Bin}(e_1, e_2), \varepsilon\rangle \;\to\; \langle \mathrm{Bin}(e_1', e_2), \varepsilon\rangle} \qquad \text{((Bin-Left))}$$

$$\frac{\langle e_2, \varepsilon\rangle \;\to\; \langle e_2', \varepsilon\rangle \quad v \in F}{\langle \mathrm{Bin}(v, e_2), \varepsilon\rangle \;\to\; \langle \mathrm{Bin}(v, e_2'), \varepsilon\rangle} \qquad \text{((Bin-Right))}$$

Instanciación: si Bin = Add obtienes Add-Left y Add-Right; si Bin = Lt, obtienes Lt-Left/Lt-Right, etc.

Unarios (Familia general)

$$\frac{\langle e, \varepsilon\rangle \;\to\; \langle e', \varepsilon\rangle}{\langle \mathrm{Un}(e), \varepsilon\rangle \;\to\; \langle \mathrm{Un}(e'), \varepsilon\rangle} \qquad \text{((Un-Arg))}$$

Instanciación: Un = Not, Un = Fst, Un = Snd, etc. (Las reglas base de cada uno van en su sección.)

Condicional (patrón del guardia):

$$\frac{\langle g, \varepsilon\rangle \;\to\; \langle g', \varepsilon\rangle}{\langle \mathrm{If}(g, t, f), \varepsilon\rangle \;\to\; \langle \mathrm{If}(g', t, f), \varepsilon\rangle} \qquad \text{((If-Guard))}$$

Las bases (*If-True*/*If-False*) irán en el bloque de condicional.

Aplicación (solo el patrón; las bases van después):

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon\rangle}{\langle \mathrm{App}(e_1, e_2), \varepsilon\rangle \to \langle \mathrm{App}(e_1', e_2), \varepsilon\rangle} \qquad \text{((App-Fun))}$$

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle \quad v \in \mathsf{F}}{\langle \mathrm{App}(v, e_2), \varepsilon\rangle \to \langle \mathrm{App}(v, e_2'), \varepsilon\rangle} \qquad \text{((App-Arg))}$$

La βv (aplicación por valor sobre closures) aparecerá en su bloque respectivo.

Catálogo de side-conditions:

Para mantener la semántica clara y determinista, fijamos estas condiciones (se *referenciarán* en las reglas base de cada operador):

- **Tipos esperados**
  - Aritmética $\text{Add}, \text{Sub}, \text{Mult}, \text{Div}, \text{Expt}$: ambos argumentos deben ser **Num**
  - Comparadores $\text{Eq}, \text{Lt}, \text{Le}, \text{Gt}, \text{Ge}$: argumentos **Num**
  - Not: argumento **Bool**
  - Fst/Snd: argumento $\textbf{Pair}(v_1, v_2)$
  - If: guardia **Bool**(**True**/**False**)
  - App: la fun debe ser $\textbf{Closure}(p, c, \varepsilon_0)$ antes de aplicar βv

Reglas que comparten las operaciones aritméticas:

$$\text{Ar} = \{\ \text{Add},\ \text{Sub},\ \text{Mult},\ \text{Div},\ \text{Expt}\ \}$$

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle Ar(i, d), \varepsilon\rangle \to \langle Ar(i', d), \varepsilon\rangle}$$ Add-Left/Right, Sub-Left/Right, Mult-Left/Right, Div-Left/Right, Expt-Left/Right

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle Ar(\text{Num}(n), d), \varepsilon\rangle \to \langle Ar(\text{Num}(n), d'), \varepsilon\rangle}$$ Add-Left/Right,Sub-Left/Right,Mult-Left/Right,Div-Left/Right, Expt-Left/Right

Reglas únicas de Add:

$$\frac{}{\langle \text{Add}(\text{Num}(n),\ \text{Num}(m)),\ \varepsilon\rangle\ \to\ \langle \text{Num}(n +_{\mathbb{Z}} m),\ \varepsilon\rangle}$$ (Add-Num)

Reglas únicas de Sub:

$$\frac{\langle \text{Sub}(\text{Num}(n),\ \text{Num}(m)),\ \varepsilon\rangle}{\langle \text{Num}(n -_{\mathbb{Z}} m),\ \varepsilon\rangle}$$ (Sub-Num)

Reglas únicas de Mult:

$$\frac{\langle \text{Mult}(\text{Num}(n),\ \text{Num}(m)),\ \varepsilon\rangle}{\langle \text{Num}(n \times_{\mathbb{Z}} m),\ \varepsilon\rangle}$$ (Mult-Num)

Reglas únicas de Div

$$\frac{}{\langle \text{Div}(\text{Num}(n), \text{Num}(0)), \varepsilon\rangle \;\to\; \text{error}} \quad \text{(Div-Cero)}$$

$\text{m} \neq 0$

$$\frac{}{\langle \text{Div}(\text{Num}(n), \text{Num}(m)), \varepsilon\rangle \;\to\; \langle \text{Num}(n \,/_{\text{Z}}\, m), \varepsilon\rangle} \quad \text{(Div-Num)}$$

Reglas únicas de Expt

$$\frac{}{\langle \text{Expt}(\text{Num}(n), \text{Num}(0)), \varepsilon\rangle \;\to\; \langle \text{Num}(1), \varepsilon\rangle} \quad \text{(Expt-Cero)}$$

$$\frac{}{\langle \text{Expt}(\text{Num}(n), \text{Num}(m)), \varepsilon\rangle \;\to\; \langle \text{Num}(n^m), \varepsilon\rangle} \quad \text{(Expt-Num)}$$

Condicional (If)

Primero se evalúa la **guardia** (patrón ya definido como *If-Guard*). Las reglas base aplican sólo cuando la guardia es **boolean**:

$$\frac{}{\langle \text{If}(\text{Bool}(\text{True}), t, f), \varepsilon\rangle \;\to\; \langle t, \varepsilon\rangle} \quad \text{(If-True)}$$

$$\frac{}{\langle \text{If}(\text{Bool}(\text{False}), t, f), \varepsilon\rangle \;\to\; \langle f, \varepsilon\rangle} \quad \text{(If-False)}$$

Si la guardia no es **boolean**, el condicional es inválido:

$$\frac{}{\langle \text{If}(v, t, f), \varepsilon\rangle \;\to\; \text{error}} \quad \text{(If-Err)}$$

si $v \notin \{\text{Bool(True)}, \text{Bool(False)}\}$

Side-condition usada: la guardia debe ser Bool(b). Esto respeta CBV y evita ambigüedad.

Negación (Not)

Primero se reduce su argumento (patrón *Un-Arg*). Bases:

$$\frac{}{\langle \mathrm{Not}(\mathrm{Bool}(b)), \varepsilon\rangle \;\to\; \langle \mathrm{Bool}(\neg b), \varepsilon\rangle} \qquad \text{(Not-Base)}$$

Argumento inválido:

$$\frac{}{\langle \mathrm{Not}(v), \varepsilon\rangle \;\to\; \mathrm{error}} \qquad \text{(Not-Err)}$$

si $v \notin \{\mathrm{Bool(True)}, \mathrm{Bool(False)}\}$

Comparadores (Eq**,** Lt**,** Le**,** Gt**,** Ge**)**

Patrón contextual (hereda Bin-Left / Bin-Right):

Para cada comparador binario $\mathrm{Cmp} \in \{\mathrm{Eq}, \mathrm{Lt}, \mathrm{Le}, \mathrm{Gt}, \mathrm{Ge}\}$ usamos las familias genéricas:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon\rangle}{\langle \mathrm{Cmp}(e_1, e_2), \varepsilon\rangle \to \langle \mathrm{Cmp}(e_1', e_2), \varepsilon\rangle} \qquad \text{(Cmp-Left)}$$

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle \quad v \in \mathcal{F}}{\langle \mathrm{Cmp}(v, e_2), \varepsilon\rangle \to \langle \mathrm{Cmp}(v, e_2'), \varepsilon\rangle} \qquad \text{(Cmp-Right)}$$

Reglas base (tipadas a enteros):

Usan la side-condition global: los comparadores trabajan sobre $\mathrm{Num}, \mathrm{Num}$*. Si no se cumple, es* `error`*.*

$$\frac{}{\langle \mathrm{Eq}(\mathrm{Num}(n), \mathrm{Num}(m)), \varepsilon\rangle \to \langle \mathrm{Bool}(n = m), \varepsilon\rangle} \qquad \text{(Eq-Num)}$$

$$\frac{}{\langle \mathrm{Lt}(\mathrm{Num}(n), \mathrm{Num}(m)), \varepsilon\rangle \to \langle \mathrm{Bool}(n < m), \varepsilon\rangle} \qquad \text{(Lt-Num)}$$

$$\frac{}{\langle \mathrm{Le}(\mathrm{Num}(n), \mathrm{Num}(m)), \varepsilon\rangle \to \langle \mathrm{Bool}(n \leq m), \varepsilon\rangle} \qquad \text{(Le-Num)}$$

$$\frac{}{\langle \mathrm{Gt}(\mathrm{Num}(n), \mathrm{Num}(m)), \varepsilon\rangle \to \langle \mathrm{Bool}(n > m), \varepsilon\rangle} \qquad \text{(Gt-Num)}$$

$$\frac{}{\langle \mathrm{Ge}(\mathrm{Num}(n), \mathrm{Num}(m)), \varepsilon\rangle \to \langle \mathrm{Bool}(n \geq m), \varepsilon\rangle} \quad \text{(Ge-Num)}$$

Tipos inválidos (mismo criterio para todos los comparadores):

$$\frac{}{\langle \mathrm{Cmp}(v_1, v_2), \varepsilon\rangle \to \mathrm{Error}} \quad \text{(Cmp-Err)}$$
si $(v_1, v_2) \notin \mathrm{Num} \times \mathrm{Num}$

Esto mantiene la semántica determinista y coherente con el catálogo: sólo Num, Num son válidos para Cmp; el resultado siempre es Bool(b).

Pares y proyecciones (Pair / Fst / Snd**)**

Valores con pares

Extendemos $\mathcal{F}$ con pares anidados:

$$\mathcal{F} \ni \mathrm{Pair}(v_1, v_2) \quad \text{si} \quad v_1 \in \mathcal{F} \ \wedge \ v_2 \in \mathcal{F}.$$

Construcción de pares

Patrón contextual:

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon\rangle}{\langle \mathrm{Pair}(e_1, e_2), \varepsilon\rangle \to \langle \mathrm{Pair}(e_1', e_2), \varepsilon\rangle} \quad \text{(Pair-Left)}$$

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle \quad v \in \mathcal{F}}{\langle \mathrm{Pair}(v, e_2), \varepsilon\rangle \to \langle \mathrm{Pair}(v, e_2'), \varepsilon\rangle} \quad \text{(Pair-Right)}$$

Regla base (creación de valor par):

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

Observación. Si $v_1, v_2 \in \mathcal{F}$, entonces $\mathrm{Pair}(v_1, v_2) \in \mathcal{F}$ (no hay paso).

Proyecciones: Fst/Snd

Patrón contextual (instancia de *Un-Arg*):

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon\rangle}{\langle \mathrm{Fst}(e), \varepsilon\rangle \to \langle \mathrm{Fst}(e'), \varepsilon\rangle} \quad \text{(Fst-Arg)}$$

$$\frac{\langle e, \varepsilon\rangle \to \langle e', \varepsilon\rangle}{\langle \mathrm{Snd}(e), \varepsilon\rangle \to \langle \mathrm{Snd}(e'), \varepsilon\rangle} \quad \text{(Snd-Arg)}$$

Reglas base (sobre valores par):

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

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

Errores por tipo inválido:

$$\frac{}{\langle \mathrm{Fst}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{si } v \neq \mathrm{Pair}(v_1, v_2) \tag{Fst-Err}$$

$$\frac{}{\langle \mathrm{Snd}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{si } v \neq \mathrm{Pair}(v_1, v_2) \tag{Snd-Err}$$

Side-condition aplicada: los argumentos de Fst/Snd deben ser **valores par**. Esto mantiene determinismo y unifica el tratamiento de errores.

Listas

Patrón, instancia de *Un-Arg*:

$$\frac{\langle e, \varepsilon \rangle \to \langle e', \varepsilon \rangle}{\langle \mathrm{Head}(e), \varepsilon \rangle \to \langle \mathrm{Head}(e'), \varepsilon \rangle} \tag{Head-Arg}$$

$$\frac{\langle e, \varepsilon \rangle \to \langle e', \varepsilon \rangle}{\langle \mathrm{Tail}(e), \varepsilon \rangle \to \langle \mathrm{Tail}(e'), \varepsilon \rangle} \tag{Tail-Arg}$$

Bases:

$$\frac{}{\langle \mathrm{Head}(\mathrm{Cons}(v, xs)), \varepsilon \rangle \to \langle v, \varepsilon \rangle} \tag{Head-Cons}$$

$$\frac{}{\langle \mathrm{Tail}(\mathrm{Cons}(v, xs)), \varepsilon \rangle \to \langle xs, \varepsilon \rangle} \tag{Tail-Cons}$$

Errores:

$$\frac{}{\langle \mathrm{Head}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{si } v \neq \mathrm{Cons}(v_1, xs) \tag{Head-Err}$$

$$\frac{}{\langle \mathrm{Tail}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{si } v \neq \mathrm{Cons}(v_1, xs) \tag{Tail-Err}$$

Funciones y Aplicación (Lambda / App / βv **con closures)**

Valores de función y cerraduras

En $\mathcal{F}$ las funciones son **cerraduras**:

$\mathcal{F} \ni \mathrm{Closure}(p, c, \varepsilon_0)$ donde $p \in \mathrm{Id}$, $c \in \mathrm{ASA}$, ε_0 es el ambiente léxico capturado.

Lambdas como cerraduras (creación)

$$\frac{}{\langle \mathrm{Lambda}(p, c), \varepsilon \rangle \to \langle \mathrm{Closure}(p, c, \varepsilon), \varepsilon \rangle} \tag{Lam-Clos}$$

Con esto, el valor de una función *siempre* es Closure(...): refleja *alcance estático* (se captura ε del punto de definición).

Aplicación (patrones contextuales; hereda App-Fun / App-Arg)

$$\frac{\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon\rangle}{\langle \mathrm{App}(e_1, e_2), \varepsilon\rangle \to \langle \mathrm{App}(e_1', e_2), \varepsilon\rangle} \quad \text{(App-Fun)}$$

$$\frac{\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle \quad v \in \mathcal{F}}{\langle \mathrm{App}(v, e_2), \varepsilon\rangle \to \langle \mathrm{App}(v, e_2'), \varepsilon\rangle} \quad \text{(App-Arg)}$$

Regla base de aplicación (βv por ambiente)

$$\frac{}{\langle \mathrm{App}(\mathrm{Closure}(p, c, \varepsilon_0), v), \varepsilon\rangle \to \langle c, \varepsilon_0[p \mapsto v]\rangle} \quad \text{(App-}\beta v\text{)}$$

El paso βv sustituye en el *ambiente capturado* ε_0; el nuevo ambiente es $\varepsilon_0[p \mapsto v]$.

Aplicación inválida (no-función)

$$\frac{}{\langle \mathrm{App}(v, w), \varepsilon\rangle \to \mathrm{error}} \quad \text{(App-Err)}$$

si $v \neq \mathrm{Closure}(p, c, \varepsilon_0)$

Side-conditions: por **CBV** (izq→der) evaluamos primero la función hasta valor, luego el argumento hasta valor; solo entonces intentamos (App-βv). Si la "fun" no es *closure*, caemos en App-Err.

Posibles errores

Convención: **error** es un estado terminal (no es un valor). Toda regla que lo produzca termina la ejecución (no hay pasos desde *error*).

1. **Variable no ligada** *(Lookup-Miss)*

$$\frac{}{\langle \mathrm{Id}(x), \varepsilon\rangle \to \mathrm{error}} \quad \text{(Lookup-Miss)}$$

si $x \notin \mathrm{dom}(\varepsilon)$.

2. **División entre cero**

$$\frac{}{\langle \mathrm{Div}(\mathrm{Num}(n), \mathrm{Num}(0)), \varepsilon\rangle \to \mathrm{error}} \quad \text{(Div-Cero)}$$

3. **Tipos inválidos (operadores aritméticos)**

$$\frac{}{\langle \mathrm{Ar}(v_1, v_2), \varepsilon\rangle \to \mathrm{error}} \quad \text{(Ar-Err)}$$

si $(v_1, v_2) \notin \mathrm{Num} \times \mathrm{Num}$, con $\mathrm{Ar} \in \{\mathrm{Add}, \mathrm{Sub}, \mathrm{Mult}, \mathrm{Div}, \mathrm{Expt}\}$.

4. **Exponentes negativos** (si aritmética entera)

$$\frac{}{\langle \mathrm{Expt}(\mathrm{Num}(n), \mathrm{Num}(m)), \varepsilon \rangle \to \mathrm{error}} \quad \text{(Expt-Neg)}$$
si $m < 0$.

5. **Comparadores fuera de tipo**

$$\frac{}{\langle \mathrm{Cmp}(v_1, v_2), \varepsilon \rangle \to \mathrm{error}} \quad \text{(Cmp-Err)}$$
si $(v_1, v_2) \notin \mathrm{Num} \times \mathrm{Num}$, con $\mathrm{Cmp} \in \{\mathrm{Eq}, \mathrm{Lt}, \mathrm{Le}, \mathrm{Gt}, \mathrm{Ge}\}$.

6. ***If* con guardia no booleana**

$$\frac{}{\langle \mathrm{If}(v, t, f), \varepsilon \rangle \to \mathrm{error}} \quad \text{(If-Err)}$$
si $v \notin \{\mathrm{Bool(True)}, \mathrm{Bool(False)}\}$.

7. ***Not* con argumento no booleano**

$$\frac{}{\langle \mathrm{Not}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{(Not-Err)}$$
si $v \notin \{\mathrm{Bool(True)}, \mathrm{Bool(False)}\}$.

8. **Proyecciones sobre no-par**

$$\frac{}{\langle \mathrm{Fst}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{(Fst-Err)}$$
si $v \neq \mathrm{Pair}(v_1, v_2)$.

$$\frac{}{\langle \mathrm{Snd}(v), \varepsilon \rangle \to \mathrm{error}} \quad \text{(Snd-Err)}$$
si $v \neq \mathrm{Pair}(v_1, v_2)$.

9. **Aplicación a no-función**

$$\frac{}{\langle \mathrm{App}(v, w), \varepsilon \rangle \to \mathrm{error}} \quad \text{(App-Err)}$$
si $v \neq \mathrm{Closure}(p, c, \varepsilon_0)$.

Ejemplos de Derivaciones Completas

(A) -por-valor con cierre + aritmética

Expresión: `App(Lambda(x, Add(x, Num(1))), Num(3))`

$$\begin{aligned}
&\langle \text{App}(\text{Lambda}(x, \text{Add}(x, \text{Num}(1))), \text{Num}(3)), \varepsilon \rangle \\
&\to \langle \text{App}(\text{Closure}(x, \text{Add}(x, \text{Num}(1)), \varepsilon), \text{Num}(3)), \varepsilon \rangle \quad \text{(Lam-Clos)} \\
&\to \langle \text{Add}(x, \text{Num}(1)), \varepsilon[x \mapsto \text{Num}(3)] \rangle \quad \text{(App-}\beta v\text{)} \\
&\to \langle \text{Add}(\text{Num}(3), \text{Num}(1)), \varepsilon[x \mapsto \text{Num}(3)] \rangle \quad \text{(Lookup-Hit + Bin-Left)} \\
&\to \langle \text{Num}(4), \varepsilon[x \mapsto \text{Num}(3)] \rangle \quad \text{(Add-Num)}
\end{aligned}$$

(B) Fst correcto y error por tipo

1. Caso correcto:

$$\langle \text{Fst}(\text{Pair}(\text{Num}(1), \text{Num}(2))), \varepsilon \rangle \to \langle \text{Num}(1), \varepsilon \rangle \quad \text{(Fst-Pair)}$$

2. Error por tipo:

$$\langle \text{Fst}(\text{Num}(7)), \varepsilon \rangle \to \text{error} \quad \text{(Fst-Err)}$$

(C) Cond desazucarado a If + cortocircuito

Expresión concreta:

```
(cond [Eq(Num(2), Num(2)) => Num(10)]
      [Else => Num(20)])
```

Desazucarado al núcleo:

$$\text{If}(\text{Eq}(\text{Num}(2), \text{Num}(2)), \text{Num}(10), \text{Num}(20))$$

$$\begin{aligned}
&\langle \text{If}(\text{Eq}(\text{Num}(2), \text{Num}(2)), \text{Num}(10), \text{Num}(20)), \varepsilon \rangle \\
&\to \langle \text{If}(\text{Bool}(\text{True}), \text{Num}(10), \text{Num}(20)), \varepsilon \rangle \quad \text{(Eq-Num)} \\
&\to \langle \text{Num}(10), \varepsilon \rangle \quad \text{(If-True)}
\end{aligned}$$

Implementación de la semántica en Haskell

Una vez formalizado en papel nuestra semántica, ya podemos implementarla en nuestro lenguaje anfitrión Haskell.

Para la implementación, se crea el módulo Interp, importando el archivo `Desugar.hs`, que tiene la gramática final ASAV del lenguaje. Luego, se define el tipo de dato `ENV = [(String, ASAV)]` que son los ambientes léxicos ε, mencionados anteriormente en el sistema de transiciones de paso pequeño. También, se define el tipo de dato `Config = (ASA, ENV)` que representan las configuraciones del sistema de paso pequeño.

Ahora bien, el módulo tiene varias funciones auxiliares para facilitar la implementación de la semántica.

Función isCanon:

La función `isCanon :: ASA ->Bool` determina si una expresión del ASA pertenece a los estados finales $F = \{\, \text{Num}(n) \mid n \in \mathbb{Z} \,\} \cup \{\, \text{Boolean}(b) \mid b \in \{\text{True}, \text{False}\} \,\} \cup \{\, \text{Pair}(v_1, v_2) \mid v_1, v_2 \in F \,\} \cup \{\, \text{Closure}(p, c, \varepsilon_0) \mid p : \text{String},\ c \in E,\ \varepsilon_0 \text{ es un ambiente léxico} \,\}$ de la semántica. Por lo tanto, determina si una expresión del ASA es un valor canónico.

```
isCanon :: ASA -> Bool
isCanon (Num _) = True
isCanon (Boolean _) = True
isCanon (Pair _ _) = True
isCanon (Fun _ _) = True -- aquí, se emplea el equivalente de Closure en ASA.
isCanon _ = False
```

Función isPair:

La función `isPair :: ASA ->Bool` determina si una expresión del ASA es un Pair.

```
isPair :: ASA -> Bool
isPair (Pair _ _) = True
isPair _ = False
```

Función toASAV:

La función `toASAV :: ASA ->ASAV` transforma una ASA que pertenezca a los valores canónicos en su forma equivalente en ASAV.

```
toASAV :: ASA -> ASAV
toASAV (Num n) = (NumV n)
toASAV (Boolean b) = (BooleanV b)
toASAV (Pair l r) = (PairV (toASAV l) (toASAV r))
toASAV (Fun x c) = Closure x c [] -- Aquí, capturamos el closure con el ambiente vacío
```

Función returnASA:

La función `returnASA :: ASAV ->ASA` transforma un ASAV a su forma equivalente en ASA.

```
returnASA :: ASAV -> ASA
returnASA (NumV n) = (Num n)
returnASA (BooleanV b) = (Boolean b)
returnASA (PairV l r) = (Pair (returnASA l) (returnASA r))
returnASA (Closure x c env) = Fun x c
returnASA (Error msg) = error msg
```

Función lookupHit:

La función `lookupHit :: String ->ENV ->ASAV` busca el valor de un identificador en un ambiente léxico ENV y lo devuelve.

- Si el ambiente está vacío, devuelve un Error con un string que explica que el error obtenido es que la variable "i" a buscar su valor es libre: `'Error LookUpMiss: El identificador ++ i ++ es una variable libre'`
- Si el ambiente no está vacío y se busca la variable "i", si "i" es igual a la variable de la primera tupla del ambiente (que es una lista de tuplas (String, ASAV)), se regresa el valor de esa tupla (el ASAV). Si no es igual a la primera tupla del ambiente, se busca en las siguientes tuplas del ambiente.

```
lookupHit :: String -> ENV -> ASAV
lookupHit i [] = Error ("Error LookUpMiss: El identificador " ++ i ++ " es una variable
lookupHit i ((x,v):xs) = if i == x then v else lookupHit i xs
```

Función eval:

La función `eval :: Config ->Config` evalúa configuraciones hasta llegar a una configuración de un valor canónico.

- Si tenemos la configuración `(e, env)` y `isCanon e` es verdadero, entonces la expresión "e" es un valor canónico y por ende ya se llegó a una configuración de un valor canónico, por lo que se regresa `(e, env)`.
- En otro caso, hace `eval (smallStep (e, env))`, para seguir evaluando `(e, env)`.

Función evalFinal:

La función `evalFinal :: Config ->ConfigASAV` evalúa configuraciones usando `eval` y el resultado lo transforma a su forma equivalente en configuraciones ASAV, ENV. Dicha función, es la que se emplea para evaluar las expresiones ASA en su totalidad a un valor canónico ASAV.

Tipo de dato ConfigASAV:

Configuraciones (tuplas) de tipo `(ASAV, ENV)`.

Función smallStep:

Ahora, describamos la función principal que implementa la semántica de paso pequeño.

La función `smallStep :: Config ->Config`, implementa las reglas de transición de paso pequeño antes mencionadas, ordenadas de menos generales a más generales, de modo que Haskell pueda evaluar primero aquellas reglas que reconocen configuraciones donde las subexpresiones ya son valores irreducibles. Esto, pues Haskell evalúa las funciones de arriba hacia abajo, por lo que con el orden descrito, los casos más específicos (como operaciones entre valores) se aplican antes que los casos más generales, respetando así la semántica de paso pequeño que se tiene.

Para aquellas reglas que tienen una subexpresión que se reduce en las premisas, $\langle e, \varepsilon\rangle \to \langle e', \varepsilon\rangle$; se realiza `let (e', env') = smallStep (e, env)`. Esto, debido a que, aplicar `smallStep` a la subexpresión "e" con el mismo ambiente de la expresión original, realiza la reducción en un paso de "e", pues `smallStep` implementa las reglas de transición de paso pequeño. Aquello, genera una expresión "e"' que "toma el lugar" de la subexpresión "e" en la expresión original (parte `in ...` de estas reglas).

Porción de código:

```
smallStep :: Config -> Config
smallStep ((Num n), env) = ((Num n), env)
smallStep ((Boolean b), env) = ((Boolean b), env)
smallStep ((Add (Num n) (Num m)), env) = (Num (n + m), env)
smallStep ((Add (Num n) e2), env) =
    let (e2', env') = smallStep (e2, env)
    in (Add (Num n) e2', env)
smallStep ((Add e1 e2), env) =
    let (e1', env') = smallStep (e1, env)
    in (Add e1' e2, env)
```

Casos a considerar de smallStep

Caso smallStep ((Id x), env):

Se define `v` como el resultado de buscar el valor del String `x` en el ambiente `env`, usando `lookupHit x env`, valor que se regresa en la configuración `(returnASA(v), env)`. Esto, es justo lo que se define en la semántica de la regla Lookup-Hit, pues `lookupHit` devuelve el valor en ASAV de un identificador en un ambiente y al regresar la configuración `(returnASA(v), env)`, el valor ASAV se devuelve en una configuración ya en su forma de ASA y con el mismo ambiente que tenía.

Ahora, en caso de que `lookupHit x env` regrese `Error(msg)`, la función `returnASA(v)` se encarga de traducirlo a un error msg de Haskell que interrumpe la ejecución del programa.

Caso de smallStep ((Fun p c), env):

`((Fun p c), env)`: No se reduce a closure para que la implementación se siga manteniendo de ASA a ASA. Sin embargo, esto no afecta la semántica del lenguaje, debido a que cuando se realizan las reglas de `smallStep` de App, sí se trata `(Fun p c)` con Closure.

Caso smallStep ((App (Fun p c) e2), env):

`= doclosure (appArg (funToClosure ((App (Fun p c) e2), env)))`: Aquí primero, `funToClosure` se encarga de aplicar la regla de paso pequeño "Lam-Clos" a `((App (Fun p c) e2), env)`, que deriva en un paso a `((AppV (Closure p c env)) a, env)`. Esto, respeta la semántica de paso pequeño. Luego, `appArg` aplicado al resultado anterior, es decir `((AppV (Closure p c env0) a), env)`; usando eval reduce con paso chico a `a` a un valor canónico con la regla "App-Arg", lo que genera `((Appv (Closure p c env) a´), env)`, respetando la semántica descrita. Finalmente, `doclosure` aplicado a `((Appv (Closure p c env) a), env)`, ya con "a" un valor canónico; da un paso con la regla "App-v", generando `(c, (p, toASAV(a)):env)`, que respeta la semántica descrita y es lo que devuelve `smallStep`.

Se tiene esta forma de aplicar `smallStep` debido a que en la implementación, dicha función va de ASA a ASA y Closure no forma parte de ASA según la sintaxis abstracta descrita. Aún así, como se explicó, respeta la semántica de paso pequeño.

Conclusiones

Reflexión crítica sobre el proceso de diseño e implementación

Optamos por una arquitectura modular (léxico → parser/SASA → desazúcar → ASA → intérprete small-step), lo que nos permitió separar claramente la teoría de la implementación. La desazucarización reduce la superficie del lenguaje a un núcleo pequeño (If, /App,

aritmética, comparadores, pares), facilitando que las reglas small-step sean deterministas con evaluación CBV izquierda → derecha. En el código, esa visión se refleja en funciones concretas: la traducción de let/let*/letrec a /App (con fix para recursión), cond a If, y la currificación de lambdas con múltiples parámetros para conservar reglas de aplicación unarias. Este alineamiento teoría-código simplificó pruebas y depuración, y ayudó a mantener consistencia entre lo especificado en el reporte y lo ejecutado por el intérprete.

Limitaciones encontradas

Ambientes y closures costaron trabajo al principio. Tuvimos idas y vueltas con la notación (ASA vs ASAV y el sufijo V). A ratos mezclamos "expresión" con "valor" y eso se notó en reglas como (App-v) y en los errores de lookup. La corrección fue estandarizar: los valores canónicos son NumV/BooleanV/PairV/Closure, y el cuerpo del closure es ASA (no un valor), más el ambiente léxico capturado.

CBV izquierda → derecha en patrones de congruencia. Nos pasó que algunas reglas reducían "de más" o "en orden distinto", por eso aparecían "variables libres" o resultados raros. Se arregló dejando claro el orden: primero función, luego argumento; sólo disparamos v cuando ambos son valor.

Proyecciones y errores de tipo. Al principio usábamos condiciones informales ("no es par") y no siempre quedaba claro. Terminamos con side-conditions explícitas tipo: "si $v \in \{\text{PairV}(v_1, v_2) \mid v_1, v_2 \in V\}$, entonces error", eso ayudó a mantener determinismo.

Azúcar sintáctica (let/let* y cond). El desugar nos generó fricción:

- `let/let*` debían traducirse a lambdas unarias (currificadas), respetando CBV (izq→der). Al principio se nos colaba aridad múltiple y rompía el intérprete.
- `cond` → anidación de if con corto-circuito; la última rama como Bool(True) para no inventar semántica nueva.

Listas y utilitarios. Aparecieron fallos puntuales en head/tail, sqrt y `expt` cuando no elevábamos literales a valores o cuando la regla base no checaba dominio (p.ej., sqrt con $n < 0$).

Pruebas y ejemplos límite. Casos con let anidado, lambdas currificadas y pares dentro de fst/snd mostraron huecos que no se ven con ejemplos cortitos. Hasta que no metimos esos tests, algunos bugs seguían escondidos.

Posibles extensiones futuras del lenguaje

Operadores numéricos completos con chequeos de dominio.
Ya tenemos aritmética básica; conviene cerrar bien Div (error si dividimos entre 0), Sqrt

(error si $n < 0$) y Expt (acotar enteros y casos negativos). Todo con reglas base alineadas a NumV.

Modo "traza" del intérprete.
Una bandera para imprimir $\langle e, \varepsilon \rangle \to \langle e', \varepsilon' \rangle$ en cada paso. Útil para tareas y para explicar CBV en clase.

Extensiones de funciones de orden superior.
Con cierres ya funcionando, es natural sumar compose, curry/uncurry, y utilitarios para trabajar con funciones como datos (sin cambiar el núcleo, sólo como biblioteca).

Errores semánticos uniformes.
Centralizar mensajes/razones de error (ejemplo: TypeErr: expected PairV, DivByZero, UnboundId x). Eso ayuda a depurar y a pasar pruebas automáticas.

Menú Interactivo

Para realizar nuestro menú interactivo tuvimos que importar los identificadores hFlush y stdout del módulo System.IO, donde el primero te permite generar una salida en consola y el segundo te permite forzar la salida pendiente en el búfer para que se escriba inmediatamente en el dispositivo asociado.

Posteriormente generamos una variable en donde evaluamos nuestros módulos en cadena hasta que nos quede un ASA desazucarizado. Creamos un menú característico y en honor a haskell y sus siglas ghc (Glasgow Haskell Compiler), decidimos ponerle a nuestro intérprete UNAM MiniLisp Compiler o UMCi, donde la i representa que es un intérprete.

Agregamos un manejador de excepciones para detectar los errores que nos envíen.

Bibliografía (formato IEEE)

[1] Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

[2] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to automata theory, languages, and computation* (3rd ed.). Pearson.

[3] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Pearson Education.

[4] 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.

[5] Simon Marlow and the Alex developers. *Alex: A tool for generating lexical analysers*

in Haskell. 2022. URL: `https://haskell-alex.readthedocs.io/`. Accedido: 2025-09-30.

[6] Simon Marlow and the Happy developers. *Happy: A parser generator for Haskell*. 2022. URL: `https://haskell-happy.readthedocs.io/`. Accedido: 2025-09-30.

[7] M. Soto Romero, *Sintaxis Abstracta*, Facultad de Ciencias UNAM, 2025. Retrieved from `https://lambdasspace.github.io/LDP/notas/ldp_n05.pdf`

[8] Graham Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, UK, 2nd edition, 2016.

[9] Lipovača, M. (2011). *Learn you a Haskell for great good!* Retrieved from `https://learnyouahaskell.github.io/starting-out.html`

[10] E. V. Gurovich, *Introducción a Autómatas y Lenguajes Formales*, 2ª ed. México: Universidad Nacional Autónoma de México, Facultad de Ciencias, 2015, pp. 90, 201.

[11] M. Soto Romero, *Sintaxis Concreta*, Facultad de Ciencias UNAM, 2025. Retrieved from `https://lambdasspace.github.io/LDP/notas/ldp_n04.pdf`

[12] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*, Springer, 2023.

[13] S. Krishnamurthi, *Programming Languages: Application and Interpretation*, 2017.

[14] M. Soto Romero, *Semántica Dinámica*, Facultad de Ciencias UNAM, 2025. Retrieved from `https://lambdasspace.github.io/LDP/notas/ldp_n06.pdf`

[15] M. Soto Romero, *Máquinas Abstractas*, Facultad de Ciencias UNAM, 2025. Retrieved from `https://lambdasspace.github.io/LDP/notas/ldp_n06.pdf`

[16] M. Soto Romero, *Ambientes de Evaluación*, Facultad de Ciencias UNAM, 2025. Retrieved from `https://lambdasspace.github.io/LDP/notas/ldp_n06.pdf`

[17] M. Soto Romero, D. Méndez Medina, J. A. Pérez Márquez, E. D. Arroyo Martínez, y M. E. Chávez Zamora, *Proyecto 1: MiniLisp*, Facultad de Ciencias, UNAM, 2025.