Universidad Nacional Autónoma de México
Facultad de Ciencias

Lenguajes de Programación

Profesor:
Manuel Soto Romero

Ayudantes de Laboratorio:
Erick Daniel Arroyo Martínez
José Alejandro Pérez Márquez
Mauro Emiliano Chávez Zamora

Proyecto 1:
MiniLisp

Integrantes:

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

Octubre 2025

Introducción

Motivación

Como parte del curso del lenguaje de programación tras haber revisado la teoría para poder construir un lenguaje de programación (MiniLisp) con este proyecto buscamos aplicar lo aprendido hasta este momento desde

Objetivos específicos

1. **Definir la sintaxis léxica, libre de contexto y representación abstracta de MiniLisp** junto con +
2. **Establecer la semántica operacional estructural del lenguaje**, especificando tanto máquinas abstractas como modelos conceptuales, como máquinas virtuales que funcionen como implementaciones ejecutables en Haskell.
3. **Modelar formalmente los ambientes de evaluación y los bindings**, considerándolos mecanismos fundamentales para garantizar la consistencia semántica y la correcta gestión de alcances y valores.
4. **Integrar el régimen de evaluación ansioso** con una estrategia de paso de parámetros por valor (*call-by-value*), analizando su impacto en la ejecución de programas y en el diseño del intérprete.
5. **Implementar de manera funcional el lenguaje MiniLisp embebido en Haskell**, mostrando la correspondencia explícita entre las reglas de la formalización teórica y su ejecución práctica.
6. **Evidenciar, mediante la eliminación de azúcar sintáctica**, el tránsito desde la expresividad del lenguaje hacia un núcleo reducido, fortaleciendo así la comprensión del puente entre teoría y 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 que 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 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.[3]

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 · 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)* pues *E* estrictamente hablando es sólo una expresión y no un lenguaje y por eso tenemos que denotarlo por aparte como *L(E)*:

Casos base: La base consiste en tres partes:

1. La constante ε y ∅ son expresiones regulares y denotan el lenguaje entre {ε} y ∅ respectivamente, esto es $L(\varepsilon)=\{\varepsilon\}$, y $L(\varnothing)=\{\varnothing\}$
2. Si a es un símbolo entonces a es una expresión regular, esta expresión denota el $L(\mathbf{a}) = \{a\}$, donde **a** se refiere a la misma a.
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 unión 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 denotando el mismo lenguaje *E*. Formalmente; $L((E)) = L(E)$.

[2]

Sintaxis Léxica de MiniLisp:

Una vez definida que 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.

Nuestro Σ={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 se refiere 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 tal que 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 eso:

```
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, que 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, los "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 la 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 carácteres se debe ignorar.

Como ya tenemos nuestra variables patrón definidas podemos usarlas para empezar a indicar nuestro tokens, los espacios en blanco los ignoramos y empezamos a definir el nombre de cada uno de nuestros tokens. En la parte 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 una de nuestros símbolos pues ya tendrán su identificador y no tendremos que definir manualmente uno por uno.

Para agregar **comentarios** a nuestro 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 si 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 GFC por sus siglas en inglés.

Gramática formal:

Una gramática formal se define formalmente como [9]:

Un 4-duplo ***G =(N, T, P, S)***, donde:

- *N* es un conjunto finito de símbolos *no terminales.*
- *T* es un conjunto de s*ímbolos terminales* ($N \cap T = \varnothing$)
- *P* es un conjunto finito de *producciones*
- *S* es el símbolo inicial tal que $S \notin (N \cup T)$ … 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 α = γAδ en la cuál 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 $\alpha \rightarrow \beta$.

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]:

- $S \rightarrow \varepsilon$
- $A \rightarrow a$

con S y $A \in N$, S es el símbolo inicial y $|A| \leq |\alpha|$, $\alpha \in (N \cup T)^+$ y si la producción $S \rightarrow \varepsilon$ 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 <> (ej: <Expr> o <Ide>).
- Reglas de producción: Se representan igual que en la notación clásica pero se cambia el símbolo “→” 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, sub1 (de incremento y decremento respectivamente), 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, #t)), (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, permitiendo dependencias entre asignaciones.
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.** A˜nadir la construcci´on 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 y que actúa como caso por defecto. Especificar en EBNF la notación de las cláusulas y documentar las restricciones de uso de else.
9. **Funciones anónimas y aplicaciones**: la superficie del lenguaje debe incluir lambdas variádicas de la forma (lambda (x1 ...) Expr), permitiendo la definición de funciones con múltiples parámetros, así como la aplicación de funciones a sus 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 de enteros

| (< <Expr> <Expr> { <Expr> }) … comparación de menor que de ent.

| (> <Expr> <Expr> { <Expr> }) … comparación de mayor que de ent.

| (>= <Expr> <Expr> { <Expr> }) … comparación de mayor igual de ent.

| (<= <Expr> <Expr> { <Expr> }) … comparación de menor igual de ent.

| (!= <Expr> <Expr> { <Expr> }) … comparación de no igual de ent.

| (<Expr> , <Expr>) … pares.

| (fst <Expr>) … proyección primer elemento

| (snd <Expr>) … proyección segundo elemento

| (let <BindLet> { <BindLet> } <Expr>) … asign. local con múltiples variables (en paralelo)

| (let* <BindLet> { <BindLet> } <Expr>)… asign. local con múltiples variables (secuencial)

| (letrec <BindLet> <Expr>) … asignación de variables recursiva

| (if <Expr> <Expr> <Expr>) … condicional “si verdadero entonces”

| (if0 <Expr> <Expr> <Expr>) … condicional “si 0 entonces”

| (lambda (<Var> { <Var> }) <Expr>) … funciones anónimas de varios argumentos.

| (<Expr> <Expr> { <Expr> }) … aplicaciones de función de varios argumentos

| nil … vacío

| “[“ <Expr> { , <Expr> } “]” … lista con elementos

| (head <Expr>) … cabeza de una lista

| (tail <Expr>) … cola de una lista

| (cond <Clause> { <Clause> } “[“ <ElseClause> “]”)… condicional por clausulado

<BindLet> ::= (<Var> <Expr>) … binding de una variable con un valor

<Clause> ::= “[“ <Expr> <Expr> ”]” … cláusula de una guarda booleana con una expresión

<ElseClause> ::= “[“ else <Expr> “]” … cláusula else

<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 [3]. 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 de 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 LSASA:
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:

---

nil LSASA

LSASA de la lista con elementos:

x SASA xs LSASA

———————————

(x:xs) LASA

SASA de <Var>:

i : String

———————

IDS(i) SASA

SASA de <Int>:

n ∈ ℤ

———————

NumS(n) SASA

SASA de <Bool>:

b ∈ { True, False }

———————

BoolS(b) SASA

SASA del vacío:

———————————————

NilS SASA

SASA de un binding de una variable con un valor:

i : String v SASA

———————————————

BindLet(i, v) SASA

SASA de una cláusula de una guarda booleana con una expresión:

i SASA d SASA

———————————————

Clause(i, d) SASA

SASA de una cáusula else:

e SASA

———————————————

ElseClause(e) SASA

SASA de la suma variádica:

as LSASA

AddS(as) SASA

SASA de la resta varíadica:

as LSASA

SubS(as) SASA

SASA de la multiplicación varíádica:

as LSASA

MultS(as) SASA

SASA de la división variádica:

as LSASA

DivS(as) SASA

SASA de la potencia:

b SASA e SASA

Expt(b, e) SASA

SASA del incremento de 1:

e SASA

Add1(e) SASA

SASA del decremento de 1:

e SASA

Sub1(e) SASA

SASA de la raíz cuadrada:

e SASA

SqrtS(e) SASA

SASA de la operación de negación:

e SASA

NotS(e) SASA

SASA de la comparación de igualdad variádica:

as LSASA

EqualS(as) SASA

SASA de la comparación de “menor que” de enteros variádica:

as LSASA

SmallerS(as) SASA

SASA de la comparación de “mayor que” de ent.. variádica:

as LSASA

BiggerS(as) SASA

SASA de la comparación de “mayor igual” de enteros:

as LSASA

BigEq(as) SASA

SASA de la comparación de “menor igual” de enteros:

as LSASA

SmallEq(as) SASA

SASA de la comparación de “no igual” de enteros:

as LSASA

NotEq(as) SASA

SASA de los pares:

l SASA r SASA

PairS(l,r) SASA

SASA de la proyección del primer elemento:

e SASA

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

SASA de la proyección del segundo elemento:

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

SASA de la asignación local con múltiples variables en paralelo:

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

SASA de la asignación local con múltiples variables secuencial:

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

SASA de la asignación de variables recursiva:

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

SASA de la condicional “si verdadero entonces”:

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

SASA de la condicional “si 0 entonces”:

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

SASA de las funciones anónimas con varios parámetros:

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

SASA de las aplicaciones de función con varios argumentos:

f SASA as LSASA

AppS(f, as) SASA

SASA de la lista con elementos:

as LSASA

List(as) SASA

SASA de la cabeza de una lista:

e SASA

HeadS(e) SASA

SASA de la cola de una lista:

e SASA

TailS(e) SASA

SASA de condición por clausulado:

as LSASA e SASA

Cond(as, e) 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 <Var>:

i : String

ID(i) ASA

ASA de <Int>:

n ∈ ℤ

Num(n) ASA

ASA de <Bool>:

b ∈ { True, False }

Boolean(b) ASA

ASA de la suma binaria:

i ASA d ASA

Add(i,d) ASA

ASA de la resta binaria:

i ASA d ASA

Sub(i,d) ASA

ASA de la multiplicación binaria:

i ASA d ASA

Mult(i,d) ASA

ASA de la división binaria:

i ASA d ASA

Div(i,d) ASA

ASA de la raíz cuadrada:

e ASA

Sqrt(e) ASA

ASA de la operación de negación:

b ASA

Not(b) ASA

ASA del vacío:

Nil ASA

ASA de la condicional “si verdadero entonces”:

c ASA t ASA e ASA

If(c, t, e) ASA

ASA de la igual de enteros binaria:

i ASA d ASA

Eq(i, d) ASA

ASA para las comparaciones:

ASA de la comparación de “menor que” de enteros variádica:

i ASA d ASA

MenorQue(i, d) ASA

ASA de la comparación de “mayor que” de ent.. variádica:

i ASA d ASA

MayorQue(i, d) ASA

ASA de los pares:

l ASA r ASA

Pair(l,r) ASA

ASA de la proyección del primer elemento:

e ASA

Fst(e) ASA

ASA de la proyección del segundo elemento:

e ASA

Snd(e) ASA

ASA de la función anónima:

p : String c ASA

Fun(p, c) ASA

ASA de la aplicación de función:

f ASA a ASA

App(f, a) ASA

ASA de la cabeza de una lista:

e ASA

Head(e) ASA

ASA de la cola de una lista:

e ASA

Tail(e) ASA

ASA de la cabeza de una lista:

e ASA

Head(e) ASA

ASA de la tupla:

e_1 ASA e_2 ASA

Cons(e_1, e_2) 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 eso:

```
cabal install happy
```

o

```
stack install happy
```

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 regla 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 sólo 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 estrategía para las listas de elementos, de tal forma que para cada una de estas se puede definir como su expresión más simple seguido 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 azucarado (SASA) esto con el fin de definir la semántica que utilizaremos después, pués Happy sólo 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 si 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 [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 desasucarización:

SUPERFICIE
—---------- (nombre-regla)
NÚCLEO

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 que son primitivas del núcleo del lenguaje.

Anteriormente definimos nuestro árbol de sintaxis azucarado o también conocido como de alto nivel, donde representamos el lenguaje tal como lo escribía 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: SASA -> ASA tal que para cualquier expresión e ∈ SASA al aplicar la función preserva el significado. Para esto tenemos que definir el núcleo funcional puro con todas las estructuras sin azúcar sintáctica de modo que:

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

Expresiones directas:

——————————————— (DES-NUM)
NumS(n) ⇒ Num(n)

——————————————— (DES-BOOL)
BoolS(b) ⇒ Boolean(b)

——————————————— (DES-ID)
IdS(x) ⇒ Id(x)

——————————————— (DES-NIL)
NilS ⇒ Nil

Reglas inductivas estructurales:

Usando una notación estándar de reglas inductivas:

Para n ≥ 2 y i = 1..n, supongamos ei ⇒ e'i.
Entonces:

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-ADD-n-left)
AddS([e1,…,en]) ⇒ foldl1 Add [e1',…,en']

y expandiendo foldl1:

AddS([e1,…,en]) ⇒ Add(... Add(Add(e1',e2'),e3') ... , en')

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————— (DES-SUB-n-left)
SubS([e1,…,en]) ⇒ foldl1 Sub [e1',…,en']

y expandiendo el foldl1:

SubS([e1,…,en]) ⇒ Sub(... Sub(Sub(e1',e2'),e3') ... , en')

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————— (DES-MUL-n-left)
MulS([e1,…,en]) ⇒ foldl1 Mul [e1',…,en']

y expandiendo el foldl1:

MulS([e1,…,en]) ⇒ Mul(... Mul(Mul(e1',e2'),e3') ... , en')

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————— (DES-DIV-n-left)
DivS([e1,…,en]) ⇒ foldl1 Div [e1',…,en']

y expandiendo el foldl1:

DivS([e1,…,en]) ⇒ Div(... Div(Div(e1',e2'),e3') ... , en')

Desugar comparaciones:
Reglas auxiliares:

Empty
——————————————————— (DES-CHAIN-EMPTY)
BiggerS([]) ⇒ Boolean True
SmallerS([]) ⇒ Boolean True
BigEq([]) ⇒ Boolean True
SmallEq([]) ⇒ Boolean True
Eq([]) ⇒ Boolean True
NotEq([]) ⇒ Boolean True

Singleton
——————————————————— (DES-CHAIN-SINGLE)
BiggerS([e1]) ⇒ Boolean True
SmallerS([e1]) ⇒ Boolean True

BigEq([e1]) ⇒ Boolean True
SmallEq([e1]) ⇒ Boolean True
Eq([e1]) ⇒ Boolean True
NotEq([e1]) ⇒ Boolean True

Regla: desazucarización binaria con recursividad

Chain n ≥ 2
e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-CHAIN-n)
CtorS([e1,…,en]) ⇒ If (ctor e1' e2') (If (ctor e2' e3') (… (If (ctor e{n-1}' e{n}') (Boolean True) (Boolean False)) …) (Boolean False)) (Boolean False)

———————————————————————————————

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:

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-BIGGERS)
BiggerS([e1,…,en]) ⇒ If (MayorQue e1' e2') (chain_desugared([e2,…,en])) (Boolean False)

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-SMALLERS)
SmallerS([e1,…,en]) ⇒ If (MenorQue e1' e2') (chain_desugared([e2,…,en])) (Boolean False)

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-BIGGEQ)
BigEq([e1,…,en]) ⇒ If (auxBE e1' e2') (chain_desugared([e2,…,en])) (Boolean False)

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-SMALLEQ)
SmallEq([e1,…,en]) ⇒ If (auxSE e1' e2') (chain_desugared([e2,…,en])) (Boolean False)

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-EQS)
EqS([e1,…,en]) ⇒ If (Eq e1' e2') (chain_desugared([e2,…,en])) (Boolean False)

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-NOTEQ)
NotEq([e1,…,en]) ⇒ If (auxNE e1' e2') (chain_desugared([e2,…,en])) (Boolean False)

Desazucarizaciones unarias

e ⇒ e'
——————————————————— (DES-ADD1)
Add1(e) ⇒ Add(e', Num 1)

e ⇒ e'
——————————————————— (DES-SUB1)
Sub1(e) ⇒ Sub(e', Num 1)

e ⇒ e'
——————————————————— (DES-SQRT)
SqrtS(e) ⇒ App (Id "sqrt") e'

a ⇒ a' b ⇒ b'
——————————————————— (DES-EXPT)
Expt(a,b) ⇒ App (App (Id "expt") a') b'

e ⇒ e'
——————————————————— (DES-NOT)
NotS(e) ⇒ Not(e')

a ⇒ a' b ⇒ b'
——————————————————— (DES-PAIR)
PairS(a,b) ⇒ Pair(a', b')

e ⇒ e'
——————————————————— (DES-FST)
FstS(e) ⇒ Fst(e')

e ⇒ e'
——————————————————— (DES-SND)
SndS(e) ⇒ Snd(e')

— para lista con n elementos (n ≥ 0) —
e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en'
——————————————————————————————— (DES-LIST)
List([e1,…,en]) ⇒ foldr Cons Nil [e1',…,en']

e ⇒ e'
——————————————————— (DES-HEAD)
HeadS(e) ⇒ Head(e')

e ⇒ e'
——————————————————————— (DES-TAIL)
TailS(e) ⇒ Tail(e')

Desazucarización de funciones, aplicación y condicionales

b ⇒ b'
——————————————————————— (DES-LAMBDA)
LambdaS(ps, b) ⇒ Fun ps b'

f ⇒ f' a ⇒ a'
——————————————————————— (DES-APP)
AppS(f, a) ⇒ App(f', a')

c ⇒ c' t ⇒ t' e ⇒ e'
——————————————————————— (DES-IF)
IfS(c, t, e) ⇒ If(c', t', e')

c ⇒ c' t ⇒ t' e ⇒ e'
——————————————————————— (DES-IF0)
If0(c, t, e) ⇒ If(Eq(c', Num 0), t', e')

e1 ⇒ e1' e2 ⇒ e2' … en ⇒ en' body ⇒ body'
——————————————————————————————————————— (DES-LETS-N)
LetS([Bind x1 e1, …, Bind xn en], body) ⇒ App (... (App (App (Fun [x1, …, xn] body') e1') e2') ... en')

LetSeq([] , body) ⇒ body'
——————————————————————————————— (DES-LETSEQ-EMPTY)
LetSeq([], body) ⇒ body'

e1 ⇒ e1' LetSeq(bs, body) ⇒ body''
——————————————————————————————————————— (DES-LETSEQ-CONS)
LetSeq(Bind x e1 : bs, body) ⇒ App (Fun [x] body'') e1'

e1 ⇒ e1' e2 ⇒ e2'
——————————————————————————————————————— (DES-LETREC)
LetRec(Bind f e1, e2) ⇒ App (Fun [f] e2') (App (Id "fix") (Fun [f] e1'))

para n cláusulas n ≥ 1 con Else e
c1 ⇒ c1' e1 ⇒ e1' … cn ⇒ cn' en ⇒ en' e ⇒ e'
——————————————————————————————————————(DES-COND-n)
Cond([Clause c1 e1, …, Clause cn en], Else e) ⇒
If(c1', e1', If(c2', e2', … If(cn', en', e') …))

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 [11].
- 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 máquinas 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 (C, →, I, F), donde:

- C = { (e, ε) | e ∈ ASA, ε un ambiente léxico } … configuraciones
- → ⊆ C × C … la relación de reducción (entre configuraciones ⟨e, ε⟩).En small step reducimos configuraciones.
- I = ASA … estados iniciales, pues en paso pequeño todas las expresiones pueden ser estados iniciales.
- F = { Num(n) | n ∈ Z } ∪ { Bool(b) | b ∈ {True, False} } ∪ { Pair(v1, v2) | v1, v2 ∈ F } ∪ { Closure(p, c, ε0) | p : String, c ∈ ASA, ε0 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>:

n ∈ Z

—----------

Num(n) ASAV

ASAV de <Bool>:

b ∈ { True, False }

—----------

Bool(b) ASAV

ASAV de los pares:

l ASAV r ASAV

—----------

Pair(l,r) ASAV

ASAV de las cerraduras de función:

p : String c ASAV ε un ambiente léxico

—----------

Closure(p,c,ε) ASAV

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:

- ε := [] | ε[x → v]
- Función de búsqueda (toma el **enlace más reciente**): (x, v) ∈ ε si ε = ε'[x → v] o (x, v) ∈ ε'

Reglas de Lookup:

—----------

⟨Id(x), ε⟩ → ⟨v, ε⟩

si (x, v) ∈ ε (Lookup-Hit)

—----------

⟨Id(x), ε⟩ → error

si x ∉ dom(ε) (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 solo cuando ambos son valores se aplica la regla base del constructor (suma, comparación, aplicación, etc.).

- **Alcance: estático (léxico).**

Las funciones se evalúan como cerraduras Closure(p, c, ε_0) que capturan el ambiente del punto de definición. La regla de βv sustituirá enlazando p ↦ v en ε_0, no en el ambiente de llamada.

- **Determinismo (esbozo):**

Con CBV y el orden L→R, para toda configuración no final ⟨e, ε⟩ existe a lo más un contexto activo y, por tanto, un siguiente paso ⟨e', ε'⟩. Esto asegura que la relación → sea determinista a nivel de un paso.

Esquemas contextuales (patrones de evaluación):

Bajo **CBV con orden izquierda→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):

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

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

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

Unarios (Familia general)

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

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

Condicional (patrón del guardia):

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

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

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

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

$\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle \; v \in F$
——————————————————————(App-Arg)
$\langle App(v, e_2), \varepsilon\rangle \to \langle App(v, e_2'), \varepsilon\rangle$

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 Add, Sub, Mult, Div, Expt: ambos argumentos deben ser **Num**
  - Comparadores Eq, Lt, Le, Gt, Ge: argumentos **Num**
  - Not: argumento **Bool**
  - Fst/Snd: argumento **Pair(v_1,v_2)**
  - If: guardia **Bool(True/False)**
  - App: la **fun** debe ser **Closure(p,c,ε_0)** antes de aplicar βv

Reglas que comparten las operaciones aritméticas:

Ar = { Add, Sub, Mult, Div, Expt }

<i, ε> → <i’, ε>

—————————————————— (Add-Left/Right, Sub-Left/Right, Mult-Left/Right, Div-Left/Right, Expt-Left/Right)

<Ar(i, d), ε> → <Ar(i’, d), ε>

<d, ε> → <d', ε>

——————————————————————————————(Add-Left/Right, Sub-Left/Right, Mult-Left/Right, Div-Left/Right, Expt-Left/Right)

<Ar(Num(n), d), ε> → <Ar(Num(n), d'), ε>

Reglas únicas de Add:

—————————————————————————————— (Add-Num)

<Add(Num(n), Num(m)), ε> → <Num(n +Z m), ε>

Reglas únicas de Sub:

—————————————————————————————— (Sub-Num)

<Sub(Num(n), Num(m)), ε> → <Num(n -Z m), ε>

.**Reglas únicas de Mult:**

—---------- (Mult-Num)

<Mult(Num(n), Num(m)), ε> → <Num(n xZ m), ε>

Reglas únicas de Div:

—---------- (Div-Cero)

<Div(Num(n), Num(0)), ε> → error

m ≠ 0

—----------(Div-Num)

<Div(Num(n), Num(m)), ε> → <Num(n /Z m), ε>

Reglas únicas de Expt:

—---------- (Expt-Cero)

<Expt(Num(n), Num(0)), ε> → <Num(1), ε>

—---------- (Expt-Num)

<Expt(Num(n), Num(m)), ε> → <Num(n^m), ε>

Condicional (If):

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

——————————————————————(If-True)

⟨If(Bool(True), t, f), ε⟩ → ⟨t, ε⟩

——————————————————————(If-False)

⟨If(Bool(False), t, f), ε⟩ → ⟨f, ε⟩

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

——————————————————————(If-Err)

⟨If(v, t, f), ε⟩ → error

si v ∉ {Bool(True), 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:

—----------(Not-Base)

⟨ Not(Bool(b)), ε ⟩ → ⟨ Bool(¬b), ε ⟩

Argumento inválido:

—----------(Not-Err)

⟨ Not(v), ε ⟩ → error

si v ∉ { Bool(True), Bool(False) }

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

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

Para cada comparador binario Cmp ∈ {Eq, Lt, Le, Gt, Ge} usamos las familias genéricas:

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

$\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle$ v ∈ F
—----------(Cmp-Right)
$\langle \text{Cmp}(v, e_2), \varepsilon\rangle \to \langle \text{Cmp}(v, e_2'), \varepsilon\rangle$

Reglas base (tipadas a enteros):

(Usan la side-condition global: los comparadores trabajan sobre Num, Num. Si no se cumple, es error.)

——————————————————————————————(Eq-Num)

⟨Eq(Num(n), Num(m)), ε⟩ → ⟨Bool(n = m), ε⟩

——————————————————————————————(Lt-Num)

⟨Lt(Num(n), Num(m)), ε⟩ → ⟨Bool(n < m), ε⟩

——————————————————————————————(Le-Num)

⟨Le(Num(n), Num(m)), ε⟩ → ⟨Bool(n ≤ m), ε⟩

——————————————————————————————(Gt-Num)

⟨Gt(Num(n), Num(m)), ε⟩ → ⟨Bool(n > m), ε⟩

——————————————————————————————(Ge-Num)

⟨Ge(Num(n), Num(m)), ε⟩ → ⟨Bool(n ≥ m), ε⟩

Tipos inválidos (mismo criterio para todos los comparadores):

——————————————————————⟨Cmp-Err),

⟨Cmp(v_1, v_2), ε⟩ →Error

Erorerror si (v_1, v_2) ∉Num × 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 F con pares anidados:

$F \ni Pair(v_1, v_2)$ si $v_1 \in F \land v_2 \in F$

Construcción de pares

Patrón contextual:

$\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon\rangle$
——————————————————————(Pair-Left)
$\langle Pair(e_1, e_2), \varepsilon\rangle \to \langle Pair(e_1', e_2), \varepsilon\rangle$

$\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle$ $v \in F$
——————————————————————(Pair-Right)
$\langle Pair(v, e_2), \varepsilon\rangle \to \langle Pair(v, e_2'), \varepsilon\rangle$

Regla base (creación de valor par):

——————————————————————(Pair-Val)

$\langle Pair(v_1, v_2), \varepsilon\rangle \to \langle Pair(v_1, v_2), \varepsilon\rangle$

Observación. Si $v1,v2 \in F$ entonces $Pair(v1,v2) \in F$ (no hay paso).

Proyecciones: Fst/Snd

Patrón contextual (instancia de Un-Arg):

$\langle e, \varepsilon\rangle \to \langle e', \varepsilon\rangle$
——————————————————(Fst-Arg)
$\langle Fst(e), \varepsilon\rangle \to \langle Fst(e'), \varepsilon\rangle$

$\langle e, \varepsilon\rangle \to \langle e', \varepsilon\rangle$
——————————————————(Snd-Arg)
$\langle Snd(e), \varepsilon\rangle \to \langle Snd(e'), \varepsilon\rangle$

Reglas base (sobre valores par):

——————————————————(Fst-Pair)

$\langle Fst(Pair(v_1, v_2)), \varepsilon\rangle \to \langle v_1, \varepsilon\rangle$

——————————————————(Snd-Pair)

$\langle Snd(Pair(v_1, v_2)), \varepsilon\rangle \to \langle v_2, \varepsilon\rangle$

Errores por tipo inválido:

——————————(Fst-Err)

⟨Fst(v), ε⟩ → error

si v ≠ Pair(v_1, v_2)

——————————(Snd-Err)

⟨Snd(v), ε⟩ →error

si v ≠ Pair(v_1, v_2)

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:

⟨e, ε⟩ → ⟨e′, ε⟩
——————————————————————(Head-Arg)
⟨Head(e), ε⟩ → ⟨Head(e′), ε⟩

⟨e, ε⟩ → ⟨e′, ε⟩
——————————————————————(Tail-Arg)
⟨Tail(e), ε⟩ → ⟨Tail(e′), ε⟩

Bases:
——————————————————————(Head-Cons)
⟨Head(Cons(v, xs)), ε⟩ → ⟨v, ε⟩

——————————————————————(Tail-Cons)
⟨Tail(Cons(v, xs)), ε⟩ → ⟨xs, ε⟩

Errores:

——————————————————————————————————(Head-Err)
⟨Head(v), ε⟩ → error si v ≠ Cons(v1, xs)

——————————————————————————————————(Tail-Err)
⟨Tail(v), ε⟩ → error si v ≠ Cons(v1, xs)

Funciones y Aplicación (Lambda / App / βv con

closures)

Valores de función y cerraduras

En nuestro conjunto de valores F las funciones son **cerraduras**:

F ∋ Closure(p, c, ε_0) donde p ∈ Id, c ∈ ASA, ε_0 es el ambiente léxico capturado

Lambdas como cerraduras (su creación): cuando una lambda aparece como expresión, captura el ambiente actual:

——————————————————————————————(Lam-Clos)

⟨Lambda(p, c), ε⟩ → ⟨Closure(p, c, ε), ε⟩

Con esto, el **valor** de una función **siempre** es closure(…), lo cual refleja el **alcance estático** (léxico): se captura ε del punto de **definición**. Al definir una lambda se crea una **cerradura** que captura el ambiente **ε** vigente (alcance estático).

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

Recordatorio de patrones (ya fijados en el Paso 2):

$\langle e_1, \varepsilon\rangle \to \langle e_1', \varepsilon\rangle$
——————————————————————————————(App-Fun)
$\langle App(e_1, e_2), \varepsilon\rangle \to \langle App(e_1', e_2), \varepsilon\rangle$

$\langle e_2, \varepsilon\rangle \to \langle e_2', \varepsilon\rangle$, v ∈ F
——————————————————————————————(App-Arg)
$\langle App(v, e_2), \varepsilon\rangle \to \langle App(v, e_2'), \varepsilon\rangle$

Regla base de aplicación (βv por ambiente)

Cuando el operador es una **cerradura** y el **operando** ya es un **valor**, hacemos el paso β-por-valor extendiendo el ambiente capturado:

——————————————————————————————(App-βv)

⟨App(Closure(p, c, ε_0), v), ε⟩ → ⟨c, $\varepsilon_0[p \mapsto v]$⟩

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

La estrategia CBV L→R: primero función, luego argumento; solo se dispara **βv** cuando la

función ya es **Closure** y el argumento un **valor**

Aplicación inválida (no-función):

——————————————————(App-Err)

⟨App(v, w), ε⟩ →error

si v ≠ Closure(p, c,ε_0)

Side-conditions: por CBV, **siempre** evaluamos (1) la fun hasta valor, (2) el arg hasta valor, y **solo entonces intentamos** (App-Bv). 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)

——————————————

⟨Id(x), ε⟩ → error

si x ∉ dom(ε)

2. **División entre cero**

——————————————————————————

⟨Div(Num(n), Num(0)), ε⟩ → error

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

——————————————————

⟨Ar(v_1, v_2), ε⟩ → error

si (v_1, v_2) ∉ Num × Num

con Ar ∈ {Add, Sub, Mult, Div, Expt}

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

—----------

⟨Expt(Num(n), Num(m)), ε⟩ → error

si m < 0

5. **Comparadores fuera de tipo**

—----------

⟨Cmp(v_1, v_2), ε⟩ → error

si (v_1, v_2) ∉ Num ×Num

con Cmp ∈ {Eq, Lt, Le, Gt, Ge}

6. **If con guardia no booleana**

—----------

⟨If(v, t, f), ε⟩ → error

Si v ∉ {Bool(True), Bool(False)}

7. **Not con argumento no booleano**

—----------

⟨Not(v), ε⟩ → error

Si v ∉ {Bool(True), Bool(False)}

8. **Proyecciones sobre no-par**

—----------

⟨Fst(v), ε⟩ → error

—----------

⟨Snd(v), ε⟩ → error si v ∉ Pair(v_1, v_2)

9. **Aplicación a no-función**

—----------

⟨App(v, w), ε⟩ → error

si v ∉ Closure(p, c, $ε_0$)

Determinismo CBV y orden L→R

En nuestro núcleo evaluamos por valor y de izquierda→derecha. En cada constructor hay un solo contexto activo (primero el operando/guardia/argumento izquierdo; hasta que sea valor pasamos al derecho). Las reglas base no se solapan gracias a sus side-conditions, por eso desde cualquier configuración no final ⟨e,ε⟩ existe a lo más un siguiente paso ⟨e′,ε′⟩: la relación → es determinista.

En otras palabras con CBV y orden L→R hay un único contexto activo y las bases están separadas por side-conditions; por tanto, desde ⟨e,ε⟩ no final hay a lo más un paso ⟨e′,ε′⟩: → es determinista.

Derivaciones completas:

(A) β-por-valor con cierre + aritmética

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

⟨App(Lambda(x, Add(x, Num(1))), Num(3)), ε⟩
→ ⟨App(Closure(x, Add(x, Num(1)), ε), Num(3)), ε⟩ (Lam-Clos)
→ ⟨Add(x, Num(1)), ε[x → Num(3)]⟩ (App-βv)
→ ⟨Add(Num(3), Num(1)), ε[x → Num(3)]⟩ (Lookup-Hit + Bin-Left)
→ ⟨Num(4), ε[x → Num(3)]⟩ (Add-Num)

(B) fst correcto y error por tipo

1. **Correcto:**

⟨Fst(Pair(Num(1), Num(2))), ε⟩
→ ⟨Num(1), ε⟩ (Fst-Pair)

2. **Error por tipo:**

⟨Fst(Num(7)), ε⟩
→ error (Fst-Err)

(C) cond desazucarado a if +

corto-circuito Expresión concreta:

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

[Else => Num(20)])

Desazúcar al núcleo :
If(Eq(Num(2), Num(2)), Num(10), Num(20))

⟨If(Eq(Num(2), Num(2)), Num(10), Num(20)), ε⟩
→ ⟨If(Bool(True), Num(10), Num(20)), ε⟩ (Eq-Num)
→ ⟨Num(10), ε⟩ (If-True)

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 = { Num(n) | n ∈ Z } ∪ { Boolean(b) | b ∈ {True, False} } ∪ { Pair(v1, v2) | v1, v2 ∈ F } ∪ { Closure(p, c, ε0) | p : String, c ∈ E, ε0 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 libre")

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).

Funcion 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, $\langle e, \varepsilon \rangle \rightarrow \langle e', \varepsilon \rangle$; 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 (Nil, env) = (Nil, env)

smallStep ((Id x), env) =

                    let v = lookupHit x env

                    in (returnASA(v), env)

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)`; da un paso pequeño con la regla “App-Arg”, lo que genera

`((AppV (Closure p c env0) a'), env)`, respetando la semántica descrita. Finalmente, `doClosure` aplicado a `((AppV (Closure p c env0) a), env)`, ya con “a” un valor canónico; da un paso con la regla “App-βv”, generando `(c, (p, toASAV(a)):env0)`, 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.

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

- *[10] M. Soto Romero, *Sintaxis Concreta*, Facultad de Ciencias UNAM, 2025. Retrieved from https://lambdasspace.github.io/LDP/notas/ldp_n04.pdf*
- *[11] M. Gabbrielli and S. Martini, "Programming Languages: Principles and Paradigms", Springer, 2023.*
- *[12] 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.*
- *[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] Lipovača, M. (2011). Learn you a Haskell for great good! Retrieved from https://learnyouahaskell.github.io/starting-out.html*