

**Universidad Nacional Autónoma de México**

Facultad de Ciencias

**Lenguajes de Programación**

Proyecto 01

**MiniLisp**

Giovanni Alejandri Espinosa (321037293)

Vania Zoë Velázquez Barrientos (321086208)

Camila Sánchez Flores (321174387)

30 de octubre de 2025

# Índice

<b>1. Introducción</b>	<b>3</b>
1.1. Motivación . . . . .	3
1.2. Objetivos . . . . .	3
<b>2. Formalización del Lenguaje</b>	<b>3</b>
2.1. Sintaxis Léxica . . . . .	3
2.2. Sintaxis Libre de Contexto (EBNF) . . . . .	4
2.3. Sintaxis Abstracta . . . . .	5
2.3.1. AST de Superficie (Surface) . . . . .	5
2.3.2. AST del Núcleo (Core) . . . . .	5
2.4. Eliminación de Azúcar Sintáctica . . . . .	6
2.4.1. Reglas Básicas . . . . .	6
2.4.2. Operadores Variádicos . . . . .	7
2.4.3. Operadores Unarios . . . . .	7
2.4.4. Condicionales . . . . .	7
2.4.5. Bindings . . . . .	7
2.4.6. Funciones . . . . .	7
2.4.7. Listas . . . . .	8
2.4.8. Proyecciones . . . . .	8
2.5. Semántica Operacional Estructural (SOS) . . . . .	8
2.5.1. Valores . . . . .	8
2.5.2. Reglas Aritméticas . . . . .	8
2.5.3. Reglas de Comparación . . . . .	9
2.5.4. Reglas Lógicas . . . . .	9
2.5.5. Reglas Condicionales . . . . .	9
2.5.6. Reglas de Binding . . . . .	9
2.5.7. Reglas de Funciones . . . . .	10
2.5.8. Reglas de Pares . . . . .	10
2.5.9. Ejemplo de Derivación Completa . . . . .	10
<b>3. Justificación de Decisiones de Diseño</b>	<b>11</b>
3.1. Elección de EBNF . . . . .	11
3.2. Separación Superficie/Núcleo . . . . .	11
3.3. Small-Step Semantics . . . . .	11
3.4. Call-by-Value . . . . .	11
3.5. Currificación Automática . . . . .	12
<b>4. Referencias Bibliográficas</b>	<b>12</b>
<b>5. Implementación en Haskell</b>	<b>12</b>
5.1. Estructura del Proyecto . . . . .	12
5.2. AST del Núcleo (AST.hs) . . . . .	13
5.3. AST de Superficie (SurfaceAST.hs) . . . . .	14
5.4. Módulo de Desazucarización (Desugar.hs) . . . . .	15
5.5. Módulo de Evaluación (Eval.hs) . . . . .	19
5.6. Parser (Parser.hs) . . . . .	24

5.7.	Main (Main.hs) . . . . .	29
5.8.	Ejemplos de Programas . . . . .	34
5.8.1.	Suma de primeros n naturales (sum.minisp) . . . . .	34
5.8.2.	Factorial (factorial.minisp) . . . . .	35
5.8.3.	Fibonacci (fibonacci.minisp) . . . . .	35
5.8.4.	Map para listas (map.minisp) . . . . .	35
5.8.5.	Filter para listas (filter.minisp) . . . . .	36
5.9.	README.md . . . . .	36
5.9.1.	Características Implementadas . . . . .	36
5.10.	Instalación . . . . .	36
5.10.1.	Requisitos . . . . .	36
5.10.2.	Pasos de Instalación . . . . .	37
5.11.	Uso . . . . .	37
5.11.1.	REPL Interactivo . . . . .	37
5.11.2.	Ejecutar Archivos . . . . .	37
5.12.	Ejemplos . . . . .	37
5.12.1.	Factorial (factorial.minisp) . . . . .	37
5.12.2.	Suma 1..n (sum.minisp) . . . . .	37
5.12.3.	Fibonacci (fibonacci.minisp) . . . . .	38
5.12.4.	Potencia (power.minisp) . . . . .	38
5.12.5.	MCD - Máximo Común Divisor (mcd.minisp) . . . . .	38
5.13.	Arquitectura del Sistema . . . . .	38
5.14.	Archivo de Configuración (minilisp.cabal) . . . . .	38
5.15.	Conclusiones . . . . .	40

# 1 Introducción

## 1.1 Motivación

El estudio formal de los lenguajes de programación trasciende el mero uso práctico de herramientas computacionales. La ciencia de la computación exige comprender los lenguajes como objetos matemáticos susceptibles de análisis riguroso, permitiendo razonar sobre programas con precisión y garantizar la corrección de sistemas software.

El presente proyecto aborda la formalización e implementación de MiniLisp, un subconjunto de Lisp diseñado con fines pedagógicos. A través de este trabajo, se recorren las etapas fundamentales del diseño de lenguajes: desde la especificación de su sintaxis léxica y libre de contexto, pasando por la eliminación de azúcar sintáctica, hasta la definición de su semántica operacional estructural.

## 1.2 Objetivos

**Objetivo General:** Formalizar exhaustivamente la sintaxis y semántica de MiniLisp y trasladar esta formalización a una implementación concreta en Haskell, garantizando coherencia entre el modelo teórico y su materialización práctica.

**Objetivos Específicos:**

1. Definir la sintaxis léxica y libre de contexto de MiniLisp
2. Establecer la sintaxis abstracta distinguiendo superficie y núcleo
3. Formalizar reglas de desazucarización sistemática
4. Especificar la semántica operacional mediante small-step semantics
5. Implementar un intérprete funcional en Haskell
6. Validar el sistema mediante casos de prueba exhaustivos

# 2 Formalización del Lenguaje

## 2.1 Sintaxis Léxica

La sintaxis léxica define las unidades básicas (tokens) mediante expresiones regulares:

```
1 IDENTIFICADOR ::= [a-zA-Z_][a-zA-Z0-9_\-?!]*
2 ENTERO ::= -?[0-9]+
3 BOOLEANO ::= #t | #f
4 LPAREN ::= (
5 RPAREN ::= )
6 LBRACKET ::= [
7 RBRACKET ::= ]
8 COMMA ::= ,
9 WHITESPACE ::= [\t\n\r]+
10 COMMENT ::= ;[^\\n]*\\n
11
12 PALABRAS_RESERVADAS ::= let | let* | letrec | if | if0 | lambda | cond
13 | else | fst | snd | head | tail | add1 | sub1 | sqrt | expt | not
```

**Justificación:** Se adopta la convención de Scheme para identificadores, permitiendo caracteres especiales comunes en programación funcional (-, ?, !). Los booleanos siguen el estándar #t/#f de Scheme. Los comentarios usan ; como es tradicional en dialectos Lisp.

## 2.2 Sintaxis Libre de Contexto (EBNF)

```

1 <Program> ::= <Expr>
2
3 <Expr> ::= <Atom> | <Pair> | <List> | <ArithOp> | <CompOp> | <LogicOp>
   | <Conditional> | <Binding> | <Function> | <Application> | <
   Projection> | <ListOp>
4
5 <Atom> ::= IDENTIFICADOR | ENTERO | BOOLEANO
6
7 <Pair> ::= "(" <Expr> "," <Expr> ")"
8 <List> ::= "[" "]" | "[" <Expr> ( "," <Expr> )* "]"
9
10 <ArithOp> ::= "(" <ArithOp2> <Expr> <Expr>+ ")"
11             | "(" <ArithOp1> <Expr> ")"
12
13 <ArithOp2> ::= "+" | "-" | "*" | "/"
14 <ArithOp1> ::= "add1" | "sub1" | "sqrt"
15
16 <CompOp> ::= "(" <Comparator> <Expr> <Expr>+ ")"
17 <Comparator> ::= "=" | "<" | ">" | ">=" | "<=" | "!="
18
19 <LogicOp> ::= "(" "not" <Expr> ")"
20
21 <Conditional> ::= "(" "if" <Expr> <Expr> <Expr> ")"
22                | "(" "if0" <Expr> <Expr> <Expr> ")"
23                | <Cond>
24
25 <Cond> ::= "(" "cond" <Clause>+ <ElseClause> ")"
26 <Clause> ::= "[" <Expr> <Expr> "]"
27 <ElseClause> ::= "[" "else" <Expr> "]"
28
29 <Binding> ::= "(" "let" "(" <Bind>+ ")" <Expr> ")"
30             | "(" "let*" "(" <Bind>+ ")" <Expr> ")"
31             | "(" "letrec" "(" <Bind>+ ")" <Expr> ")"
32
33 <Bind> ::= "(" IDENTIFICADOR <Expr> ")"
34
35 <Function> ::= "(" "lambda" "(" IDENTIFICADOR* ")" <Expr> ")"
36
37 <Application> ::= "(" <Expr> <Expr>* ")"
38
39 <Projection> ::= "(" "fst" <Expr> ")"
40                | "(" "snd" <Expr> ")"
41
42 <ListOp> ::= "(" "head" <Expr> ")"
43             | "(" "tail" <Expr> ")"
44             | "(" "expt" <Expr> <Expr> ")"

```

Notas sobre la gramática:

- $\langle Expr \rangle +$  denota una o más repeticiones
- $\langle Expr \rangle *$  denota cero o más repeticiones
- La gramática es ambigua en  $\langle Application \rangle$ , se resuelve durante el parsing mediante precedencia

## 2.3 Sintaxis Abstracta

### 2.3.1. AST de Superficie (Surface)

Representa todas las construcciones del lenguaje incluyendo azúcar sintáctica:

```

1 data SExpr = SVar String
2   | SInt Integer
3   | SBool Bool
4   | SPair SExpr SExpr
5   | SList [SExpr]
6   -- Operadores vari dicos
7   | SAdd [SExpr]
8   | SSub [SExpr]
9   | SMul [SExpr]
10  | SDiv [SExpr]
11  | SEq [SExpr]
12  | SLt [SExpr]
13  | SGt [SExpr]
14  | SLe [SExpr]
15  | SGe [SExpr]
16  | SNe [SExpr]
17  -- Operadores unarios
18  | SAdd1 SExpr
19  | SSub1 SExpr
20  | SSqrt SExpr
21  | SNot SExpr
22  -- Operadores binarios especiales
23  | SExpt SExpr SExpr
24  -- Condicionales
25  | SIf SExpr SExpr SExpr
26  | SIf0 SExpr SExpr SExpr
27  | SCond [(SExpr, SExpr)] SExpr
28  -- Bindings
29  | SLet [(String, SExpr)] SExpr
30  | SLetStar [(String, SExpr)] SExpr
31  | SLetRec [(String, SExpr)] SExpr
32  -- Funciones
33  | SLambda [String] SExpr
34  | SApp SExpr [SExpr]
35  -- Proyecciones
36  | SFst SExpr
37  | SSnd SExpr
38  | SHead SExpr
39  | STail SExpr

```

### 2.3.2. AST del Núcleo (Core)

El núcleo contiene solo las construcciones primitivas esenciales:

```

1 data Expr = Var String
2   | IntLit Integer
3   | BoolLit Bool
4   | Pair Expr Expr
5   | Nil
6   | Cons Expr Expr
7   -- Operadores binarios    nicamente
8   | Add Expr Expr
9   | Sub Expr Expr
10  | Mul Expr Expr
11  | Div Expr Expr
12  | Eq Expr Expr
13  | Lt Expr Expr
14  -- Operadores unarios
15  | Not Expr
16  -- Condicional    nico
17  | If Expr Expr Expr
18  -- Binding    nico
19  | Let String Expr Expr
20  -- Funci n currificada
21  | Lambda String Expr
22  | App Expr Expr
23  -- Proyecciones
24  | Fst Expr
25  | Snd Expr

```

### Decisiones de diseño del núcleo:

1. Operadores binarios únicamente: Todos los operadores variádicos se desazucarizan
2. Funciones de un parámetro: Lambda currificada
3. Let simple: let\* y letrec se traducen a let
4. Condicional booleano: if0 se traduce a if
5. Listas como pares: Cons + Nil (estilo tradicional Lisp)

## 2.4 Eliminación de Azúcar Sintáctica

Definimos la función de desazucarización:  $\cdot_s : \text{SExpr} \rightarrow \text{Expr}$

### 2.4.1. Reglas Básicas

- [DS-VAR] Variables:  $x_s = x$
- [DS-INT] Enteros:  $n_s = n$
- [DS-BOOL] Booleanos:  $b_s = b$
- [DS-PAIR] Pares:  $(e_1, e_2)_s = \text{Pair } e_{1s} e_{2s}$

### 2.4.2. Operadores Variádicos

- [DS-ADD-BIN] Suma binaria (caso base):  $(+e_1e_2)_s = \text{Add } e_{1s}e_{2s}$
- [DS-ADD-VAR] Suma variádica ( $n \geq 3$ ):  $(+e_1e_2 \dots e_n)_s = \text{Add } e_{1s}(+e_2 \dots e_n)_s$
- [DS-EQ-BIN] Igualdad binaria:  $(= e_1e_2)_s = \text{Eq } e_{1s}e_{2s}$
- [DS-EQ-VAR] Igualdad variádica:  $(= e_1e_2e_3 \dots e_n)_s = \text{If } (\text{Eq } e_{1s}e_{2s})(= e_2e_3 \dots e_n)_s(\text{BoolLit False})_s$
- [DS-LT-VAR] Menor que variádico:  $(< e_1e_2e_3 \dots e_n)_s = \text{If } (\text{Lt } e_{1s}e_{2s})(< e_2e_3 \dots e_n)_s(\text{BoolLit False})_s$

### 2.4.3. Operadores Unarios

- [DS-ADD1] Incremento:  $(\text{add1 } e)_s = \text{Add } e_s(\text{IntLit } 1)_s$
- [DS-SUB1] Decremento:  $(\text{sub1 } e)_s = \text{Sub } e_s(\text{IntLit } 1)_s$
- [DS-SQRT] Raíz cuadrada:  $(\text{sqrt } e)_s = (\text{expt } e0,5)_s$
- [DS-EXPT] Potencia:  $(\text{expt base exp})_s = \text{If } (\text{Eq } \text{exp}_s(\text{IntLit } 0)) (\text{IntLit } 1) (\text{Mul base}_s(\text{expt base exp})_s)$

### 2.4.4. Condicionales

- [DS-IF] If booleano:  $(\text{if } c \text{ t } e)_s = \text{If } c_s t_s e_s$
- [DS-IF0] If0 a if:  $(\text{if0 } e \text{ t } f)_s = \text{If } (\text{Eq } e_s(\text{IntLit } 0)) t_s f_s$
- [DS-COND-BASE] Cond caso base:  $(\text{cond } [\text{else } e])_s = e_s$
- [DS-COND-REC] Cond recursivo:  $(\text{cond } [g1 \text{ e1}] \dots [gn \text{ en}] [\text{else } ee])_s = \text{If } g1_s e1_s (\text{cond } [g2 \text{ e2}] \dots [gn \text{ en}] [\text{else } ee])_s$

### 2.4.5. Bindings

- [DS-LET-SINGLE] Let simple:  $(\text{let } ((x \text{ e})) \text{ body})_s = \text{Let } x e_s \text{ body}_s$
- [DS-LET-MULTI] Let multiple:  $(\text{let } ((x_1 \text{ e}_1) \dots (x_n \text{ e}_n)) \text{ body})_s = \text{App } (\text{Lambda } x_1 (\dots (\text{Lambda } x_n \text{ body}) \dots)) e_{1s} \dots e_{ns}$
- [DS-LETSTAR-BASE] Let\* caso base:  $(\text{let}^* ((x \text{ e})) \text{ body})_s = \text{Let } x e_s \text{ body}_s$
- [DS-LETSTAR-REC] Let\* recursivo:  $(\text{let}^* ((x_1 \text{ e}_1)(x_2 \text{ e}_2) \dots (x_n \text{ e}_n)) \text{ body})_s = \text{Let } x_1 e_{1s} (\text{let}^* ((x_2 \text{ e}_2) \dots (x_n \text{ e}_n)) \text{ body})_s$
- [DS-LETREC] Letrec usando combinador Y:  $(\text{letrec } ((f \text{ e})) \text{ body})_s = \text{Let } f (\text{Fix } (\text{Lambda } f e_s)) \text{ body}_s$

### 2.4.6. Funciones

- [DS-LAMBDA-ZERO] Lambda sin parámetros:  $(\text{lambda } () \text{ body})_s = \text{Lambda } \_ \text{body}_s$
- [DS-LAMBDA-ONE] Lambda un parámetro:  $(\text{lambda } (x) \text{ body})_s = \text{Lambda } x \text{ body}_s$
- [DS-LAMBDA-CURRY] Lambda currificada ( $n \geq 2$ ):  $(\text{lambda } (x_1 \text{ } x_2 \dots x_n) \text{ body})_s = \text{Lambda } x_1 (\text{Lambda } x_2 (\dots (\text{Lambda } x_n \text{ body}_s \dots)))$
- [DS-APP-ZERO] Aplicación sin argumentos:  $(f)_s = \text{App } f_s(\text{BoolLit True})_s$



- [DS-APP-ONE] Aplicación un argumento:  $(f\ e)_s = \text{App } f_s e_s$
- [DS-APP-MULTI] Aplicación múltiple:  $(f\ e_1\ e_2 \dots e_n)_s = \text{App } (\text{App } \dots (\text{App } f_s e_{1s}) \dots e_{n-1s}) e_{ns}$

#### 2.4.7. Listas

- [DS-NIL] Lista vacía:  $[]_s = \text{Nil}$
- [DS-LIST-SINGLE] Lista un elemento:  $[e]_s = \text{Cons } e_s \text{Nil}$
- [DS-LIST-MULTI] Lista múltiple:  $[e_1, e_2, \dots, e_n]_s = \text{Cons } e_{1s} [e_2, \dots, e_n]_s$
- [DS-HEAD] Cabeza de lista:  $(\text{head } e)_s = \text{Fst } e_s$
- [DS-TAIL] Cola de lista:  $(\text{tail } e)_s = \text{Snd } e_s$

#### 2.4.8. Proyecciones

- [DS-FST] Primera proyección:  $(\text{fst } e)_s = \text{Fst } e_s$
- [DS-SND] Segunda proyección:  $(\text{snd } e)_s = \text{Snd } e_s$

### 2.5 Semántica Operacional Estructural (SOS)

Adoptamos **small-step semantics** con la notación:  $e \rightarrow e'$

Un juicio de transición  $e \rightarrow e'$  significa que la expresión  $e$  reduce en un paso a  $e'$ .

#### 2.5.1. Valores

Definimos el conjunto de valores (formas normales):

$$v ::= n \mid \#t \mid \#f \mid (v_1, v_2) \mid \text{nil} \mid (\text{cons } v_1\ v_2) \mid (\text{lambda } (x)\ e)$$

#### 2.5.2. Reglas Aritméticas

- [E-ADD] Suma de valores:  $\frac{}{n_1 + n_2 \rightarrow n_3}$  donde  $n_3 = n_1 + n_2$
- [E-ADD-L] Reducir operando izquierdo:  $\frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2}$
- [E-ADD-R] Reducir operando derecho:  $\frac{e_2 \rightarrow e'_2}{v_1 + e_2 \rightarrow v_1 + e'_2}$
- [E-DIV-ZERO] División por cero:  $\frac{}{v/0 \rightarrow \text{ERROR}}$

**Reglas análogas para:** Sub, Mul, Div

### 2.5.3. Reglas de Comparación

- [E-EQ-TRUE] Igualdad verdadera:  $\frac{}{n = n \rightarrow \#t}$
- [E-EQ-FALSE] Igualdad falsa:  $\frac{n_1 \neq n_2}{n_1 = n_2 \rightarrow \#f}$
- [E-EQ-L] Reducir izquierda:  $\frac{e_1 \rightarrow e'_1}{e_1 = e_2 \rightarrow e'_1 = e_2}$
- [E-EQ-R] Reducir derecha:  $\frac{e_2 \rightarrow e'_2}{v_1 = e_2 \rightarrow v_1 = e'_2}$

Reglas análogas para: Lt, Gt (con comparación numérica)

### 2.5.4. Reglas Lógicas

- [E-NOT-TRUE]  $\frac{}{\text{not } \#t \rightarrow \#f}$
- [E-NOT-FALSE]  $\frac{}{\text{not } \#f \rightarrow \#t}$
- [E-NOT-STEP]  $\frac{e \rightarrow e'}{\text{not } e \rightarrow \text{not } e'}$

### 2.5.5. Reglas Condicionales

- [E-IF-TRUE]  $\frac{}{\text{if } \#t \ e_2 \ e_3 \rightarrow e_2}$
- [E-IF-FALSE]  $\frac{}{\text{if } \#f \ e_2 \ e_3 \rightarrow e_3}$
- [E-IF-COND]  $\frac{e_1 \rightarrow e'_1}{\text{if } e_1 \ e_2 \ e_3 \rightarrow \text{if } e'_1 \ e_2 \ e_3}$
- [E-IF-ERROR] Condición no booleana:  $\frac{v \notin \{\#t, \#f\}}{\text{if } v \ e_2 \ e_3 \rightarrow \text{ERROR}}$

### 2.5.6. Reglas de Binding

- [E-LET] Let con valor:  $\frac{}{\text{let } x = v \text{ in } e \rightarrow e[x := v]}$
- [E-LET-STEP] Reducir definición:  $\frac{e_1 \rightarrow e'_1}{\text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e'_1 \text{ in } e_2}$

Donde  $e[x := v]$  denota sustitución captura-evitando.

### 2.5.7. Reglas de Funciones

- [E-APP-BETA] Beta-reducción:  $\frac{}{((\text{lambda } (x) \ e_1) \ v_2) \rightarrow e_1[x := v_2]}$
- [E-APP-FUN] Reducir función:  $\frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2}$
- [E-APP-ARG] Reducir argumento:  $\frac{e_2 \rightarrow e'_2}{v_1 \ e_2 \rightarrow v_1 \ e'_2}$
- [E-APP-ERROR] Aplicar no-función:  $\frac{v_1 \notin (\text{lambda } \dots)}{v_1 \ v_2 \rightarrow \text{ERROR}}$

### 2.5.8. Reglas de Pares

- [E-PAIR-L] Reducir componente izquierdo:  $\frac{e_1 \rightarrow e'_1}{(e_1, e_2) \rightarrow (e'_1, e_2)}$
- [E-PAIR-R] Reducir componente derecho:  $\frac{e_2 \rightarrow e'_2}{(v_1, e_2) \rightarrow (v_1, e'_2)}$
- [E-FST] Primera proyección:  $\frac{}{\text{fst } (v_1, v_2) \rightarrow v_1}$
- [E-FST-STEP] Reducir par:  $\frac{e \rightarrow e'}{\text{fst } e \rightarrow \text{fst } e'}$
- [E-SND] Segunda proyección:  $\frac{}{\text{snd } (v_1, v_2) \rightarrow v_2}$
- [E-SND-STEP]  $\frac{e \rightarrow e'}{\text{snd } e \rightarrow \text{snd } e'}$

### 2.5.9. Ejemplo de Derivación Completa

**Programa:**  $(+2(*34))$

**Árbol de derivación:**

$$\frac{\frac{}{(*34) \rightarrow 12} \quad \frac{\frac{}{(+212) \rightarrow 14}}{(+2(*34)) \rightarrow (+212)}}{(+2(*34)) \rightarrow^* 14}$$

**Programa:**  $(\text{let } ((x5))(+x3))$

**Árbol de derivación:**

$$\frac{\frac{}{(\text{let } ((x5))(+x3)) \rightarrow (+53)} \quad \frac{}{(+53) \rightarrow 8}}{(\text{let } ((x5))(+x3)) \rightarrow^* 8}$$

## 3 Justificación de Decisiones de Diseño

### 3.1 Elección de EBNF

Se utiliza EBNF (Extended Backus-Naur Form) por ser un estándar reconocido en la especificación de sintaxis de lenguajes de programación [Aho et al., 2007]. La notación permite expresar repeticiones (+, \*) de forma concisa sin necesidad de reglas recursivas adicionales.

### 3.2 Separación Superficie/Núcleo

La distinción entre sintaxis de superficie y núcleo es fundamental en el diseño de lenguajes [Felleisen et al., 1996]. El núcleo minimalista simplifica:

- La especificación de la semántica operacional
- La corrección del intérprete
- La verificación formal de propiedades

### 3.3 Small-Step Semantics

Se adopta small-step semantics (SOS de paso pequeño) por las siguientes razones [Plotkin, 1981]:

- **Precisión:** Captura el proceso de evaluación paso a paso
- **Depuración:** Permite observar estados intermedios
- **Formalización:** Facilita razonamiento sobre propiedades de terminación
- **Modularidad:** Las reglas son composicionales

### 3.4 Call-by-Value

Se implementa evaluación ansiosa (eager evaluation) con estrategia call-by-value porque:

- Es la estrategia estándar en Scheme/Racket
- Simplifica el modelo de ejecución
- Permite efectos secundarios predecibles
- Es la estrategia más común en lenguajes funcionales estrictos

### 3.5 Currificación Automática

Las funciones multi-parámetro se currifican automáticamente porque:

- Unifica el modelo teórico (lambda cálculo admite solo un parámetro)
- Permite aplicación parcial de funciones
- Simplifica la semántica operacional
- Es consistente con lenguajes funcionales como Haskell

## 4 Referencias Bibliográficas

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2007.

Matthias Felleisen and Daniel P. Friedman. *A Little Java, a Few Patterns*. MIT Press, Cambridge, MA, 1996.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016.

Gordon D. Plotkin. *A structural approach to operational semantics*. Technical Report FN-19, DAIMI, Computer Science Department, Aarhus University, 1981.

Michael Sperber et al. *Revised<sup>6</sup> Report on the Algorithmic Language Scheme*. Cambridge University Press, 2009.

|

## 5 Implementación en Haskell

### 5.1 Estructura del Proyecto

```
1 minilisp/  
2   src/  
3       AST.hs           -- Definición de AST (Core)  
4       SurfaceAST.hs    -- AST de superficie  
5       Parser.hs        -- Analizador sintáctico  
6       Desugar.hs       -- Eliminación de azúcar  
7       Eval.hs          -- Interpretador/evaluador  
8       Main.hs          -- REPL y entrada principal  
9   examples/  
10      sum.minisp        -- Suma de números naturales  
11      factorial.minisp   -- Factorial  
12      fibonacci.minisp  -- Fibonacci  
13      map.minisp        -- Map para listas  
14      filter.minisp     -- Filter para listas  
15   test/  
16      Tests.hs         -- Suite de pruebas
```

```

17      README.md
18      minilisp.cabal
19      Proyecto01.pdf

```

## 5.2 AST del Núcleo (AST.hs)

```

1  {-# LANGUAGE DeriveGeneric #-}
2  {-# LANGUAGE DeriveAnyClass #-}
3
4  module AST
5      ( Expr(..)
6      , prettyExpr
7      ) where
8
9  import GHC.Generics (Generic)
10 import Control.DeepSeq (NFData)
11
12 -- | Sintaxis Abstracta del Núcleo de MiniLisp
13 -- Todas las construcciones aquí son primitivas
14 data Expr
15     -- Valores atómicos
16     = Var String           -- Variable
17     | IntLit Integer       -- Literal entera
18     | BoolLit Bool         -- Literal booleana
19     -- Estructuras de datos
20     | Pair Expr Expr       -- Par ordenado (e1, e2)
21     | Nil                  -- Lista vacía
22     | Cons Expr Expr       -- Constructor de lista
23     -- Operadores aritméticos binarios
24     | Add Expr Expr        -- Suma
25     | Sub Expr Expr        -- Resta
26     | Mul Expr Expr        -- Multiplicación
27     | Div Expr Expr        -- División
28     -- Operadores de comparación binarios
29     | Eq Expr Expr         -- Igualdad
30     | Lt Expr Expr         -- Menor que
31     | Gt Expr Expr         -- Mayor que
32     | Le Expr Expr         -- Menor o igual
33     | Ge Expr Expr         -- Mayor o igual
34     | Ne Expr Expr         -- Diferente
35     -- Operadores lógicos
36     | Not Expr             -- Negación
37     -- Control de flujo
38     | If Expr Expr Expr    -- Condicional
39     -- Binding
40     | Let String Expr Expr -- Definición local
41     -- Funciones (curricadas)
42     | Lambda String Expr   -- Función anónima
43     | App Expr Expr        -- Aplicación
44     -- Proyecciones
45     | Fst Expr             -- Primera proyección
46     | Snd Expr             -- Segunda proyección
47     deriving (Eq, Show, Generic, NFData)
48
49 -- | Pretty printer para expresiones del núcleo
50 prettyExpr :: Expr -> String

```

```

51 prettyExpr = go 0
52   where
53     indent n = replicate (n * 2) ' '
54
55     go _ (Var x) = x
56     go _ (IntLit n) = show n
57     go _ (BoolLit True) = "#t"
58     go _ (BoolLit False) = "#f"
59     go _ Nil = "nil"
60     go n (Pair e1 e2) =
61       "(" ++ go n e1 ++ ", " ++ go n e2 ++ ")"
62     go n (Cons e1 e2) =
63       "(cons " ++ go n e1 ++ " " ++ go n e2 ++ ")"
64     go n (Add e1 e2) =
65       "(+ " ++ go n e1 ++ " " ++ go n e2 ++ ")"
66     go n (Sub e1 e2) =
67       "(- " ++ go n e1 ++ " " ++ go n e2 ++ ")"
68     go n (Mul e1 e2) =
69       "(* " ++ go n e1 ++ " " ++ go n e2 ++ ")"
70     go n (Div e1 e2) =
71       "(/ " ++ go n e1 ++ " " ++ go n e2 ++ ")"
72     go n (Eq e1 e2) =
73       "(= " ++ go n e1 ++ " " ++ go n e2 ++ ")"
74     go n (Lt e1 e2) =
75       "< " ++ go n e1 ++ " " ++ go n e2 ++ ")"
76     go n (Not e) =
77       "(not " ++ go n e ++ ")"
78     go n (If c t e) =
79       "(if " ++ go n c ++ "\n" ++
80         indent (n+1) ++ go (n+1) t ++ "\n" ++
81         indent (n+1) ++ go (n+1) e ++ ")"
82     go n (Let x e1 e2) =
83       "(let " ++ x ++ " " ++ go n e1 ++ "\n" ++
84         indent (n+1) ++ go (n+1) e2 ++ ")"
85     go n (Lambda x body) =
86       "(lambda (" ++ x ++ ") " ++ go n body ++ ")"
87     go n (App e1 e2) =
88       "(" ++ go n e1 ++ " " ++ go n e2 ++ ")"
89     go n (Fst e) = "(fst " ++ go n e ++ ")"
90     go n (Snd e) = "(snd " ++ go n e ++ ")"
91     go n e = show e

```

### 5.3 AST de Superficie (SurfaceAST.hs)

```

1 {-# LANGUAGE DeriveGeneric #-}
2
3 module SurfaceAST
4   ( SExpr(..)
5   , Binding
6   ) where
7
8 import GHC.Generics (Generic)
9
10 type Binding = (String, SExpr)
11
12 -- | Sintaxis Abstracta de Superficie

```

```

13 -- Incluye todas las extensiones y az car sint ctica
14 data SExpr
15   -- Valores at micos
16   = SVar String
17   | SInt Integer
18   | SBool Bool
19   -- Estructuras de datos
20   | SPair SExpr SExpr
21   | SList [SExpr]
22   -- Operadores aritm ticos vari dicos
23   | SAdd [SExpr]      -- (+ e1 e2 ... en)
24   | SSub [SExpr]      -- (- e1 e2 ... en)
25   | SMul [SExpr]      -- (* e1 e2 ... en)
26   | SDiv [SExpr]      -- (/ e1 e2 ... en)
27   -- Operadores aritm ticos unarios
28   | SAdd1 SExpr      -- (add1 e)
29   | SSub1 SExpr      -- (sub1 e)
30   | SSqrt SExpr      -- (sqrt e)
31   -- Operador de potencia
32   | SExpt SExpr SExpr -- (expt base exp)
33   -- Operadores de comparaci n vari dicos
34   | SEq [SExpr]      -- (= e1 e2 ... en)
35   | SLt [SExpr]      -- (< e1 e2 ... en)
36   | SGt [SExpr]      -- (> e1 e2 ... en)
37   | SLe [SExpr]      -- (<= e1 e2 ... en)
38   | SGe [SExpr]      -- (>= e1 e2 ... en)
39   | SNe [SExpr]      -- (!= e1 e2 ... en)
40   -- Operadores l gicos
41   | SNot SExpr      -- (not e)
42   -- Condicionales
43   | SIf SExpr SExpr SExpr -- (if cond then else)
44   | SIf0 SExpr SExpr SExpr -- (if0 e then else)
45   | SCond [(SExpr, SExpr)] SExpr -- (cond [g1 e1] ... [else ee])
46   -- Bindings
47   | SLet [Binding] SExpr -- (let ((x1 e1) ...) body)
48   | SLetStar [Binding] SExpr -- (let* ((x1 e1) ...) body)
49   | SLetRec [Binding] SExpr -- (letrec ((x1 e1) ...) body)
50   -- Funciones
51   | SLambda [String] SExpr -- (lambda (x1 ... xn) body)
52   | SApp SExpr [SExpr] -- (f e1 e2 ... en)
53   -- Proyecciones y operaciones sobre listas
54   | SFst SExpr      -- (fst e)
55   | SSnd SExpr      -- (snd e)
56   | SHead SExpr      -- (head e)
57   | STail SExpr      -- (tail e)
58   deriving (Eq, Show, Generic)

```

## 5.4 M3dulo de Desazucarizaci3n (Desugar.hs)

```

1 module Desugar
2   ( desugar
3   , desugarOp
4   ) where
5
6 import AST
7 import SurfaceAST

```



```

8 import qualified Data.List as L
9
10 -- | Funci n principal de desazucarizaci n
11 -- Implementa todas las reglas [DS-*] definidas formalmente
12 desugar :: SExpr -> Expr
13 desugar se = case se of
14   -- [DS-VAR], [DS-INT], [DS-BOOL]
15   SVar x -> Var x
16   SInt n -> IntLit n
17   SBool b -> BoolLit b
18
19   -- [DS-PAIR]
20   SPair e1 e2 -> Pair (desugar e1) (desugar e2)
21
22   -- [DS-NIL], [DS-LIST-*]
23   SList [] -> Nil
24   SList [e] -> Cons (desugar e) Nil
25   SList (e:es) -> Cons (desugar e) (desugar (SList es))
26
27   -- Operadores aritm ticos vari dicos
28   SAdd es -> desugarOp "+" es
29   SSub es -> desugarOp "-" es
30   SMul es -> desugarOp "*" es
31   SDiv es -> desugarOp "/" es
32
33   -- [DS-ADD1], [DS-SUB1]
34   SAdd1 e -> Add (desugar e) (IntLit 1)
35   SSub1 e -> Sub (desugar e) (IntLit 1)
36
37   -- [DS-SQRT] - implementado como expt e 0.5
38   -- Para enteros, usamos aproximaci n
39   SSqrt e -> desugarSqrt (desugar e)
40
41   -- [DS-EXPT]
42   SExpt base exp -> desugarExpt (desugar base) (desugar exp)
43
44   -- Operadores de comparaci n vari dicos
45   SEq es -> desugarComp Eq es
46   SLt es -> desugarComp Lt es
47   SGt es -> desugarComp Gt es
48   SLe es -> desugarComp Le es
49   SGe es -> desugarComp Ge es
50   SNe es -> desugarComp Ne es
51
52   -- [DS-NOT]
53   SNot e -> Not (desugar e)
54
55   -- [DS-IF]
56   SIf c t e -> If (desugar c) (desugar t) (desugar e)
57
58   -- [DS-IF0]
59   SIf0 e t f -> If (Eq (desugar e) (IntLit 0))
60                   (desugar t)
61                   (desugar f)
62
63   -- [DS-COND-*]
64   SCond clauses elseExpr -> desugarCond clauses elseExpr
65

```

```

66 -- [DS-LET-*]
67 Slet bindings body -> desugarLet bindings body
68
69 -- [DS-LETSTAR-*]
70 SletStar bindings body -> desugarLetStar bindings body
71
72 -- [DS-LETREC]
73 SletRec bindings body -> desugarLetRec bindings body
74
75 -- [DS-LAMBDA-*]
76 SLambda params body -> curryLambda params (desugar body)
77
78 -- [DS-APP-*]
79 SApp f args -> applyMany (desugar f) (map desugar args)
80
81 -- [DS-FST], [DS-SND]
82 SFst e -> Fst (desugar e)
83 SSnd e -> Snd (desugar e)
84
85 -- [DS-HEAD], [DS-TAIL]
86 SHead e -> Fst (desugar e) -- head = fst (cons)
87 STail e -> Snd (desugar e) -- tail = snd (cons)
88
89 -- | Desazucariza operadores aritm ticos vari dicos
90 -- [DS-ADD-BIN], [DS-ADD-VAR]
91 desugarOp :: String -> [SExpr] -> Expr
92 desugarOp op exprs = case exprs of
93   [] -> error $ "Operator " ++ op ++ " requires at least 2 arguments"
94   [_] -> error $ "Operator " ++ op ++ " requires at least 2 arguments"
95   [e1, e2] -> applyBinOp op (desugar e1) (desugar e2)
96   (e1:e2:rest) -> applyBinOp op
97                     (desugar e1)
98                     (desugarOp op (e2:rest))
99   where
100     applyBinOp "+" = Add
101     applyBinOp "-" = Sub
102     applyBinOp "*" = Mul
103     applyBinOp "/" = Div
104     applyBinOp _ = error "Unknown operator"
105
106 -- | Desazucariza operadores de comparaci n vari dicos
107 -- [DS-EQ-VAR], [DS-LT-VAR], etc.
108 desugarComp :: (Expr -> Expr -> Expr) -> [SExpr] -> Expr
109 desugarComp _ [] = error "Comparison requires at least 2 arguments"
110 desugarComp _ [_] = error "Comparison requires at least 2 arguments"
111 desugarComp cons [e1, e2] = cons (desugar e1) (desugar e2)
112 desugarComp cons (e1:e2:rest) =
113   If (cons (desugar e1) (desugar e2))
114     (desugarComp cons (e2:rest))
115     (BoolLit False)
116
117 -- | Desazucariza cond a if anidados
118 -- [DS-COND-BASE], [DS-COND-REC]
119 desugarCond :: [(SExpr, SExpr)] -> SExpr -> Expr
120 desugarCond [] elseExpr = desugar elseExpr
121 desugarCond ((guard, body):rest) elseExpr =
122   If (desugar guard)
123     (desugar body)

```

```

124     (desugarCond rest elseExpr)
125
126 -- | Desazucariza let a lambdas
127 -- [DS-LET-SINGLE], [DS-LET-MULTI]
128 desugarLet :: [Binding] -> SExpr -> Expr
129 desugarLet [] body = desugar body
130 desugarLet bindings body =
131     let (vars, exprs) = unzip bindings
132         lambda = foldr Lambda (desugar body) vars
133         args = map desugar exprs
134     in applyMany lambda args
135
136 -- | Desazucariza let* a let anidados
137 -- [DS-LETSTAR-BASE], [DS-LETSTAR-REC]
138 desugarLetStar :: [Binding] -> SExpr -> Expr
139 desugarLetStar [] body = desugar body
140 desugarLetStar [(x, e)] body =
141     Let x (desugar e) (desugar body)
142 desugarLetStar ((x, e):rest) body =
143     Let x (desugar e) (desugarLetStar rest body)
144
145 -- | Desazucariza letrec usando combinador Y
146 -- [DS-LETREC]
147 desugarLetRec :: [Binding] -> SExpr -> Expr
148 desugarLetRec bindings body =
149     -- Para letrec, necesitamos el combinador de punto fijo
150     -- Por simplicidad, usamos una aproximaci3n:
151     -- letrec transforma cada binding en una funci3n recursiva
152     let desugarBinding (x, e) restBody =
153         Let x (fixCombinator x (desugar e)) restBody
154     in foldr desugarBinding (desugar body) bindings
155 where
156     -- Combinador Y simplificado para call-by-value
157     fixCombinator :: String -> Expr -> Expr
158     fixCombinator f body =
159         App (Lambda f body)
160             (Lambda "_" (App (Var f) (Var "_")))
161
162 -- | Currifica lambdas multi-par metro
163 -- [DS-LAMBDA-ZERO], [DS-LAMBDA-ONE], [DS-LAMBDA-CURRY]
164 curryLambda :: [String] -> Expr -> Expr
165 curryLambda [] body = Lambda "_" body -- Unit lambda
166 curryLambda [x] body = Lambda x body
167 curryLambda (x:xs) body = Lambda x (curryLambda xs body)
168
169 -- | Aplica funci3n a m3ltiples argumentos
170 -- [DS-APP-ZERO], [DS-APP-ONE], [DS-APP-MULTI]
171 applyMany :: Expr -> [Expr] -> Expr
172 applyMany f [] = App f (BoolLit True) -- Unit application
173 applyMany f [e] = App f e
174 applyMany f (e:es) = applyMany (App f e) es
175
176 -- | Desazucariza sqrt (implementaci3n simplificada para enteros)
177 desugarSqrt :: Expr -> Expr
178 desugarSqrt e =
179     -- sqrt(n) usando el m3todo de Newton-Raphson
180     -- Para simplificar, usamos una funci3n auxiliar
181     Let "x" e

```

```

182     (Let "guess" (IntLit 1)
183       (desugarSqrtHelper (Var "x") (Var "guess")))
184   where
185     desugarSqrtHelper x guess =
186       -- if guess * guess = x then guess
187       -- else sqrt_helper x ((guess + x/guess) / 2)
188       If (Eq (Mul guess guess) x)
189         guess
190         (desugarSqrtHelper x
191           (Div (Add guess (Div x guess)) (IntLit 2)))
192
193   -- | Desazucariza expt (exponenciaci n)
194   desugarExpt :: Expr -> Expr -> Expr
195   desugarExpt base exp =
196     If (Eq exp (IntLit 0))
197       (IntLit 1)
198       (If (Lt exp (IntLit 0))
199         (error "Negative exponents not supported")
200         (Mul base (desugarExpt base (Sub exp (IntLit 1)))))

```

## 5.5 Módulo de Evaluación (Eval.hs)

```

1 {-# LANGUAGE LambdaCase #-}
2
3 module Eval
4   ( Value(..)
5   , Env
6   , emptyEnv
7   , eval
8   , evalSteps
9   , prettyValue
10  ) where
11
12 import AST
13 import qualified Data.Map.Strict as Map
14 import Control.Monad (foldM)
15
16 -- | Valores semánticos (formas normales)
17 data Value
18   = VInt Integer
19   | VBool Bool
20   | VPair Value Value
21   | VNil
22   | VCons Value Value
23   | VClosure Env String Expr -- Closure con entorno capturado
24   deriving (Eq, Show)
25
26 -- | Entorno de evaluación (binding de variables a valores)
27 type Env = Map.Map String Value
28
29 -- | Entorno vacío
30 emptyEnv :: Env
31 emptyEnv = Map.empty
32
33 -- | Evaluador principal (big-step para simplicidad de implementación)
34 -- Nota: La especificación formal usa small-step, pero big-step

```

```

35 -- es equivalente y m s eficiente para la implementaci n
36 eval :: Env -> Expr -> Either String Value
37 eval env = \case
38   -- [E-VAR]
39   Var x ->
40     case Map.lookup x env of
41       Just v -> Right v
42       Nothing -> Left $ "Unbound variable: " ++ x
43
44   -- Valores
45   IntLit n -> Right (VInt n)
46   BoolLit b -> Right (VBool b)
47   Nil -> Right VNil
48
49   -- [E-PAIR-*]
50   Pair e1 e2 -> do
51     v1 <- eval env e1
52     v2 <- eval env e2
53     return $ VPair v1 v2
54
55   -- [E-CONS]
56   Cons e1 e2 -> do
57     v1 <- eval env e1
58     v2 <- eval env e2
59     return $ VCons v1 v2
60
61   -- [E-ADD-*]
62   Add e1 e2 -> do
63     v1 <- eval env e1
64     v2 <- eval env e2
65     case (v1, v2) of
66       (VInt n1, VInt n2) -> Right $ VInt (n1 + n2)
67       _ -> Left "Type error: + expects integers"
68
69   -- [E-SUB-*]
70   Sub e1 e2 -> do
71     v1 <- eval env e1
72     v2 <- eval env e2
73     case (v1, v2) of
74       (VInt n1, VInt n2) -> Right $ VInt (n1 - n2)
75       _ -> Left "Type error: - expects integers"
76
77   -- [E-MUL-*]
78   Mul e1 e2 -> do
79     v1 <- eval env e1
80     v2 <- eval env e2
81     case (v1, v2) of
82       (VInt n1, VInt n2) -> Right $ VInt (n1 * n2)
83       _ -> Left "Type error: * expects integers"
84
85   -- [E-DIV-*], [E-DIV-ZERO]
86   Div e1 e2 -> do
87     v1 <- eval env e1
88     v2 <- eval env e2
89     case (v1, v2) of
90       (VInt _, VInt 0) -> Left "Division by zero"
91       (VInt n1, VInt n2) -> Right $ VInt (n1 `div` n2)
92       _ -> Left "Type error: / expects integers"

```

```

93
94 -- [E-EQ-*]
95 Eq e1 e2 -> do
96   v1 <- eval env e1
97   v2 <- eval env e2
98   case (v1, v2) of
99     (VInt n1, VInt n2) -> Right $ VBool (n1 == n2)
100     (VBool b1, VBool b2) -> Right $ VBool (b1 == b2)
101     _ -> Left "Type error: = expects same types"
102
103 -- [E-LT-*]
104 Lt e1 e2 -> do
105   v1 <- eval env e1
106   v2 <- eval env e2
107   case (v1, v2) of
108     (VInt n1, VInt n2) -> Right $ VBool (n1 < n2)
109     _ -> Left "Type error: < expects integers"
110
111 -- Similar para Gt, Le, Ge, Ne
112 Gt e1 e2 -> do
113   v1 <- eval env e1
114   v2 <- eval env e2
115   case (v1, v2) of
116     (VInt n1, VInt n2) -> Right $ VBool (n1 > n2)
117     _ -> Left "Type error: > expects integers"
118
119 Le e1 e2 -> do
120   v1 <- eval env e1
121   v2 <- eval env e2
122   case (v1, v2) of
123     (VInt n1, VInt n2) -> Right $ VBool (n1 <= n2)
124     _ -> Left "Type error: <= expects integers"
125
126 Ge e1 e2 -> do
127   v1 <- eval env e1
128   v2 <- eval env e2
129   case (v1, v2) of
130     (VInt n1, VInt n2) -> Right $ VBool (n1 >= n2)
131     _ -> Left "Type error: >= expects integers"
132
133 Ne e1 e2 -> do
134   v1 <- eval env e1
135   v2 <- eval env e2
136   case (v1, v2) of
137     (VInt n1, VInt n2) -> Right $ VBool (n1 /= n2)
138     (VBool b1, VBool b2) -> Right $ VBool (b1 /= b2)
139     _ -> Left "Type error: != expects same types"
140
141 -- [E-NOT-*]
142 Not e -> do
143   v <- eval env e
144   case v of
145     VBool b -> Right $ VBool (not b)
146     _ -> Left "Type error: not expects boolean"
147
148 -- [E-IF-*]
149 If cond thenE elseE -> do
150   v <- eval env cond

```

```

151     case v of
152         VBool True  -> eval env thenE
153         VBool False -> eval env elseE
154         _           -> Left "Type error: if expects boolean condition"
155
156     -- [E-LET-*]
157     Let x e1 e2 -> do
158         v1 <- eval env e1
159         eval (Map.insert x v1 env) e2
160
161     -- [E-LAMBDA]
162     Lambda x body ->
163         Right $ VClosure env x body
164
165     -- [E-APP-*]
166     App e1 e2 -> do
167         v1 <- eval env e1
168         v2 <- eval env e2
169         case v1 of
170             VClosure closureEnv param body ->
171                 eval (Map.insert param v2 closureEnv) body
172             _ -> Left "Type error: application of non-function"
173
174     -- [E-FST-*]
175     Fst e -> do
176         v <- eval env e
177         case v of
178             VPair v1 _ -> Right v1
179             VCons v1 _ -> Right v1 -- head
180             _ -> Left "Type error: fst expects pair or cons"
181
182     -- [E-SND-*]
183     Snd e -> do
184         v <- eval env e
185         case v of
186             VPair _ v2 -> Right v2
187             VCons _ v2 -> Right v2 -- tail
188             _ -> Left "Type error: snd expects pair or cons"
189
190 -- | Evaluaci n paso a paso (small-step)
191 -- Para depuraci n y visualizaci n
192 evalSteps :: Env -> Expr -> [Either String Expr]
193 evalSteps env expr = expr : go expr
194     where
195         go e = case step env e of
196             Left _ -> []
197             Right e' -> e' : go e'
198
199 -- | Un paso de evaluaci n (small-step)
200 step :: Env -> Expr -> Either String Expr
201 step env = \case
202     -- Valores no reducen
203     IntLit _ -> Left "Value"
204     BoolLit _ -> Left "Value"
205     Nil -> Left "Value"
206     Lambda _ _ -> Left "Value"
207
208     -- Variables se sustituyen

```

```

209 Var x -> case Map.lookup x env of
210   Just (VInt n) -> Right (IntLit n)
211   Just (VBool b) -> Right (BoolLit b)
212   Just VNil -> Right Nil
213   Nothing -> Left $ "Unbound: " ++ x
214   _ -> Left "Complex value"
215
216 -- Operadores: evaluar operandos primero
217 Add e1 e2 -> do
218   case (e1, e2) of
219     (IntLit n1, IntLit n2) -> Right $ IntLit (n1 + n2)
220     (IntLit _, _) -> do
221       e2' <- step env e2
222       Right $ Add e1 e2'
223     _ -> do
224       e1' <- step env e1
225       Right $ Add e1' e2
226
227 -- If eval a condici n primero
228 If (BoolLit True) e2 _ -> Right e2
229 If (BoolLit False) _ e3 -> Right e3
230 If e1 e2 e3 -> do
231   e1' <- step env e1
232   Right $ If e1' e2 e3
233
234 -- Let sustituye
235 Let x (IntLit n) body -> Right $ substExpr x (IntLit n) body
236 Let x (BoolLit b) body -> Right $ substExpr x (BoolLit b) body
237 Let x e1 e2 -> do
238   e1' <- step env e1
239   Right $ Let x e1' e2
240
241 -- Aplicaci n: beta-reducci n
242 App (Lambda x body) v@(IntLit _) -> Right $ substExpr x v body
243 App (Lambda x body) v@(BoolLit _) -> Right $ substExpr x v body
244 App e1@(Lambda _ _) e2 -> do
245   e2' <- step env e2
246   Right $ App e1 e2'
247 App e1 e2 -> do
248   e1' <- step env e1
249   Right $ App e1' e2
250
251 e -> Left $ "Cannot step: " ++ show e
252
253 -- | Sustituci n capture-avoiding
254 substExpr :: String -> Expr -> Expr -> Expr
255 substExpr var val = go
256   where
257     go (Var x) | x == var = val
258               | otherwise = Var x
259     go (IntLit n) = IntLit n
260     go (BoolLit b) = BoolLit b
261     go Nil = Nil
262     go (Pair e1 e2) = Pair (go e1) (go e2)
263     go (Cons e1 e2) = Cons (go e1) (go e2)
264     go (Add e1 e2) = Add (go e1) (go e2)
265     go (Sub e1 e2) = Sub (go e1) (go e2)
266     go (Mul e1 e2) = Mul (go e1) (go e2)

```



```

267 go (Div e1 e2) = Div (go e1) (go e2)
268 go (Eq e1 e2) = Eq (go e1) (go e2)
269 go (Lt e1 e2) = Lt (go e1) (go e2)
270 go (Gt e1 e2) = Gt (go e1) (go e2)
271 go (Le e1 e2) = Le (go e1) (go e2)
272 go (Ge e1 e2) = Ge (go e1) (go e2)
273 go (Ne e1 e2) = Ne (go e1) (go e2)
274 go (Not e) = Not (go e)
275 go (If c t e) = If (go c) (go t) (go e)
276 go (Let x e1 e2) | x == var = Let x (go e1) e2 -- No sustituir en
scope
277 | otherwise = Let x (go e1) (go e2)
278 go (Lambda x body) | x == var = Lambda x body -- No sustituir en
scope
279 | otherwise = Lambda x (go body)
280 go (App e1 e2) = App (go e1) (go e2)
281 go (Fst e) = Fst (go e)
282 go (Snd e) = Snd (go e)
283
284 -- | Pretty printer para valores
285 prettyValue :: Value -> String
286 prettyValue (VInt n) = show n
287 prettyValue (VBool True) = "#t"
288 prettyValue (VBool False) = "#f"
289 prettyValue VNil = "[]"
290 prettyValue (VPair v1 v2) = "(" ++ prettyValue v1 ++ ", " ++
prettyValue v2 ++ ")"
291 prettyValue (VCons v1 v2) = "[" ++ prettyValue v1 ++ go v2 ++ "]"
292 where
293 go VNil = ""
294 go (VCons v vr) = ", " ++ prettyValue v ++ go vr
295 go v = " | " ++ prettyValue v -- Improper list
296 prettyValue (VClosure _ x _) = "<function: " ++ x ++ ".?>"

```

## 5.6 Parser (Parser.hs)

```

1 {-# LANGUAGE OverloadedStrings #-}
2
3 module Parser
4   ( parseExpr
5   , parseProgram
6   , Parser
7   ) where
8
9 import SurfaceAST
10 import Control.Monad (void)
11 import Data.Void (Void)
12 import Text.Megaparsec
13 import Text.Megaparsec.Char
14 import qualified Text.Megaparsec.Char.Lexer as L
15 import Data.Functor (($>))
16
17 type Parser = Parsec Void String
18
19 -- | Espacios en blanco y comentarios
20 sc :: Parser ()

```

```

21 sc = L.space
22   space1
23   (L.skipLineComment ";")
24   empty
25
26 -- | Lexema: parsea y consume espacios
27 lexeme :: Parser a -> Parser a
28 lexeme = L.lexeme sc
29
30 -- | Símbolo: parsea string y consume espacios
31 symbol :: String -> Parser String
32 symbol = L.symbol sc
33
34 -- | Entre paréntesis
35 parens :: Parser a -> Parser a
36 parens = between (symbol "(") (symbol ")")
37
38 -- | Entre corchetes
39 brackets :: Parser a -> Parser a
40 brackets = between (symbol "[") (symbol "]")
41
42 -- | Palabras reservadas
43 reserved :: [String]
44 reserved =
45   [ "let", "let*", "letrec", "if", "if0", "lambda"
46   , "cond", "else", "not", "fst", "snd", "head", "tail"
47   , "add1", "sub1", "sqrt", "expt"
48   ]
49
50 -- | Identificador
51 identifier :: Parser String
52 identifier = lexeme $ try $ do
53   first <- letterChar <|> char '_'
54   rest <- many (alphaNumChar <|> char '_' <|> char '-' <|> char '?')
55   <|> char '!')
56   let name = first : rest
57   if name `elem` reserved
58   then fail $ "Reserved word: " ++ name
59   else return name
60
61 -- | Número entero
62 integer :: Parser Integer
63 integer = lexeme $ L.signed sc L.decimal
64
65 -- | Booleano
66 boolean :: Parser Bool
67 boolean = lexeme $
68   (string "#t" $> True) <|>
69   (string "#f" $> False)
70
71 -- | Expresión
72 expr :: Parser SExpr
73 expr = choice
74   [ try pairExpr
75   , listExpr
76   , atomExpr
77   , parens compoundExpr
78   ]

```

```

78
79 -- | tomo (variable, n mero, booleano)
80 atomExpr :: Parser SExpr
81 atomExpr = choice
82   [ SInt <$> integer
83     , SBool <$> boolean
84     , SVar <$> identifier
85   ]
86
87 -- | Par ordenado: (e1, e2)
88 pairExpr :: Parser SExpr
89 pairExpr = parens $ do
90   e1 <- expr
91   symbol ","
92   e2 <- expr
93   return $ SPair e1 e2
94
95 -- | Lista: [e1, e2, ..., en] o []
96 listExpr :: Parser SExpr
97 listExpr = brackets $ do
98   exprs <- expr 'sepBy' symbol ","
99   return $ SList exprs
100
101 -- | Expresiones compuestas
102 compoundExpr :: Parser SExpr
103 compoundExpr = choice
104   [ ifExpr
105     , if0Expr
106     , condExpr
107     , letExpr
108     , letStarExpr
109     , letRecExpr
110     , lambdaExpr
111     , notExpr
112     , add1Expr
113     , sub1Expr
114     , sqrtExpr
115     , exptExpr
116     , fstExpr
117     , sndExpr
118     , headExpr
119     , tailExpr
120     , opExpr
121     , appExpr
122   ]
123
124 -- | If: (if cond then else)
125 ifExpr :: Parser SExpr
126 ifExpr = do
127   symbol "if"
128   c <- expr
129   t <- expr
130   e <- expr
131   return $ SIf c t e
132
133 -- | If0: (if0 e then else)
134 if0Expr :: Parser SExpr
135 if0Expr = do

```

```

136     symbol "if0"
137     e <- expr
138     t <- expr
139     f <- expr
140     return $ SIf0 e t f
141
142 -- | Cond: (cond [g1 e1] ... [else ee])
143 condExpr :: Parser SExpr
144 condExpr = do
145     symbol "cond"
146     clauses <- many clause
147     elseClause <- elseClause
148     return $ SCond clauses elseClause
149 where
150     clause = brackets $ do
151         guard <- expr
152         body <- expr
153         return (guard, body)
154     elseClause = brackets $ do
155         symbol "else"
156         expr
157
158 -- | Let: (let ((x1 e1) ...) body)
159 letExpr :: Parser SExpr
160 letExpr = do
161     symbol "let"
162     bindings <- parens $ many binding
163     body <- expr
164     return $ SLet bindings body
165
166 -- | Let*: (let* ((x1 e1) ...) body)
167 letStarExpr :: Parser SExpr
168 letStarExpr = do
169     symbol "let*"
170     bindings <- parens $ many binding
171     body <- expr
172     return $ SLetStar bindings body
173
174 -- | LetRec: (letrec ((f1 e1) ...) body)
175 letRecExpr :: Parser SExpr
176 letRecExpr = do
177     symbol "letrec"
178     bindings <- parens $ many binding
179     body <- expr
180     return $ SLetRec bindings body
181
182 -- | Binding: (x e)
183 binding :: Parser Binding
184 binding = parens $ do
185     x <- identifier
186     e <- expr
187     return (x, e)
188
189 -- | Lambda: (lambda (x1 ... xn) body)
190 lambdaExpr :: Parser SExpr
191 lambdaExpr = do
192     symbol "lambda"
193     params <- parens $ many identifier

```

```

194     body <- expr
195     return $ SLambda params body
196
197 -- | Not: (not e)
198 notExpr :: Parser SExpr
199 notExpr = do
200     symbol "not"
201     e <- expr
202     return $ SNot e
203
204 -- | Add1: (add1 e)
205 add1Expr :: Parser SExpr
206 add1Expr = do
207     symbol "add1"
208     e <- expr
209     return $ SAdd1 e
210
211 -- | Sub1: (sub1 e)
212 sub1Expr :: Parser SExpr
213 sub1Expr = do
214     symbol "sub1"
215     e <- expr
216     return $ SSub1 e
217
218 -- | Sqrt: (sqrt e)
219 sqrtExpr :: Parser SExpr
220 sqrtExpr = do
221     symbol "sqrt"
222     e <- expr
223     return $ SSqrt e
224
225 -- | Expt: (expt base exp)
226 exptExpr :: Parser SExpr
227 exptExpr = do
228     symbol "expt"
229     base <- expr
230     exp <- expr
231     return $ SExpt base exp
232
233 -- | Fst: (fst e)
234 fstExpr :: Parser SExpr
235 fstExpr = do
236     symbol "fst"
237     e <- expr
238     return $ SFst e
239
240 -- | Snd: (snd e)
241 sndExpr :: Parser SExpr
242 sndExpr = do
243     symbol "snd"
244     e <- expr
245     return $ SSnd e
246
247 -- | Head: (head e)
248 headExpr :: Parser SExpr
249 headExpr = do
250     symbol "head"
251     e <- expr

```

```

252     return $ SHead e
253
254 -- | Tail: (tail e)
255 tailExpr :: Parser SExpr
256 tailExpr = do
257     symbol "tail"
258     e <- expr
259     return $ STail e
260
261 -- | Operadores: (+/-/*/= e1 e2 ...)
262 opExpr :: Parser SExpr
263 opExpr = do
264     op <- choice
265     [ symbol "+" $> SAdd
266     , symbol "-" $> SSub
267     , try (symbol "*") $> SMul
268     , symbol "/" $> SDiv
269     , try (symbol "=") $> SEq
270     , try (symbol "<=") $> SLe
271     , try (symbol ">=") $> SGe
272     , try (symbol "!=") $> SNe
273     , symbol "<" $> SLt
274     , symbol ">" $> SGt
275     ]
276     exprs <- some expr
277     return $ op exprs
278
279 -- | Aplicaci n: (f e1 ... en)
280 appExpr :: Parser SExpr
281 appExpr = do
282     f <- expr
283     args <- many expr
284     return $ SApp f args
285
286 -- | Parsea expresi n completa
287 parseExpr :: String -> Either String SExpr
288 parseExpr input =
289     case runParser (sc *> expr <* eof) "" input of
290     Left err -> Left $ errorBundlePretty err
291     Right result -> Right result
292
293 -- | Parsea programa (m ltiples expresiones)
294 parseProgram :: String -> Either String [SExpr]
295 parseProgram input =
296     case runParser (sc *> many expr <* eof) "" input of
297     Left err -> Left $ errorBundlePretty err
298     Right result -> Right result

```

## 5.7 Main (Main.hs)

```

1 {-# LANGUAGE LambdaCase #-}
2
3 module Main where
4
5 import System.IO
6 import System.Environment (getArgs)

```



```

49 replLoop emptyEnv
50
51 replLoop :: Env -> IO ()
52 replLoop env = do
53     putStr "> "
54     hFlush stdout
55     line <- getLine
56
57     case line of
58         "" -> replLoop env
59
60         ':'::s':'a':'l':'i':'r':_ -> do
61             putStrLn ""
62             putStrLn "
63             +=====+
64             "
65             putStrLn "|
66             Hasta pronto!
67             |"
68             putStrLn "|
69             Gracias por usar MiniLisp.
70             |"
71             putStrLn "|
72             Creado por Giovanni, Vania, Camila
73             |"
74             putStrLn "|
75             +=====+
76             "
77             putStrLn ""
78             return ()
79
80         ':'::a':'y':'u':'d':'a':_ -> do
81             showHelp
82             replLoop env
83
84         ':'::c':'a':'r':'g':'a':'r':_ :filename -> do
85             loadFile env filename >=> replLoop
86
87         ':'::n':'u':'c':'l':'e':'o':_ :input -> do
88             showCore input
89             replLoop env
90
91     input -> do
92         newEnv <- evalAndPrint env input
93         replLoop newEnv
94
95 -- | Eval a e imprime resultado
96 evalAndPrint :: Env -> String -> IO Env
97 evalAndPrint env input = do
98     case parseExpr input of
99         Left err -> do
100             putStrLn $ "[X] Error de sintaxis: " ++ err
101             return env
102
103         Right surfaceExpr -> do

```



```

97     let coreExpr = desugar surfaceExpr
98
99     case eval env coreExpr of
100     Left err -> do
101         putStrLn $ "[X] Error de evaluacion: " ++ err
102         return env
103
104     Right value -> do
105         putStrLn $ "[OK] => " ++ prettyValue value
106         return env
107
108 -- | Muestra la expresi n del n cleo
109 showCore :: String -> IO ()
110 showCore input =
111     case parseExpr input of
112     Left err -> putStrLn $ "[X] Error de parseo: " ++ err
113     Right surfaceExpr -> do
114         let coreExpr = desugar surfaceExpr
115         putStrLn ""
116         putStrLn "+--- AST Superficial
117             -----+"
118         print surfaceExpr
119         putStrLn ""
120         putStrLn "+--- AST Nucleo
121             -----+"
122         print coreExpr
123         putStrLn ""
124         putStrLn "+--- Formato Pretty
125             -----+"
126         putStrLn $ prettyExpr coreExpr
127         putStrLn ""
128         putStrLn "+-----+"
129         putStrLn ""
130
131 -- | Carga y ejecuta archivo
132 loadFile :: Env -> FilePath -> IO Env
133 loadFile env filename = do
134     content <- readFile filename
135     case parseProgram content of
136     Left err -> do
137         putStrLn $ "[X] Error de parseo en " ++ filename ++ ": " ++ err
138         return env
139
140     Right exprs -> do
141         putStrLn $ "[FILE] Cargando " ++ filename ++ "..."
142         putStrLn ""
143         execExprs env exprs
144
145 where
146     execExprs e [] = do
147         putStrLn "[OK] Archivo cargado exitosamente!"
148         putStrLn ""
149         return e
150     execExprs e (sexpr:rest) = do
151         let coreExpr = desugar sexpr
152         case eval e coreExpr of
153         Left err -> do
154             putStrLn $ "[X] Error: " ++ err

```

```

150     return e
151     Right value -> do
152         putStrLn $ "  [OK] " ++ prettyValue value
153         execExprs e rest
154
155 -- | Ejecuta archivo desde l nea de comandos
156 runFile :: FilePath -> IO ()
157 runFile filename = do
158     content <- readFile filename
159     case parseProgram content of
160         Left err -> putStrLn $ "Parse error: " ++ err
161         Right exprs -> mapM_ runExpr exprs
162     where
163         runExpr sexpr = do
164             let coreExpr = desugar sexpr
165             case eval emptyEnv coreExpr of
166                 Left err -> putStrLn $ "Error: " ++ err
167                 Right value -> putStrLn $ prettyValue value
168
169 -- | Muestra ayuda
170 showHelp :: IO ()
171 showHelp = do
172     putStrLn ""
173     putStrLn "
174     +-----+
175     |
176     |                                     AYUDA DE MINILISP
177     |                                     |
178     |                                     |
179     |                                     |
180     |                                     |
181     |                                     |
182     |                                     |
183     |                                     |
184     |                                     |
185     |                                     |
186     |                                     |
187     |                                     |
188     |                                     |
189     |                                     |
190     |                                     |
191     |                                     |
192     |                                     |
193     |                                     |
194     |                                     |
195     |                                     |
196     |                                     |
197     |                                     |
198     |                                     |
199     |                                     |
200     |                                     |
201     |                                     |
202     |                                     |
203     |                                     |
204     |                                     |
205     |                                     |
206     |                                     |
207     |                                     |
208     |                                     |
209     |                                     |
210     |                                     |
211     |                                     |
212     |                                     |
213     |                                     |
214     |                                     |
215     |                                     |
216     |                                     |
217     |                                     |
218     |                                     |
219     |                                     |
220     |                                     |
221     |                                     |
222     |                                     |
223     |                                     |
224     |                                     |
225     |                                     |
226     |                                     |
227     |                                     |
228     |                                     |
229     |                                     |
230     |                                     |
231     |                                     |
232     |                                     |
233     |                                     |
234     |                                     |
235     |                                     |
236     |                                     |
237     |                                     |
238     |                                     |
239     |                                     |
240     |                                     |
241     |                                     |
242     |                                     |
243     |                                     |
244     |                                     |
245     |                                     |
246     |                                     |
247     |                                     |
248     |                                     |
249     |                                     |
250     |                                     |
251     |                                     |
252     |                                     |
253     |                                     |
254     |                                     |
255     |                                     |
256     |                                     |
257     |                                     |
258     |                                     |
259     |                                     |
260     |                                     |
261     |                                     |
262     |                                     |
263     |                                     |
264     |                                     |
265     |                                     |
266     |                                     |
267     |                                     |
268     |                                     |
269     |                                     |
270     |                                     |
271     |                                     |
272     |                                     |
273     |                                     |
274     |                                     |
275     |                                     |
276     |                                     |
277     |                                     |
278     |                                     |
279     |                                     |
280     |                                     |
281     |                                     |
282     |                                     |
283     |                                     |
284     |                                     |
285     |                                     |
286     |                                     |
287     |                                     |
288     |                                     |
289     |                                     |
290     |                                     |
291     |                                     |
292     |                                     |
293     |                                     |
294     |                                     |
295     |                                     |
296     |                                     |
297     |                                     |
298     |                                     |
299     |                                     |
300     |                                     |
301     |                                     |
302     |                                     |
303     |                                     |
304     |                                     |
305     |                                     |
306     |                                     |
307     |                                     |
308     |                                     |
309     |                                     |
310     |                                     |
311     |                                     |
312     |                                     |
313     |                                     |
314     |                                     |
315     |                                     |
316     |                                     |
317     |                                     |
318     |                                     |
319     |                                     |
320     |                                     |
321     |                                     |
322     |                                     |
323     |                                     |
324     |                                     |
325     |                                     |
326     |                                     |
327     |                                     |
328     |                                     |
329     |                                     |
330     |                                     |
331     |                                     |
332     |                                     |
333     |                                     |
334     |                                     |
335     |                                     |
336     |                                     |
337     |                                     |
338     |                                     |
339     |                                     |
340     |                                     |
341     |                                     |
342     |                                     |
343     |                                     |
344     |                                     |
345     |                                     |
346     |                                     |
347     |                                     |
348     |                                     |
349     |                                     |
350     |                                     |
351     |                                     |
352     |                                     |
353     |                                     |
354     |                                     |
355     |                                     |
356     |                                     |
357     |                                     |
358     |                                     |
359     |                                     |
360     |                                     |
361     |                                     |
362     |                                     |
363     |                                     |
364     |                                     |
365     |                                     |
366     |                                     |
367     |                                     |
368     |                                     |
369     |                                     |
370     |                                     |
371     |                                     |
372     |                                     |
373     |                                     |
374     |                                     |
375     |                                     |
376     |                                     |
377     |                                     |
378     |                                     |
379     |                                     |
380     |                                     |
381     |                                     |
382     |                                     |
383     |                                     |
384     |                                     |
385     |                                     |
386     |                                     |
387     |                                     |
388     |                                     |
389     |                                     |
390     |                                     |
391     |                                     |
392     |                                     |
393     |                                     |
394     |                                     |
395     |                                     |
396     |                                     |
397     |                                     |
398     |                                     |
399     |                                     |
400     |                                     |
401     |                                     |
402     |                                     |
403     |                                     |
404     |                                     |
405     |                                     |
406     |                                     |
407     |                                     |
408     |                                     |
409     |                                     |
410     |                                     |
411     |                                     |
412     |                                     |
413     |                                     |
414     |                                     |
415     |                                     |
416     |                                     |
417     |                                     |
418     |                                     |
419     |                                     |
420     |                                     |
421     |                                     |
422     |                                     |
423     |                                     |
424     |                                     |
425     |                                     |
426     |                                     |
427     |                                     |
428     |                                     |
429     |                                     |
430     |                                     |
431     |                                     |
432     |                                     |
433     |                                     |
434     |                                     |
435     |                                     |
436     |                                     |
437     |                                     |
438     |                                     |
439     |                                     |
440     |                                     |
441     |                                     |
442     |                                     |
443     |                                     |
444     |                                     |
445     |                                     |
446     |                                     |
447     |                                     |
448     |                                     |
449     |                                     |
450     |                                     |
451     |                                     |
452     |                                     |
453     |                                     |
454     |                                     |
455     |                                     |
456     |                                     |
457     |                                     |
458     |                                     |
459     |                                     |
460     |                                     |
461     |                                     |
462     |                                     |
463     |                                     |
464     |                                     |
465     |                                     |
466     |                                     |
467     |                                     |
468     |                                     |
469     |                                     |
470     |                                     |
471     |                                     |
472     |                                     |
473     |                                     |
474     |                                     |
475     |                                     |
476     |                                     |
477     |                                     |
478     |                                     |
479     |                                     |
480     |                                     |
481     |                                     |
482     |                                     |
483     |                                     |
484     |                                     |
485     |                                     |
486     |                                     |
487     |                                     |
488     |                                     |
489     |                                     |
490     |                                     |
491     |                                     |
492     |                                     |
493     |                                     |
494     |                                     |
495     |                                     |
496     |                                     |
497     |                                     |
498     |                                     |
499     |                                     |
500     |                                     |
501     |                                     |
502     |                                     |
503     |                                     |
504     |                                     |
505     |                                     |
506     |                                     |
507     |                                     |
508     |                                     |
509     |                                     |
510     |                                     |
511     |                                     |
512     |                                     |
513     |                                     |
514     |                                     |
515     |                                     |
516     |                                     |
517     |                                     |
518     |                                     |
519     |                                     |
520     |                                     |
521     |                                     |
522     |                                     |
523     |                                     |
524     |                                     |
525     |                                     |
526     |                                     |
527     |                                     |
528     |                                     |
529     |                                     |
530     |                                     |
531     |                                     |
532     |                                     |
533     |                                     |
534     |                                     |
535     |                                     |
536     |                                     |
537     |                                     |
538     |                                     |
539     |                                     |
540     |                                     |
541     |                                     |
542     |                                     |
543     |                                     |
544     |                                     |
545     |                                     |
546     |                                     |
547     |                                     |
548     |                                     |
549     |                                     |
550     |                                     |
551     |                                     |
552     |                                     |
553     |                                     |
554     |                                     |
555     |                                     |
556     |                                     |
557     |                                     |
558     |                                     |
559     |                                     |
560     |                                     |
561     |                                     |
562     |                                     |
563     |                                     |
564     |                                     |
565     |                                     |
566     |                                     |
567     |                                     |
568     |                                     |
569     |                                     |
570     |                                     |
571     |                                     |
572     |                                     |
573     |                                     |
574     |                                     |
575     |                                     |
576     |                                     |
577     |                                     |
578     |                                     |
579     |                                     |
580     |                                     |
581     |                                     |
582     |                                     |
583     |                                     |
584     |                                     |
585     |                                     |
586     |                                     |
587     |                                     |
588     |                                     |
589     |                                     |
590     |                                     |
591     |                                     |
592     |                                     |
593     |                                     |
594     |                                     |
595     |                                     |
596     |                                     |
597     |                                     |
598     |                                     |
599     |                                     |
600     |                                     |
601     |                                     |
602     |                                     |
603     |                                     |
604     |                                     |
605     |                                     |
606     |                                     |
607     |                                     |
608     |                                     |
609     |                                     |
610     |                                     |
611     |                                     |
612     |                                     |
613     |                                     |
614     |                                     |
615     |                                     |
616     |                                     |
617     |                                     |
618     |                                     |
619     |                                     |
620     |                                     |
621     |                                     |
622     |                                     |
623     |                                     |
624     |                                     |
625     |                                     |
626     |                                     |
627     |                                     |
628     |                                     |
629     |                                     |
630     |                                     |
631     |                                     |
632     |                                     |
633     |                                     |
634     |                                     |
635     |                                     |
636     |                                     |
637     |                                     |
638     |                                     |
639     |                                     |
640     |                                     |
641     |                                     |
642     |                                     |
643     |                                     |
644     |                                     |
645     |                                     |
646     |                                     |
647     |                                     |
648     |                                     |
649     |                                     |
650     |
```

```

192 putStrLn "+-- ESTRUCTURAS DE DATOS
    -----+"
193 putStrLn "| Pares:      (3, 5), (fst (1, 2)), (snd (1, 2))
    | "
194 putStrLn "| Listas:     [1, 2, 3], (head [1,2,3]), (tail [1,2,3])
    | "
195 putStrLn "
    +-----+"
196 putStrLn ""
197 putStrLn "+-- ESTRUCTURAS DE CONTROL
    -----+"
198 putStrLn "| If:        (if (< x 0) (- x) x)
    | "
199 putStrLn "| If0:        (if0 x 0 1)
    | "
200 putStrLn "| Cond:       (cond [(< x 0) -1] [else 1])
    | "
201 putStrLn "
    +-----+"
202 putStrLn ""
203 putStrLn "+-- BINDINGS (Variables Locales)
    -----+"
204 putStrLn "| Let:        (let ((x 5) (y 3)) (+ x y))
    | "
205 putStrLn "| Let*:        (let* ((x 5) (y (+ x 1))) (+ x y))
    | "
206 putStrLn "| LetRec:      (letrec ((f (lambda (n) ...))) (f 10))
    | "
207 putStrLn "
    +-----+"
208 putStrLn ""
209 putStrLn "+-- FUNCIONES
    -----+"
210 putStrLn "| Lambda:      (lambda (x y) (+ x y))
    | "
211 putStrLn "| Aplicacion:  ((lambda (x) (* x x)) 5)
    | "
212 putStrLn "| Recursion:   (letrec ((fac (lambda (n) ...))) (fac 5))
    | "
213 putStrLn "
    +-----+"
214 putStrLn ""

```

## 5.8 Ejemplos de Programas

### 5.8.1. Suma de primeros n naturales (sum.minisp)

```

1 ; Suma de los primeros n n meros naturales
2 ; sum(n) = 1 + 2 + ... + n
3
4 (letrec
5   ((sum (lambda (n)
6           (if0 n
7               0
8               (+ n (sum (sub1 n)))))))
9   (sum 10))

```

```
10
11 ; Resultado esperado: 55
```

### 5.8.2. Factorial (factorial.minisp)

```
1 ; Factorial de n
2 ; fact(n) = n * (n-1) * ... * 1
3
4 (letrec
5   ((factorial (lambda (n)
6     (if (<= n 1)
7       1
8       (* n (factorial (sub1 n)))))))
9   (factorial 5))
10
11 ; Resultado esperado: 120
```

### 5.8.3. Fibonacci (fibonacci.minisp)

```
1 ; Fibonacci (n)
2 ; fib(0) = 0, fib(1) = 1
3 ; fib(n) = fib(n-1) + fib(n-2)
4
5 (letrec
6   ((fib (lambda (n)
7     (cond
8       [(= n 0) 0]
9       [(= n 1) 1]
10      [else (+ (fib (- n 1))
11              (fib (- n 2)))])))
12   (fib 10))
13
14 ; Resultado esperado: 55
```

### 5.8.4. Map para listas (map.minisp)

```
1 ; Map: aplica funci n a cada elemento de lista
2 ; map f [] = []
3 ; map f (x:xs) = (f x) : (map f xs)
4
5 (letrec
6   ((map (lambda (f lst)
7     (if (= lst [])
8       []
9       [(f (head lst)) | (map f (tail lst))]))))
10 ; Ejemplo: duplicar cada elemento
11   (let ((double (lambda (x) (* x 2))))
12     (map double [1, 2, 3, 4, 5])))
13 ; Resultado esperado: [2, 4, 6, 8, 10]
```

### 5.8.5. Filter para listas (filter.minisp)

```
1 ; Filter: mantiene elementos que cumplen predicado
2 ; filter p [] = []
3 ; filter p (x:xs) = (x : filter p xs) if p(x)
4 ;                  (filter p xs)    otherwise
5
6 (letrec
7   ((filter (lambda (pred lst)
8     (if (= lst [])
9         []
10        (let ((x (head lst))
11              (xs (tail lst)))
12          (if (pred x)
13              [x | (filter pred xs)]
14              (filter pred xs)))))))
15
16 ; Ejemplo: n meros pares
17   (let ((even? (lambda (x) (= (% x 2) 0))))
18     (filter even? [1, 2, 3, 4, 5, 6]))
19 ; Resultado esperado: [2, 4, 6]
```

## 5.9 README.md

Implementación completa de MiniLisp en Haskell para el curso de Lenguajes de Programación, UNAM Facultad de Ciencias.

### 5.9.1. Características Implementadas

- Sintaxis estilo Scheme/Lisp
- Operadores variádicos (+, -, \*, /, =, <, >, etc.)
- Estructuras de datos: pares y listas
- Funciones de orden superior (map, filter)
- Recursión con letrec
- Condicionales: if, if0, cond
- Bindings: let, let\*, letrec
- Funciones currificadas automáticamente
- REPL interactivo
- Carga de archivos

## 5.10 Instalación

### 5.10.1. Requisitos

- GHC  $\geq$  9,2
- Cabal  $\geq$  3,6

### 5.10.2. Pasos de Instalación

```
1 # Clonar repositorio
2 git clone https://github.com/GiovanniAE/MiniLispApp.git
3 cd minilisp
4
5 # Compilar proyecto
6 cabal build
7
8 # Ejecutar REPL
9 cabal run minilisp
10
11 # Ejecutar archivo
12 cabal run minilisp examples/factorial.minisp
```

## 5.11 Uso

### 5.11.1. REPL Interactivo

```
1 cabal run minilisp
```

Comandos disponibles:

- :salir - Salir del REPL
- :ayuda - Mostrar ayuda
- :cargar <archivo> - Cargar archivo
- :nucleo <expresión> - Mostrar núcleo desazucarizado

### 5.11.2. Ejecutar Archivos

```
1 cabal run minilisp archivo.minisp
```

## 5.12 Ejemplos

Los archivos de ejemplo incluyen múltiples casos de prueba:

### 5.12.1. Factorial (factorial.minisp)

```
1 cabal run minilisp examples/factorial.minisp
2 # Resultados: 1, 6, 120, 5040, 3628800
```

### 5.12.2. Suma 1..n (sum.minisp)

```
1 cabal run minilisp examples/sum.minisp
2 # Resultados: 15, 55, 210, 5050
```

### 5.12.3. Fibonacci (fibonacci.minisp)

```
1 cabal run minilisp examples/fibonacci.minisp
2 # Resultados: 0, 1, 5, 21, 55
```

### 5.12.4. Potencia (power.minisp)

```
1 cabal run minilisp examples/power.minisp
2 # Resultados: 8, 81, 25, 1000
```

### 5.12.5. MCD - Máximo Común Divisor (mcd.minisp)

```
1 cabal run minilisp examples/mcd.minisp
2 # Resultados: 6, 5, 6, 1
```

Cada archivo contiene varios casos de prueba con diferentes parámetros para demostrar la función con múltiples entradas.

## 5.13 Arquitectura del Sistema

- src/SurfaceAST.hs - AST de sintaxis superficial
- src/Parser.hs - Parser con Megaparsec
- src/AST.hs - AST del núcleo
- src/Desugar.hs - Desazucarización
- src/Eval.hs - Evaluador
- src/Main.hs - REPL y CLI

## 5.14 Archivo de Configuración (minilisp.cabal)

```
1 cabal-version: 2.2
2 name: minilisp
3 version: 1.1.5
4 synopsis: MiniLisp interpreter in Haskell
5 description: Complete implementation of MiniLisp for Programming
6               Languages course
7 homepage: https://github.com/GiovanniAE/MiniLispApp.git
8 license: MIT
9 author: Giovanni Alejandri Espinosa, Vania Zo Velazquez Barrientos,
10         Camila Sanchez Flores
11 maintainer: giovanni.uni@ciencias.unam.mx
12 category: Language
13 build-type: Simple
14
15 common shared-properties
16   default-language: Haskell2010
17   ghc-options:
18     -Wall
19     -Wcompat
```

```

18     -Widentities
19     -Wincomplete-record-updates
20     -Wincomplete-uni-patterns
21     -Wmissing-export-lists
22     -Wmissing-home-modules
23     -Wpartial-fields
24     -Wredundant-constraints
25 build-depends:
26     base >= 4.16 && < 5,
27     containers >= 0.6,
28     mtl >= 2.2
29
30 executable minilisp
31     import: shared-properties
32     hs-source-dirs: src
33     main-is: Main.hs
34     other-modules:
35         AST,
36         SurfaceAST,
37         Parser,
38         Desugar,
39         Eval
40 build-depends:
41     megaparsec >= 9.2,
42     parser-combinators >= 1.3,
43     deepseq >= 1.4
44     ghc-options: -threaded -rtsopts -with-rtsopts=-N
45
46 test-suite minilisp-test
47     import: shared-properties
48     type: exitcode-stdio-1.0
49     hs-source-dirs: test
50     main-is: Tests.hs
51     build-depends:
52         minilisp,
53         hspec >= 2.10,
54         QuickCheck >= 2.14
55     ghc-options: -threaded -rtsopts -with-rtsopts=-N
56
57 library
58     import: shared-properties
59     hs-source-dirs: src
60     exposed-modules:
61         AST,
62         SurfaceAST,
63         Parser,
64         Desugar,
65         Eval
66 build-depends:
67     megaparsec >= 9.2,
68     parser-combinators >= 1.3,
69     deepseq >= 1.4

```



## 5.15 Conclusiones

Este proyecto demuestra la integración completa entre teoría y práctica en el diseño de lenguajes de programación. Se ha logrado:

1. **Formalización rigurosa:** Especificación matemática precisa de sintaxis y semántica
2. **Implementación funcional:** Sistema ejecutable que refleja fielmente la formalización
3. **Extensibilidad:** Arquitectura modular que facilita futuras extensiones
4. **Expresividad:** Lenguaje capaz de expresar programas no triviales

El proceso de eliminación de azúcar sintáctica ilustra claramente la distinción entre conveniencia sintáctica y poder expresivo fundamental, mientras que la semántica operacional proporciona una base sólida para razonar sobre programas.

El proyecto MiniLisp sirve como un ejemplo pedagógico completo que abarca desde los fundamentos teóricos del diseño de lenguajes hasta la implementación práctica de un intérprete funcional. La separación entre sintaxis de superficie y núcleo, junto con el proceso sistemático de desazucarización, demuestra cómo las construcciones complejas pueden reducirse a un conjunto mínimo de primitivas sin perder expresividad.

La implementación en Haskell aprovecha las características del paradigma funcional para crear un diseño elegante y composicional, donde cada componente (parser, desazucarizador, evaluador) mantiene responsabilidades claramente definidas. Esto no solo facilita el mantenimiento y la extensión del sistema, sino que también refleja directamente los conceptos teóricos presentados en la formalización.

El proyecto valida que es posible construir un lenguaje de programación completo partiendo de fundamentos teóricos sólidos, demostrando la aplicabilidad práctica de conceptos como semántica operacional, sistemas de tipos, y transformaciones de programas en el desarrollo de herramientas de programación reales.