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)

5 de noviembre de 2025

Índice

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:

```
IDENTIFICADOR ::= [a-zA-Z_][a-zA-Z0-9_\-?!]*
ENTERO ::= -?[0-9]+
BOOLEANO ::= #t | #f
LPAREN ::= (
RPAREN ::= )
LBRACKET ::= [
RBRACKET ::= ]
COMMA ::= ,
WHITESPACE ::= [ \t\n\r]+
COMMENT ::= ;[^\n]*\n

PALABRAS_RESERVADAS ::= let | let* | letrec | if | if0 | lambda | cond
   | else | fst | snd | head | tail | addl | subl | sqrt | expt | not

```

```
OPERADORES ::= + | - | * | / | = | < | > | >= | <= | !=
```

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)

```
<Program> ::= <Expr>

<Expr> ::= <Atom> | <Pair> | <List> | <ArithOp> | <CompOp> | <LogicOp>
   | <Conditional> | <Binding> | <Function> | <Application> | <
   Projection> | <ListOp>

<Atom> ::= IDENTIFICADOR | ENTERO | BOOLEANO

<Pair> ::= "(" <Expr> "," <Expr> ")"
<List> ::= "[" "]" | "[" <Expr> ( "," <Expr> )* "]"

<ArithOp> ::= "(" <ArithOp2> <Expr> <Expr>+ ")"
            | "(" <ArithOp1> <Expr> ")"

<ArithOp2> ::= "+" | "-" | "*" | "/"
<ArithOp1> ::= "add1" | "sub1" | "sqrt"

<CompOp> ::= "(" <Comparator> <Expr> <Expr>+ ")"
<Comparator> ::= "=" | "<" | ">" | ">=" | "<=" | "!="

<LogicOp> ::= "(" "not" <Expr> ")"

<Conditional> ::= "(" "if" <Expr> <Expr> <Expr> ")"
                | "(" "if0" <Expr> <Expr> <Expr> ")"
                | <Cond>

<Cond> ::= "(" "cond" <Clause>+ <ElseClause> ")"
<Clause> ::= "[" <Expr> <Expr> "]"
<ElseClause> ::= "[" "else" <Expr> "]"

<Binding> ::= "(" "let" "(" <Bind>+ ")" <Expr> ")"
            | "(" "let*" "(" <Bind>+ ")" <Expr> ")"
            | "(" "letrec" "(" <Bind>+ ")" <Expr> ")"

<Bind> ::= "(" IDENTIFICADOR <Expr> ")"

<Function> ::= "(" "lambda" "(" IDENTIFICADOR* ")" <Expr> ")"

<Application> ::= "(" <Expr> <Expr>* ")"

<Projection> ::= "(" "fst" <Expr> ")"
               | "(" "snd" <Expr> ")"

<ListOp> ::= "(" "head" <Expr> ")"
           | "(" "tail" <Expr> ")"
           | "(" "expt" <Expr> <Expr> ")"
```

Notas sobre la gramática:

- $< Expr > +$ denota una o más repeticiones
- $< Expr > *$ denota cero o más repeticiones
- La gramática es ambigua en $< Application >$, 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:

```haskell
data SExpr = SVar String
    | SInt Integer
    | SBool Bool
    | SPair SExpr SExpr
    | SList [SExpr]
    -- Operadores vari dicos
    | SAdd [SExpr]
    | SSub [SExpr]
    | SMul [SExpr]
    | SDiv [SExpr]
    | SEq [SExpr]
    | SLt [SExpr]
    | SGt [SExpr]
    | SLe [SExpr]
    | SGe [SExpr]
    | SNe [SExpr]
    -- Operadores unarios
    | SAddl SExpr
    | SSubl SExpr
    | SSqrt SExpr
    | SNot SExpr
    -- Operadores binarios especiales
    | SExpt SExpr SExpr
    -- Condicionales
    | SIf SExpr SExpr SExpr
    | SIf0 SExpr SExpr SExpr
    | SCond [(SExpr, SExpr)] SExpr
    -- Bindings
    | SLet [(String, SExpr)] SExpr
    | SLetStar [(String, SExpr)] SExpr
    | SLetRec [(String, SExpr)] SExpr
    -- Funciones
    | SLambda [String] SExpr
    | SApp SExpr [SExpr]
    -- Proyecciones
    | SFst SExpr
    | SSnd SExpr
    | SHead SExpr
    | STail SExpr
```

2.3.2. AST del Núcleo (Core)

El núcleo contiene solo las construcciones primitivas esenciales:

```haskell
data Expr = Var String
    | IntLit Integer
    | BoolLit Bool
    | Pair Expr Expr
    | Nil
    | Cons Expr Expr
    -- Operadores binarios   nicamente
    | Add Expr Expr
    | Sub Expr Expr
    | Mul Expr Expr
    | Div Expr Expr
    | Eq Expr Expr
    | Lt Expr Expr
    -- Operadores unarios
    | Not Expr
    -- Condicional  nico
    | If Expr Expr Expr
    -- Binding  nico
    | Let String Expr Expr
    -- Funci n currificada
    | Lambda String Expr
    | App Expr Expr
    -- Proyecciones
    | Fst Expr
    | 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: $\cdot_s : \text{SExpr} \to \text{Expr}$

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

2.4.3. Operadores Unarios

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

2.4.4. Condicionales

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

2.4.5. Bindings

- [**DS-LET-SINGLE**] Let simple: $(\text{let } ((x\ e))\ \text{body})_s = \text{Let } xe_s\text{body}_s$
- [**DS-LET-MULTI**] Let multiple: $(\text{let } ((x_1\ e_1) \ldots (x_n\ e_n))\ \text{body})_s = \text{App (Lambda } x_1(\ldots(\text{Lambd}$
- [**DS-LETSTAR-BASE**] Let* caso base: $(\text{let* } ((x\ e))\ \text{body})_s = \text{Let } xe_s\text{body}_s$
- [**DS-LETSTAR-REC**] Let* recursivo: $(\text{let* } ((x_1\ e_1)(x_2\ e_2) \ldots (x_n\ e_n))\ \text{body})_s =$ $\text{Let } x_1e_{1s}(\text{let* } ((x_2\ e_2) \ldots (x_n\ e_n))\ \text{body})_s$
- [**DS-LETREC**] Letrec usando combinador Y: $(\text{letrec } ((f\ e))\ \text{body})_s = \text{Let } f(\text{Fix (Lambda } fe_s))\text{b}$

2.4.6. Funciones

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

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

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: $[e_1, e_2, \ldots, e_n]_s = \text{Cons}\ e_{1s} [e_2, \ldots, e_n]_s$
- **[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: $e \to e'$
Un juicio de transición $e \to e'$ 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: $\dfrac{}{n_1 + n_2 \to n_3}$ donde $n_3 = n_1 + n_2$
- **[E-ADD-L]** Reducir operando izquierdo: $\dfrac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$
- **[E-ADD-R]** Reducir operando derecho: $\dfrac{e_2 \to e_2'}{v_1 + e_2 \to v_1 + e_2'}$
- **[E-DIV-ZERO]** División por cero: $\dfrac{}{v/0 \to \text{ERROR}}$

Reglas análogas para: Sub, Mul, Div

2.5.3. Reglas de Comparación

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

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

2.5.4. Reglas Lógicas

- [**E-NOT-TRUE**] $\dfrac{}{\text{not } \#t \to \#f}$
- [**E-NOT-FALSE**] $\dfrac{}{\text{not } \#f \to \#t}$
- [**E-NOT-STEP**] $\dfrac{e \to e'}{\text{not } e \to \text{not } e'}$

2.5.5. Reglas Condicionales

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

2.5.6. Reglas de Binding

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

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

2.5.7. Reglas de Funciones

- [**E-APP-BETA**] Beta-reducción: $\dfrac{}{((\text{lambda } (x)\ e_1)\ v_2) \to e_1[x := v_2]}$
- [**E-APP-FUN**] Reducir función: $\dfrac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$
- [**E-APP-ARG**] Reducir argumento: $\dfrac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$
- [**E-APP-ERROR**] Aplicar no-función: $\dfrac{v_1 \notin (\text{lambda } \ldots)}{v_1\ v_2 \to \text{ERROR}}$

2.5.8. Reglas de Pares

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

2.5.9. Ejemplo de Derivación Completa

Programa: $(+2(*34))$
Árbol de derivación:

$$\dfrac{\dfrac{}{(*34) \to 12} \quad \dfrac{\dfrac{}{(+212) \to 14}}{(+2(*34)) \to (+212)}}{(+2(*34)) \to^* 14}$$

Programa: $(\text{let } ((x5))(+x3))$
Árbol de derivación:

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

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*^{6} *Report on the Algorithmic Language Scheme*. Cambridge University Press, 2009.

5 Implementación en Haskell

5.1 Estructura del Proyecto

```
minilisp/
          src/
                AST.hs              -- Definici n de AST (Core)
                SurfaceAST.hs       -- AST de superficie
                Parser.hs           -- Analizador sint ctico
                Desugar.hs          -- Eliminaci n de az car
                Eval.hs             -- Int rprete/evaluador
                Main.hs             -- REPL y entrada principal
          examples/
                sum.minisp          -- Suma de n naturales
                factorial.minisp    -- Factorial
                fibonacci.minisp    -- Fibonacci
                map.minisp          -- Map para listas
                filter.minisp       -- Filter para listas
          test/
                Tests.hs            -- Suite de pruebas
```

```
          Doc/
                Proyecto01.pdf
                Proyecto01.tex
          README.md
          minilisp.cabal
```

5.2 AST del Núcleo (AST.hs)

```haskell
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveAnyClass #-}

module AST
   ( Expr(..)
   , prettyExpr
   ) where

import GHC.Generics (Generic)
import Control.DeepSeq (NFData)

-- | Sintaxis Abstracta del N cleo de MiniLisp
-- Todas las construcciones aqu  son primitivas
data Expr
   -- Valores at micos
   = Var String              -- Variable
   | IntLit Integer          -- Literal entera
   | BoolLit Bool            -- Literal booleana
   -- Estructuras de datos
   | Pair Expr Expr          -- Par ordenado (e1, e2)
   | Nil                     -- Lista vac a
   | Cons Expr Expr          -- Constructor de lista
   -- Operadores aritm ticos binarios
   | Add Expr Expr           -- Suma
   | Sub Expr Expr           -- Resta
   | Mul Expr Expr           -- Multiplicaci n
   | Div Expr Expr           -- Divisi n
   -- Operadores de comparaci n binarios
   | Eq Expr Expr            -- Igualdad
   | Lt Expr Expr            -- Menor que
   | Gt Expr Expr            -- Mayor que
   | Le Expr Expr            -- Menor o igual
   | Ge Expr Expr            -- Mayor o igual
   | Ne Expr Expr            -- Diferente
   -- Operadores l gicos
   | Not Expr                -- Negaci n
   -- Control de flujo
   | If Expr Expr Expr       -- Condicional
   -- Binding
   | Let String Expr Expr    -- Definici n local
   -- Funciones (currificadas)
   | Lambda String Expr      -- Funci n an nima
   | App Expr Expr           -- Aplicaci n
   -- Proyecciones
   | Fst Expr                -- Primera proyecci n
   | Snd Expr                -- Segunda proyecci n
   deriving (Eq, Show, Generic, NFData)

```

```haskell
-- | Pretty printer para expresiones del n cleo
prettyExpr :: Expr -> String
prettyExpr = go 0
  where
    indent n = replicate (n * 2) ' '

    go _ (Var x) = x
    go _ (IntLit n) = show n
    go _ (BoolLit True) = "#t"
    go _ (BoolLit False) = "#f"
    go _ Nil = "nil"
    go n (Pair e1 e2) =
        "(" ++ go n e1 ++ ", " ++ go n e2 ++ ")"
    go n (Cons e1 e2) =
        "(cons " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Add e1 e2) =
        "(+ " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Sub e1 e2) =
        "(- " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Mul e1 e2) =
        "(* " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Div e1 e2) =
        "(/ " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Eq e1 e2) =
        "(= " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Lt e1 e2) =
        "(< " ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Not e) =
        "(not " ++ go n e ++ ")"
    go n (If c t e) =
        "(if " ++ go n c ++ "\n" ++
        indent (n+1) ++ go (n+1) t ++ "\n" ++
        indent (n+1) ++ go (n+1) e ++ ")"
    go n (Let x e1 e2) =
        "(let " ++ x ++ " " ++ go n e1 ++ "\n" ++
        indent (n+1) ++ go (n+1) e2 ++ ")"
    go n (Lambda x body) =
        "(lambda (" ++ x ++ ") " ++ go n body ++ ")"
    go n (App e1 e2) =
        "(" ++ go n e1 ++ " " ++ go n e2 ++ ")"
    go n (Fst e) = "(fst " ++ go n e ++ ")"
    go n (Snd e) = "(snd " ++ go n e ++ ")"
    go n e = show e
```

5.3 AST de Superficie (SurfaceAST.hs)

```haskell
{-# LANGUAGE DeriveGeneric #-}

module SurfaceAST
   ( SExpr(..)
   , Binding
   ) where

import GHC.Generics (Generic)

type Binding = (String, SExpr)
```

```haskell

-- | Sintaxis Abstracta de Superficie
-- Incluye todas las extensiones y az car sint ctica
data SExpr
   -- Valores at micos
   = SVar String
   | SInt Integer
   | SBool Bool
   -- Estructuras de datos
   | SPair SExpr SExpr
   | SList [SExpr]
   -- Operadores aritm ticos vari dicos
   | SAdd [SExpr]          -- (+ e1 e2 ... en)
   | SSub [SExpr]          -- (- e1 e2 ... en)
   | SMul [SExpr]          -- (* e1 e2 ... en)
   | SDiv [SExpr]          -- (/ e1 e2 ... en)
   -- Operadores aritm ticos unarios
   | SAdd1 SExpr           -- (add1 e)
   | SSub1 SExpr           -- (sub1 e)
   | SSqrt SExpr           -- (sqrt e)
   -- Operador de potencia
   | SExpt SExpr SExpr     -- (expt base exp)
   -- Operadores de comparaci n vari dicos
   | SEq [SExpr]           -- (= e1 e2 ... en)
   | SLt [SExpr]           -- (< e1 e2 ... en)
   | SGt [SExpr]           -- (> e1 e2 ... en)
   | SLe [SExpr]           -- (<= e1 e2 ... en)
   | SGe [SExpr]           -- (>= e1 e2 ... en)
   | SNe [SExpr]           -- (!= e1 e2 ... en)
   -- Operadores l gicos
   | SNot SExpr            -- (not e)
   -- Condicionales
   | SIf SExpr SExpr SExpr -- (if cond then else)
   | SIf0 SExpr SExpr SExpr -- (if0 e then else)
   | SCond [(SExpr, SExpr)] SExpr -- (cond [g1 e1] ... [else ee])
   -- Bindings
   | SLet [Binding] SExpr  -- (let ((x1 e1) ...) body)
   | SLetStar [Binding] SExpr -- (let* ((x1 e1) ...) body)
   | SLetRec [Binding] SExpr -- (letrec ((x1 e1) ...) body)
   -- Funciones
   | SLambda [String] SExpr -- (lambda (x1 ... xn) body)
   | SApp SExpr [SExpr]    -- (f e1 e2 ... en)
   -- Proyecciones y operaciones sobre listas
   | SFst SExpr            -- (fst e)
   | SSnd SExpr            -- (snd e)
   | SHead SExpr           -- (head e)
   | STail SExpr           -- (tail e)
   deriving (Eq, Show, Generic)
```

5.4 Módulo de Desazucarización (Desugar.hs)

```haskell
module Desugar
   ( desugar
   , desugarOp
   ) where

```

```haskell
import AST
import SurfaceAST
import qualified Data.List as L

-- | Funci n principal de desazucarizaci n
-- Implementa todas las reglas [DS-*] definidas formalmente
desugar :: SExpr -> Expr
desugar se = case se of
   -- [DS-VAR], [DS-INT], [DS-BOOL]
   SVar x -> Var x
   SInt n -> IntLit n
   SBool b -> BoolLit b

   -- [DS-PAIR]
   SPair e1 e2 -> Pair (desugar e1) (desugar e2)

   -- [DS-NIL], [DS-LIST-*]
   SList [] -> Nil
   SList [e] -> Cons (desugar e) Nil
   SList (e:es) -> Cons (desugar e) (desugar (SList es))

   -- Operadores aritm ticos vari dicos
   SAdd es -> desugarOp "+" es
   SSub es -> desugarOp "-" es
   SMul es -> desugarOp "*" es
   SDiv es -> desugarOp "/" es

   -- [DS-ADD1], [DS-SUB1]
   SAdd1 e -> Add (desugar e) (IntLit 1)
   SSub1 e -> Sub (desugar e) (IntLit 1)

   -- [DS-SQRT] - implementado como expt e 0.5
   -- Para enteros, usamos aproximaci n
   SSqrt e -> desugarSqrt (desugar e)

   -- [DS-EXPT]
   SExpt base exp -> desugarExpt (desugar base) (desugar exp)

   -- Operadores de comparaci n vari dicos
   SEq es -> desugarComp Eq es
   SLt es -> desugarComp Lt es
   SGt es -> desugarComp Gt es
   SLe es -> desugarComp Le es
   SGe es -> desugarComp Ge es
   SNe es -> desugarComp Ne es

   -- [DS-NOT]
   SNot e -> Not (desugar e)

   -- [DS-IF]
   SIf c t e -> If (desugar c) (desugar t) (desugar e)

   -- [DS-IF0]
   SIf0 e t f -> If (Eq (desugar e) (IntLit 0))
                   (desugar t)
                   (desugar f)

   -- [DS-COND-*]
```

```
    SCond clauses elseExpr -> desugarCond clauses elseExpr

    -- [DS-LET-*]
    SLet bindings body -> desugarLet bindings body

    -- [DS-LETSTAR-*]
    SLetStar bindings body -> desugarLetStar bindings body

    -- [DS-LETREC]
    SLetRec bindings body -> desugarLetRec bindings body

    -- [DS-LAMBDA-*]
    SLambda params body -> curryLambda params (desugar body)

    -- [DS-APP-*]
    SApp f args -> applyMany (desugar f) (map desugar args)

    -- [DS-FST], [DS-SND]
    SFst e -> Fst (desugar e)
    SSnd e -> Snd (desugar e)

    -- [DS-HEAD], [DS-TAIL]
    SHead e -> Fst (desugar e)  -- head = fst (cons)
    STail e -> Snd (desugar e)  -- tail = snd (cons)

-- | Desazucariza operadores aritm ticos vari dicos
-- [DS-ADD-BIN], [DS-ADD-VAR]
desugarOp :: String -> [SExpr] -> Expr
desugarOp op exprs = case exprs of
    [] -> error $ "Operator " ++ op ++ " requires at least 2 arguments"
    [_] -> error $ "Operator " ++ op ++ " requires at least 2 arguments"
    [e1, e2] -> applyBinOp op (desugar e1) (desugar e2)
    (e1:e2:rest) -> applyBinOp op
                      (desugar e1)
                      (desugarOp op (e2:rest))
 where
    applyBinOp "+" = Add
    applyBinOp "-" = Sub
    applyBinOp "*" = Mul
    applyBinOp "/" = Div
    applyBinOp _ = error "Unknown operator"

-- | Desazucariza operadores de comparaci n vari dicos
-- [DS-EQ-VAR], [DS-LT-VAR], etc.
desugarComp :: (Expr -> Expr -> Expr) -> [SExpr] -> Expr
desugarComp _ [] = error "Comparison requires at least 2 arguments"
desugarComp _ [_] = error "Comparison requires at least 2 arguments"
desugarComp cons [e1, e2] = cons (desugar e1) (desugar e2)
desugarComp cons (e1:e2:rest) =
    If (cons (desugar e1) (desugar e2))
       (desugarComp cons (e2:rest))
       (BoolLit False)

-- | Desazucariza cond a if anidados
-- [DS-COND-BASE], [DS-COND-REC]
desugarCond :: [(SExpr, SExpr)] -> SExpr -> Expr
desugarCond [] elseExpr = desugar elseExpr
desugarCond ((guard, body):rest) elseExpr =
```

```haskell
    If (desugar guard)
       (desugar body)
       (desugarCond rest elseExpr)

-- | Desazucariza let a lambdas
-- [DS-LET-SINGLE], [DS-LET-MULTI]
desugarLet :: [Binding] -> SExpr -> Expr
desugarLet [] body = desugar body
desugarLet bindings body =
    let (vars, exprs) = unzip bindings
        lambda = foldr Lambda (desugar body) vars
        args = map desugar exprs
    in applyMany lambda args

-- | Desazucariza let* a let anidados
-- [DS-LETSTAR-BASE], [DS-LETSTAR-REC]
desugarLetStar :: [Binding] -> SExpr -> Expr
desugarLetStar [] body = desugar body
desugarLetStar [(x, e)] body =
    Let x (desugar e) (desugar body)
desugarLetStar ((x, e):rest) body =
    Let x (desugar e) (desugarLetStar rest body)

-- | Desazucariza letrec usando combinador Y
-- [DS-LETREC]
desugarLetRec :: [Binding] -> SExpr -> Expr
desugarLetRec bindings body =
    -- Para letrec, necesitamos el combinador de punto fijo
    -- Por simplicidad, usamos una aproximaci n:
    -- letrec transforma cada binding en una funci n recursiva
    let desugarBinding (x, e) restBody =
          Let x (fixCombinator x (desugar e)) restBody
    in foldr desugarBinding (desugar body) bindings
  where
    -- Combinador Y simplificado para call-by-value
    fixCombinator :: String -> Expr -> Expr
    fixCombinator f body =
       App (Lambda f body)
           (Lambda "_" (App (Var f) (Var "_")))

-- | Currifica lambdas multi-par metro
-- [DS-LAMBDA-ZERO], [DS-LAMBDA-ONE], [DS-LAMBDA-CURRY]
curryLambda :: [String] -> Expr -> Expr
curryLambda [] body = Lambda "_" body  -- Unit lambda
curryLambda [x] body = Lambda x body
curryLambda (x:xs) body = Lambda x (curryLambda xs body)

-- | Aplica funci n a m ltiples argumentos
-- [DS-APP-ZERO], [DS-APP-ONE], [DS-APP-MULTI]
applyMany :: Expr -> [Expr] -> Expr
applyMany f [] = App f (BoolLit True)  -- Unit application
applyMany f [e] = App f e
applyMany f (e:es) = applyMany (App f e) es

-- | Desazucariza sqrt (implementaci n simplificada para enteros)
desugarSqrt :: Expr -> Expr
desugarSqrt e =
    -- sqrt(n) usando el m todo de Newton-Raphson
```

```haskell
    -- Para simplificar, usamos una funci n auxiliar
    Let "x" e
      (Let "guess" (IntLit 1)
        (desugarSqrtHelper (Var "x") (Var "guess")))
  where
    desugarSqrtHelper x guess =
      -- if guess * guess = x then guess
      -- else sqrt_helper x ((guess + x/guess) / 2)
      If (Eq (Mul guess guess) x)
        guess
        (desugarSqrtHelper x
          (Div (Add guess (Div x guess)) (IntLit 2)))

-- | Desazucariza expt (exponenciaci n)
desugarExpt :: Expr -> Expr -> Expr
desugarExpt base exp =
    If (Eq exp (IntLit 0))
      (IntLit 1)
      (If (Lt exp (IntLit 0))
        (error "Negative exponents not supported")
        (Mul base (desugarExpt base (Sub exp (IntLit 1)))))
```

5.5 Módulo de Evaluación (Eval.hs)

```haskell
{-# LANGUAGE LambdaCase #-}

module Eval
    ( Value(..)
    , Env
    , emptyEnv
    , eval
    , evalSteps
    , prettyValue
    ) where

import AST
import qualified Data.Map.Strict as Map
import Control.Monad (foldM)

-- | Valores sem nticos (formas normales)
data Value
    = VInt Integer
    | VBool Bool
    | VPair Value Value
    | VNil
    | VCons Value Value
    | VClosure Env String Expr  -- Closure con entorno capturado
    deriving (Eq, Show)

-- | Entorno de evaluaci n (binding de variables a valores)
type Env = Map.Map String Value

-- | Entorno vac o
emptyEnv :: Env
emptyEnv = Map.empty

```

```haskell
-- | Evaluador principal (big-step para simplicidad de implementaci n)
-- Nota: La especificaci n formal usa small-step, pero big-step
-- es equivalente y m s eficiente para la implementaci n
eval :: Env -> Expr -> Either String Value
eval env = \case
   -- [E-VAR]
   Var x ->
      case Map.lookup x env of
         Just v -> Right v
         Nothing -> Left $ "Unbound variable: " ++ x

   -- Valores
   IntLit n -> Right (VInt n)
   BoolLit b -> Right (VBool b)
   Nil -> Right VNil

   -- [E-PAIR-*]
   Pair e1 e2 -> do
      v1 <- eval env e1
      v2 <- eval env e2
      return $ VPair v1 v2

   -- [E-CONS]
   Cons e1 e2 -> do
      v1 <- eval env e1
      v2 <- eval env e2
      return $ VCons v1 v2

   -- [E-ADD-*]
   Add e1 e2 -> do
      v1 <- eval env e1
      v2 <- eval env e2
      case (v1, v2) of
         (VInt n1, VInt n2) -> Right $ VInt (n1 + n2)
         _ -> Left "Type error: + expects integers"

   -- [E-SUB-*]
   Sub e1 e2 -> do
      v1 <- eval env e1
      v2 <- eval env e2
      case (v1, v2) of
         (VInt n1, VInt n2) -> Right $ VInt (n1 - n2)
         _ -> Left "Type error: - expects integers"

   -- [E-MUL-*]
   Mul e1 e2 -> do
      v1 <- eval env e1
      v2 <- eval env e2
      case (v1, v2) of
         (VInt n1, VInt n2) -> Right $ VInt (n1 * n2)
         _ -> Left "Type error: * expects integers"

   -- [E-DIV-*], [E-DIV-ZERO]
   Div e1 e2 -> do
      v1 <- eval env e1
      v2 <- eval env e2
      case (v1, v2) of
         (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"
```

```haskell

    -- Variables se sustituyen
    Var x -> case Map.lookup x env of
       Just (VInt n) -> Right (IntLit n)
       Just (VBool b) -> Right (BoolLit b)
       Just VNil -> Right Nil
       Nothing -> Left $ "Unbound: " ++ x
       _ -> Left "Complex value"

    -- Operadores: evaluar operandos primero
    Add e1 e2 -> do
       case (e1, e2) of
          (IntLit n1, IntLit n2) -> Right $ IntLit (n1 + n2)
          (IntLit _, _) -> do
             e2' <- step env e2
             Right $ Add e1 e2'
          _ -> do
             e1' <- step env e1
             Right $ Add e1' e2

    -- If eval a condici n primero
    If (BoolLit True) e2 _ -> Right e2
    If (BoolLit False) _ e3 -> Right e3
    If e1 e2 e3 -> do
       e1' <- step env e1
       Right $ If e1' e2 e3

    -- Let sustituye
    Let x (IntLit n) body -> Right $ substExpr x (IntLit n) body
    Let x (BoolLit b) body -> Right $ substExpr x (BoolLit b) body
    Let x e1 e2 -> do
       e1' <- step env e1
       Right $ Let x e1' e2

    -- Aplicaci n: beta-reducci n
    App (Lambda x body) v@(IntLit _) -> Right $ substExpr x v body
    App (Lambda x body) v@(BoolLit _) -> Right $ substExpr x v body
    App e1@(Lambda _ _) e2 -> do
       e2' <- step env e2
       Right $ App e1 e2'
    App e1 e2 -> do
       e1' <- step env e1
       Right $ App e1' e2

    e -> Left $ "Cannot step: " ++ show e

-- | Sustituci n capture-avoiding
substExpr :: String -> Expr -> Expr -> Expr
substExpr var val = go
  where
    go (Var x) | x == var = val
               | otherwise = Var x
    go (IntLit n) = IntLit n
    go (BoolLit b) = BoolLit b
    go Nil = Nil
    go (Pair e1 e2) = Pair (go e1) (go e2)
    go (Cons e1 e2) = Cons (go e1) (go e2)
    go (Add e1 e2) = Add (go e1) (go e2)
```

```haskell
    go (Sub e1 e2) = Sub (go e1) (go e2)
    go (Mul e1 e2) = Mul (go e1) (go e2)
    go (Div e1 e2) = Div (go e1) (go e2)
    go (Eq e1 e2) = Eq (go e1) (go e2)
    go (Lt e1 e2) = Lt (go e1) (go e2)
    go (Gt e1 e2) = Gt (go e1) (go e2)
    go (Le e1 e2) = Le (go e1) (go e2)
    go (Ge e1 e2) = Ge (go e1) (go e2)
    go (Ne e1 e2) = Ne (go e1) (go e2)
    go (Not e) = Not (go e)
    go (If c t e) = If (go c) (go t) (go e)
    go (Let x e1 e2) | x == var = Let x (go e1) e2  -- No sustituir en scope
                     | otherwise = Let x (go e1) (go e2)
    go (Lambda x body) | x == var = Lambda x body  -- No sustituir en scope
                       | otherwise = Lambda x (go body)
    go (App e1 e2) = App (go e1) (go e2)
    go (Fst e) = Fst (go e)
    go (Snd e) = Snd (go e)

-- | Pretty printer para valores
prettyValue :: Value -> String
prettyValue (VInt n) = show n
prettyValue (VBool True) = "#t"
prettyValue (VBool False) = "#f"
prettyValue VNil = "[]"
prettyValue (VPair v1 v2) = "(" ++ prettyValue v1 ++ ", " ++ prettyValue v2 ++ ")"
prettyValue (VCons v1 v2) = "[" ++ prettyValue v1 ++ go v2 ++ "]"
  where
    go VNil = ""
    go (VCons v vr) = ", " ++ prettyValue v ++ go vr
    go v = " | " ++ prettyValue v  -- Improper list
prettyValue (VClosure _ x _) = "<function:   " ++ x ++ ".?>"
```

5.6 Parser (Parser.hs)

```haskell
{-# LANGUAGE OverloadedStrings #-}

module Parser
   ( parseExpr
   , parseProgram
   , Parser
   ) where

import SurfaceAST
import Control.Monad (void)
import Data.Void (Void)
import Text.Megaparsec
import Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L
import Data.Functor (($>))

type Parser = Parsec Void String

```

```haskell
-- | Espacios en blanco y comentarios
sc :: Parser ()
sc = L.space
   space1
   (L.skipLineComment ";")
   empty

-- | Lexema: parsea y consume espacios
lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc

-- | S mbolo: parsea string y consume espacios
symbol :: String -> Parser String
symbol = L.symbol sc

-- | Entre par ntesis
parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")

-- | Entre corchetes
brackets :: Parser a -> Parser a
brackets = between (symbol "[") (symbol "]")

-- | Palabras reservadas
reserved :: [String]
reserved =
   [ "let", "let*", "letrec", "if", "if0", "lambda"
   , "cond", "else", "not", "fst", "snd", "head", "tail"
   , "add1", "sub1", "sqrt", "expt"
   ]

-- | Identificador
identifier :: Parser String
identifier = lexeme $ try $ do
   first <- letterChar <|> char '_'
   rest <- many (alphaNumChar <|> char '_' <|> char '-' <|> char '?'
      <|> char '!')
   let name = first : rest
   if name `elem` reserved
      then fail $ "Reserved word: " ++ name
      else return name

-- | N mero entero
integer :: Parser Integer
integer = lexeme $ L.signed sc L.decimal

-- | Booleano
boolean :: Parser Bool
boolean = lexeme $
   (string "#t" $> True) <|>
   (string "#f" $> False)

-- | Expresi n
expr :: Parser SExpr
expr = choice
   [ try pairExpr
   , listExpr
   , atomExpr
```

```
76      , parens compoundExpr
77      ]
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)
```

```haskell
if0Expr :: Parser SExpr
if0Expr = do
   symbol "if0"
   e <- expr
   t <- expr
   f <- expr
   return $ SIf0 e t f

-- | Cond: (cond [g1 e1] ... [else ee])
condExpr :: Parser SExpr
condExpr = do
   symbol "cond"
   clauses <- many clause
   elseClause <- elseClause
   return $ SCond clauses elseClause
 where
   clause = brackets $ do
      guard <- expr
      body <- expr
      return (guard, body)
   elseClause = brackets $ do
      symbol "else"
      expr

-- | Let: (let ((x1 e1) ...) body)
letExpr :: Parser SExpr
letExpr = do
   symbol "let"
   bindings <- parens $ many binding
   body <- expr
   return $ SLet bindings body

-- | Let*: (let* ((x1 e1) ...) body)
letStarExpr :: Parser SExpr
letStarExpr = do
   symbol "let*"
   bindings <- parens $ many binding
   body <- expr
   return $ SLetStar bindings body

-- | LetRec: (letrec ((f1 e1) ...) body)
letRecExpr :: Parser SExpr
letRecExpr = do
   symbol "letrec"
   bindings <- parens $ many binding
   body <- expr
   return $ SLetRec bindings body

-- | Binding: (x e)
binding :: Parser Binding
binding = parens $ do
   x <- identifier
   e <- expr
   return (x, e)

-- | Lambda: (lambda (x1 ... xn) body)
lambdaExpr :: Parser SExpr
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
```

```haskell
    symbol "head"
    e <- expr
    return $ SHead e

-- | Tail: (tail e)
tailExpr :: Parser SExpr
tailExpr = do
    symbol "tail"
    e <- expr
    return $ STail e

-- | Operadores: (+/-/*/= e1 e2 ...)
opExpr :: Parser SExpr
opExpr = do
    op <- choice
        [ symbol "+" $> SAdd
        , symbol "-" $> SSub
        , try (symbol "*") $> SMul
        , symbol "/" $> SDiv
        , try (symbol "=") $> SEq
        , try (symbol "<=") $> SLe
        , try (symbol ">=") $> SGe
        , try (symbol "!=") $> SNe
        , symbol "<" $> SLt
        , symbol ">" $> SGt
        ]
    exprs <- some expr
    return $ op exprs

-- | Aplicaci n: (f e1 ... en)
appExpr :: Parser SExpr
appExpr = do
    f <- expr
    args <- many expr
    return $ SApp f args

-- | Parsea expresi n completa
parseExpr :: String -> Either String SExpr
parseExpr input =
    case runParser (sc *> expr <* eof) "" input of
        Left err -> Left $ errorBundlePretty err
        Right result -> Right result

-- | Parsea programa (m ltiples expresiones)
parseProgram :: String -> Either String [SExpr]
parseProgram input =
    case runParser (sc *> many expr <* eof) "" input of
        Left err -> Left $ errorBundlePretty err
        Right result -> Right result
```

5.7 Main (Main.hs)

```haskell
{-# LANGUAGE LambdaCase #-}

module Main where

```

```haskell
import System.IO
import System.Environment (getArgs)
import Control.Monad (when, forever)
import Data.List (isPrefixOf)

import SurfaceAST
import Parser
import Desugar
import Eval
import AST (prettyExpr)

-- | REPL principal
main :: IO ()
main = do
  args <- getArgs
  case args of
    [] -> repl
    [filename] -> runFile filename
    _ -> putStrLn "Usage: minilisp [filename]"

-- | REPL interactivo
repl :: IO ()
repl = do
  putStrLn ""
  putStrLn "
     +=================================================================+"
  putStrLn "|                                                                 |"
  putStrLn "|                    Proyecto 1 - MiniLisp
                        |"
  putStrLn "|                                                                 |"
  putStrLn "|               Interprete Funcional en Haskell
                  |"
  putStrLn "|               UNAM - Facultad de Ciencias 2025
                 |"
  putStrLn "|                                                                 |"
  putStrLn "
     +=================================================================+"
  putStrLn ""
  putStrLn "
     +-----------------------------------------------------------------+"
  putStrLn "|  Comandos disponibles:
                                    |"
  putStrLn "
     +-----------------------------------------------------------------+"
  putStrLn "|  :salir             -> Salir del REPL
                          |"
  putStrLn "|  :ayuda             -> Mostrar ayuda completa
                   |"
  putStrLn "|  :cargar <archivo> -> Cargar y ejecutar archivo
                |"
  putStrLn "|  :nucleo <expr>    -> Ver nucleo desazucarizado
                |"
  putStrLn "
     +-----------------------------------------------------------------+"
  putStrLn ""
```

```
  putStrLn "Escribe una expresion Lisp para evaluarla..."
  putStrLn ""
  replLoop emptyEnv

replLoop :: Env -> IO ()
replLoop env = do
  putStr "> "
  hFlush stdout
  line <- getLine

  case line of
    "" -> replLoop env

    ':':'s':'a':'l':'i':'r':_ -> do
      putStrLn ""
      putStrLn "
          +================================================================+
          "
      putStrLn "|                                                                |"
      putStrLn "|                          Hasta  pronto!                        |"
      putStrLn "|                                                                |"
      putStrLn "|                   Gracias por usar MiniLisp.                   |"
      putStrLn "|              Creado por Giovanni, Vania, Camila                |"
      putStrLn "|                                                                |"
      putStrLn "
          +================================================================+
          "
      putStrLn ""
      return ()

    ':':'a':'y':'u':'d':'a':_ -> do
      showHelp
      replLoop env

    ':':'c':'a':'r':'g':'a':'r':' ':filename -> do
      loadFile env filename >>= replLoop

    ':':'n':'u':'c':'l':'e':'o':' ':input -> do
      showCore input
      replLoop env

    input -> do
      newEnv <- evalAndPrint env input
      replLoop newEnv

-- | Eval a e imprime resultado
evalAndPrint :: Env -> String -> IO Env
evalAndPrint env input = do
  case parseExpr input of
    Left err -> do
      putStrLn $ "[X] Error de sintaxis: " ++ err
      return env
```

```
95
96      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       print surfaceExpr
118       putStrLn ""
119       putStrLn "+--- AST Nucleo
          -----------------------------------------------+"
120       print coreExpr
121       putStrLn ""
122       putStrLn "+--- Formato Pretty
          ----------------------------------------+"
123       putStrLn $ prettyExpr coreExpr
124       putStrLn "
          +-------------------------------------------------------------+
          "
125       putStrLn ""
126
127 -- | Carga y ejecuta archivo
128 loadFile :: Env -> FilePath -> IO Env
129 loadFile env filename = do
130   content <- readFile filename
131   case parseProgram content of
132     Left err -> do
133       putStrLn $ "[X] Error de parseo en " ++ filename ++ ": " ++ err
134       return env
135
136     Right exprs -> do
137       putStrLn $ "[FILE] Cargando " ++ filename ++ "..."
138       putStrLn ""
139       execExprs env exprs
140   where
141     execExprs e [] = do
142       putStrLn "[OK] Archivo cargado exitosamente!"
143       putStrLn ""
144       return e
145     execExprs e (sexpr:rest) = do
146       let coreExpr = desugar sexpr
147       case eval e coreExpr of
```

```
148         Left err -> do
149           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   putStrLn "|
                                                                |"
175   putStrLn "|                       AYUDA DE MINILISP
                           |"
176   putStrLn "|
                                                                |"
177   putStrLn "
      +=============================================================+"
178   putStrLn ""
179   putStrLn "+-- SINTAXIS BASICA
      ----------------------------------------+"
180   putStrLn "|  Numeros:     42, -17, 0
                                    |"
181   putStrLn "|  Booleanos:   #t, #f
                                        |"
182   putStrLn "|  Variables:   x, foo, my-var
                                  |"
183   putStrLn "
      +-------------------------------------------------------------+"
184   putStrLn ""
185   putStrLn "+-- OPERADORES
      ---------------------------------------------+"
186   putStrLn "|  Aritmeticos: (+ 1 2 3), (- 10 5), (* 2 3 4), (/ 10 2)
      |"
187   putStrLn "|  Comparacion: (= 1 1), (< 1 2), (> 3 2), (<= 1 2)
      |"
188   putStrLn "|  Logicos:     (not #t)
                                      |"
189   putStrLn "|  Unarios:     (add1 5), (sub1 10), (sqrt 16)
                  |"
```

```
190   putStrLn "
          +-------------------------------------------------------------+"
191   putStrLn ""
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
```

```
            0
            (+ n (sum (sub1 n)))))))
    (sum 10))

; Resultado esperado: 55
```

5.8.2. Factorial (factorial.minisp)

```
; Factorial de n
; fact(n) = n * (n-1) * ... * 1

(letrec
    ((factorial (lambda (n)
        (if (<= n 1)
            1
            (* n (factorial (sub1 n)))))))
    (factorial 5))

; Resultado esperado: 120
```

5.8.3. Fibonacci (fibonacci.minisp)

```
; Fibonacci (n)
; fib(0) = 0, fib(1) = 1
; fib(n) = fib(n-1) + fib(n-2)

(letrec
    ((fib (lambda (n)
        (cond
            [(= n 0) 0]
            [(= n 1) 1]
            [else (+ (fib (- n 1))
                     (fib (- n 2)))]))))
    (fib 10))

; Resultado esperado: 55
```

5.8.4. Map para listas (map.minisp)

```
; Map: aplica funci n a cada elemento de lista
; map f [] = []
; map f (x:xs) = (f x) : (map f xs)

(letrec
    ((map (lambda (f lst)
        (if (= lst [])
            []
            [(f (head lst)) | (map f (tail lst))]))))
; Ejemplo: duplicar cada elemento
    (let ((double (lambda (x) (* x 2))))
        (map double [1, 2, 3, 4, 5])))
; Resultado esperado: [2, 4, 6, 8, 10]
```

5.8.5. Filter para listas (filter.minisp)

```
; Filter: mantiene elementos que cumplen predicado
; filter p [] = []
; filter p (x:xs) = (x : filter p xs) if p(x)
;                   (filter p xs)    otherwise

(letrec
    ((filter (lambda (pred lst)
        (if (= lst [])
            []
            (let ((x (head lst))
                  (xs (tail lst)))
                (if (pred x)
                    [x | (filter pred xs)]
                    (filter pred xs)))))))

; Ejemplo: n meros pares
    (let ((even? (lambda (x) (= (% x 2) 0))))
        (filter even? [1, 2, 3, 4, 5, 6])))
; 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 >= 9,2
- Cabal >= 3,6

5.10.2. Pasos de Instalación

```
# Clonar repositorio
git clone https://github.com/GiovanniAE/MiniLisp.git
cd minilisp

# Compilar proyecto
cabal build

# Ejecutar REPL
cabal run minilisp

# Ejecutar archivo
cabal run minilisp examples/factorial.minisp
```

5.11 Uso

5.11.1. REPL Interactivo

```
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

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

5.12 Ejemplos

Los archivos de ejemplo incluyen múltiples casos de prueba:

5.12.1. Factorial (factorial.minisp)

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

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

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

5.12.3. Fibonacci (fibonacci.minisp)

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

5.12.4. Potencia (power.minisp)

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

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

```
cabal run minilisp examples/mcd.minisp
# 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)

```
cabal-version: 2.2
name: minilisp
version: 1.2.0
synopsis: MiniLisp interpreter in Haskell
description: Complete implementation of MiniLisp for Programming
   Languages course
homepage: https://github.com/GiovanniAE/MiniLisp.git
license: MIT
author: Giovanni Alejandri Espinosa, Vania Zo  Velazquez Barrientos,
   Camila S nchez Flores
maintainer: giovanni.uni@ciencias.unam.mx
category: Language
build-type: Simple

common shared-properties
  default-language: Haskell2010
  ghc-options:
    -Wall
    -Wcompat
```

```
    -Widentities
    -Wincomplete-record-updates
    -Wincomplete-uni-patterns
    -Wmissing-export-lists
    -Wmissing-home-modules
    -Wpartial-fields
    -Wredundant-constraints
  build-depends:
    base >= 4.16 && < 5,
    containers >= 0.6,
    mtl >= 2.2

executable minilisp
  import: shared-properties
  hs-source-dirs: src
  main-is: Main.hs
  other-modules:
    AST,
    SurfaceAST,
    Parser,
    Desugar,
    Eval
  build-depends:
    megaparsec >= 9.2,
    parser-combinators >= 1.3,
    deepseq >= 1.4
  ghc-options: -threaded -rtsopts -with-rtsopts=-N

test-suite minilisp-test
  import: shared-properties
  type: exitcode-stdio-1.0
  hs-source-dirs: test
  main-is: Tests.hs
  build-depends:
    minilisp,
    hspec >= 2.10,
    QuickCheck >= 2.14
  ghc-options: -threaded -rtsopts -with-rtsopts=-N

library
  import: shared-properties
  hs-source-dirs: src
  exposed-modules:
    AST,
    SurfaceAST,
    Parser,
    Desugar,
    Eval
  build-depends:
    megaparsec >= 9.2,
    parser-combinators >= 1.3,
    deepseq >= 1.4
```

5.15 Concluciones

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.