UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Lenguajes de Programación

MiniLisp

Proyecto 1

Presenta:
Lugo Díaz Ordaz Gretel Alexandra
Ramírez Juárez María Fernanda
Rojo Peña Manuel Ianluck

Profesor:
Manuel Soto Romero

Ayudantes:
Diego Méndez Medina
Erick Daniel Arroyo Martínez

Grupo: 7121, 2026-1

Fecha de entrega:
11 de octubre, 2025

Índice

Capítulo 1

Resultados

Bien, una vez hemos visto toda la teoría de nuestro MiniLisp y además de que hemos mostrado el código mismo. En este capítulo presentamos los resultados obtenidos tras la implementación completa del lenguaje MiniLisp . Se mostrará el funcionamiento de cada una de las etapas principales del lenguaje (***lexer***, ***parser***, ***desugar***, ***interprete***), acompañadas de ejemplos que ilustran tanto su entrada como su salida.
Además, se presenta el ***menú interactivo*** del proyecto, que permite al usuario ejecutar comandos y evaluar expresiones sin hacer el proceso paso a paso. Este componente sirve como punto de unión entre todas las fases del lenguaje, permitiendo observar la interacción completa desde la lectura del código fuente hasta la obtención del resultado final.

En conjunto, buscamos que este capítulo tenga como propósito mostrar de forma integrada y funcional el resultado del trabajo desarrollado a lo largo de este reporte. Más que solo validar la implementación, buscamos evidenciar la coherencia entre el diseño teórico del lenguaje y su comportamiento práctico, demostrando que los principios formales de la sintaxis y la semántica pueden efectivamente traducirse en un sistema ejecutable, expresivo y consistente.

1.1. MiniLisp

Primero mostramos los resultados individuales de cada fase del código implementado que le da vida a nuestro lenguaje MiniLisp -0.2cm.
Utilizamos el intérprete interactivo de Haskell `GHCi` para compilar, cargar módulos y ejecutar las pruebas de de nuestra implementación. En el archivo `README.md` mostramos más a detalle como inicializar y compilar nuestro proyecto.

1.1.1. Lexer

Como se vió a inicios del reporte, el *analizador léxico* es el primer paso en la ejecución del lenguaje. Recibe el programa del usuario como cadena de caracteres y devuelve una lista de *Tokens* ya definido. Generamos el analizador léxico con `Alex`:

```
$ alex Lexer.x
```

Iniciamos el intérprete interactico de Haskell y para no tener que escribir siempre el nombre completo del módulo `Lexer`, cargalo:

```
$ ghci
GHCi, version 9.4.5: https://www.haskell.org/ghc/  :? for help
ghci> :l Lexer.hs
```

Ya con esto podemos probar nuestra función `lexer`, que no es muy impresionante, como dijimos, va verificando la entrada detectando los *Tokens*:

- **Variables:**

```
ghci> lexer "var12 #f 512 #t sum 90ba"
[TokenVar "var12",TokenBool False,TokenNum 512,
  TokenBool True,TokenVar "sum",TokenNum 90,TokenVar "ba"]
```

  Podemos ver que el `lexer` separa adecuadamente entre variables, números y booleanos asignándolos a su respectivo `Token`. Incluso *var12* se guarda completo como `TokenVar "var12"` justo como lo definimos, donde las variables comienzan siempre con un caracter seguido de una combinación de caracteres o números. Se puede apreciar también con el vaso de *90ba* donde lo separa como dos *Token* distintos, pues lo números no puden tener caracteres ni las variables pueden comenzar con números.

- **Operadores:**

  Veamos la lista de operadores generada por una cadena de operaciones (para acortar la extensión de esta sección):

```
ghci> lexer "(expt (+ (- (* 10 3) (/ (add1 5) (sub1 4))) (sqrt 81)))"
[TokenPA,TokenExpt,
  TokenPA,TokenAdd,
   TokenPA,TokenSub,
    TokenPA,TokenMul,TokenNum 10,TokenNum 3,TokenPC,
     TokenPA,TokenDiv,
      TokenPA,TokenAdd1,TokenNum 5,TokenPC,
      TokenPA,TokenSub1,TokenNum 4,TokenPC,
      TokenPC,TokenPC,
   TokenPA,TokenSqrt,TokenNum 81,TokenPC,TokenPC,TokenPC]
```

  Como se puede apreciar, los símbolos y palabras reservadas para operadores son detectadas correctamente por el `lexer`. No verificamos aún que los argumentos sean válidos en cantidad y tipo, pero si verificamos que los símbolos sean solo los definidos:

```
ghci> lexer "(+ 3)"
[TokenPA,TokenAdd,TokenNum 3,TokenPC]
ghci> lexer "(sqrt hola)"
[TokenPA,TokenSqrt,TokenVar "hola",TokenPC]
ghci> lexer "(% 3 5)"
[TokenPA,*** Exception: Lexical error:
  caracter no reconocido = "%" | codepoints = [37]
  CallStack (from HasCallStack):
  error, called at Lexer.hs:10816:24 in main:Lexer
```

- **Comparadores:**

  De manera análoga, tenemos los comparadores:

```
ghci> lexer "(!= 0 9 9) (= 6 6 1)
             (> 1 1) (< 4 3) (>= 7 3 5) (<= 22 22 2) (not #t)"
[TokenPA,TokenNeq,TokenNum 0,TokenNum 9,TokenNum 9,TokenPC,
  TokenPA,TokenEq,TokenNum 6,TokenNum 6,TokenNum 1,TokenPC,
  TokenPA,TokenGt,TokenNum 1,TokenNum 1,TokenPC,
  TokenPA,TokenLt,TokenNum 4,TokenNum 3,TokenPC,
  TokenPA,TokenGeq,TokenNum 7,TokenNum 3,TokenNum 5,TokenPC,
  TokenPA,TokenLeq,TokenNum 22,TokenNum 22,TokenNum 2,TokenPC,
  TokenPA,TokenNot,TokenBool True,TokenPC]
```

  Como se mencionó, en este punto no es relevante para el `lexer` los argumentos de cada comparador.

- **Condicionales:**

```
ghci> lexer "(cond [(= (sqrt 1000) (expt 10)) -1]
              [(!= (sub1 9) (add1 8)) 1] [else 0])"
[TokenPA,TokenCond,
  TokenLI,TokenPA,TokenEq,TokenPA,TokenSqrt,TokenNum 1000,TokenPC,
  TokenPA,TokenExpt,TokenNum 10,TokenPC,TokenPC,TokenNum (-1),TokenLD,
  TokenLI,TokenPA,TokenNeq,TokenPA,TokenSub1,TokenNum 9,TokenPC,
  TokenPA,TokenAdd1,TokenNum 8,TokenPC,TokenPC,TokenNum 1,TokenLD,
  TokenLI,TokenElse,TokenNum 0,TokenLD,TokenPC]
```

- **Pares y Listas:**

```
ghci> lexer "[(8,10),[],(#t,#f)]"
[TokenLI,
  TokenPA,TokenNum 8,TokenComma,TokenNum 10,TokenPC,TokenComma,
```

```
  TokenLI,TokenLD,TokenComma,
  TokenPA,TokenBool True,TokenComma,TokenBool False,TokenPC,
 TokenLD]
```

- **Lets y Expresiones Lambda:**

```
ghci> lexer "(let (x 10) (expt x))"
[TokenPA,TokenLet,TokenPA,TokenVar "x",TokenNum 10,TokenPC,
  TokenPA,TokenExpt,TokenVar "x",TokenPC,TokenPC]
```

```
ghci> lexer "(let* ((x 2)) (+ x 3))"
[TokenPA,TokenLetStar,TokenPA,TokenPA,TokenVar "x",TokenNum 2,TokenPC,TokenPC,
  TokenPA,TokenAdd,TokenVar "x",TokenNum 3,TokenPC,TokenPC]
```

```
ghci> lexer "((lambda (x) (+ x 1)) 5)"
[TokenPA,TokenPA,TokenLambda,TokenPA,TokenVar "x",TokenPC,
  TokenPA,TokenAdd,TokenVar "x",TokenNum 1,TokenPC,TokenPC,TokenNum 5,TokenPC]
```

Así, aunque los resultados del `lexer` no son tan emocionantes como lo pudieran ser para el `desugar` o `eval`, hemos mostrado con estos ejemplos que hasta el momento, el análisis sintáctico para el lenguaje funciona.
A partir de este momento no mostraremos los resultados de aplicar las fases a únicamente variables ya que es redundante su procedimiento pues podemos ver su progreso en las fases del lenguaje a través de las demás expresiones.

1.1.2. Parser

En la fase del `Parser` la situación se vuelve más interesante pues es donde aplicamos la gramática del lenguaje y decidimos las estructuras del programa que son válidas. Veamos ejemplos para algunas expresiones donde son rechazados por el `Parser`:

```
ghci> tokens = lexer "(+ 3)"
ghci> parse tokens
*** Exception: Error al Parsear los Tokens
CallStack (from HasCallStack):
error, called at ./Grammar.hs:1265:16 in main:Grammar
```

Falla porque la suma es un operador variádico que requiere de al menos dos elementos, por eso hay erroe en el `Parser`. De manera similar, fallan los operadores de resta, multiplicación y división con un solo argumentos, pues, requieren también de dos argumentos como mínimo.

Los ejemplos válidos serían:

```
ghci> tokens = lexer "(+ 52 34 42)"
ghci> parse tokens
Add [Num 52,Num 34,Num 42]
ghci> tokens = lexer "(- 22 -11 7)"
ghci> parse tokens
Sub [Num 22,Num (-11),Num 7]
ghci> tokens = lexer "(* 2 200)"
ghci> parse tokens
Mul [Num 2,Num 200]
ghci> tokens = lexer "(/ 21 0)"
ghci> parse tokens
Div [Num 21,Num 0]
```

Nótese que en la división no marcamos error al dividir por cero, pues recrodemos que eso es trabajo de la semántica, estamos en el *análisis sintáctico*.

Otro caso son los operadores unarios:

```
ghci> tokens = lexer "(sqrt 33 81)"
ghci> parse tokens
*** Exception: Error al Parsear los Tokens
CallStack (from HasCallStack):
error, called at ./Grammar.hs:1265:16 in main:Grammar
ghci> tokens = lexer "(expt 21 1 3)"
ghci> parse tokens
*** Exception: Error al Parsear los Tokens
CallStack (from HasCallStack):
error, called at ./Grammar.hs:1265:16 in main:Grammar
```

`Sqrt` y `Expt` fallan porque son operadores unarios, la gramática rechaza tener más de uno:

```
ghci> tokens = lexer "(sqrt 81)"
ghci> parse tokens
Sqrt (Num 81)
ghci> tokens = lexer "(expt 16)"
ghci> parse tokens
Expt (Num 16)
```

Las demás expresiones funcionan igual, dan error en el `Parser` con una gramática inválida, por ello veamos como queda el resultado de parsear programas válidos:

- **Comparadores:**

```
ghci> tokens = lexer "(= (+ 2 3) (* 5 1))"
ghci> parse tokens
Equal [Add [Num 2,Num 3],Mul [Num 5,Num 1]]
```

```
ghci> tokens = lexer "(< (+ 1 2) (* 2 3))"
ghci> parse tokens
Less [Add [Num 1,Num 2],Mul [Num 2,Num 3]]
```

```
ghci> tokens = lexer "(> (* 3 3) (+ 4 2))"
ghci> parse tokens
Greater [Mul [Num 3,Num 3],Add [Num 4,Num 2]]
```

```
ghci> tokens = lexer "(!= (* 2 3) (+ 3 3))"
ghci> parse tokens
Diff [Mul [Num 2,Num 3],Add [Num 3,Num 3]]
```

```
ghci> tokens = lexer "(<= (+ 2 3) (* 2 3))"
ghci> parse tokens
Leq [Add [Num 2,Num 3],Mul [Num 2,Num 3]]
```

```
ghci> tokens = lexer "(>= (* 3 3) (+ 4 2))"
ghci> parse tokens
Geq [Mul [Num 3,Num 3],Add [Num 4,Num 2]]
```

Notemos que las estructuras se muestran como se deben, con sus argumento guardados como listas y sus etiquetas respectivas a sus *Tokens*.

- **Condicionales:**

```
ghci> tokens = lexer "(if (> 3 2) (+ 1 2) (* 2 2))"
ghci> parse tokens
If (Greater [Num 3,Num 2]) (Add [Num 1,Num 2]) (Mul [Num 2,Num 2])
```

```
ghci> tokens = lexer "(if0 (- 3 3) (+ 1 2) (* 3 3))"
ghci> parse tokens
If0 (Sub [Num 3,Num 3]) (Add [Num 1,Num 2]) (Mul [Num 3,Num 3])
```

```
ghci> tokens = lexer "(cond [(< x 0) (- 0 x)] [(= x 0) 0] [else (+ x 1)])"
ghci> parse tokens
Cond [(Less [Var "x",Num 0],Sub [Num 0,Var "x"]),
  (Equal [Var "x",Num 0],Num 0)] (Add [Var "x",Num 1])
```

- **Pares y Listas:**

```
ghci> tokens = lexer "((+ 1 2), (* 3 4))"
ghci> parse tokens
Pair (Add [Num 1,Num 2]) (Mul [Num 3,Num 4])
```

```
ghci> tokens = lexer "(fst ((+ 1 2), (sqrt 9)))"
ghci> parse tokens
Fst (Pair (Add [Num 1,Num 2]) (Sqrt (Num 9)))
```

```
ghci> tokens = lexer "(snd ((sqrt 16), (+ 3 5)))"
ghci> parse tokens
Snd (Pair (Sqrt (Num 16)) (Add [Num 3,Num 5]))
```

```
ghci> tokens = lexer "[[1, 2, (3, 4)], #t, (+ 1 2)]"
ghci> parse tokens
List [List [Num 1,Num 2,Pair (Num 3) (Num 4)],
  Boolean True,Add [Num 1,Num 2]]
```

```
ghci> tokens = lexer "(head [[1, 2], (+ 3 4), #f])"
ghci> parse tokens
Head (List [List [Num 1,Num 2],Add [Num 3,Num 4],Boolean False])
```

```
ghci> tokens = lexer "(tail [[(+ 1 2)], (* 3 4), #t])"
ghci> parse tokens
Tail (List [List [Add [Num 1,Num 2]],Mul [Num 3,Num 4],Boolean True])
```

- **Lets y Expresiones Lambda:**

```
ghci> tokens = lexer "(let ((x 2) (y (* x 3))) (+ x y))"
ghci> parse tokens
Let [("x",Num 2),("y",Mul [Var "x",Num 3])] (Add [Var "x",Var "y"])
```

```
ghci> tokens = lexer "(let* ((x 2) (y (+ x 3)) (z (* y 2))) (+ x y z))"
ghci> parse tokens
LetStar [("x",Num 2),("y",Add [Var "x",Num 3]),
  ("z",Mul [Var "y",Num 2])] (Add [Var "x",Var "y",Var "z"])
```

```
ghci> tokens =
       lexer "(letrec (fact
               (lambda (n) (if0 n 1 (* n (fact (sub1 n)))))) (fact 5))"
ghci> parse tokens
LetRec "fact" (Lambda ["n"] (If0 (Var "n") (Num 1)
(Mul [Var "n",App (Var "fact") [Sub1 (Var "n")]])))
(App (Var "fact") [Num 5])
```

```
ghci> tokens = lexer "(lambda (x y) (if (> x y) (- x y) (+ x y)))"
ghci> parse tokens
Lambda ["x","y"] (If (Greater [Var "x",Var "y"])
(Sub [Var "x",Var "y"]) (Add [Var "x",Var "y"]))
```

```
ghci> tokens = lexer "((lambda (f x) (f x)) (lambda (y) (* y y)) 4)"
ghci> parse tokens
App (Lambda ["f","x"] (App (Var "f") [Var "x"]))
[Lambda ["y"] (Mul [Var "y",Var "y"]),Num 4]
```

Nótese que estas estructuras son **ASA** con azúcar, pues usamos listas en Haskell para ciertas estructuras. Aún nos falta la fase de desazucarización.

1.1.3. Desugar

1.2. Menú interactivo

Para mostrar los resultados del intérprete, utilizaremos un menú interactivo que tendrá la función de recibir la entrada del usuario, para procesarla a través de todas las fases de `Lexer`, `Parser` y `Desugar` para eventualmente realizar el proceso de evaluación semántica.

El menú interactivo queda definido en el archivo `MiniLisp.hs` como sigue:

```haskell
module MiniLisp where
import Token
import ASA
import AST
import ASV
import Lexer
import Grammar
import Desugar
import Interprete
import EvalStrict
import Saca
import Control.Exception (catch, SomeException)
-- Combinador Z
combZ :: String
combZ = "(lambda (f) ((lambda (x) (f (lambda (v) ((x x) v)))) (lambda (x)
   (f (lambda (v) ((x x) v))))))"
-- Evaluamos el combinador Z
z :: ASV
z = evalS (desugar $ parse $ lexer combZ) []
-- Punto de entrada principal
main :: IO ()
main =
  do
  putStrLn "\nBienvenido a MiniLisp. "
  putStrLn "Escriba (exit) para salir."
  minilisp
-- Bucle principal del interprete
minilisp =
  do
    putStr "[MiniLisp]> "
    str <- getLine
    if str == ""
      then minilisp
      else if str == ":q"
           then putStrLn "Bye :)"
      else do
      run str
      minilisp
-- Envuelve la evaluacion con manejo de errores
run :: String -> IO ()
run input =
  catch
    (do
      let tokens = lexer input
      let asa = parse tokens
      let ast = desugar asa
      let asv = evalS (ast) [("Z", z)]
      putStrLn (saca asv))
    errors
-- Manejador de errores
errors :: SomeException -> IO ()
errors e = putStrLn $ "[Error]: " ++ show e
```

Código 1.1: Menú interactivo de MiniLisp

Importamos todos los módulos a usar en el proyecto. Definimos el combinador **Z** y aplicamos us evaluación como es requerido para la recursión en MiniLisp -0.2cm.
El punto de entrada pincipal del lenguaje es la función `main`, donde damos la bienvenida al usuario y comnzamos el intérprete interactivo de MiniLisp -0.2cm. De este modo hacemos lo siguiente para correr el proyecto:

```
$ ghci
GHCi, version 9.4.5: https://www.haskell.org/ghc/  :? for help
ghci> :l MiniLisp.hs
ghci> main

Bienvenido a MiniLisp.
Escriba (exit) para salir.
[MiniLisp]>
```

El bucle principal del intérprete que, en una arrebato increíble de originalidad lo llamamos `minilisp`, es donde leemos la entrada del usuario, y la pasamos a una función `run` para que sea procesada. En este bucle antes de mandar la cadena a ser evaluada comprobamos que si es vacía o la cadena reservada para salir del intérprete.

La función `run`

Nos hace falta explicar la función `saca`. Esta función es auxiliar al momeno de mostrarle el resultado al usuairo, ya que no es conveniente regresar un valor canónico, necesitamos entonces implementar una manera de extraer el valor real del resultado de la evaluación y que este valor sea el que se muestre al usuario.

La función `saca` queda implementada en el archivo `Saca.hs` como sigue:

1.3. Funciones de prueba

1.3.1. Suma primeros n números naturales

1.3.2. Factorial

1.3.3. Fibonacci

1.3.4. Función `map` para listas

1.3.5. Función `filter` para listas

Bibliografía

[1] https://weblibrary.mila.edu.my/upload/ebook/engineering/2017_Book_FoundationsOfProgr

[2] Aho, A. V., Lam, S. M., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools. [Second Edition]. 2007.

[3] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n04.pdf

[4] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n05.pdf

[5] Documentación Haskell. Disponible en: https://www.haskell.org

[6] Documentación Alex(Haskell) The Alex Lexer Generator for Haskell Programming in Haskell (Graham Hutton, 2nd Edition). Sección sobre parsers y lexers. Disponible en: https://www.haskell.org/alex/

[7] Marlow, S., Gill, A. (2009). Happy. Disponible en: https://www.haskell.org/happy/

[8] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n08.pdf

[9] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n09.pdf

[10] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n10.pdf

[11] Disponible en: https://docs.racket-lang.org/reference/let.html

[12] Disponible en: https://www.lispworks.com/documentation/HyperSpec/Body/s_let_l.htm

Bibliografía

[1] Referencias de gretel para evitar conflictos al ultimo se uniran

Bibliografía

[1] *Especificación Formal de los Lenguajes de Programación. Sintaxis Concreta*,(M.Soto, lenguajes de programación, 2025).

[2] *Introduction to Automata Theory, Languages, and Computation*,(Hopcroft y Ullman).

[3] *Introduction to Automata Theory, Languages, and Computation*,(Hopcroft y Ullman)