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]
```

Podemos observar que las salidas generadas por el parser corresponden correctamente a la estructura del ***Árbol de Sintaxis Abstracta*** definido para MiniLisp . Cada expresión se traduce en una construcción interna con sus operadores y argumentos organizados en listas. En los ejemplos de comparadores y condicionales, se refleja cómo las expresiones se agrupan jerárquicamente, respetando el orden y los paréntesis del código fuente. Las secciones de pares y listas muestran la correcta interpretación de estructuras anidadas y de funciones de acceso como `fst`, `snd`, `head` y `tail`. Por último, las construcciones de `let`, `let*`, `letrec` y `lambda` evidencian el manejo de entornos locales y funciones como valores, preparando el terreno para su posterior desazucarización y evaluación semántica.

En conjunto, estos resultados confirman que la gramática y el parser generan correctamente los ASA esperados para cada tipo de expresión del lenguaje MiniLisp. Nótese que además que estas estructuras son **ASA** con azúcar, pues se puede apreciar por ejemplo, el uso listas en Haskell para ciertas estructuras. Aún nos falta la fase de desazucarización.

1.1.3. Desugar

Veremos los resultados interesantes del proceso de desazucarización:

```
ghci> tokens = lexer "(add1 (* 2 3))"
ghci> asa = parse tokens
ghci> desugar asa
AddC (MulC (NumC 2) (NumC 3)) (NumC 1)
```

```
ghci> tokens = lexer "(sub1 (+ 4 (* 2 3)))"
ghci> asa = parse tokens
ghci> desugar asa
SubC (AddC (NumC 4) (MulC (NumC 2) (NumC 3))) (NumC 1)
```

```
ghci> tokens = lexer "(expt (add1 (* 2 2)))"
ghci> asa = parse tokens
ghci> desugar asa
MulC (AddC (MulC (NumC 2) (NumC 2)) (NumC 1))
     (AddC (MulC (NumC 2) (NumC 2)) (NumC 1))
```

Como se puede ver, `add1`, `sub1` y `expt`, se convierten en suma, resta y multiplicación respectivamente.

```
ghci> tokens = lexer "(>= (* 3 3) (+ 4 2))"
ghci> asa = parse tokens
ghci> desugar asa
GeqC (MulC (NumC 3) (NumC 3)) (AddC (NumC 4) (NumC 2))
```

```
ghci> tokens = lexer "(= 4 0 (+ 9 3))"
ghci> asa = parse tokens
ghci> desugar asa
IfC (EqualC (NumC 4) (NumC 0)) (EqualC (NumC 0) (AddC (NumC 9) (NumC 3)))
    (BoolC False)
```

```
ghci> tokens = lexer "(!= 1 1 (- 7 7))"
ghci> asa = parse tokens
```

```
ghci> desugar asa
IfC (DiffC (NumC 1) (NumC 1)) (DiffC (NumC 1) (SubC (NumC 7) (NumC 7)))
    (BoolC False)
```

De igual manera, los comparadores ya no son una lista de comparaciones, sino un encadenamiento de condicionales. De manera similar con las condiciones `If0` y `Cond` que pasan a `IfC`.

```
ghci> tokens = lexer "(if0 (- 3 3) (+ 1 2) (* 3 3))"
ghci> asa = parse tokens
ghci> desugar asa
IfC (EqualC (SubC (NumC 3) (NumC 3)) (NumC 0)) (AddC (NumC 1) (NumC 2))
    (MulC (NumC 3) (NumC 3))
```

```
ghci> tokens = lexer "(cond [(< x 0) (- 0 x)] [(= x 0) 0] [else (+ x 1)])"
ghci> asa = parse tokens
ghci> desugar asa
IfC (LessC (VarC "x") (NumC 0)) (SubC (NumC 0) (VarC "x"))
    (IfC (EqualC (VarC "x") (NumC 0)) (NumC 0) (AddC (VarC "x") (NumC 1)))
```

En el caso de las listas notemos que se han convertido en una encadenación de *cons*.

```
ghci> tokens = lexer "[1, 2, 3]"
ghci> asa = parse tokens
ghci> desugar asa
ConS (NumC 1) (ConS (NumC 2) (NumC 3))
ghci> tokens = lexer "[[1, 2, (3, 4)], #t, (+ 1 2)]"
ghci> asa = parse tokens
ghci> desugar asa
ConS (ConS (NumC 1) (ConS (NumC 2) (PairC (NumC 3) (NumC 4))))
     (ConS (BoolC True) (AddC (NumC 1) (NumC 2)))
```

Para los lets, como se puede apreciar a continuación, se han convertido en aplicaciones de funciones.

```
ghci> tokens = lexer "(let ((x 5)) (+ x 1))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "x" (AddC (VarC "x") (NumC 1))) (NumC 5)
ghci> tokens = lexer "(let ((x 2) (y (* x 3))) (+ x y))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "x" (AppC (FunC "y" (AddC (VarC "x") (VarC "y")))
     (MulC (VarC "x") (NumC 3)))) (NumC 2)
```

```
ghci> tokens = lexer "(let* ((x 2) (y (+ x 3)) (z (* y 2))) (+ x y z))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "x" (AppC (FunC "y" (AppC (FunC "z" (AddC (VarC "x")
       (AddC (VarC "y") (VarC "z")))) (MulC (VarC "y") (NumC 2))))
     (AddC (VarC "x") (NumC 3)))) (NumC 2)
```

```
ghci> tokens = lexer "(letrec (fact (lambda (n)
                      (if0 n 1 (* n (fact (sub1 n)))))) (fact 5))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "fact" (AppC (VarC "fact") (NumC 5))) (AppC (VarC "Z")
       (FunC "fact" (FunC "n" (IfC (EqualC (VarC "n") (NumC 0))
     (NumC 1) (MulC (VarC "n") (AppC (VarC "fact") (SubC (VarC "n") (NumC 1)))
     )))))
```

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
```

```haskell
  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]>
```

Como primer instancia, definimos y evaluamos el **combinador Z**. Este combinador es una herramienta fundamental para implementar la recursión en MiniLisp , ya que el lenguaje carece de recursión directa a nivel del núcleo. El resultado de evaluar el combinador se guarda en la variable `z`, que se introduce en el ambiente inicial de evaluación. Así, los programas que

usan `letrec` pueden implementar funciones recursivas correctamente.
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` es la que coordina todas las fases de análisis, evaluación y muestreo al usuario. Desde iniciar el proceso pasando como argumento la cadena recibida al `Lexer`, como su evaluación en `AST` (con `EvalS`) dentro de un ambiente inicial que contiene la definición del combinador **Z**, necesario para la recursión. Esta función mete de inicio a ese ambiente, la variable **Z** con su respectiva evaluación. `run` se ejecuta dentro de un bloque `catch` permite capturar cualquier excepción que ocurra durante el análisis o la evaluación, evitando que el intérprete se detenga ante un error. En su lugar, se muestra un mensaje informativo en pantalla y el programa continúa su ejecución de forma segura.

Finalmente, para la visualización de los resultados, nos hace falta explicar la función `saca`. Esta función es una función auxiliar que se encarga de mostrar el resultado de la evaluación al usuario. Esto es necesario porque los valores evaluados en MiniLisp se representan mediante constructores internos del tipo `ASV`, los cuales no son legibles directamente. La función `saca` transforma estos valores en una representación textual clara y amigable para el usuario.

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

```haskell
module Saca where

import ASV

--Funcion para obtener el resultado e imprimirlo como cadena y no como
    tipo de dato ASV
saca :: ASV -> String
saca (NiV) = "[]"
saca (NumV n) = show n
saca (BoolV b)
  | b == True = "#t"
  | otherwise = "#f"
saca (ClosureF p c e) = "#<procedure>"
saca (ConV f s) = "[" ++ sacaElems (ConV f s) ++ "]"
saca (PairV f s) = "(" ++ saca f ++ "," ++ saca s ++ ")"
saca _ = "#<unknown>"

-- Funcion auxiliar para recorrer ConV
sacaElems :: ASV -> String
sacaElems NiV = "[]"
sacaElems (ConV x xs) = saca x ++ ", " ++ sacaElems xs
sacaElems x = saca x
```

Código 1.2: Procedimiento `saca` para mostrar el resultado al usuario

Nos apoyamos de *pattern matching* para proceder con los casos correspondientes, estos son solo los valores canónicos:

- `NiV`: devolvemos la lista vacía y nada más.

- `NumV`: devolvemos el valor n que guarda `NumV`, este ya es un número en Haskell y utilizamos `show` para mostrarlo en su representación de salida.
- `BoolV`: comprobamos que valor tiene b. Si es `True` devolvemos la cadena "$\#t$", en otro caso "$\#f$"para `False`.
- `ClosureF`: si el valor es una una cerradura, no se imprime su contenido interno sino que se representa como #`<procedure>`. Esto es un indicador textual, no un valor de lenguaje; se usa para que el usuario sepa que el valor es una función, pero no puede imprimirse directamente, no es un valor real del lenguaje, sino una representación simbólica para el usuario.
- `ConV`: cuando caemos en este valor, quiere decir que debemos representarlo como listas. Para ello nos auxiliaremos en la función `sacaElems` para formar una cadena con el formato adecuado y así devolver una representación fiel de los elementos.

  Esta función `sacaElems` recorre el encadenamiento de `ConV`, si hay dos elementos representamos estos recursivamente con la funciónsaca separados por comas, cuando hemos llegado al último elemento, lo regresamos tal cual.
- `PairV`: es más simple que `ConV`, solo es representar dos elementos recursivamente con `saca` entre paréntesis y separados por una coma.
- Si el valor canónico `ASV` recibido no es ninguno de los anteriores, entonces este no es válido, cosa que no debería ni puede suceder pero lo ponemos por completitud.

Ya con esto, veamos los resultados finales del lenguaje MiniLisp -0.2cm:

```
[MiniLisp]> (+ (* 2 3 -1) (- 10 4 9 -22) (sqrt 16) (expt 36) (/ 1 2))
1331
```

Comprobemos arugmento por argumento que la expresion anterior es correcta:

```
[MiniLisp]> (* 2 3 -1)
-6
[MiniLisp]> (- 10 4 9 -22)
37
[MiniLisp]> (sqrt 16)
4
[MiniLisp]> (expt 36)
1296
[MiniLisp]> (/ 1 2)
0
[MiniLisp]>
```

Al sumar los resultado se puede apreciar que los valores coinciden.

```
[MiniLisp]> (+ -6 37 4 1296 0)
1331
```

Veamos otro ejemplo:

```
[MiniLisp]> (let ((y 31) (x -22)) (cond [(< x 0 y) (- 0 x y)] [(= x 0) 0]
                                        [else (+ x 1)]))
53
```

Aquí, al resolver paso por paso, tenemos que se asigna 31 a y y -22 a x, por lo que la primer comprobación debería ser correcta al sustituir las variables:

```
[MiniLisp]> (< -22 0 31)
#t
```

Por lo que se resuelve su cláusula:

```
[MiniLisp]> (- 0 -22 31)
53
```

Por último veamos un ejemplo de evaluación para listas, se espera que cada elemento de la lista se evalúe, regresando una lista con valores evaluados. Hacemos el recordatorio también de que nuestras listas son heterogéneas:

```
[MiniLisp]> [[1, 2], (3, 4), (not #t), (+ 1 2 3 4 5 6), (!= 9 7 3 5),
             [], (/ 2 4), (not (not (= 9 9))), (tail [1, 2, 3]),
             ((lambda (f x) (f x)) (lambda (y) (* y y)) 4)]
[[1, 2], (3,4), #f, 21, #t, [], 0, #t, 3, 16]
```

Al evaluar cada elemento por separado, se puede apreciar que estos resultados coinciden:

```
[MiniLisp]> [1, 2]
[1, 2]
[MiniLisp]> (3, 4)
(3,4)
[MiniLisp]> (not #t)
#f
[MiniLisp]> (+ 1 2 3 4 5 6)
21
[MiniLisp]> (!= 9 7 3 5)
#t
[MiniLisp]> []
[]
[MiniLisp]> (/ 2 4)
```

```
0
[MiniLisp]> (not (not (= 9 9)))
#t
[MiniLisp]> (tail [1, 2, 3])
3
[MiniLisp]> ((lambda (f x) (f x)) (lambda (y) (* y y)) 4)
16
```

En el archivo `README.md` se incluyen ejemplos más detallados para cada expresión.

1.3. Funciones de prueba

En esta sección implementamos tres funciones especiales: la suma de los primeros n números naturales, el factorial de un número y el n-ésimo número de Fibonacci. Estas funciones nos sirven como ejemplos prácticos para probar las capacidades del lenguaje MiniLisp -0.2cm y en especial, de su capacidades recursivas.
Debido a que cada una de estas funciones se define naturalmente mediante recursión, nos auxiliamos de la expresión `letrec`, con la cual definimos funciones recursivas dentro del lenguaje.

Para poder ejecutar estas funciones directamente desde el menú interactivo, modificamos la función `run` de nuestro archivo `MiniLisp.hs`. Agregamos lógica que detecta cuando la entrada del usuario comienza con alguna palabra clave: `fact`, `sum` o `fibo`.

`fact(Int)` `sum(Int)` `fibo(Int)`

Cuando se detecta uno de estos comandos, se genera dinámicamente una expresión `MiniLisp` equivalente que utiliza `letrec` y luego se evalúa normalmente.

```haskell
-- Envuelve la evaluacion con manejo de errores
run :: String -> IO ()
run input =
  catch
    (do
        expr <-
          if "fact" `isPrefixOf` input
          then return $ fact (read (last (words input)))
          else if "sum" `isPrefixOf` input
               then return $ sumSum (read (last (words input)))
          else if "fibo" `isPrefixOf` input
               then return $ fibo (read (last (words input)))
          else return input

        let tokens = lexer expr
        let asa = parse tokens
        let ast = desugar asa
        let asv = evalS (ast) [("Z", z)]
        putStrLn (saca asv))
    errors
```

Código 1.3: Implementación de las funciones especiales de MiniLisp

La función `isPrefixOf` de Haskell, nos permite verificar si la cadena introducida por el usuario comienza con un determinado prefijo. De esta forma, si el usuario escribe `fact 5`, el intérprete genera internamente el código `MiniLisp` equivalente y lo evalúa como si el usuario lo hubiese escrito explícitamente.

1.3.1. Suma primeros n números naturales

Para calcular la suma de los primeros n números naturales se utiliza una definición recursiva simple:

$$\texttt{sum}(n) = \begin{cases} 0, & \text{si } n{=}0, \\ n{+}\texttt{sum}(n-1) & \text{en otro caso} \end{cases}$$

La implementación en Haskell genera una expresión en MiniLisp con `letrec` que define y ejecuta esta función:

```haskell
--Generamos la suma de los primeros n numeros con letrec
sumSum :: Int -> String
sumSum n = "(letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum
    " ++ show n ++ "))"
```

Código 1.4: Función para obtener el factorial de un número con `letrec`

Algunos ejemplos de ejecución son:

```
[MiniLisp]> sum 15
120
[MiniLisp]> sum 5
15
[MiniLisp]> sum 50
1275
[MiniLisp]> sum 3
6
```

El intérprete convierte internamente esta entrada en la expresión:

```
(letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum 5))
```

1.3.2. Factorial

De manera análoga, para el factorial usamos la definición recursiva clásica:

$$\texttt{fact}(n) = \begin{cases} 1, & \text{si } n{=}0, \\ n \times \texttt{fact}(n-1) & \text{en otro caso} \end{cases}$$

La función generadora en Haskell construye la expresión correspondiente:

```haskell
--Generamos la funcion factorial con letrec
fact :: Int -> String
fact n = "(letrec (fact (lambda (n) (if0 n 1 (* n (fact (- n 1)))))) (fact
    " ++ show n ++ "))"
```

Código 1.5: Función para obtener la suma de los primeros n números naturales con `letrec`

Algunos ejemplos de ejecución serían:

```
[MiniLisp]> fact 5
120
[MiniLisp]> fact 8
40320
[MiniLisp]> fact 3
6
[MiniLisp]> fact 7
5040
```

Donde la entrada `fact 7` por ejemplo, se traduce internamente como:

```
(letrec (fact (lambda (n) (if0 n 1 (* n (fact (- n 1)))))) (fact 5))
```

1.3.3. Fibonacci

La sucesión de Fibonacci se define recursivamente como:

$$\texttt{fibo}(n) = \begin{cases} 0, & \text{si } n{=}0, \\ 1, & \text{si } n{=}1, \\ \texttt{fibo}(n-1){+}\texttt{fibo}(n-2) & \text{en otro caso} \end{cases}$$

Su implementación en Haskell genera el código MiniLisp -0.2cm correspondiente con `letrec`:

```haskell
--Generamos el n-esimo numero de Fibonacci con letrec
fibo :: Int -> String
fibo n = "(letrec (fib (lambda (n) (if0 n 0 (if0 (- n 1) 1 (+ (fib (- n 1)
    ) (fib (- n 2)))))))) (fib " ++ show n ++ "))"
```

Código 1.6: Función para obtener el n-ésimo número de la suceción de Fibonacci con con `letrec`

Como ejemplos de ejecución tenemos:

```
[MiniLisp]> fibo 12
144
[MiniLisp]> fibo 18
2584
[MiniLisp]> fibo 7
```

```
13
[MiniLisp]> fibo 2
1
```

El intérprete genera internamente la siguiente definición:

```
(letrec (fib (lambda (n) (if0 n 0 (if0 (- n 1) 1 (+ (fib (- n 1)) (fib (- n 2)))))))
```

Estas funciones de prueba no sólo demuestran la expresividad del lenguaje donde el sistema de evaluación de MiniLisp -0.2cm maneja correctamente la recursión, sino también que la implementación del combinador **Z** y las expresiones `letrec` permiten definir y evaluar funciones complejas sin necesidad de estructuras externas. Además, muestran cómo el menú interactivo puede extenderse para admitir comandos personalizados que simplifican la interacción del usuario con el intérprete.

Capítulo 2

Conclusiones

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] Landin.P.J.(1965) *The Next 700 Programming Languages*.Univac Division of Sperry Rand Corp,9(3),157-166.

[4] Winske.G.(1993) *The Formal Semantics of Programming Languages*, Massachusetts Institute of Technology.

[5] Plotkin.G.D.(1981) *A Structural Approach to Operational Semantics*, University of Aarhus.