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

Introducción

Como hemos visto en el curso Leguajes de Programación, al menos hasta la fecha de entrega del presente proyecto, en el desarrollo de un lenguaje de programación resulta fundamental comprender cómo se definen formalmente sus componentes y cómo estos se traducen a estructuras que una computadora puede interpretar y ejecutar.

1.1. Motivación

En el desarrollo de lenguajes de programación, Una de las motivaciones principales de este proyecto es acercarse al diseño de un lenguaje minimalista —en este caso, MINILISP— que permita practicar la construcción de gramáticas formales, analizadores léxicos y sintácticos, así como el modelado de árboles de sintaxis abstracta en un entorno académico.

1.2. Objetivos

El objetivo del proyecto es implementar un subconjunto del lenguaje Lisp con un conjunto reducido pero representativo de operaciones: expresiones aritméticas y booleanas, estructuras de control (if, cond), mecanismos de definición local (let, letrec, let*), funciones anónimas (lambda), listas y pares. Para ello, se diseña una gramática en notación BNF/EBNF, se define un conjunto de tokens para el análisis léxico y se construyen las estructuras de datos necesarias en Haskell para representar el árbol de sintaxis abstracta (ASA). De esta manera, se busca no solo capturar la semántica básica del lenguaje, sino también poner en práctica técnicas de diseño de compiladores a pequeña escala.

1.3. Delimitación del Proyecto

La delimitación del proyecto consiste en que MINILISP no pretende ser una implementación completa de Lisp, sino una versión simplificada con fines didácticos. Se restringe el conjunto de operaciones soportadas, se omite el manejo de macros y de entrada/salida, y se centra únicamente en el análisis sintáctico y la representación interna de programas. Con

esto, se logra un balance entre la complejidad teórica y la viabilidad de implementación en el tiempo disponible.

Capítulo 2

Sintaxis Concreta/Léxica

El análisis léxico constituye la fase inicial en el proceso de interpretación de lenguajes de programación. Su objetivo es transformar una secuencia de caracteres sin estructura en una secuencia de tokens, que representan las unidades mínimas con significado léxico en nuestro lenguaje. Cada token encapsula información sobre el tipo de elemento reconocido y, cuando es relevante, su valor específico.

Herramienta Utilizada: Alex Para la implementación del analizador léxico de nuestro miniLisp, seleccionamos Alex, el generador de analizadores léxicos estándar para Haskell. Esta elección se fundamenta en varias ventajas significativas: + Reducción de errores: Alex automatiza la generación de código robusto, minimizando errores comunes en implementaciones manuales. + Expresividad: Utiliza expresiones regulares extendidas para definir patrones léxicos de manera clara y concisa. + Integración con Haskell: Genera código Haskell nativo que se integra perfectamente con el resto de nuestro intérprete. + Eficiencia: Produce analizadores de alto rendimiento mediante algoritmos de coincidencia optimizados.

Estructura del Archivo Lexer.x Lo primero que hacemos es importar las definiciones de tokens desde Tokens, que contiene el tipo de datos Token con todos los constructores necesarios. Después, definimos los patrones básicos que establecen los bloques fundamentales para construir patrones más complejos, promoviendo la reutilización y claridad. Estas líneas no son código Haskell, sino instrucciones para Alex. Le indican a Alex: Çuando veas $digit en las reglas, reemplázalo por 0-9". Continuamos con los delimitadores estructurales compuestos por: + Patrón o expresión regular (lo que buscamos): La secuencia de caracteres que el lexer debe reconocer; en este caso nos referimos a "(", "let", "+", etc. Es importante mencionar que los caracteres los podemos definir a gusto personal, pero para evitar crear un lenguaje confuso usaremos para cada token los ya reconocidos; es decir, para la suma "+", para la resta "-", etc. + Bloque de acción: Es el código Haskell que se ejecuta cuando se reconoce el patrón. Su propósito es generar el token correspondiente .... + Expresión lambda: Es todo lo que está en el bloque de acción, es decir, "_ ->TokenPA", donde _ representa la cadena de texto que coincidió con el patrón, ->separa el parámetro del resultado, y TokenPA es el constructor del token que devolvemos.

Continuamos con las literales e identificadores: son los elementos fundamentales que representan los valores básicos y nombres en nuestro lenguaje. Son las "palabras" que contienen datos específicos en el programa. Para realizar el lexer tomamos como referencia lo visto en clase con el profesor (en su GitHub), además de usar la documentación oficial de Alex para

desarrollar nuestro lexer.

```haskell
data Token
  = TokenVar String
  | TokenNum Int
  | TokenBool Bool
  | TokenAdd
  | TokenSub
  | TokenMul
  | TokenDiv
  | TokenAdd1
  | TokenSub1
  | TokenSqrt
  | TokenExpt
  | TokenNot
  | TokenEq
  | TokenLt
  | TokenGt
  | TokenNeq
  | TokenLeq
  | TokenGeq
  | TokenFirst
  | TokenSecond
  | TokenHead
  | TokenTail
  | TokenLet
  | TokenLetRec
  | TokenLetStar
  | TokenIf0
  | TokenIf
  | TokenCond
  | TokenElse
  | TokenLambda
  | TokenApp
  | TokenLI
  | TokenLD
  | TokenComma
  | TokenPA
  | TokenPC
  deriving (Show, Eq)
```

Código 2.1: Estructura de Tokens

Para ello usamos la dicumetación oficial de Alex. The Alex Lexer Generator for Haskell https://www.haskell.org/alex/ Programming in Haskell (Graham Hutton, 2nd Edition). Sección sobre parsers y lexers.

Capítulo 3

Sintaxis Libre de Contexto

Definimos la Gramática para MiniLisp en **EBNF**:

Gramática MINILISP

```
<Expr>  ::=  <Var>
          |  <Num>
          |  <Bool>
          |  (+ <Expr> <Expr> {<Expr>})
          |  (- <Expr> <Expr> {<Expr>})
          |  (* <Expr> <Expr> {<Expr>})
          |  (/ <Expr> <Expr> {<Expr>})
          |  (add1 <Expr>)
          |  (sub1 <Expr>)
          |  (sqrt <Expr>)
          |  (expt <Expr>)
          |  (not <Expr>)
          |  (= <Expr> <Expr> {<Expr>})
          |  (<  <Expr> <Expr> {<Expr>})
          |  (>  <Expr> <Expr> {<Expr>})
          |  (<= <Expr> <Expr> {<Expr>})
          |  (>= <Expr> <Expr> {<Expr>})
          |  (!= <Expr> <Expr> {<Expr>})
          |  (<Expr>, <Expr>)
          |  (fst <Expr>)
          |  (snd <Expr>)
          |  (let ((<Var> <Expr>) {<Var> <Expr>}) <Expr>)
          |  (letrec <Var> <Expr> <Expr>)
          |  (let* ((<Var> <Expr>) {<Var> <Expr>}) <Expr>)
          |  (if0 <Expr> <Expr> <Expr>)
          |  (if <Expr> <Expr> <Expr>)
          |  (lambda (<Var> {<Var>}) <Expr>)
          |  (<Expr> <Expr>)
          |  [<Expr> {, <Expr>}]
          |  (head <Expr>)
          |  (tail <Expr>)
          |  (cond [<Expr> <Expr>] {[<Expr> <Expr>]} [else <Expr>])

 <Var>  ::=  Identificador de variable
 <Num>  ::=  Constante entera
<Bool>  ::=  #t | #f
```

Hacemos un abuso de notación para aclarar que el uso de '**[**' '**]**' no es para indicar opcionalidad de la notación de EBNF sino que son los símbolos que usamos para representar

listas, también por eso están en negritas.

Capítulo 4

Sintaxis Abstracta

Capítulo 5

Azúcar sintáctica

5.1. Sintaxis Abstracta sin azúcar

5.2. Desugar

Capítulo 6

Semántica operacional

6.1. Paso pequeño

6.1.1. Evaluación perezosa

6.1.2. Evaluación ansiosa

Capítulo 7

Intérprete

Capítulo 8

Resultados

8.1. Funciones de prueba

8.1.1. Suma primeros n números naturales

8.1.2. Factorial

8.1.3. Fibonacci

8.1.4. Función `map` para listas

8.1.5. Función `filter` para listas

Capítulo 9

Conclusiones

Bibliografía

[1] Autor. *Título del libro*. Editorial, Año.

[2] Autor. .ªrtículo". Revista, Año.