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.

En conclusión, con la ***sintaxis concreta*** de nuestro lenguaje MiniLisp definimos las reglas exactas de escritura con las que creamos un lenguaje de programación. Lo que nos da una visión como programadores de como se crea esta herramienta que mas usamos, los lenguajes, y más adelante ver como el intérprete maneja lo que nosotros entedemos a como, de manera al menos sencilla por ser MiniLisp , lo convierte en un resultado.

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

Antes de entrar de fondo en programar nuestro MiniLisp en `Haskell`, es necesario definir la ***sintaxis concreta*** que utilizaremos para el lenguaje.

Citando al profesor, en su archivo `PDF` compartido ***Especificación Formal de los Lenguajes de Programación. Sintaxis Concreta***.

> *En el contexto de la teoría de lenguajes de programación y lenguajes formales, la* ***sintaxis concreta*** *se refiere a la estructura específica de un lenguaje de programación que define exactamente cómo se deben escribir los programas. Matemáticamente, esto se describe mediante una gramática formal que especifica las reglas de formación para las secuencias válidas de símbolos en el lenguaje.*

Esta especificación formal se divide en ***sintaxis léxica*** y ***sintaxis libre de contexto***, con los cuales podemos construir programas válidos y sin ambigüedades, asegurando que nuestro lenguaje pueda transformarse sin problemas en su correspondiente representación abstracta. En términos simples, la sintaxis describe *cómo se ve el programa*, es la forma exacta en la que el usuario debe escribir las expresiones, instrucciones y estructuras del lenguaje.

Podemos decir que la sintaxis, constituye la **puerta de entrada entre el usuario y el compilador o intérprete**, definiendo los símbolos, operadores, delimitadores y palabras reservadas que el lenguaje reconoce. Mientras que la *sintaxis abstracta* (**ASA**, Árbol de Sintaxis Abstracta) representa la estructura lógica del programa, la sintaxis concreta establece las **reglas formales de escritura** que garantizan que un programa pueda ser reconocido y analizado correctamente. Su correcta definición es fundamental para el funcionamiento del analizador léxico (*Lexer*) y del analizador sintáctico (*Parser*), ya que determina las entradas válidas que ambos deben procesar.

Para nuestro lenguaje MiniLisp , hemos definido las expresiones:

- **Variables:** cualquier secuencia de caracteres de la forma $[a-z+A-Z][a-zA-Z0-9]^*$.
- **Números enteros:** $x \in \mathbb{Z}$.
- **Booleanos:** `#t` (verdadero) y `#f` (falso), junto con la negación `(not)`.

- **Operadores aritméticos:** `+`, `-`, `*`, `/`, `++`, `--`, raíz cuadrada (`sqrt`) y potencia (`**`).
- **Predicados y comparaciones:** igualdad y desigualdad (`=`, `!=`), así como comparaciones numéricas (`<`, `>`, `<=`, `>=`).
- **Asignaciones y funciones:** construcciones `let`, `let*`, `letrec`, funciones anónimas con `lambda`, y aplicación de funciones.
- **Pares ordenados y proyecciones:** `(e1, e2)`, `first` y `second`.
- **Condicionales:** `if`, `if0` y `cond`.
- **Listas:** delimitadas por corchetes `[` y `]`, con elementos separados por comas `,`, junto con operaciones básicas `head` y `tail`.

Cabe destacar que, algunas de las operaciones dadas, tendrán la característica de ser variádicas. Entraremos en este tema más adelante.

2.1. Sintaxis 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.

Para ayudarnos con el análisis léxico y transformar esa cadena ingresada por el usuario, utilizamos la herramienta Alex.

2.1.1. Tokens

```
data Token
  = TokenVar String | TokenNum Int | TokenBool Bool
  | TokenAdd
  | TokenSub
  | TokenMul
  | TokenDiv
  | TokenAdd1
  | TokenSub1
  | TokenSqrt
  | TokenExpt
  | TokenNot
  | TokenEq
  | TokenLt
  | TokenGt
  | TokenNeq
  | TokenLeq
  | TokenGeq
  | TokenIf0
  | TokenIf
```

```
20    | TokenCond
21    | TokenElse
22    | TokenFirst
23    | TokenSecond
24    | TokenHead
25    | TokenTail
26    | TokenLet
27    | TokenLetRec
28    | TokenLetStar
29    | TokenLambda
30    | TokenLI
31    | TokenLD
32    | TokenComma
33    | TokenPA
34    | TokenPC
35    deriving (Show, Eq)
```

Código 2.1: Estructura de Tokens

2.1.2. Alex como Analizador Léxico

Alex es 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.

Para ello usamos la dicumetación oficial de Alex [1]. The Alex Lexer Generator for Haskell https://www.haskell.org/alex/ Programming in Haskell (Graham Hutton, 2nd Edition). Sección sobre parsers y lexers.

2.2. 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>})
          |  (++ <Expr>)
          |  (-- <Expr>)
          |  (sqrt <Expr>)
          |  (** <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 `[` y `]` no es para indicar opcionalidad de la notación de EBNF sino que son los símbolos que usamos para representar listas, en general, en nuestra gramática no tenemos expreciones opcionales.

Capítulo 3

Sintaxis Abstracta

3.1. Parser con Happy

Capítulo 4

Azúcar sintáctica

4.1. Sintaxis Abstracta sin azúcar

4.2. Desugar

Capítulo 5

Semántica operacional

5.1. Paso pequeño

5.1.1. Evaluación perezosa

5.1.2. Evaluación ansiosa

Capítulo 6

Intérprete

6.1. Paso pequeño

6.2. Evaluación

Capítulo 7

Resultados

7.1. Menú interactivo

7.2. Funciones de prueba

7.2.1. Suma primeros n números naturales

7.2.2. Factorial

7.2.3. Fibonacci

7.2.4. Función `map` para listas

7.2.5. Función `filter` para listas

Capítulo 8

Conclusiones

Bibliografía

[1] Documentación Alex(Haskell). Disponible en: https://www.haskell.org/alex/

[2] Autor. .ªrtículo". Revista, Año.