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

Lorem ipsum dolor sit amet consectetur adipiscing elit, varius dictum egestas quisque hac rutrum. Posuere sociosqu leo vitae dui metus proin massa feugiat laoreet non sodales potenti, fusce cursus pharetra risus elementum hendrerit litora pretium cubilia dapibus vulputate luctus, dictumst egestas enim nibh tellus id in mattis orci ut quis. Non phasellus molestie luctus curae turpis viverra condimentum pretium, proin dis auctor odio nunc facilisis eros morbi mauris, nam penatibus senectus vel placerat quisque convallis.

Convallis feugiat ullamcorper massa nisi dictum justo vitae, suscipit luctus proin libero tortor mattis, felis dictumst lobortis ac est leo. Sagittis imperdiet urna vehicula pretium platea porta fames nam mollis massa facilisis laoreet accumsan, aptent molestie ligula dictumst neque ad ullamcorper dictum class interdum quis dignissim. Laoreet fringilla nam orci ut aptent luctus, venenatis quis a dictum bibendum scelerisque nec, aliquet habitant felis sagittis suscipit.

1.1. Motivación

Lorem ipsum dolor sit amet consectetur adipiscing elit, varius dictum egestas quisque hac rutrum. Posuere sociosqu leo vitae dui metus proin massa feugiat laoreet non sodales potenti, fusce cursus pharetra risus elementum hendrerit litora pretium cubilia dapibus vulputate luctus, dictumst egestas enim nibh tellus id in mattis orci ut quis. Non phasellus molestie luctus curae turpis viverra condimentum pretium, proin dis auctor odio nunc facilisis eros morbi mauris, nam penatibus senectus vel placerat quisque convallis.

Convallis feugiat ullamcorper massa nisi dictum justo vitae, suscipit luctus proin libero tortor mattis, felis dictumst lobortis ac est leo. Sagittis imperdiet urna vehicula pretium platea porta fames nam mollis massa facilisis laoreet accumsan, aptent molestie ligula dictumst neque ad ullamcorper dictum class interdum quis dignissim. Laoreet fringilla nam orci ut aptent luctus, venenatis quis a dictum bibendum scelerisque nec, aliquet habitant felis sagittis suscipit.

1.2. Objetivos

Lorem ipsum dolor sit amet consectetur adipiscing elit, varius dictum egestas quisque hac rutrum. Posuere sociosqu leo vitae dui metus proin massa feugiat laoreet non sodales potenti, fusce cursus pharetra risus elementum hendrerit litora pretium cubilia dapibus vulputate luctus, dictumst egestas enim nibh tellus id in mattis orci ut quis. Non phasellus molestie luctus curae turpis viverra condimentum pretium, proin dis auctor odio nunc facilisis eros morbi mauris, nam penatibus senectus vel placerat quisque convallis.

Convallis feugiat ullamcorper massa nisi dictum justo vitae, suscipit luctus proin libero tortor mattis, felis dictumst lobortis ac est leo. Sagittis imperdiet urna vehicula pretium platea porta fames nam mollis massa facilisis laoreet accumsan, aptent molestie ligula dictumst neque ad ullamcorper dictum class interdum quis dignissim. Laoreet fringilla nam orci ut aptent luctus, venenatis quis a dictum bibendum scelerisque nec, aliquet habitant felis sagittis suscipit.

1.3. Delimitación del Proyecto

Lorem ipsum dolor sit amet consectetur adipiscing elit, varius dictum egestas quisque hac rutrum. Posuere sociosqu leo vitae dui metus proin massa feugiat laoreet non sodales potenti, fusce cursus pharetra risus elementum hendrerit litora pretium cubilia dapibus vulputate luctus, dictumst egestas enim nibh tellus id in mattis orci ut quis. Non phasellus molestie luctus curae turpis viverra condimentum pretium, proin dis auctor odio nunc facilisis eros morbi mauris, nam penatibus senectus vel placerat quisque convallis.

Convallis feugiat ullamcorper massa nisi dictum justo vitae, suscipit luctus proin libero tortor mattis, felis dictumst lobortis ac est leo. Sagittis imperdiet urna vehicula pretium platea porta fames nam mollis massa facilisis laoreet accumsan, aptent molestie ligula dictumst neque ad ullamcorper dictum class interdum quis dignissim. Laoreet fringilla nam orci ut aptent luctus, venenatis quis a dictum bibendum scelerisque nec, aliquet habitant felis sagittis suscipit.

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.

Para nuestro lenguaje MiniLisp, como una introducción a la implementación que mostraremos, hemos definido las expresiones:

- **Variables.**
- **Números entero.**
- **Booleanos.**
- **Operadores aritméticos.**
- **Predicados y comparaciones.**
- **Asignaciones y funciones.**

- **Pares ordenados y proyecciones.**
- **Condicionales.**
- **Listas.**

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

En conclusión, podemos pensar en la sintaxis concreta como las secuencias de caracteres del alfabeto Σ que se convierten en programas válidos del lenguaje. 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. Esto se logra mediante un **Análisis léxico** y un **Análisis sintáctico**.

2.1. Sintaxis Léxica

La definición léxica se establece mediante un conjunto de **expresiones regulares**, las cuales constituyen la base formal sobre la que se construyen los componentes básicos de un lenguaje de programación. Dichas expresiones definen los patrones válidos de caracteres que pueden formar identificadores, números, operadores, palabras reservadas y otros símbolos que componen el vocabulario fundamental del lenguaje.

En pocas palabras, citando al profesor:

> "*Formalmente, la sintaxis léxica se define usando* ***expresiones regulares*** *y* ***autómatas finitos***."

La **sintaxis léxica**, dentro del estudio de los lenguajes formales, representa la primera capa estructural de un lenguaje de programación. Su propósito es definir el *alfabeto* del lenguaje y describir cómo las secuencias de símbolos de dicho alfabeto se agrupan en unidades con significado propio. No describe la estructura lógica o gramatical del programa (de estos e encarga la ***sintaxis libre de contexto***), sino que se encarga de definir los elementos básicos que lo conforman.

En términos prácticos, esta especificación léxica permitirá posteriormente implementar un *analizador léxico*, encargado de recorrer la entrada del usuario y separar cada componente del programa según las reglas aquí definidas.

2.1.1. Análisis Léxico

Nuestra sintaxis se constituye de un **Análisis léxico**. El análisis léxico constituye la fase inicial en el proceso de interpretación de lenguajes de programación. Cumple una función fundamental dentro del proceso de compilación o interpretación, ya que actúa como un filtro inicial entre el texto fuente escrito por el usuario y las estructuras sintácticas que procesará

el analizador sintáctico.

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 (palabras reservadas, identificadores , literales, operadores y delimitadores) que simplifican el trabajo del parser. Cada token encapsula información sobre el tipo de elemento reconocido y, cuando es relevante, su valor específico.

> Definimos una función `lexer:` $\sum^* \to$ `[`$Token$`]`, que toma una cadena de caracteres y produce la lista de *tokens* según las expresiones regulares que hayamos definido en nuestro lenguaje.

Anteriormente hicimos una breve mención de las expresiones que nuestro MiniLisp va a manejar en la sintaxis léxica. Ya que el propósito de este proyecto es académico, basta con implementar los tipos de datos más simples, como lo son los números (`Num`)y booleanos (`Boolean`), también implementamos cadenas (`String`) pero no tendremos ningún programa que opere con cadenas de caracteres, únicamente las usaremos como asignación de variables.

Tenemos entonces, el alfabeto Σ de nuestro lenguaje:

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a - z, A - Z, \texttt{-}, \texttt{+}, \texttt{*}, \texttt{/}, \texttt{=}, \texttt{>}, \texttt{<}, \texttt{!}, \texttt{\#}, \texttt{[}, \texttt{]}, \texttt{,}, \texttt{(}, \texttt{)}\}$$

Ahora, los *Tokens* de nuestro lenguaje serán justamente las cadenas reservados o caracteres que podemos formar con dichos símbolos. Una vez tenemos encuenta todo lo anterior, definimos los siguientes tipos de *Tokens* para nuestro lenguaje MiniLisp:

- **Paréntesis:** `(` y `)`, con los que indicamos cuando comienzan y terminan nuestras expresion (por eso se llaman *delimitadores*).
- **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`.

Como primera parte de nuestra implementación en Haskell del ***análisis léxico***, utilizamos la palabra reservada `data` que nos permite definir nuevos tipos de datos y los constructores asociados a ellos.

2.1.2. Tokens

La estructura del tipo *Token* son las piezas fundamentales que permiten construir la sintaxis del lenguaje de manera estructurada y libre de ambigüedades. Incluso, no solo nos permiten clasificar y representar las unidades léxicas mínimas reconocibles por el lenguaje, sino que también facilitan el trabajo del *parser*.

Para nuestro proyecto MiniLisp definimos el tipo de dato `Token` en Haskell dentro del archivo `Tokens.hs`, con el cual representamos cada posible componente léxico del lenguaje.

Queda definido como sigue:

```haskell
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
  | TokenCond
  | TokenElse
  | TokenFirst
  | TokenSecond
  | TokenHead
  | TokenTail
  | TokenLet
  | TokenLetRec
  | TokenLetStar
  | TokenLambda
  | TokenLI
  | TokenLD
  | TokenComma
  | TokenPA
  | TokenPC
  deriving (Show, Eq)
```

Código 2.1: Estructura de Tokens

Nótese que, los *Tokens*: `TokenVar`, `TokenNum`, `TokenBool`, además de encapsular el tipo de elemento reconocido, guardan su valor específico asociado a dichos *Tokens* con los tipos de datos en el lenguaje anfitrión (`String`, `Int` y `Bool`).

2.1.3. Alex

Cada vez que el analizador léxico identifica un patrón en la entrada, genera el token correspondiente y al final, esta función `lexer`, construirá una lista de *Tokens* la cual recibirá wl analizador sintáctico. Utilizamos la herramienta `Alex` que nos ayudará con la implementación de este `lexer` en Haskell.

Alex es el generador de analizadores léxicos estándar para Haskell, toma una descripción de tokens basada en expresiones regulares y genera un Haskell `module` que contiene código para escanear texto de manera eficiente. 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.

Implementamos Alex en el archivo `Lexer.x`, su estructura es la siguiente:

Lo primero que hacemos es importar los *Tokens* definidos y construidos en el archivo `Tokens.hs` e importar `Data.Char` para usar la función `isSpace` con la que normalizaremos espacios Unicode.

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, con ellos le indicamos a Alex: "*Cuando veas $digit en las reglas, reemplázalo por 0-9*". Lo mismo para $*alpha* con `[a-zA-Z]` y $*alphnum* `[a-zZ-Z0-9]`.

Además de incluir con la definición de los espacios (*whitespaces*): espacio ASCII (\ x20), tabulador (\ x09), LF (\ x0A), CR (\ x0D), FF (\ x0C), VT (\ x0B). Definirlos explícitamente nos ayuda a ignorarlos al definir la regla de construcción o de lectura para generar los *Tokens* de la cadena recibida.

Por último `tokens :-` marca el comienzo de la sección de patrones de las expresiones regulares que Alex convertirá en la lista de *Tokens* `regex { Token }`. Declarando también la regla de ignorar los espacios y salto de (`$white+`).

```
{
module Lexer where

import Token
import Data.Char (isSpace)
}

%wrapper "basic"

-- Definiciones de patrones
$digit    = 0-9
$alpha    = [a-zA-Z]
$alnum    = [a-zA-Z0-9]

-- Usamos codigos hex para los espacios en blanco Unicode mas comunes:
--   \x20 = ' ' (space), \x09 = tab, \x0A = LF, \x0D = CR, \x0C = FF, \x0B = VT
$white = [\x20\x09\x0A\x0D\x0C\x0B]

tokens :-

-- Ignoramos espacios y saltos de linea
$white+                         ;
```

Código 2.2: Lexer con Alex.

Continuamos con la definición de los **delimitadores estructurales** y los **operadores básicos** de nuestro lenguaje, los cuales constituyen los símbolos fundamentales que permiten organizar y expresar la estructura de los programas en MiniLisp .

Cada una de estas reglas dentro del analizador léxico de Alex consta de dos componentes principales:

- **Patrón o expresión regular:** Es la secuencia de caracteres que el lexer debe reconocer. En este caso, se trata de los símbolos estructurales o palabras clave como `(`, `let`, `+`, etc. Cabe mencionar que estos caracteres pueden definirse de manera personalizada; sin embargo, para mantener la coherencia con la notación tradicional de los lenguajes de programación, utilizamos los símbolos comúnmente aceptados, como `+` para la suma y `-` para la resta.

- **Bloque de acción:** Es el fragmento de código en Haskell que se ejecuta cuando se reconoce el patrón. Su función es generar el token correspondiente, por ejemplo: `{ _ ->TokenPA }`.

- **Expresión lambda:** Dentro del bloque de acción, la expresión lambda define cómo se construye el token. En el ejemplo anterior, `_ ->TokenPA`, el símbolo `_` representa la cadena de texto que coincidió con el patrón (la entrada reconocida), el operador `->` separa el parámetro del resultado, y `TokenPA` es el constructor del token que se devuelve al análisis sintáctico.

Es importante resaltar el caso de las **palabras reservadas**, como `let*`, `letrec`, `!=`, `++`, entre otros. En el diseño del lexer, estas reglas deben escribirse *antes* que las reglas más generales o más cortas (por ejemplo, `let`, `<`, `!`, `+`).

Esto se debe a que el generador de analizadores léxicos Alex aplica la estrategia conocida como *longest match*, que selecciona la coincidencia más larga posible. En caso de empate entre dos patrones de igual longitud, prevalece la primera regla definida en el archivo.

Por lo tanto, si definiéramos la regla de `let` antes que `let*`, la cadena `let*` nunca coincidiría correctamente, ya que la primera regla (más corta) interceptaría el patrón. Este ordenamiento de las reglas garantiza un análisis léxico preciso y evita ambigüedades en el reconocimiento de tokens.

```
1  \(                          { \_ -> TokenPA }
2  \)                          { \_ -> TokenPC }
3  \[                          { \_ -> TokenLI }
4  \]                          { \_ -> TokenLD }
5  \,                          { \_ -> TokenComma }
6  \+                          { \_ -> TokenAdd }
7  \-                          { \_ -> TokenSub }
8  \*                          { \_ -> TokenMul }
9  \/                          { \_ -> TokenDiv }
10 \=                          { \_ -> TokenEq }
11 \<                          { \_ -> TokenLt }
12 \>                          { \_ -> TokenGt }
13 "++"                        { \_ -> TokenAdd1 }
14 "--"                        { \_ -> TokenSub1 }
15 "sqrt"                      { \_ -> TokenSqrt }
16 "**"                        { \_ -> TokenExpt }
17 "!="                        { \_ -> TokenNeq }
18 "<="                        { \_ -> TokenLeq }
19 ">="                        { \_ -> TokenGeq }
20 "not"                       { \_ -> TokenNot }
21 "if0"                       { \_ -> TokenIf0 }
22 "if"                        { \_ -> TokenIf }
23 "first"                     { \_ -> TokenFst }
24 "second"                    { \_ -> TokenSnd }
25 "letrec"                    { \_ -> TokenLetRec }
26 "let*"                      { \_ -> TokenLetStar }
27 "let"                       { \_ -> TokenLet }
28 "lambda"                    { \_ -> TokenLambda }
29 "head"                      { \_ -> TokenHead }
30 "tail"                      { \_ -> TokenTail }
31 "cond"                      { \_ -> TokenCond }
32 "else"                      { \_ -> TokenElse }
33 "#t"                        { \_ -> TokenBool True }
34 "#f"                        { \_ -> TokenBool False }
35 $digit+                     { \s -> TokenNum (read s) }
36 $alpha ($alnum)*            { \s -> TokenVar s }
```

Código 2.3: Lexer con Alex.

Nótese que tenemos las reglas para booleanos y literales con #t y #f para `TokenBool`, $digit+ para uno o más digitos a partir de la cadena *s* y las variables con $alpha ($alnum*). Son los elementos fundamentales que representan los valores básicos y nombres en nuestro lenguaje, son las expresiones que contienen datos específicos en el programa usando el lenguaje anfitrión para guardar estos datos.

Por último definimos un *catch-all* para diagnosticar caracteres inesperados. Es una depuración útil, si el usuario introduce un carácter inválido, el `lexer` falla con un mensaje claro y el código Unicode del carácter. Además definimos la función normalizeSpaces para que los espacios en Unicode los consuma $white+.[1]

```
  -- Catch-all para diagnosticar caracteres inesperados
  .                       { \s -> error ("Lexical error: caracter no
     reconocido = "
                                        ++ show s
                                        ++ " | codepoints = "
                                        ++ show (map fromEnum s)) }

  {
    -- Normaliza cualquier espacios en blanco Unicode a ' ' para que
       $white+ lo consuma
    normalizeSpaces :: String -> String
    normalizeSpaces = map (\c -> if isSpace c then '\x20' else c)

    lexer :: String -> [Token]
    lexer = alexScanTokens . normalizeSpaces
  }
```

Código 2.4: Lexer con Alex.

Finalmente, definimos la firma del `lexer` como `lexer :: String ->[Token]`, cumpliendo así con la función esencial del *análisis léxico*: recibir una cadena de entrada (el código fuente escrito por el usuario en nuestro lenguaje MiniLisp) y transformarla en una secuencia de `Tokens` reconocibles.

En el capítulo dedicado a los **Resultados** (7), se muestran distintos ejemplos de ejecución de esta módulo, donde mostramos la *tokenizacin* de expresiones dadas dentro del lenguaje MiniLisp .

[1]Para realizar el lexer tomamos como referencia lo visto en clase con el profesor y el material compartido en su GitHub, además de usar la documentación oficial de Alex[5] para desarrollar nuestro lexer.

2.2. Sintaxis Libre de Contexto

La ***sintaxis libre de contexto*** se refiere a la estructura de un lenguaje de programación en la que las reglas de formación de sus sentencias se pueden describir mediante una gramática libre de contexto. En ella especificamos cómo se pueden combinar las secuencias de *tokens* para formar expresiones y sentencias válidas para el lenguaje. Sin la gramática no podemos darle la estructura necesaria a para que, tanto el usuario como el interprete puedan hacer su trabajo.

En otras palabras, la ***sintaxis libre de contexto*** constituye el *esqueleto sintáctico* del lenguaje. Si el **análisis léxico** segmenta la entrada en *Tokens*, el **análisis sintáctico** (guiado por una *gramática libre de contexto*) se encarga de verificar que dichos *Tokens* se ensamblen de manera coherente conforme a las reglas del lenguaje. Sin una gramática bien definida, no sería posible darle forma ni estructura a los programas escritos en MiniLisp, ni mucho menos permitir que el intérprete los procese correctamente. Necesitamos de la gramática para dar orden, decidir qué aceptamos y cómo lo aceptamos, de este modo damos más formalidad y menos ambigüedad al lenguaje.

2.2.1. La gramática de MiniLisp

Formalmente, una ***gramática libre de contexto*** es una tupla:

$$G = (V, \Sigma, P, S)$$

donde:

- V es un conjunto finito de símbolos **no terminales** o variables las cuales representan conjuntos de cadenas que están siendo definidos recursivamente, es decir, cada variable genera un lenguaje.
- Σ es un conjunto finito de símbolos **terminales**. Son los símbolos básicos del lenguaje.
- P es el conjunto finito de **reglas de producción**, o reglas gramaticales, que contiene a las definiciones recursivas.
- S es el símbolo inicial, $S \in V$. Es una variable especial que genera a derivación de las cadenas del lenguaje deseado.

En nuestro proyecto, la gramática de MiniLisp está definida mediante la notación **BNF** (*Backus-Naur Form*), en particular la notación de **EBNF**.

Al rededor de los años 1950 y 1960, John Backus y Peter Naur desarrollaron esta notación (**BNF**) como una solución a la necesidad de definir de manera clara y precisa la sintaxis de los lenguajes de programación. Sin embargo, aunque **BNF** es efectiva, tiene ciertas limitaciones en términos de expresividad, especialmente para describir repeticiones y agrupaciones de una manera más compacta.

Con la notación **EBNF**:

- Las **variables** o no terminales se denotan entre los símbolos `<>`.
- Las **reglas de producción** se escriben con el operador `::=`.
- El símbolo `|` se utiliza para indicar **alternativas**, permitiendo expresar diferentes formas de una misma construcción sintáctica.
- La extensión de **EBNF** agrega el uso de `{ }` para indicar **repetición** de cero o más veces.

De esta manera, cada producción de la gramática define cómo los *tokens* generados por el analizador léxico (como `TokenAdd`, `TokenIf`, `TokenLet`, etc.) se combinan para formar expresiones válidas en el lenguaje. Recordemos que en la sección de **Sintaxis Léxica** se estableció la correspondencia entre patrones de texto y sus respectivos tokens; ahora, en esta etapa, esos mismos tokens se convierten en los símbolos terminales de nuestra gramática.

Las reglas sintácticas que definen la forma de las expresiones en esta versión de MiniLisp son las siguientes:

- Toda expresión está delimitada por paréntesis.
- Usamos la notación prefija, donde el operador precede a sus argumentos (operandos).
- Las operaciones aritméticas `+`, `-`, `*` y `/` son *n-arias* (*varidicas*), permitiendo una cantidad arbitraria de argumentos.
- Los predicados sobre enteros (igualdad y comparaciones) `=`, `<`, `>`, `>=`, `<=` y `!=` también admiten múltiples argumentos.
- Las asignaciones `let` y `let*` son igualmente son variádicas, es decir, permiten realizar asiganaciones locales con múltiples variables.
- Las listas se denotan mediante el uso de `[ ]` con la característica de que cada elemento (*expresin*) nuevo en la lista es separado del anterior con una coma `,` .
- Por último la expresión condicional `cond`, permite escribir múltiples condiciones de forma ordenada.

Cabe resaltar que en nuestra gramática se hace un abuso de notación: el uso de corchetes `[` y `]` no son para indicar opcionalidad de la notación en **EBNF**, sino que son los símbolos literales del lenguaje que usamos para representar listas, y en el caso de `cond` para delimitar las expresiones de una condición. En general, MiniLisp no define expresiones opcionales,

Con esto explicado, definimos la Gramática para MiniLisp en notación **EBNF** como sigue:

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
```

2.2.2. Análisis sintáctico

Una vez definida la **gramática libre de contexto** para MiniLisp , podemos pasar a la etapa de **análisis sintáctico**, también conocida como *parsing*.

Como bien mencionamos, mientras que el **análisis léxico** se encarga de transformar la cadena de entrada en una secuencia de *Tokens*, el **análisis sintáctico** tiene la tarea de verificar que dicha secuencia respete las reglas estructurales del lenguaje, tal como fueron establecidas por la gramática.

En otras palabras, el analizador sintáctico organiza los tokens generados por el `lexer` conforme a las producciones de la gramática, construyendo una representación estructurada del programa. Esta representación se denomina **árbol de sintaxis abstracta** (*ASA* o *Abstract Syntax Tree*-**AST**), el cual captura la estructura lógica del programa, eliminando detalles superficiales como los paréntesis o separadores que solo sirven para dar forma a la sintaxis concreta.

Formalmente, definimos una función sintáctica `parser`:

$$\texttt{parser: [Token]} \to \texttt{ASA}$$

Toma una secuencia de tokens y produce un **árbol de sintaxis abstracta** (*ASA*) según la gramática. Si el programa no respeta las reglas de sintaxis, este árbol no puede ser construido, lo que implica un **error sintáctico**.

El análisis sintáctico representa entonces, una etapa intermedia y esencial dentro del proceso de interpretación: traduce la estructura lineal de los tokens en una forma jerárquica que puede ser fácilmente interpretada y evaluada por etapas posteriores de MiniLisp .

Con todo lo visto en este capítulo, podemos concebir la ***Sintaxis Concreta*** de nuestro lenguaje como la composición funcional entre el **analizador léxico** y el **analizador sintáctico**, donde ambos trabajan en conjunto para transformar una cadena de caracteres en una estructura interna coherente:

$$\texttt{(parser} \circ \texttt{lexer):}\ \Sigma^* \to \texttt{ASA}$$

donde Σ^* representa todas las cadenas posibles de símbolos del alfabeto del lenguaje, y **ASA** (*Árbol de Sintaxis Abstracta*) es la estructura resultante. Y con esto cubrimos las fases que onforman el puente entre la entrada textual del usuario y las representaciones internas que permiten la evaluación del lenguaje.

A partir de este punto, continuaremos con las definiciones formales que dan estructura interna a nuestro lenguaje MiniLisp , avanzando hacia la construcción del **Árbol de Sintaxis Abstracta**, la implementación del **analizador sintáctico** usando Happy.

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 para MiniLisp

6.2. Ambientes

6.3. Evaluación en Haskell

Capítulo 7

Resultados

Bien, una vez hemos visto toda la teoría de nuestro MiniLisp y además de que hemos mostrado el código que lo implementa en Haskell, veamos como funciona:

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] Link del profe sintaxis concreta.

[2] Otro libro.

[3] Docs Haskell.

[4] 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/

[5] Docs Happy

[6] Autor. .ªrtículo". Revista, Año.