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 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:** `+`, `-`, `*`, `/`, `Add1`(incremento), `Sub1`(decremento), 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:** (*e*1`,` *e*2), `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 el 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`, `!=`, `add1`, 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.

```
  \(                          { \_ -> TokenPA }
  \)                          { \_ -> TokenPC }
  \[                          { \_ -> TokenLI }
  \]                          { \_ -> TokenLD }
  \,                          { \_ -> TokenComma }
  \+                          { \_ -> TokenAdd }
  \-                          { \_ -> TokenSub }
  \*                          { \_ -> TokenMul }
  \/                          { \_ -> TokenDiv }
  \=                          { \_ -> TokenEq }
  \<                          { \_ -> TokenLt }
  \>                          { \_ -> TokenGt }
  "add1"                        { \_ -> TokenAdd1 }
  "sub1"                        { \_ -> TokenSub1 }
  "sqrt"                      { \_ -> TokenSqrt }
  "**"                        { \_ -> TokenExpt }
  "!="                        { \_ -> TokenNeq }
  "<="                        { \_ -> TokenLeq }
  ">="                        { \_ -> TokenGeq }
  "not"                       { \_ -> TokenNot }
  "if0"                       { \_ -> TokenIf0 }
  "if"                        { \_ -> TokenIf }
  "first"                     { \_ -> TokenFst }
  "second"                    { \_ -> TokenSnd }
  "letrec"                    { \_ -> TokenLetRec }
  "let*"                      { \_ -> TokenLetStar }
  "let"                       { \_ -> TokenLet }
  "lambda"                    { \_ -> TokenLambda }
  "head"                      { \_ -> TokenHead }
  "tail"                      { \_ -> TokenTail }
  "cond"                      { \_ -> TokenCond }
  "else"                      { \_ -> TokenElse }
  "#t"                        { \_ -> TokenBool True }
  "#f"                        { \_ -> TokenBool False }
  "-"?$digit+                 { \s -> TokenNum (read s) }
  $alpha ($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, mientras que con ¿$digit+ para uno o más digitos a partir de la cadena *s* incluyendo los números negativos con ¿ 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**, 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[?] 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

Según Hopcroft y Ullman es su libro ***Introduction to Automata Theory, Languages, and Computation*** [?], una ***gramática libre de contexto*** se define formalmente como una tupla:

$$G = (V, T, P, S)$$

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.
- T es un conjunto finito, disjunto de V **de simbolos terminales**.
- P es el conjunto finito de **reglas de producción**; cada producción es de la forma $A \to \alpha$, donde A es una variable [en V] y α es una cadena de símbolos en $(V \cup T)^*$
- S es una variable en V llamada el símbolo inicial.

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.

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>})
         | (add1 <Expr>)
         | (sub1 <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> {","<Expr>} ] “]”
         | (head <Expr>)
         | (tail <Expr>)
         | (cond "["<Expr> <Expr>"]" {"["<E> <E>"]"} "[” else <Expr>"]")

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

Como se puede apreciar, definimos en las reglas para la gramática, que los operadores aritméticos(suma, resta, multiplicación y división) que son variádicos, efectivamente lo sean. Decidimos forzar que cada uno de ellos reciba al menos dos expresiones, ya que el uso de las llaves en la notación **EBNF** indica que puede haber cero o más repeticiones. Por ello, cualquier invocación de un operador aritmético con menos de dos operandos no será aceptada por el lenguaje.

En contraste, los operadores de incremento y decremento se definieron para aceptar únicamente una expresión. Ya que así modelamos su comportamiento natural: ambos operan sobre un solo valor, aumentando o disminuyendo su contenido en una unidad. De forma similar, en el caso de la raíz cuadrada, solo se requiere una expresión, dado que su propósito es calcular la raíz cuadrada de un único número. Para el operador del exponente, decidimos mantener el mismo comportamiento, por lo que en nuestro lenguaje, este operador eleva al cuadrado el valor de la expresión proporcionada, así solo necesita de un argumento. El operador not, su regla también refleja ese uso unario, pues su función es negar el valor booleano de un único argumento.

De manera análoga a los operadores aritméticos, las operaciones de comparación (=, <, >, <=, >=, !=), al también ser definidos como variádicos, exigimos que al menos se especifiquen dos expresiones y damos la posibilidad de que haya más, ya que una sola no permitiría realizar una comparación válida.

En cuanto a las expresiones de asignación y alcance como `let`, `letrec` y `let*`, establecimos que debe haber al menos un par `(<Var> <Expr>)`, permitiendo además la inclusión de múltiples pares adicionales. Así reflejamos la posibilidad de definir una o más asociaciones dentro de un mismo bloque, manteniendo la flexibilidad solicitada para el proyecto.

Smilarmente con la aplicación de funciones y las funciones λ, donde especificamos que debe haber almenos , una variable para la función lambda y dos expresiones expresiones para la aplicación de función: donde la primera corresponde a la función a aplicar y la segunda a su primer argumento; seguidas opcionalmente de más variables para la función o más argumentos para la aplicación. Con esto aseguramos que la aplicación de funciones y las funciones lambda siempre sean válidas y tengan sentido semántico.

Por último, cabe resaltar que en nuestra gramática el uso de los corchetes `[` y `]` tiene dos propósitos: En **EBNF**, los corchetes se utilizan para denotar opcionalidad, sin embargo, en MiniLisp decidimos emplear comillas dobles alrededor de los corchetes literales (`"["` y `"]"`) para distinguirlos de los usados por la notación formal. De esta manera, los corchetes con comillas representan la sintaxis concreta del lenguaje (las listas y condicionales `cond`), mientras que los corchetes sin comillas siguen indicando opcionalidad en la notación formal. Así, la regla:

```
"["[ <Expr> { ","<Expr> } ] "]"
```

permite definir listas que pueden estar vacías o contener una o más expresiones separadas por comas, representando correctamente la flexibilidad del manejo de listas dentro del lenguaje.

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 jerárquica 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:

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

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, entramos en el tema de la construcción del **Árbol de Sintaxis Abstracta** y la implementación del **analizador sintáctico** (*parser*) usando Happy.

Capítulo 3

Sintaxis Abstracta

La *sintaxis abstracta* es la representación interna de la estructura del lenguaje, se enfoca en los componentes esenciales y en cómo se relacionan entre sí, ignorando los detalles concretos del código fuente escritos por el usuario (detalles necesarios para nosotros como programadores, pero generalmente irrelevantes para el intérprete).

Está enfocada en capturar la **lógica** y **jerarquía** del programa dejando de lado elementos puramente sintácticos como los paréntesis o el formato. Formalizar en ella nos permitirá desarrollar un intérprete más eficiente y robusto, ya que nos facilita la detección de errores, la optimización del código y de incorporar en un futuro nuevas funcionalidades sin mayor problema. Justo como en todo nuestro campo de trabajo, buscamos hacer que nuestro código sea expandible y para ello tenemos que definir una estructura sólida desde el comienzo.

En comparación con la sintaxis concreta, la sintaxis abstracta es más clara y simple, pues elimina los detalles sintácticos (como paréntesis), enfocándose en la estructura lógica de las operaciones. Esta simplificación reduce la complejidad del análisis y mejora la eficiencia de las herramientas que operan sobre el código, como analizadores, optimizadores e intérpretes.

Mientras la sintaxis concreta nos da una representación más cercana al lenguaje humano (legible y expresiva), la sintaxis abstracta nos brinda una representación más adecuada para el procesamiento automático. Ambas son complementarias: la primera facilita la escritura del código, y la segunda permite su interpretación y evaluación.

Cabe mencionar que existe un concepto intermedio denominado **azúcar sintáctica**, el cual se refiere a aquellas construcciones del lenguaje que hacen más legible el código sin agregar nueva funcionalidad. En términos prácticos, la relación entre la sintaxis concreta, la abstracta y la azúcar sintáctica puede entenderse como un proceso progresivo de simplificación: primero eliminamos los elementos puramente sintácticos (paréntesis, separadores, etc.), y posteriormente reducimos aún más la estructura, obteniendo así una versión mínima que el intérprete pueda evaluar directamente y nos facilite la implementación del mismo. Profundizaremos más adelante en este aspecto al tratar la reducción de expresiones y la eliminación del azúcar sintáctica.

3.1. Árboles de Sintaxis Abstracta

A menudo, la sintaxis abstracta suele representarse como un ***Árbol de Sintaxis Abstracta***. Esta es una representación jerárquica modela la estructura lógica del programa: cada nodo del árbol corresponde a una construcción del lenguaje, y las hojas representan valores o identificadores.

A diferencia de la sintaxis concreta, en un **ASA** los paréntesis, comas y demás símbolos delimitadores no se representan explícitamente, pues su propósito es estructural, no semántico. Lo que sí se conserva es la relación jerárquica entre las partes del programa: qué elementos dependen de otros y cómo se combinan.

Formalmente, un *Árbol de Sintaxis Abstracta* puede definirse como una tupla ordenada $A = (N, E, R)$ donde:

- N es un conjunto finito de **nodos**, que representan las construcciones del lenguaje mediante etiquetas y las hojas representan a sus respectivos valores.
- $E \subseteq N \times N$ es el conjunto de **aristas dirigidas** que conectan los nodos, representando las relaciones jerárquicas entre ellos.
- $R \in N$ es la **raíz** del árbol, correspondiente a la expresión o programa principal.

Cada subárbol dentro del **ASA** puede interpretarse como una subexpresión del programa, lo que permite recorrerlo de forma recursiva para su evaluación, análisis o transformación. De esta manera, el *Árbol de Sintaxis Abstracta* constituye el puente entre la entrada textual del usuario y la representación interna que manipula el intérprete de nuestro lenguaje MiniLisp.

3.2. ASA para MiniLisp

Con lo anterior establedico, necesitamos ahora definir formalmente las reglas que nos permitan especificar los **ASA**'s de MiniLisp .

Para formalizar esta descripción, definimos la relación:

$$X \text{ ASA}$$

que se lee como "*X es un Árbol de Sintaxis Abstracta*". A partir de esta relación, especificamos las reglas que determinan qué estructuras son consideradas válidas como **ASA** dentro del lenguaje. De forma intuitiva, cada expresión definida en la gramática tiene su respectiva etiqueta, en el **ASA** es el *nodo padre*, y cada sub-expresión asociada a esta nueva etiqueta serán sus *nodos hijo*.

Hacemos la pequeña pero importante aclaración de que, las expresiones para pares, listas, aplicación de funciones , etc., que no hayamos definido una palabra reservada o algún caracter que el usuario deba escribir para identificarlos como tales, tendrán su etiqueta. Pues aunque no tengas estas palabras clave, se disntinguen por su sintaxis definida en la gramática.

Con un extraordinario uso de la imaginación, tenemos las siguientes etiquetas para nuestras expresiones de MiniLisp y definimos la descripción de sus reglas:

3.2.1. Expresiones atómicas

- **Variables**

  $Var(s)$ es un **ASA** si s es una cadena válida en el lenguaje (en particular si s es de tipo `String` en el lenguaje anfitrión Haskell).

$$\frac{s \in \texttt{String}}{Var(s)\ \textbf{ASA}}$$

  Su **ASA**, dado que es expresión atómica es la siguiente:

$$\begin{array}{c} \text{Var} \\ | \\ s \end{array}$$

- **Números**

  $Num(n)$ es un **ASA** si $n \in \mathbb{Z}$ (n es de tipo `Int` en el lenguaje anfitrión Haskell)

$$\frac{n \in \mathbb{Z}}{Num(n)\ \textbf{ASA}}$$

  Su **ASA**, dado que es expresión atómica es la siguiente:

$$\begin{array}{c} \text{Num} \\ | \\ n \end{array}$$

- **Booleanos**

  $Boolean(b)$ es un **ASA** si b es `True` o `False` (b es de tipo `Bool` en Haskell).

$$\frac{b \in \{\texttt{True}, \texttt{False}\}}{Boolean(b)\ \textbf{ASA}}$$

  Su **ASA**, dado que es expresión atómica es la siguiente:

$$\begin{array}{c} \text{Boolean} \\ | \\ b \end{array}$$

Las reglas anteriores podemos considerarlas como los *nodos hoja* de MiniLisp ya que no tenemos que evaluar nada más.

3.2.2. Operadores aritméticos

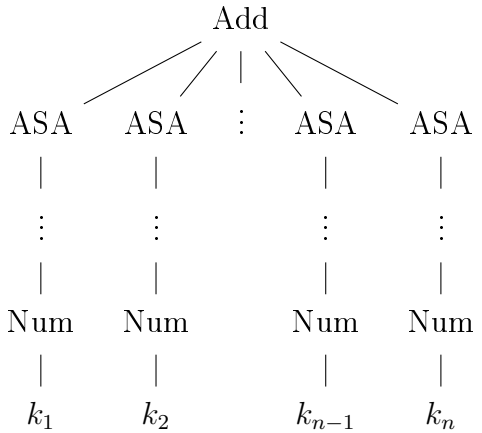
Como bien explicamos, dentro de nuestro lenguaje algunos de los operadores aritméticos se caracterizan por ser *variádicos*, por lo que su representación dentro del **ASA** se confroma de un árbol *n-ario*, mientras que el resto de los operadores aplican únicamente a un único argumento, es decir, son *unarios*. De este modo tenemos los dos casos:

- Variádicos

  Los **ASA** de estos operadores son prácticamente los mismos, donde el *nodo raíz*[1] corresponde al operador, y cada uno de sus hijos representa un "sub-árbol" asociado a las expresiones que son parte de la operación. Cada una de estas expresiones debe ser a su vez un **ASA** válido, y en última instancia debe corresponder a un **ASA** de tipo *Num*.

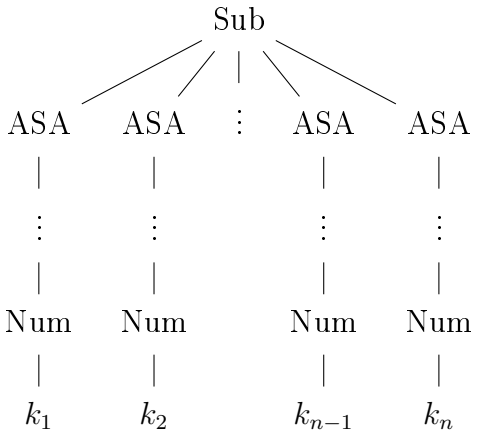
$Add(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**. Es un árbol *n-ario*.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Add(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



$Sub(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

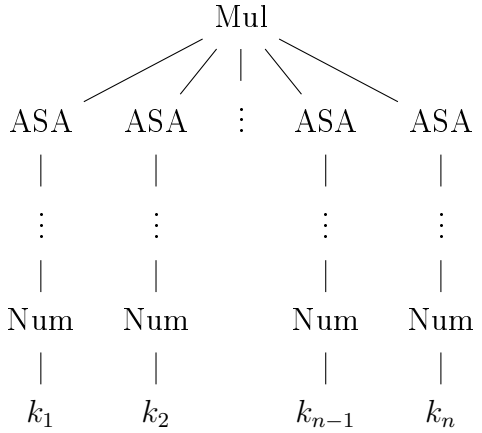
$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Sub(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



[1] Aunque no es precisamente un nodo raíz, lo tomamos como tal para indicar que es el nodo principal de donde parten sus expresiones.

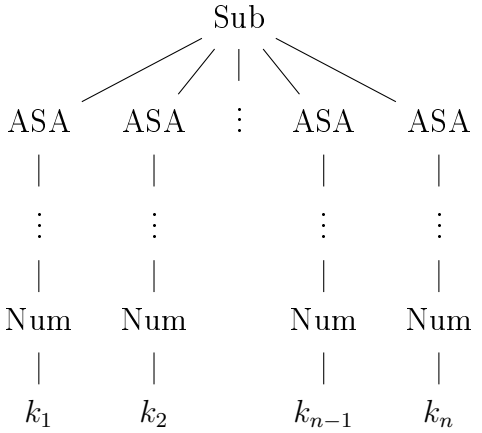
$Mul(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Mul(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



$Div(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Div(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$

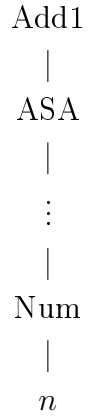


- No variádicos

  En su representación abstracta, estos operadores se modelan como árboles *unarios*, en los cuales el *nodo raíz* contiene la etiqueta del operador y posee exactamente un hijo, el cual representa la expresión sobre la que opera. Del mismo modo, la última instancia debe corresponder a un **ASA** de tipo *Num*.

Add1 (e) es un **ASA** si e es un **ASA**.

$$\frac{e \ \textbf{ASA}}{Add1(e) \ \textbf{ASA}}$$



Sub1 (e) es un **ASA** si e es un **ASA**.

$$\frac{e \ \textbf{ASA}}{Add1(e) \ \textbf{ASA}}$$

Sqrt(e) es un **ASA** si e es un **ASA**.

$$\frac{e \textbf{ ASA}}{\textit{Sqrt}(e) \textbf{ ASA}}$$

Sqrt
|
ASA
|
⋮
|
Num
|
n

Expt(e) es un **ASA** si e es un **ASA**.

$$\frac{e \textbf{ ASA}}{\textit{Expt}(e) \textbf{ ASA}}$$

Expt
|
ASA
|
⋮
|
Num
|
n

3.2.3. Predicados y comparaciones

Aquí también tenemos el caso donde la negación es unaria.

Not(p) es un **ASA** si p es un **ASA**. Debe concluir en un **ASA** de tipo *Boolean*.

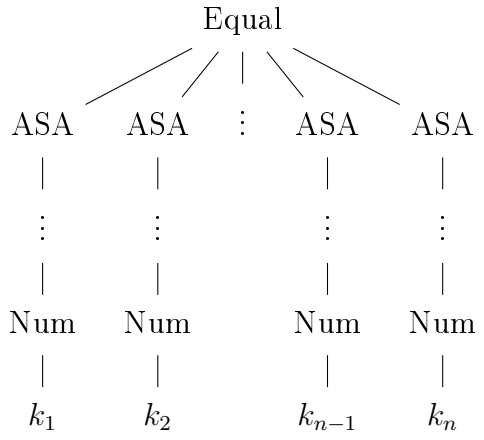
$$\frac{p \textbf{ ASA}}{\textit{Not}(p) \textbf{ ASA}}$$

Not
|
ASA
|
⋮
|
Boolean
|
n

Para el resto de expresiones, son **ASA** *n-arios* y cada uno debe terminar con **ASA** de tipo *Num*:

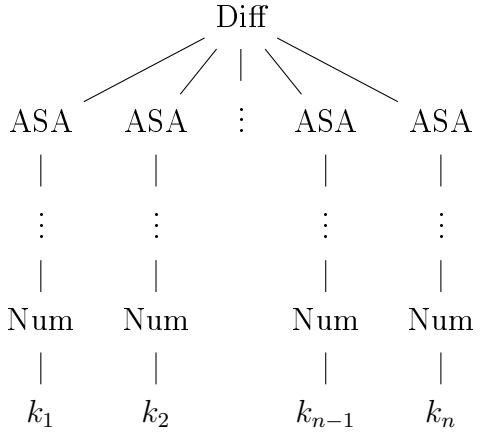
Equal$(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{\textit{Equal}(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



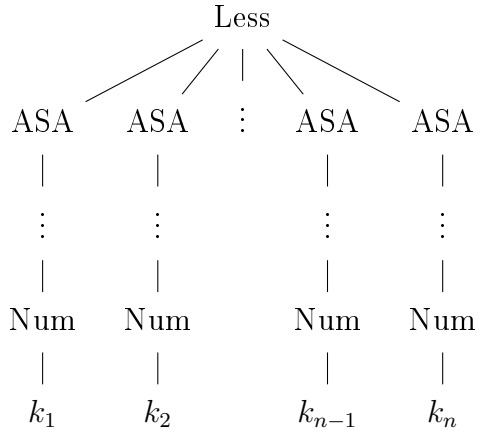
Diff$(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{\textit{Diff}(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



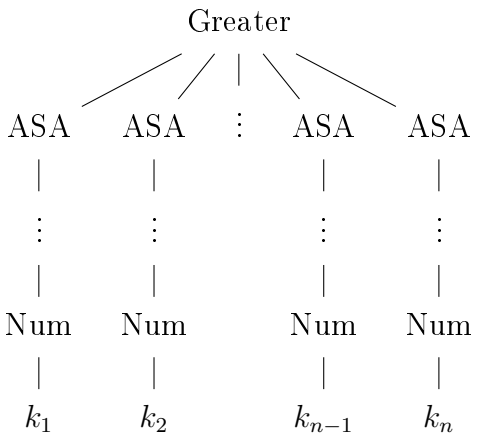
$Less(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Less(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



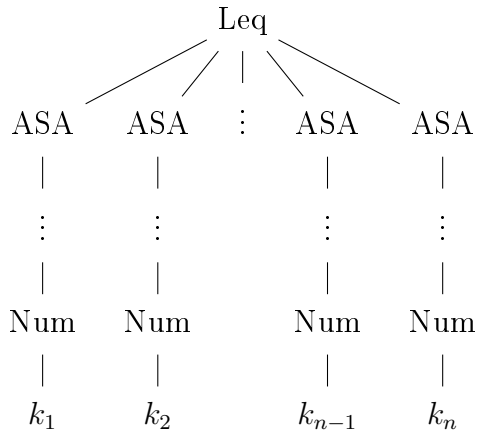
$Greater(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Greater(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



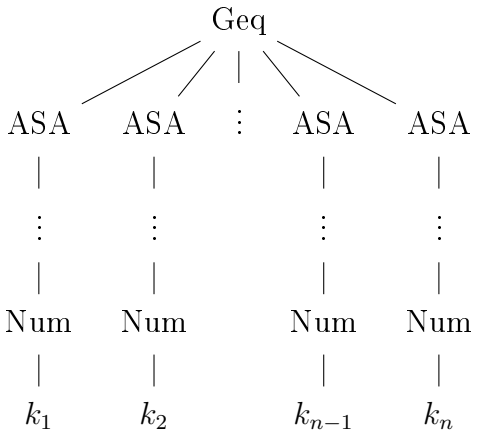
$Leq(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Leq(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



$Geq(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \ldots, e_n \ \textbf{ASA}}{Geq(e_1, e_2, \ldots, e_n) \ \textbf{ASA}}$$



3.2.4. Expresiones Let y Términos λ

Para estas reglas tenemos un caso especial, ya que no todas las expresiones en cada regla involucradas son variádicas. A diferencia de los operadores aritméticos, aquí las estructuras del **ASA** reflejan la manera en que se manejan los entornos y la aplicación de funciones dentro de nuestro lenguaje MiniLisp .

En primer lugar, las expresiones de tipo `let`, `let*` y `letrec` se utilizan para introducir nuevas asociaciones de variables dentro de un entorno local. Demanera informal, una expresión `let` consta de tres elementos básicos: **identificadores**, **valores** y un **cuerpo**.

Como se pudo ver en la **Sintaxis Concreta**, las tres expresiones `let` tienen el par de (**identificador valor**) y una tercera expresión que vendría siendo el **body**, con la característica de que `let` y `let*` tienen el par de asignación variádico.

Cada una de estas construcciones se representa en el **ASA** mediante una lista de pares (`Var`, `ASA`), donde cada par asocia un identificador con su correspondiente expresión. De este modo, el analizador sintáctico puede reconstruir de manera estructurada las relaciones entre las variables y sus valores dentro del entorno.

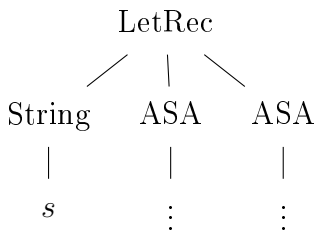
En particular:

- `let` define un nuevo entorno donde las variables se evalúan en paralelo (**alcance estático**).
- `let*` permite una evaluación secuencial, donde las definiciones anteriores pueden ser utilizadas en las siguientes (**alcance dinámico**).
- `letrec` introduce definiciones recursivas, es decir, variables que pueden hacer referencia a sí mismas dentro de sus expresiones. En este caso, el **ASA** no es variádico, ya que su estructura se limita a dos componentes bien definidos: la lista de asociaciones y el cuerpo de la expresión.

Estos identificadores deben ser **ASA** de tipo `String`.
Por lo que una vez explicado lo anterior, tenemos las siguientes reglas:

LetRec(i, v, b) es un **ASA** si cada identifcador i es de tipo `String`, el valor v y el cuerpo b son todos **ASA**.

$$\frac{i\text{: } \texttt{String} \;\; v, b \;\; \textbf{ASA}}{\textit{LetRec}(i, v, b) \;\; \textbf{ASA}}$$

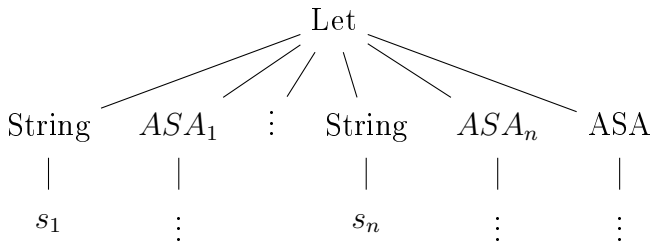


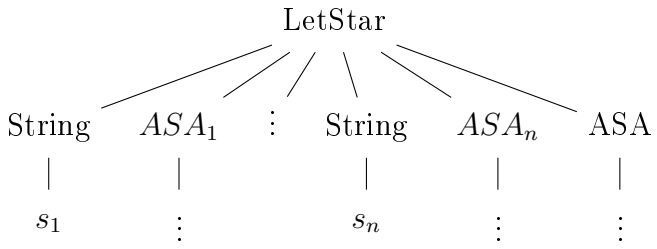
Let$((i_1, v_1), (i_2, v_2), \cdots, (i_n, v_n), b)$ es un **ASA** si cada identificador i_j es de tipo `String` y cada valor v_j y cuerpo b son todos **ASA**.

$$\frac{i_1, i_2 \cdots, i_n\texttt{: String } v_1, v_2, \cdots, v_n, b \textbf{ ASA}}{\textit{Let}((i_1, v_1), (i_2, v_2), \cdots, (i_n, v_n), b) \textbf{ ASA}}$$

LetStar$((i_1, v_1), (i_2, v_2), \cdots, (i_n, v_n), b)$ es un **ASA** si cada identificador i_j es de tipo `String` y cada valor v_j y cuerpo b son todos **ASA**.

$$\frac{i_1, i_2 \cdots, i_n\texttt{: String } v_1, v_2, \cdots, v_n, b \textbf{ ASA}}{\textit{LetStar}((i_1, v_1), (i_2, v_2), \cdots, (i_n, v_n), b) \textbf{ ASA}}$$





Por otra parte, tenemos los términos λ, para dicha implementación recordemos que los términos lambda se dividen en: **variables**, **abstracciones** λ y la aplicación de funciones.

Nuestras funciones `Lambda` se representan en el **ASA** como una lista de encabezados y una expresión que constituye el cuerpo de la función, que vendrían sinedo las **variables** y **abstracciones** λ.

Cada parámetro debe ser un identificador válido, por lo que en el **ASA** estos terminan **ASA** de tipo *String*. Con este diseño nos permitimos modelar funciones con múltiples argumentos de manera flexible, ya que el número de parámetros puede variar según la definición.

Finalmente, para la aplicación de funciones (`App`).

Dada la expresión de una aplicación de función e_0 con n expresiones (e_1, ..., e_2), se dice que e_0 es la posición de la función lambda y que cada e_i están en la posición de argumentos de la función. De esta forma, una aplicación representa el proceso de evaluar una función con sus respectivos n argumentos.

De este modo, la estructura del **ASA** distingue entre dos partes:

1. Como explicamos, la primer expresión representa la función que será aplicada (intuitivamente una expresión `Lambda` en nuestro lenguaje), esta no es variádica.

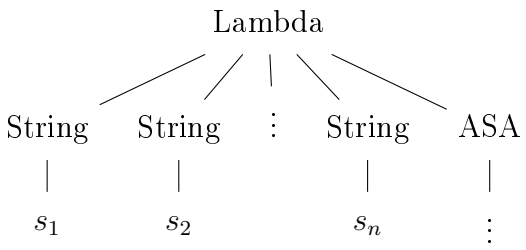
2. La segunda expresión corresponde a un lista de n argumentos sobre los cuales se aplicará la función anterior, y sí es variádica, ya que puede contener un número arbitrario (n) de expresiones.

De esta forma, en el **árbol de sintaxis abstracta** conservamos una representación fiel de la sintaxis para después implementar la semántica funcional de nuestro lenguaje.

Las cuales definimos como sigue:

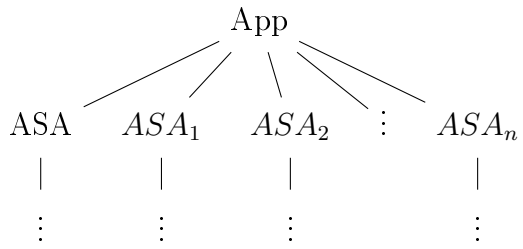
$Lambda(i_1, i_2, \cdots, i_n, c)$ es un **ASA** si cada identificador i_j es de tipo `String` que representa el nombre de su parámetro y b es **ASA** que representa el cuerpo.

$$\frac{i_1, i_2 \cdots, i_n\texttt{: String } b \textbf{ ASA}}{Lambda(i_1, i_2, \cdots, i_n, b) \textbf{ ASA}}$$



$App(f, e_1, e_2, \cdots, e_n)$ es un **ASA** si f que representa la función que se aplicará es un **ASA** y cada expresión e_i (los argumentos) son todos **ASA**.

$$\frac{f \textbf{ ASA } e_1, e_2, \cdots, e_n \textbf{ ASA}}{App(e_0, e_1, e_2, \cdots, e_n) \textbf{ ASA}}$$

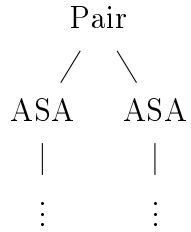


3.2.5. Pares ordenados y Proyecciones

Para los pares ordenados y sus proyecciones basta con tener en cuenta las siguientes reglas:

$Pair(f, s)$ es un **ASA** si las expresiones f y s son **ASA**.

$$\frac{f \textbf{ ASA } s \textbf{ ASA}}{Pair(f, s) \textbf{ ASA}}$$



$Fst(p)$ y $Snd(p)$ son **ASA** si la expresión p es **ASA**.

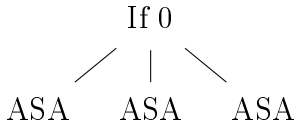
$$\frac{p \textbf{ ASA}}{Fst(p) \textbf{ ASA}} \quad \frac{p \textbf{ ASA}}{Snd(p) \textbf{ ASA}}$$

3.2.6. Condicionales

Tenemos dos condicionales `if0` e `if`, ambas son similares en cuanto sintaxis abstracta y por ende, en cuanto a su **ASA**:

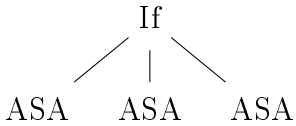
$If0(c, t, e)$ es un **ASA** si c, t y e son **ASA**.

$$\frac{c \textbf{ ASA} t \textbf{ ASA} e \textbf{ ASA}}{If0(c, t, e) \textbf{ ASA}}$$



$If(c, t, e)$ es un **ASA** si c, t y e son **ASA**.

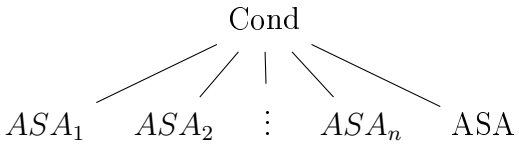
$$\frac{c \textbf{ ASA} t \textbf{ ASA} e \textbf{ ASA}}{If0(c, t, e) \textbf{ ASA}}$$



No obstante, la condicional `cond` al ser variádica, genera un **ASA** al menos *tri-ario*, pero puede variar en más ramas:

Cond$((c_1, t_1), (c_2, t_2), \ldots, (c_n, t_n), e)$ es un **ASA** si cada par de c_i, t_i son todos **ASA** y la expresión e también es **ASA**.

$$\frac{c_1, t_1, c_2, t_2, \ldots, c_n, t_n,\ e\ \textbf{ASA}}{\textit{Cond}((c_1, t_1), (c_2, t_2), \ldots, (c_n, t_n), e)\ \textbf{ASA}}$$

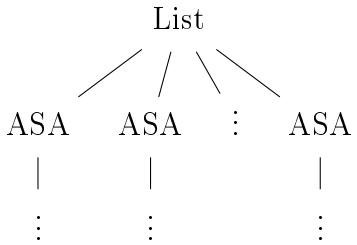


3.2.7. Listas

Por último pero no menos importante tenemos los siguientes **ASA** para las listas.

List$(e_1, e_2, \cdots, e_n)$ es un **ASA** si cada expresión e_i es un **ASA**. Es un árbol *n-ario*.

$$\frac{e_1, e_2, \ldots, e_n\ \textbf{ASA}}{\textit{List}(e_1, e_2, \ldots, e_n)\ \textbf{ASA}}$$



En cambio `head` y `tail`, solo requieren una expresión en nuestro lenguaje.

Head(l) es un **ASA** si l es un **ASA**, en particular una lista.

$$\frac{l\ \textbf{ASA}}{\textit{Head}(l)\ \textbf{ASA}}$$

Head
|
ASA
|
vdots

Tail(l) es un **ASA** si l es un **ASA**, en particular una lista.

$$\frac{l\ \textbf{ASA}}{\textit{Tail}(l)\ \textbf{ASA}}$$

Tail
|
ASA
|
vdots

3.3. ASA en Haskell

Las etiquetas que hemos definido para cada expresión, funcionarán como los constructores de nuestro tipo de dato en Haskell, ya que modelamos el tipo de dato **ASA** (*Árbol de Sintaxis Abstracta*) en Haskell mediante el tipo algebraico de datos, así es más sencillo expresar la variedad de formas que pueden adoptar las expresiones en MiniLisp.

Definimos el tipo de dato **ASA** en Haskell para MiniLisp en el archivos `ASA.hs` como sigue:

```haskell
data ASA
  = Var String
  | Num Int
  | Boolean Bool
  | Add [ASA]
  | Sub [ASA]
  | Mul [ASA]
  | Div [ASA]
  | Add1 ASA
  | Sub1 ASA
  | Sqrt ASA
  | Expt ASA
  | Not ASA
  | Equal [ASA]
  | Less [ASA]
  | Greater [ASA]
  | Diff [ASA]
  | Leq [ASA]
  | Geq [ASA]
  | Pair ASA ASA
  | Fst ASA
  | Snd ASA
  | Let [(String, ASA)] ASA
  | LetRec String ASA ASA
  | LetStar [(String, ASA)] ASA
  | If0 ASA ASA ASA
  | If ASA ASA ASA
  | Lambda [String] ASA
  | App ASA [ASA]
  | List [ASA]
  | Head ASA
  | Tail ASA
  | Cond [(ASA, ASA)] ASA
  deriving (Show, Eq)
```

Código 3.1: Tipo de dato ASA con azúcar

Con este diseño reflejamos directamente la estructura lógica del lenguaje que hemos definido en las reglas gramaticales y sus respectivas reglas para **ASA**.

- **Expresiones atómicas.** Los constructores `Var`, `Num` y `Boolean` representan las expresiones más simples del lenguaje. Cada una encapsula directamente un valor del tipo

correspondiente en Haskell: `String`, `Int`, `Bool`. Y como explicamos, constituyen las *hojas* del **ASA**, pues no se descomponen en subexpresiones.

- **Operadores aritméticos y Not.** Para los operadores *variádicos* (`Add`, `Sub`, `Mul`, `Div`, etc.), se implementó el uso de listas de expresiones `[ASA]`, de modo que cada operador pueda aplicarse a un número arbitrario de argumentos.

  De este modo, podemos representar eficientemente construcciones como:

  $$(+\ 1\ 2\ 3\ 4) \Rightarrow \texttt{Add [Num 1, Num 2, Num 3, Num 4]}$$

  En cambio, los operadores **unarios** (`Add1`, `Sub1`, `Sqrt`, `Expt`, `Not`) reciben únicamente un argumento `ASA`, justo como los hemos definido para el lenguaje.

- **Expresiones `Let`** Se representan como una lista de pares `[(String, ASA)]` para las expresiones `let` y `let*`, donde cada par vincula el nombre de la variable con la expresión que se le asigna. El segundo argumento `ASA` representa el cuerpo en el que dichas variables estarán disponibles.

  Por ejemplo:

$$\texttt{(let ((x 2) (y 3)) (+ x y))} \Rightarrow \texttt{Let [("x", Num 2), (“y”, Num 3)] (Add [Var "x", Var “y”])}$$

- **Expresiones condicionales.** Los constructores `If0` e `If` representan las estructuras condicionales del lenguaje. Cada una contiene tres subexpresiones: la condición, la rama verdadera y la rama falsa. Por su parte, el constructor `Cond` modela una estructura condicional más general, en la que se evalúan múltiples condiciones. Este se representa como una lista de pares `[(ASA, ASA)]`, donde cada par asocia una condición con su expresión correspondiente a ejecutar en caso de que se cumpla, permitiendo así una evaluación secuencial de casos.

  Tenemos por ejemplo:

  $$\texttt{(if 0 1 -1)} \Rightarrow \texttt{If0(Num 0)(Num 1)(Num -1)}$$

  $$\texttt{(cond [(>x 0) 1] [(<x 0) 2] [else 3])} \Rightarrow \texttt{Cond [(Greater [Var "x", Num 0], Num 1)(Less [}$$

- **Funciones y aplicación.** La abstracción lambda (`Lambda`) se define con una lista de encabezados `[String]` y un cuerpo `ASA`. La aplicación de funciones (`App`) se modela mediante dos expresiones formados por la expresión que representa la función y una lista de argumentos `[ASA]`, que serán evaluados y aplicados en orden.

  Por ejemplo:

  $$\texttt{(lambda(x y)(+ x y))} \Rightarrow \texttt{Lambda ["x",“y”](Add [Var "x", Var “y”])}$$

$$\texttt{((lambda(x y)(+ x y))2 3)} \Rightarrow \texttt{App(Lambda [“x”, “y”](Add [Var “x”, Var “y”])[Num 2, Num 3])}$$

- **Listas y operaciones sobre listas.** Las listas se modelan naturalmente como `List [ASA]`, donde cada elemento de la lista es a su vez un **ASA**. Los constructores `Head` y `Tail` representan las operaciones de acceso al primer elemento y al resto de la lista, respectivamente, y poseen un único hijo que corresponde a la lista sobre la que se aplican.

  Un ejemplo sería:

  $$(\texttt{[1, 2, 3, 4]}) \Rightarrow \texttt{List [Num 1, Num 2, Num 3, Num 4]}$$

- **Pares y proyecciones.** Los constructores `Pair`, `Fst` y `Snd` permiten representar estructuras de pares ordenados, así como las operaciones para obtener su primer y segundo elemento. Estos casos son binarios y unarios, respectivamente, de acuerdo con su aridad.

  Por ejemplo:

  $$(\texttt{(+ 3 2),(- 2 1)}) \Rightarrow \texttt{Pair(Add(Num 3, Num 2)Sub(Num 2, Num 1))}$$

Produndizaremos más en esto y veremos las pruebas de que nuestra implementación es correcta en la siguiente sección.

3.4. Parser con Happy

Happy es es un sistema generador de **analizadores sintácticos** (*parsers*) para Haskell. Su función es transformar una especificación de una **gramática libre de contexto**, escrita en una notación similar a **BNF**, en un módulo de Haskell que implementa automáticamente el parser correspondiente.

En nuestro caso, Happy toma como entrada el conjunto de *Tokens* producidos por nuestro `Lexer` en Alex, y construye a partir de ellos una estructura de ASA (definida en `ASA.hs`), que representa el programa en MiniLisp.

La idea central al usar Happy es definir formalmente la gramática de MiniLisp y dejar que Happy genere el código que realice el proceso de parseo. Esto nos permite integrar el módulo generado (`Grammar.hs`) al resto del proyecto, de modo que podamos transformar cualquier secuencia de *Tokens* válida en una estructura **ASA** con azúcar. De este modo podemos desazucarala sin problemas y una vez desazucarada, tener la estructura semántica lista para ser evaluada por nuestro intérprete.

En Happy, cada producción de la gramática se asocia con una acción semántica en Haskell, que construye el nodo correspondiente en el **ASA**. De este modo, Happy no solo valida la estructura del programa, sino que también construye automáticamente la representación estructural.

Mostramos a continuación nuestra implementación del **analizador léxico** con Happy.

```
{
module Grammar where

import Lexer
import Token
import ASA
}

%name parse
%tokentype { Token }
%error { parseError }

...

{
parseError :: [Token] -> a
parseError _ = error "Parser Error"
}
```

Código 3.2: Parser de Gramática con Happy.

No hay mucho que enfatizar en el comienzo y definición de errores del parser, solo es sintaxis de Happy:

- En el encabezado Haskell, definimos el módulo que Happy generará y los imports necesarios.
  - `Lexer` provee el flujo de *Tokens*, la entrada al parser.
  - `Token` define los tipos de *Tokens* reconocidos por el analizador léxico.
  - `ASA` contiene las definiciones de los constructores de los **ASA**.
- `%name parse`: indica el nombre de la función principal que Happy generará. Esta es de la forma:

  $$\texttt{parse :: [Token]} \rightarrow \texttt{ASA}$$

  y será el punto de entrada del parser.
- `%tokentype { Token }`: especifica el tipo de dato que Happy debe esperar como entrada (nuestros *Tokens*).
- Y con %error { parseError } definimos la función que se ejecutará en caso de error sintáctico.

  En este caso, parseError simplemente lanza una excepción con el mensaje "*Parser Error*", indicando que la cadena no cumple la gramática definida. esta función la definimos al final del archivo `Grammar.y`.

Una vez inicializamos el parser en Happy, continuamos con la declaración de *Tokens*:

```
%token
  var             { TokenVar $$ }
  num             { TokenNum $$ }
  boolean         { TokenBool $$ }
  '('             { TokenPA }
  ')'             { TokenPC }
  '['             { TokenLI }
  ']'             { TokenLD }
  ','             { TokenComma }
  '+'             { TokenAdd }
  '-'             { TokenSub }
  '*'             { TokenMul }
  '/'             { TokenDiv }
  '='             { TokenEq }
  '<'             { TokenLt }
  '>'             { TokenGt }
  "!="            { TokenNeq }
  "<="            { TokenLeq }
  ">="            { TokenGeq }
  "++"            { TokenAdd1 }
  "--"            { TokenSub1 }
  "**"            { TokenExpt }
  "sqrt"          { TokenSqrt }
  "not"           { TokenNot }
  "if0"           { TokenIf0 }
  "if"            { TokenIf }
  "first"         { TokenFst }
  "second"        { TokenSnd }
  "let"           { TokenLet }
  "letrec"        { TokenLetRec }
  "let*"          { TokenLetStar }
  "head"          { TokenHead }
  "tail"          { TokenTail }
  "lambda"        { TokenLambda }
  "cond"          { TokenCond }
  "else"          { TokenElse }
```

Código 3.3: Declaración de Tokens en Happy.

En esta sección declaramos los *Tokens* terminales que Happy reconocerá como símbolos del lenguaje. Son los elementos básicos que el parser reconocerá. Cada entrada de `%token` indica cómo un token léxico (producido por el **analizador léxico** `Lexer`) se relaciona con un nombre dentro de la gramática.

Para nuestros valores, por ejemplo:`var { TokenVar $$ }`, indicamos que cuando el lexer produzca un `TokenVar "x"`, el parser reconocerá `var` y podrá acceder al valor x a través de $$.

Los símbolos reservados o caracteres especiales del lenguaje los denotamos comillas simples ' y las palabras reservadas con comillas dobles , asociando cada una con su respectivo `Token`.

Esta correspondencia permite que Happy comprenda la estructura léxica y la relacione con la estructura sintáctica del lenguaje MiniLisp.

Continuando, tenemos las reglas de producción del lenguaje, sección delimitada por %% donde definimos cómo se construye el **ASA** a partir de los *Tokens*.

Intuitivamente, esta sección es prácticamente igual a nuestra definición de la gramática para MiniLisp, por lo que en Happy cada regla tiene la forma:

NoTerminal : simbolos_de_produccion { accion_semantica }

Donde:

- **NoTerminal** es una categoría gramatical, como `ASA`.
- **simbolos_de_produccion** son tokens o no terminales.
- { **accion_semantica** } es código Haskell que construye el nodo del **ASA** correspondiente.

Ya con esto podemos definir las reglas de la gramática y producción del **ASA**.

```
%%

ASA
  : var                                          { Var $1 }
  | num                                          { Num $1 }
  | boolean                                      { Boolean $1 }
  | '(' '+' opArgs ')'                           { Add (reverse $3) }
  | '(' '-' opArgs ')'                           { Sub (reverse $3) }
  | '(' '*' opArgs ')'                           { Mul (reverse $3) }
  | '(' '/' opArgs ')'                           { Div (reverse $3) }
  | '(' '=' opArgs ')'                           { Equal (reverse $3) }
  | '(' '<' opArgs ')'                           { Less (reverse $3) }
  | '(' '>' opArgs ')'                           { Greater (reverse $3) }
  | '(' "!=" opArgs ')'                          { Diff (reverse $3) }
  | '(' "<=" opArgs ')'                          { Leq (reverse $3) }
  | '(' ">=" opArgs ')'                          { Geq (reverse $3) }
  | '(' "++" ASA ')'                             { Add1 $3 }
  | '(' "--" ASA ')'                             { Sub1 $3 }
  | '(' "sqrt" ASA ')'                           { Sqrt $3 }
  | '(' "**" ASA ')'                             { Expt $3 }
  | '(' "not" ASA ')'                            { Not $3 }
  | '(' ASA ',' ASA ')'                          { Pair $2 $4 }
  | '(' "first" ASA ')'                          { Fst $3 }
  | '(' "second" ASA ')'                         { Snd $3 }
  | '(' "let" '(' ids ')' ASA ')'                { Let (reverse $4) $6 }
  | '(' "letrec" '(' var ASA ')' ASA ')'         { LetRec $4 $5 $7 }
  | '(' "let*" '(' ids ')' ASA ')'               { LetStar (reverse $4) $6 }
  | '(' "if0" ASA ASA ASA ')'                    { If0 $3 $4 $5 }
  | '(' "if" ASA ASA ASA ')'                     { If $3 $4 $5 }
  | '(' "lambda" '(' vars ')' ASA ')'            { Lambda (reverse $4) $6 }
  | '(' ASA appArgs ')'                          { App $2 (reverse $3) }
  | '(' '[' listArgs ']' ')'                     { List (reverse $3) }
  | '(' "head" ASA ')'                           { Head $3 }
  | '(' "tail" ASA ')'                           { Tail $3 }
  | '(' "cond" condis '[' "else" ASA ']' ')'     { Cond (reverse $3) $6 }
```

Código 3.4: Reglas principales de la gramática con Happy.

El uso de `reverse` es importante pues, durante el análisis, Happy construye las listas en orden inverso por eficiencia (debido a la *recursión por izquierda*). Aplicar reverse al final restaura el orden original de los argumentos según fueron escritos por el usuario.

Nótese además que, para algunas producciones definimos nuevas reglas, estas reglas las definimos para llevar un mejor control de su **análisis sintáctico**. Las podemos ver como sigue:

```
opArgs
  : ASA ASA                                { [$2, $1] }
  | opArgs ASA                             { $2 : $1 }

ids
  : id                                     { [$1] }
  | ids id                                 { $2 : $1 }

id
  : '(' var ASA ')'                        { ($2, $3) }

vars
  : var                                    { [$1] }
  | vars var                               { $2 : $1 }

appArgs
  : ASA                                    { [$1] }
  | appArgs ASA                            { $2 : $1 }

listArgs
  : {- empty -}                            { [] }
  | ASA                                    { [$1] }
  | listArgs ',' ASA                       { $3 : $1 }

condis
  : condy                                  { [$1] }
  | condis condy                           { $2 : $1 }

condy
  : '[' ASA ASA ']'                        { ($2, $3) }
```

Código 3.5: Reglas auxiliares para la gramática.

Estas reglas complementarias definen la estructura interna de las construcciones del lenguaje:

- `opArgs`: permite operadores aritméticos con un número variable de argumentos. La forma `[$2, $1]` y `($2 : $1)` implementa recursión por izquierda. Gracias a ello, el parser consume la entrada de forma eficiente, usando espacio constante en la pila. Luego, `reverse` corrige el orden de evaluación. Con esta regla nos aseguramos que los operadores aritméticos dados por el usuario tengan al menos dos expresiones `ASA ASA` para ser válidos.

- `ids` e `id`: definimos los pares `(var ASA)` de las expresiones `let`, `id` verifica que sea un par correcto y `ids` acumula los pares variádicos.
- `vars`: define las listas de variables, lo usamos para la lista de encabezados para la expresión `lambda`.
- `appArgs`: generamos una lista de expresiones, los argumentos de la aplicación de función.
- `listArgs` manejamos los elementos de la lista, permitiendo la lista vacia y por otro lado el manejo de la lista con sus elementos separados por comas.
- `condis` y `condy`: podemos definir la estructura variádica para `cond`, además de establecer que las expresiones están delimitadas por corchetes.

De este modo hemos definimos correctamente las reglas para el parser y obtenemos correctamente los **ASA** de cada expresión.

La razón por la cual usamos *recursión izquierda* en Happy es porque así podemos definir las reglas de producción de forma más eficiente, ya que construimos el resultado del parser en una sola pasada utilizando espacio constante en la pila. Mientras que la *recursión por derecha* podría incrementar la complejidad de tiempo y memoria de manera significativa.

Por ello utilizamos *recursión por izquierda* de la forma `{ [$2, $1]}` y `{$2 : $1}`, así construimos la lista de expresiones variádicas hacia la izquierda y, apoyándonos de `reverse`, hacemos que esta lista de expresiones vuelva al orden de como el usuario escribió las reglas.

Y así, como hemos visto en el caítulo, el estudio y la construcción de la **sintaxis abstracta** representan uno de los pasos más importantes en el diseño de un lenguaje de programación (como es el caso con nuestra implementación de MiniLisp). A través del *Árbol de Sintaxis Abstracta (ASA)*, logramos capturar la estructura esencial de los programas sin los elementos superficiales de la **sintaxis concreta**, lo que nos permite trabajar directamente con la lógica y la jerarquía de las expresiones. Esta representación no solo facilita el análisis y la transformación del código, sino que también hace posible implementar de manera más limpia y coherente cada parte del lenguaje.

Sin embargo, nuestro MiniLisp aún no está listo para pasar directamente a la etapa del intérpretE. Aunque el **ASA** ya elimina mucho del ruido"sintáctico, todavía contiene construcciones que, si las evaluáramos directamente, requerirían una gran cantidad de reglas semánticas adicionales. Por eso, antes de interpretar, necesitamos un paso intermedio: el proceso de **desazucarización**, distinguir la **azúcar sintáctica** y eliminarla.

Lo que buscamos es simplificar aún más nuestra **sintaxis abstracta**, transformando las construcciones más complejas o redundantes en formas más básicas que el intérprete pueda manejar de manera uniforme.

Capítulo 4

Azúcar Sintáctica

Una vez que hemos "*limpiado*" la Sintaxis Concreta y Léxica nos queda la Sintaxis Abstracta y como vimos, con ella logramos capturar la estructura esencial del programa. Sin embargo, podemos reducir aún más estos **ASA** porque a pesar de ya haber elimida los elementos superficiales de la Sintaxis Concreta, hay expresiones redundantes o que pueden representarse como otras, lo que nos reduce en mayor medida las reglas debemos implementar al momento de la evaluación. Por ejemplo:

$$\texttt{(if0 (+ 8 -8) 0 -1)} \Rightarrow \texttt{(if (= (+ 8 -8) 0) 0 -1)}$$

`if0` es análogo a if con la comprobación de que el resultado sea igual a cero, pero con `if0` el usuario se ahorra hacer cada vez esa comprobación de igualdad, pero esto sigue siendo azúcar para nuestro intérprete.

Estas expresiones **ASA** sin azúcar sintáctica pertenecen al conjunto que denominamos como *núcleo*, o *core*.

Desarrollar más la introducción...

4.1. Sintaxis Abstracta sin azúcar en MiniLisp

Necesitamos definir una función que realice el prodecimientod e desazucarar los **ASA** en núcleos. Por lo que nombramos a esta función como `desugar` y queda definida como sigue:

$$\texttt{desugar ::=} \Rightarrow \texttt{Sugared\_ASA} \rightarrow \texttt{Desugared\_ASA}$$

- **Variables:** Las variables no necesitan desazucarizarse pues ya son expresiones atómicas, ya pertenecen al núcleo, simplemente renombramos a ASA sin azúcar preservando el valor.

$$\begin{aligned} \texttt{desugar(SugarVar i)} &\Rightarrow \texttt{Var i} \\ \texttt{desugar(SugarNum n)} &\Rightarrow \texttt{Num n} \\ \texttt{desugar(SugarBool b)} &\Rightarrow \texttt{Bool b} \end{aligned}$$

- **Operadores:** Nuestros operadores en su gran mayoría son variádicos, preservan la lista de ASA (los operandos), esto es azúcar sintáctica para el intérprete ya que nuestros operadores `+`, `-`, `*` y `/` son binarios. Por lo que podemos representar a los operadores variádicos como un encademiento del mismo. Por ejemplo:

$$\texttt{desugar(SugarAdd[} n_1, n_2, \ldots, n_k\texttt{]}) \Rightarrow \texttt{Add } n_1 \ (\texttt{Add } n_2 \ldots (\texttt{Add } n_{k-1}\ n_k))$$

  Para el caso de los operadores unarios, estos ya de por sí son azúcar sintáctica- a excepción de Sqrt el cuál es un operador único por lo que ya es *núcleo* -. En el caso de `Add1` n y `Sub1` n podemos reexpreasarlos como $n+1$ y $n-1$ respectivamente,y de manera similar con `Expt`, dado que en nuestro lenguaje representa elevar al cuadrado el número n, entonces `Expt` es azúcar sitáctica y lo reexpresamos como una multiplicación se $n \times n$:

$$\begin{gathered}\texttt{desugar(Add1 } n\texttt{)} \Rightarrow \texttt{Add } n\ 1 \\ \texttt{desugar(SugarSqrt } n\texttt{)} \Rightarrow \texttt{Sqrt } n \\ \texttt{desugar(Expt } n\texttt{)} \Rightarrow \texttt{Mul } n\ n\end{gathered}$$

- **Comparadores:** En el caso de los comparadores, es muy similar su representación en **ASA** sin azúcar como lo fue para los operadores variádicos, ya que estos también lo son. El núcleo de cada comparadaor es el mismo pues forzosamente tenemos que definir uno para cada uno de ellos, ya que necesitamos preservar el tipo de comparación. Sin embargo, a diferencia de los operadores, no es conveniente representarlos como un encademiento de comparadores, pues al evaluar la comparación entre dos números, el resultado es de tipo `Bool`Sino como un encademiento de condicionales if. Donde la condición inicial es la comparación de los dos primeros argumentos y el consecuente son las comparaciones de la argumentos restantes, si alguna de las condiciones no se cumple entonces caemos en el else False y de otro modo las comparaciones son válidas y el resultado es `True`:

$$\texttt{desugar(SugarEqual[} n_1, n_2, \ldots, n_k\texttt{]}) \Rightarrow \texttt{If (Equal } n_1\ n_2\texttt{) (Equal } n_2\ n_3\texttt{) } \ldots \texttt{ (Equal } n_{k-1}\ n_k\texttt{) (Bool False)}$$

- **Not y Pares:** `Not` y las **ASA** sobre pares `Pair`, `Fst` y `Snd` ya son núcleos, no hace falta definir una desazucarización específica.

- **Condicionales:** `If0` y `Cond` solo son azúcar sintáctica de `If`. `If0` como mencionamos es comprobar que el resultado al terminar de evaluarse sea igual a cero. Mientras que `Cond` es igualmente un encadenamiento de `If`. Por ello estas tres expresiones las representamos como un único núcleo If:

$$\begin{gathered}\texttt{desugar(If0 c t e)} \Rightarrow \texttt{If (Equal c 0) t e} \\ \texttt{desugar(Cond [}x_1\ e_1\texttt{] [}x_2\ e_2\texttt{] } \ldots \texttt{ [}x_n\ e_n\texttt{] [else } e_k\texttt{])} \Rightarrow \texttt{If } (x_1)\ e_1\ (\text{If } (x_2)\ n_2) \\ \ldots (\text{If } (x_n)\ e_n\ (e_k))\end{gathered}$$

- **Lets:** Los `Let` son solo la azúcar de la aplicación de funciones donde la sustitucion del valor con el identificador, el par (*id, value*) es el argumento, y el cuerpo del `let` es la función que se va a aplicar. Por otro lado, `LetStar` es azúcar sintácica de `Let`, de modo que `LetStar` se reexpresa como `lets` anidados.

  Por lo que el proceso de desazucarización sería similar a:

  $$\texttt{desugar}(\texttt{Let}(id, value)\ body) \Rightarrow \texttt{App}\ (body)\ (id, value)$$
  $$\texttt{desugar}(\texttt{LetStar}(id_1, value_1)\ (id_2, value_2)\ \ldots\ (id_n, value_n)\ body) \Rightarrow \texttt{Let}\ ((id_1, value_1)\ (\texttt{Let}\ (id_2, value_2)\ \ldots\ (\texttt{Let}(id_n, value_n)\ body)))$$

  Falta la definición de LetRec como azúcar...

- **Expresiones lambda:** Nuestras expresiones lambda son variádicas, por lo que para representarlas en núcleo necesitamos currificarlas., es decir, necesitamos convertiralas en funciones de un solo argumento:

  $$\texttt{desugar}(\texttt{Lambda [}x_1\texttt{, }x_2\texttt{, }\ldots\ x_n\texttt{]}\ b) \Rightarrow (\texttt{Fun}\ x_1\ (\texttt{Fun}\ x_2\ \ldots\ (\texttt{Fun}\ x_n\ b)))$$

  De igual forma para la Aplicación de funciones, ya que `App` en **ASA** maneja una lista de argumentos, necesitamos currificar estos argumentos ya que las funciones ya están en forma de un argumento a la vez:

  $$\texttt{desugar}(\texttt{App}\ e\ \texttt{[}x_1\texttt{, }x_2\texttt{, }\ldots\ x_n\texttt{]}) \Rightarrow (\texttt{App}\ (\texttt{App}\ (\texttt{App}\ \ldots\ (\texttt{App}\ e\ x_1)\ x_2)\ \ldots)\ x_n))$$

- **Listas:** Para las listas definimos su desazucaricación con el uso de `Nil` y `Cons`, el cuál funciona de manera similar al encadenamiento de pares:

  $$\texttt{desugar}(\texttt{List [}x_1\texttt{, }x_2\texttt{, }\ldots\ x_n\texttt{]}) \Rightarrow (\texttt{Cons}\ x_1\ (\texttt{Cons}\ x_2\ \ldots\ (\texttt{Con}\ x_{n-1}\ x_n))$$

  `Head`, `Tail` ya no trabajan sobre listas **ASA** sino con `Nil` y `Cons`, por lo que estos seran nuestro núcleos para manejar las listas.

  Creo estaría bien mencionar que algunos ya no son árboles n-arios sino binarios...

4.1.1. Desugar en Haskell

Para nuestro proyecto en MiniLisp definimos el siguiente tipo de dato:

```haskell
module AST where

-- ASA sin azucar (AST)
data AST
  = VarC String
  | NumC Int
  | BoolC Bool
  | AddC AST AST
```

```
9   | SubC AST AST
10  | MulC AST AST
11  | DivC AST AST
12  | SqrtC AST
13  | NotC AST
14  | EqualC AST AST
15  | LessC AST AST
16  | GreaterC AST AST
17  | DiffC AST AST
18  | LeqC AST AST
19  | GeqC AST AST
20  | PairC AST AST
21  | FstC AST
22  | SndC AST
23  | IfC AST AST AST
24  | FunC String AST
25  | AppC AST AST
26  | ConS AST AST
27  | HeadC AST
28  | TailC AST
29  | NiL
30  deriving (Show, Eq)
```

Código 4.1: Tipo de dato ASA sin azúcar, AST

AST (*Abstract Sintaxis Tree*) es nuestro tipo de dato **ASA** sin azúcar, no hay un razón especial por la que la hayamos nombrado **AST**, nos pareció práctico y nada más. En esta seccón nos referiremos como **AST** a nuestra sintaxis abstracta sin azúcar en MiniLisp -0.2cm. Notemos que, los tipos de dato que trabajaban sobre listas

Anteriormente hicimos una breve mención, casi de manera superficial, de cómo se reexpresan nuestro **ASA** a **AST** a través de un función especial conocida como `desugar`, de tal modo que nos quedamos con las estructuras núcleo y no ahorramos futuras reglas para el intérprete.

En nuestro proyecto de MiniLisp -0.2cm, definimos la función `desugar` en el arhcivo `Desugar.hs` como sigue:

```
1   module Desugar where
2
3   import ASA
4   import AST
5   import ASV
6
7   {-- Desazucaramos los ASA --}
8   desugar :: ASA -> AST
9   -- Casos base
10  desugar (Var x) = VarC x
11  desugar (Num n) = NumC n
12  desugar (Boolean b) = BoolC b
```

Código 4.2: Firma y casos base de la función desugar

La firma de la función refleja nuestro objetivo, dado una estructura `ASA`, `desugar` lo proecesa hasta obtener un `AST`. Nótese además que las expresiones atómicas no cambian su estructura, únicamente las renombramos de tipo `ASA` a `AST`.

```
desugar :: ASA -> AST
-- Operaciones aritmeticas
desugar (Add xs) = desugarOps AddC xs
desugar (Sub xs) = desugarOps SubC xs
desugar (Mul xs) = desugarOps MulC xs
desugar (Div xs) = desugarOps DivC xs
desugar (Add1 n) = AddC (desugar n) (NumC 1)
desugar (Sub1 n) = SubC (desugar n) (NumC 1)
desugar (Expt n) = MulC (desugar n) (desugar n)
desugar (Sqrt n) = SqrtC (desugar n)
```

Código 4.3: Sección de la función `desugar` para operadores aritméticos

Previamente, al hacer mención de la función `desugar` omitimos explicar que los argumentos de las expresiones deben pasar también por el proceso de desazucarización, sin embargo es necesario definir la desazucarización recursivamente ya que como sabemos, un **AST** es **AST** si todos sus hijos lo son.
Para ello definimos una función auxiliar `desugarOps` que generaliza el trabajo de desazucarar las operaciones aritméticas pues estas pasan por el mismo procedimiento solo que cambian la etiqueta de su estructura. Mientras que `Add1` y `Sub1` como mencionamos, son azúcar para n + 1 / n - 1 respectivamente, además de que `Expt` es azúcar de $n \times n$. Por otro lado, `Sqrt` solo pasa a ser `SqrtC` además que aplica `desugar` a su único hijo.

```
desugar :: ASA -> AST
-- Operaciones aritmeticas
desugar (Add xs) = desugarOps AddC xs
desugar (Sub xs) = desugarOps SubC xs
desugar (Mul xs) = desugarOps MulC xs
desugar (Div xs) = desugarOps DivC xs

--Funcion auxiliar para desazucarar los operadores
desugarOps :: (AST -> AST -> AST) -> [ASA] -> AST
desugarOps _ [] = error "[desugarOps Error]: Lista vacia (no deberia
    suceder)"
desugarOps _ [x] = desugar x
desugarOps op (x:xs) = op (desugar x) (desugarOps op xs)
```

Código 4.4: Función `desugarOps` como auxiliar para desazucarar operadores

La función `desugarOps` recibe una tupla de `AST` (`AST ->AST ->AST`) y una lista de `ASA` y devuelve un `AST` donde en la tupla, el primero es la etiqueta asociada al operador que vamos a desazucarar y los otros dos son los hijos del operador, que recordemos, en `AST` ya son árboles binarios. Y la lista de `ASA` es la lista de los operandos que vamos a separar.
De esta forma no perdemos la referencia de qué tipo de operador `AST` estamos desazucarando mientras mantemos una única función `desugarOps` y así no tenemos que definir una función para cada operador.
Tenemos dos casos base para la función, donde `[ASA]` es vacía, cosa que no debería suceder pues en la gramática definida en Happy justo lo implementamos de modo que los operadores

rechacen un número de argumentos inválidos; además de que tampoco se puede llegar a la lista vacía por el siguiente caso base donde si la lista tiene un elemento es donde termina la recursión y devolvemos ese elemento desazucarado con `desugar`. Por otro lado el paso recursivo es donde tomamos la cabeza de la lista el cual desazucaramos para ser el primer arguemento del operador, mientras que la cola recursivamente se aplica `desugarOps` y que será el segundo argumento.

Continuando con los comparadores, intuitivamente pensamos en implementarlo de igual forma que con los operadores (un encadenamiento de comparadores). Sin embargo, al momento de pensar en su interpretación, nos topamos con el problema de que, al hacer la comparación entre un `Num n` y `Num m`, el resultado es de tipo `Bool`, y esto nos da una inconsistencia de tipos al momento de continuar con las evaluaciones posteriores ya que no es imposible comparar un `Num` con un `Bool`.
Por ello cambiamos su implementación a encadenamiento de condicionales `If`, pues es la única forma en nuestro lenguaje de preservar las comparaciones correctas y detectar en donde no se cumple la comparación.

```
1  desugar :: ASA -> AST
2  -- Not
3  desugar (Not x) = NotC (desugar x)
4  -- Comparaciones
5  desugar (Equal xs) = desugarComp EqualC xs
6  desugar (Less xs) = desugarComp LessC xs
7  desugar (Greater xs) = desugarComp GreaterC xs
8  desugar (Diff xs) = desugarComp DiffC xs
9  desugar (Leq xs) = desugarComp LeqC xs
10 desugar (Geq xs) = desugarComp GeqC xs
11
12 --Funcion auxiliar para desazucarar los comparadores
13 desugarComp :: (AST -> AST -> AST) -> [ASA] -> AST
14 desugarComp _ [] = BoolC True
15 desugarComp _ [_] = BoolC True
16 desugarComp op [i, d] = op (desugar i) (desugar d)
17 desugarComp op (i:d:is) = IfC (op (desugar i) (desugar d))
18                             (desugarComp op (d:is))
19                             (BoolC False)
```

Código 4.5: Función `desugarComp` como auxiliar para desazucarar comparadores

De manera similar como fue con los operadores, definimos una función `desugarComp` que recibe una tupla de `AST` para preservar la etiqueta a desazucarar y la lista de elementos a separar que se van a comparar. Los primeros dos casos, son los casos base, donde igualmente, no podemos tener una lista de uno o ningún elemento, pues. La siguiente instrucción sería nuestro caso base real, donde establece que al tener solo dos elementos en la lista, simplemente se devuela la comparación de ambos elementos, mientras que si todavía quedan elementos en la lista, iniciemos la cadena de `IfC`, donde los primeros dos elementos se comparan en la condición y en caso de cumplirse continuamos en el entonces con la llamada recursiva de `desugarComp` del segundo elemento con el resto de la lista, y en caso de no cumplirse, el else es `BoolC False`.
Esta separación y comparación de elementos es válida para cualquier lista de n elementos sin

importar si n es $2k$ o $2k$ - 1, es decir, si la lista tiene un número impar o par de elementos; ya que siempre hacemos la comparación de elemento por elemento hasta llegar al caso donde quedan 2 elementos en la lista que es cuando simplemente se devuelve la comparación de ambos.

Los pares como se mencionó no es necesario desazucararlos más que recursivamente desazucarar a sus hijos:

```
desugar :: ASA -> AST
-- Pares
desugar (Pair f s) = PairC (desugar f) (desugar s)
desugar (Fst p) = FstC (desugar p)
desugar (Snd p) = SndC (desugar p)
```

Código 4.6: Desazucarización de los Pares

Como bien explicamos, las condicionales `If0` y `Cond` son azúcar sintáctica de `If`. `If0` pasa a `IfC` con la comprobación de que el valor en la condición sea igual a cero y nada más.

```
desugar :: ASA -> AST
--Condicionales
desugar (If0 c t e) = IfC (EqualC (desugar c) (NumC 0)) (desugar t) (
   desugar e)
desugar (If c t e) = IfC (desugar c) (desugar t) (desugar e)
desugar (Cond cs e) = desugarCond cs e
```

Código 4.7: Desazucarización de los condicionales

Por otro lado para `Cond`, tenemos que definir una función auxiliar que nos realice el paso a encadenamiento de condicionales `IfC`

```
desugar :: ASA -> AST
--Condicionales
desugar (Cond cs e) = desugarCond cs e

--Funcion auxiliar para desazucarar el operador cond
desugarCond :: [(ASA, ASA)] -> ASA -> AST
desugarCond [] e = desugar e
desugarCond ((c, t):cs) e = IfC (desugar c) (desugar t) (desugarCond cs
   e)
```

Código 4.8: Desazucarización de Cond

Como se puede ver, la función `desugarCond` recibe una lista de pares (*condición, expresión*) junto con una expresión final (el caso `else` implícito). Si la lista de pares es vacía, basta con devolver la expresión por defecto desazucarada. En caso contrario, se construye una estructura `IfC` donde la primera condición se evalúa en el ***if***, la primera expresión en el ***then***, y el resto de los pares en el ***else***, aplicando recursivamente `desugarCond`.
De esta manera, la estructura `Cond` se traduce en una sucesión de evaluaciones `IfC` anidadas, logrando así preservar el mismo comportamiento semántico que tendría en su forma azucarada. Y así garantizamos que sólo se ejecute el cuerpo correspondiente a la primera condición verdadera, respetando la naturaleza secuencial del condicional múltiple.

Como se mencionó, los `Let` en nuestro lenguaje no son construcciones primitivas, sino azúcar sintáctica que se puede expresar completamente a partir de funciones y aplicaciones. Intuitivamente, un `Let` introduce una variable local asociada a un valor dentro de un cuerpo de expresión. Sin embargo, esta noción de "sustitución" puede modelarse directamente mediante la aplicación de una función anónima a un argumento.
Esta equivalencia se refleja directamente en nuestra función auxiliar `desugarLet`, encargada de traducir cualquier `Let` del **ASA** a su correspondiente expresión **AST** basada en aplicaciones de funciones:

```
desugar :: ASA -> AST
--Lets
desugar (Let iv b) = desugarLet iv b
desugar (LetStar [] body) = desugar body
desugar (LetStar (iv:ivs) b) = desugar (Let [iv] (LetStar ivs b))

--Funciones auxiliares para desazucarar let
desugarLet :: [(String, ASA)] -> ASA -> AST
desugarLet [] b = desugar b
desugarLet ((p, v):ps) b = AppC (FunC p (desugarLet ps b)) (desugar v)
```

Código 4.9: Desazucarización de Let y LetStar

La función `desugarLet` recibe una lista de pares `(id, valor)` y el cuerpo del `Let`. En el caso base, cuando no hay más pares, simplemente se desazucara el cuerpo, ya que no hay variables locales restantes por introducir. En el caso general, se construye una función anónima con el primer identificador `p` y cuerpo el resultado de seguir desazucarando los pares restantes junto con el cuerpo `b`. A continuación, esta función se aplica (`AppC`) al valor `v` correspondiente, el cual también se desazucara antes de la aplicación.

Además, `LetStar` -como también se mencionó en la sección anterior- es simplemente azúcar sintáctica de `Let`. El `LetStar` permite escribir múltiples asignaciones secuenciales en un mismo bloque, pero semánticamente equivale a una serie de `Let` anidados. Su desazucarización se implementa recursivamente, construyendo un `Let` por cada par `(id, valor)` y utilizando como cuerpo el siguiente `LetStar`, hasta llegar al cuerpo final.

Por otro lado:

Falta completar lo de LetRec

```
desugar :: ASA -> AST
---LetRec
desugar (LetRec i v b) = desugar (LetStar [(i, App (Lambda [i] v) [
    Lambda [i] v])] b)
---
```

Código 4.10: Desazucarización de LetRec

Una vez establecido lo anterior, continuamos con el caso de los **ASA** `Lambda` en nuestro lenguaje, operan con una lista de parámetros siendo esto azúcar sintáctica, por lo que definimos la función auxiliar `desugarLmb` done "*currificamos*" la función en `FunC` que trabaja sobre un parámetro:

```haskell
desugar :: ASA -> AST
--Expresiones lambda
desugar (Lambda ps b) = desugarLmb ps b

--Funcion auxiliar para desazucarar las funcioneslambda
desugarLmb :: [String] -> ASA -> AST
desugarLmb [] b = desugar b
desugarLmb (p:ps) b = FunC p (desugarLmb ps b)
```

Código 4.11: Desazucarización de las funciones lambda

Si la lista de parámetros está vacía, simplemente se devuelve el cuerpo desazucarado; en caso contrario, se crea un `FunC` con el primer parámetro y como cuerpo el resultado de desazucarar los parámetros restantes junto con el cuerpo.

De igual manera, las aplicaciones de función en `ASA` trabajan con una lista de argumentos, por lo que debemos desazucarlo en aplicaciones sucesivas de `AppC`.

```haskell
desugar :: ASA -> AST
--Expresiones lambda
desugar (App f as) = desugarApp (desugar f) as

--Funcion auxiliar para desazucarar las aplicaciones de funcion
desugarApp :: AST -> [ASA] -> AST
desugarApp f [] = f
desugarApp f (a:as) = desugarApp (AppC f (desugar a)) as
```

Código 4.12: Desazucarización de la aplicación de función

Donde el caso base es que al quedarnos sin argumentos que desazcuarar, devolvemos la función a aplicar, ya que esta fue desazucarada en `AST` desde la primer llamada a `desugarApp`. Por otra parte, si quedan argumentos en la lista continuamos con la llamda recursiva para que formen las aplicaciones sucesivas de `AppC`.

Finalmente tenemos `desugar` para listas. En un principio, la idea fue implementar las listas como encademaiento de pares, no obstante, esto nos trajo problemas al momento de implementar la función que devuelve el resultado al usuario[1], pero aunque realizar esta desazucarización requiere de definir cuatro `AST`, nos facilita el trabajo al momento de evaluar acertadamente lo que el usuario da como programa.

```haskell
desugar :: ASA -> AST
--Listas
desugar (List l) = desugarList l
desugar (Head l) = HeadC (desugar l)
desugar (Tail l) = TailC (desugar l)

--Funcion auxiliar para construir listas como cons y nil
desugarList :: [ASA] -> AST
desugarList [] = NiL
```

[1]Abordaremos más sobre esta situación en próximos capítulos pero en términos simples usar `ConS` y `NiL` mejoró la parte de diferenciar totalementre entre una lista con pares a un par con listas, entre otras combinaciones para poder reflejar correctamente la entrada del usuario sin modificaciones inesperadas.

```
10  desugarList [x] = desugar x
11  desugarList (x:xs) = ConS (desugar x) (desugarList xs)
```

Código 4.13: Desazucarización de Listas

De manera muy similar a como fuimos creando el encadenamiento de operadores o de condicionales `IfC` para los comparadores, en este caso, vamos elemento por elemento de la lista creando un encadenamiento de únicamente `ConS`, mas no utilizamos `NiL`. Este `AST` lo utilizamos de manera reservada para representar las listas vacías, mientras que `HeadC` y `TailC` son similares a `FstC` y `SndC` pero sobre listas.

De este modo terminamos con el algoritmo de la función `desugar` para desazucarar nuestros **ASA** y convertilos a `AST`:

```
1   {-- Desazucaramos los ASA  --}
2   desugar :: ASA -> AST
3   -- Casos base
4   desugar (Var x) = VarC x
5   desugar (Num n) = NumC n
6   desugar (Boolean b) = BoolC b
7   -- Operaciones aritmeticas
8   desugar (Add xs) = desugarOps AddC xs
9   desugar (Sub xs) = desugarOps SubC xs
10  desugar (Mul xs) = desugarOps MulC xs
11  desugar (Div xs) = desugarOps DivC xs
12  desugar (Add1 n) = AddC (desugar n) (NumC 1)
13  desugar (Sub1 n) = SubC (desugar n) (NumC 1)
14  desugar (Expt n) = MulC (desugar n) (desugar n)
15  desugar (Sqrt n) = SqrtC (desugar n)
16  -- Not
17  desugar (Not x) = NotC (desugar x)
18  -- Comparaciones
19  desugar (Equal xs) = desugarComp EqualC xs
20  desugar (Less xs) = desugarComp LessC xs
21  desugar (Greater xs) = desugarComp GreaterC xs
22  desugar (Diff xs) = desugarComp DiffC xs
23  desugar (Leq xs) = desugarComp LeqC xs
24  desugar (Geq xs) = desugarComp GeqC xs
25  -- Pares
26  desugar (Pair f s) = PairC (desugar f) (desugar s)
27  desugar (Fst p) = FstC (desugar p)
28  desugar (Snd p) = SndC (desugar p)
29  --Condicionales
30  desugar (If0 c t e) = IfC (EqualC (desugar c) (NumC 0)) (desugar t) (
        desugar e)
31  desugar (If c t e) = IfC (desugar c) (desugar t) (desugar e)
32  desugar (Cond cs e) = desugarCond cs e
33  --Lets
34  desugar (Let iv b) = desugarLet iv b
35  desugar (LetStar [] body) = desugar body
36  desugar (LetStar (iv:ivs) b) = desugar (Let [iv] (LetStar ivs b))
37  ---
38  ---desugar (LetRec i v b) = desugar (LetStar [(i, App (Lambda [i] v) [
        Lambda [i] v])] b)
```

```haskell
  ---
  --Expresiones lambda
  desugar (Lambda ps b) = desugarLmb ps b
  desugar (App f as) = desugarApp (desugar f) as
  --Listas
  desugar (List l) = desugarList l
  desugar (Head l) = HeadC (desugar l)
  desugar (Tail l) = TailC (desugar l)

  {-- Funciones auxiliares para desugar --}
  --Funcion auxiliar para desazucarar los operadores
  desugarOps :: (AST -> AST -> AST) -> [ASA] -> AST
  desugarOps _ [] = error "[desugarOps Error]: Lista vacia (no deberia
     suceder)"
  desugarOps _ [x] = desugar x
  desugarOps op (x:xs) = op (desugar x) (desugarOps op xs)

  --Funcion auxiliar para desazucarar los comparadores
  desugarComp :: (AST -> AST -> AST) -> [ASA] -> AST
  desugarComp _ [] = BoolC True
  desugarComp _ [_] = BoolC True
  desugarComp op [i, d] = op (desugar i) (desugar d)
  desugarComp op (i:d:is) = IfC (op (desugar i) (desugar d))
  (desugarComp op (d:is))
  (BoolC False)

  --Funcion auxiliar para desazucarar el operador cond
  desugarCond :: [(ASA, ASA)] -> ASA -> AST
  desugarCond [] e = desugar e
  desugarCond ((c, t):cs) e = IfC (desugar c) (desugar t) (desugarCond cs
     e)

  --Funciones auxiliares para desazucarar let
  desugarLet :: [(String, ASA)] -> ASA -> AST
  desugarLet [] b = desugar b
  desugarLet ((p, v):ps) b = AppC (FunC p (desugarLet ps b)) (desugar v)

  --Funcion auxiliar para desazucarar las funcioneslambda
  desugarLmb :: [String] -> ASA -> AST
  desugarLmb [] b = desugar b
  desugarLmb (p:ps) b = FunC p (desugarLmb ps b)

  --Funcion auxiliar para desazucarar las aplicaciones de funcion
  desugarApp :: AST -> [ASA] -> AST
  desugarApp f [] = f
  desugarApp f (a:as) = desugarApp (AppC f (desugar a)) as

  --Funcion auxiliar para construir listas como cons y nil
  desugarList :: [ASA] -> AST
  desugarList [] = NiL
  desugarList [x] = desugar x
  desugarList (x:xs) = ConS (desugar x) (desugarList xs)
```

Código 4.14: Algoritmo para función **desugar** en Haskell completo

Así concluímos con la implementación de desazucarar nuestros Árboles de Sintaxis Abstracta , quedando como resultado los núcleos de nuestro lenguaje MiniLisp -0.2cm. Como vimos, este proceso nos ayuda enormemente a reducir las futuras reglas del interprete, disminuyendo la carga de trabajo y evitando operaciones redundantes y hasta cierto punto innecesarias, quedándonos con lo escencial para facilitar la evaluación y además prevalece nuestro mayorobjetivo que es no alterar ni arriesgar la entrada original del usuario.

Aquí podríamos extender más el cierre del tema de azúcar sintáctica

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] 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

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

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

Bibliografía

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

Bibliografía

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