



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

1. Introducción	3
1.1. Motivación	3
1.2. Objetivos	4
1.3. Delimitación del Proyecto	4
2. Sintaxis Concreta	5
2.1. Sintaxis Léxica	6
2.1.1. Análisis Léxico	6
2.1.2. Tokens	8
2.1.3. Alex	9
2.2. Sintaxis Libre de Contexto	13
2.2.1. La gramática de MINILISP	13
2.2.2. Análisis sintáctico	17
3. Azúcar Sintáctica	19
3.1. Sintaxis Abstracta sin azúcar en MINILISP	19
3.1.1. Desugar en Haskell	21
4. Semántica operacional	27
4.1. Paso pequeño	27
4.1.1. Evaluación perezosa	27
4.1.2. Evaluación ansiosa	27
5. Intérprete	29
5.1. Paso pequeño para MINILISP	29
5.2. Ambientes	29
5.3. Evaluación en Haskell	29
6. Resultados	31
6.1. Menú interactivo	31
6.2. Funciones de prueba	31
6.2.1. Suma primeros n números naturales	31
6.2.2. Factorial	31
6.2.3. Fibonacci	31
6.2.4. Función <code>map</code> para listas	31
6.2.5. Función <code>filter</code> para listas	31

7. Conclusiones	33
Bibliografía	34

Capítulo 1

Introducción

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

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

1.1. Motivación

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

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

1.2. Objetivos

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

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

1.3. Delimitación del Proyecto

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

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

Capítulo 2

Sintaxis Concreta

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

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

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

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

Podemos decir que la sintaxis, constituye la **puerta de entrada entre el usuario y el compilador o intérprete**, definiendo los símbolos, operadores, delimitadores y palabras reservadas que el lenguaje reconoce.

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

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

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

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

En conclusión, podemos pensar en la sintaxis concreta como las secuencias de caracteres del alfabeto Σ que se convierten en programas válidos del lenguaje. Mientras que la *sintaxis abstracta* (**ASA**, Árbol de Sintaxis Abstracta) representa la estructura lógica del programa, la sintaxis concreta establece las **reglas formales de escritura** que garantizan que un programa pueda ser reconocido y analizado correctamente. Su correcta definición es fundamental para el funcionamiento del analizador léxico (*Lexer*) y del analizador sintáctico (*Parser*), ya que determina las entradas válidas que ambos deben procesar. Esto se logra mediante un **Análisis léxico** y un **Análisis sintáctico**.

2.1. Sintaxis Léxica

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

En pocas palabras, citando al profesor:

*“Formalmente, la sintaxis léxica se define usando **expresiones regulares** y **autómatas finitos**.”*

La **sintaxis léxica**, dentro del estudio de los lenguajes formales, representa la primera capa estructural de un lenguaje de programación. Su propósito es definir el *alfabeto* del lenguaje y describir cómo las secuencias de símbolos de dicho alfabeto se agrupan en unidades con significado propio. No describe la estructura lógica o gramatical del programa (de estos se 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`: $\Sigma^* \rightarrow [Token]$, 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:

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

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

- **Paréntesis:** (y), con los que indicamos cuando comienzan y terminan nuestras expresiones (por eso se llaman *delimitadores*).
- **Variables:** cualquier secuencia de caracteres de la forma $[a - z + A - Z][a - zA - Z0 - 9]^*$.
- **Números enteros:** $x \in \mathbb{Z}$.
- **Booleanos:** `#t` (verdadero) y `#f` (falso), junto con la negación (`not`).
- **Operadores aritméticos:** `+`, `-`, `*`, `/`, `++`, `--`, raíz cuadrada (`sqrt`) y potencia (`**`).
- **Predicados y comparaciones:** igualdad y desigualdad (`=`, `!=`), así como comparaciones numéricas (`<`, `>`, `<=`, `>=`).
- **Asignaciones y funciones:** construcciones `let`, `let*`, `letrec`, funciones anónimas con `lambda`, y aplicación de funciones.
- **Pares ordenados y proyecciones:** (`e1`, `e2`), `first` y `second`.
- **Condicionales:** `if`, `if0` y `cond`.
- **Listas:** delimitadas por corchetes [y], con elementos separados por comas , , junto con operaciones básicas `head` y `tail`.

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

2.1.2. Tokens

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

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

Queda definido como sigue:

```
1  data Token
2    = TokenVar String
3    | TokenNum Int
4    | TokenBool Bool
5    | TokenAdd
6    | TokenSub
7    | TokenMul
8    | TokenDiv
9    | TokenAdd1
10   | TokenSub1
11   | TokenSqrt
12   | TokenExpt
13   | TokenNot
14   | TokenEq
15   | TokenLt
16   | TokenGt
17   | TokenNeq
18   | TokenLeq
19   | TokenGeq
20   | TokenIf0
21   | TokenIf
22   | TokenCond
23   | TokenElse
24   | TokenFirst
25   | TokenSecond
26   | TokenHead
27   | TokenTail
28   | TokenLet
29   | TokenLetRec
30   | TokenLetStar
31   | TokenLambda
32   | TokenLI
33   | TokenLD
34   | TokenComma
35   | TokenPA
36   | TokenPC
37  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 contruidos 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-zA-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+`).

```

1  {
2  module Lexer where
3
4  import Token
5  import Data.Char (isSpace)
6  }
7
8  %wrapper "basic"
9
10 -- Definiciones de patrones
11 $digit    = 0-9
12 $alpha    = [a-zA-Z]
13 $alnum    = [a-zA-Z0-9]
14
15 -- Usamos codigos hex para los espacios en blanco Unicode mas comunes:
16 -- \x20 = ' ' (space), \x09 = tab, \x0A = LF, \x0D = CR, \x0C = FF, \
   x0B = VT
17 $white    = [\x20\x09\x0A\x0D\x0C\x0B]
18
19 tokens :-
20
21 -- Ignoramos espacios y saltos de linea
22 $white+
   ;

```

Código 2.2: Lexer con Alex.

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

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

- **Patrón o expresión regular:** Es la secuencia de caracteres que el lexer debe reconocer. En este caso, se trata de los símbolos estructurales o palabras clave como (, let, +, etc. Cabe mencionar que estos caracteres pueden definirse de manera personalizada; sin embargo, para mantener la coherencia con la notación tradicional de los lenguajes de programación, utilizamos los símbolos comúnmente aceptados, como + para la suma y - para la resta.
- **Bloque de acción:** Es el fragmento de código en Haskell que se ejecuta cuando se reconoce el patrón. Su función es generar el token correspondiente, por ejemplo:
`{ _ ->TokenPA }`.
- **Expresión lambda:** Dentro del bloque de acción, la expresión lambda define cómo se construye el token. En el ejemplo anterior, `_ ->TokenPA`, el símbolo `_` representa la cadena de texto que coincidió con el patrón (la entrada reconocida), el operador `->` separa el parámetro del resultado, y `TokenPA` es el constructor del token que se devuelve al análisis sintáctico.

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

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

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

```

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

```

Código 2.3: Lexer con Alex.

Nótese que tenemos las reglas para booleanos y literales con `#t` y `#f` para `TokenBool`, mientras que con `¿$digit+` para uno o más dígitos a partir de la cadena `s` incluyendo los números negativos con `¿` y las variables con `$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

```

1  -- Catch-all para diagnosticar caracteres inesperados
2  . { \s -> error ("Lexical error: caracter no
    reconocido = "
3
4      ++ show s
5      ++ " | codepoints = "
6      ++ show (map fromEnum s)) }
7
8  {
9      -- Normaliza cualquier espacios en blanco Unicode a ' ' para que
10     $white+ lo consuma
11     normalizeSpaces :: String -> String
12     normalizeSpaces = map (\c -> if isSpace c then '\x20' else c)
13
14     lexer :: String -> [Token]
15     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 *tokenización* de expresiones dadas dentro del lenguaje MINILISP.

¹Para 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[4] para desarrollar nuestro lexer.

2.2. Sintaxis Libre de Contexto

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

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

2.2.1. La gramática de MINILISP

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

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

donde:

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

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

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

Con la notación **EBNF**:

- Las **variables** o no terminales se denotan entre los símbolos $\langle \rangle$.
- 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* (*variádicas*), 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 variádicas, es decir, permiten realizar asignaciones 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>})
        | (++) <Expr>
        | (--) <Expr>
        | (sqrt <Expr>)
        | (** <Expr>)
        | (not <Expr>)
        | (= <Expr> <Expr> {<Expr>})
        | (< <Expr> <Expr> {<Expr>})
        | (> <Expr> <Expr> {<Expr>})
        | (<= <Expr> <Expr> {<Expr>})
        | (>= <Expr> <Expr> {<Expr>})
        | (!= <Expr> <Expr> {<Expr>})
        | (<Expr>, <Expr>)
        | (fst <Expr>)
        | (snd <Expr>)
        | (let ((<Var> <Expr>) {<Var> <Expr>}) <Expr>)
        | (letrec (<Var> <Expr>) <Expr>)
        | (let* ((<Var> <Expr>) {<Var> <Expr>}) <Expr>)
        | (if0 <Expr> <Expr> <Expr>)
        | (if <Expr> <Expr> <Expr>)
        | (lambda (<Var> {<Var>}) <Expr>)
        | (<Expr> <Expr> {<Expr>})
        | “[” [ <Expr> {“,”<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 al menos, 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**:

$$\text{parser}: [\text{Token}] \rightarrow \text{ASA}$$

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:

$$(\text{parser} \circ \text{lexer}): \Sigma^* \rightarrow \text{ASA}$$

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

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 eliminado 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:

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

`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...

3.1. Sintaxis Abstracta sin azúcar en MINILISP

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

$$\text{desugar} ::= \Rightarrow \text{Sugared_ASA} \rightarrow \text{Desugared_ASA}$$

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

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

- **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 encadementio del mismo. Por ejemplo:

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

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 reexpresarlos 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 sintáctica y lo reexpresamos como una multiplicación $n \times n$:

$$\begin{aligned} \text{desugar}(\text{Add1 } n) &\Rightarrow \text{Add } n \ 1 \\ \text{desugar}(\text{SugarSqrt } n) &\Rightarrow \text{Sqrt } n \\ \text{desugar}(\text{Expt } n) &\Rightarrow \text{Mul } n \ n \end{aligned}$$

- **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 comparador 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 encadementio de comparadores, pues al evaluar la comparación entre dos números, el resultado es de tipo `Bool`. Si como un encadementio de condicionales `if`. Donde la condición inicial es la comparación de los dos primeros argumentos y el consecuente son las comparaciones de los 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`:

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

- **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`:

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

- **Lets:**

Falta la definición de Lets sin azúcar...

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

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

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:

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

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

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

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

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

3.1.1. Desugar en Haskell

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

```

1 module AST where
2
3 -- ASA sin azúcar (AST)
4 data AST
5   = VarC String
6   | NumC Int
7   | BoolC Bool
8   | 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 3.1: Tipo de dato ASA sin azúcar, AST

AST (*Abstract Syntax 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 sección 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 re-expresan 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 archivo **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 3.2: Firma y casos base de la función desugar

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

```

1  desugar :: ASA -> AST
2  -- Operaciones aritmeticas
3  desugar (Add xs) = desugarOps AddC xs
4  desugar (Sub xs) = desugarOps SubC xs
5  desugar (Mul xs) = desugarOps MulC xs
6  desugar (Div xs) = desugarOps DivC xs

```

```

7  desugar (Add1 n) = AddC (desugar n) (NumC 1)
8  desugar (Sub1 n) = SubC (desugar n) (NumC 1)
9  desugar (Expt n) = MulC (desugar n) (desugar n)
10 desugar (Sqrt n) = SqrtC (desugar n)

```

Código 3.3: 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.

```

1  desugar :: ASA -> AST
2  -- Operaciones aritmeticas
3  desugar (Add xs) = desugarOps AddC xs
4  desugar (Sub xs) = desugarOps SubC xs
5  desugar (Mul xs) = desugarOps MulC xs
6  desugar (Div xs) = desugarOps DivC xs
7
8  --Funcion auxiliar para desazucarar los operadores
9  desugarOps :: (AST -> AST -> AST) -> [ASA] -> AST
10 desugarOps _ [] = error "[desugarOps Error]: Lista vacia (no deberia
    suceder)"
11 desugarOps _ [x] = desugar x
12 desugarOps op (x:xs) = op (desugar x) (desugarOps op xs)

```

Código 3.4: 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 mantenemos 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 argumento 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 3.5: 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 devuelve 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:

```

1  desugar :: ASA -> AST
2  -- Pares
3  desugar (Pair f s) = PairC (desugar f) (desugar s)
4  desugar (Fst p) = FstC (desugar p)
5  desugar (Snd p) = SndC (desugar p)

```

Código 3.6: 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.

```

1  desugar :: ASA -> AST
2  -- Condicionales
3  desugar (If0 c t e) = IfC (EqualC (desugar c) (NumC 0)) (desugar t) (
    desugar e)
4  desugar (If c t e) = IfC (desugar c) (desugar t) (desugar e)
5  desugar (Cond cs e) = desugarCond cs e

```

Código 3.7: 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`

```

1  desugar :: ASA -> AST
2  -- Condicionales
3  desugar (Cond cs e) = desugarCond cs e
4
5  -- Funcion auxiliar para desazucarar el operador cond
6  desugarCond :: [(ASA, ASA)] -> ASA -> AST
7  desugarCond [] e = desugar e
8  desugarCond ((c, t):cs) e = IfC (desugar c) (desugar t) (desugarCond cs
    e)

```

Código 3.8: Desazucarización de los Pares

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.

Aquí va lo que falta de explicar de la desazucarización de los Lets

Lambda

Capítulo 4

Semántica operacional

4.1. Paso pequeño

4.1.1. Evaluación perezosa

4.1.2. Evaluación ansiosa

Capítulo 5

Intérprete

5.1. Paso pequeño para `MINILISP`

5.2. Ambientes

5.3. Evaluación en Haskell

Capítulo 6

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:

6.1. Menú interactivo

6.2. Funciones de prueba

6.2.1. Suma primeros n números naturales

6.2.2. Factorial

6.2.3. Fibonacci

6.2.4. Función `map` para listas

6.2.5. Función `filter` para listas

Capítulo 7

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] Documentación Haskell. Disponible en: <https://www.haskell.org>
- [4] Documentación Alex(Haskell) The Alex Lexer Generator for Haskell Programming in Haskell (Graham Hutton, 2nd Edition). Sección sobre parsers y lexers. Disponible en: <https://www.haskell.org/alex/>
- [5] Marlow, S., Gill, A. (2009). Happy. Disponible en: <https://www.haskell.org/happy/>
- [6] Autor. "Artículo". Revista, Año.