



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:

3 de noviembre, 2025

Índice

1. Introducción	3
1.1. Motivación	3
1.2. Objetivos	3
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. Sintaxis Abstracta	19
3.1. Árboles de Sintaxis Abstracta	20
3.2. ASA para MINILISP	20
3.2.1. Expresiones atómicas	21
3.2.2. Operadores aritméticos	22
3.2.3. Predicados y comparaciones	24
3.2.4. Expresiones Let y Términos λ	26
3.2.5. Pares ordenados y Proyecciones	29
3.2.6. Condicionales	29
3.2.7. Listas	30
3.3. ASA en Haskell	31
3.4. Parser con Happy	33
4. Azúcar Sintáctica	39
4.1. Sintaxis Abstracta sin azúcar en MINILISP	40
4.1.1. Desugar en Haskell	43
5. Semántica Operacional	53
5.1. ¿Qué es la semántica operacional?	53
5.2. Semántica Estructural (Paso pequeño)	54
5.2.1. Sistema de transición	55

5.2.2.	Estados Finales en MINILISP	56
5.2.3.	Reglas de evaluación	58
5.3.	Intérprete para MINILISP	63
5.3.1.	Función paso pequeño en MINILISP	63
6.	Puntos Estrictos	73
6.1.	Reglas semánticas perezosas con strict	73
6.2.	Evaluación perezosa para MINILISP	75
6.3.	Cierre: intérprete, recursión y evaluación glotona	80
6.3.1.	Cierre del intérprete (closures)	80
6.3.2.	Recursión y combinadores	80
6.3.3.	Evaluación glotona (estricta): reglas contrastantes	81
6.3.4.	Cierre	81
7.	Resultados	83
7.1.	MINILISP	83
7.1.1.	Lexer	83
7.1.2.	Parser	86
7.1.3.	Desugar	91
7.2.	Menú interactivo	93
7.3.	Funciones de prueba	98
7.3.1.	Suma primeros n números naturales	99
7.3.2.	Factorial	100
7.3.3.	Fibonacci	100
8.	Conclusiones	103
	Bibliografía	105
	Bibliografía	106
	Bibliografía	108

Capítulo 1

Introducción

1.1. Motivación

El estudio de los lenguajes de programación es uno de los pilares fundamentales de las ciencias de la computación. Comprender cómo se definen, interpretan y ejecutan permite no solo utilizarlos como herramientas, sino también analizarlos y extenderlos desde la perspectiva formal. En este contexto, la implementación de un lenguaje estilo **Lisp** ofrece un terreno fértil para explorar la relación entre teoría y práctica: su sintaxis minimalista, su semántica clara y su estructura recursiva facilitan su análisis formal.

El proyecto **MiniLisp** surge con la intención de integrar los conceptos teóricos vistos en clase dentro de una implementación concreta en **Haskell**. De este modo, este trabajo busca fortalecer la comprensión del proceso completo de formalización de un lenguaje de programación: desde la definición léxica y gramatical, hasta la evaluación de programas mediante un intérprete funcional.

1.2. Objetivos

Objetivo general

Desarrollar e implementar una versión extendida del lenguaje **MiniLisp** que formalice su sintaxis y semántica, y que permita ejecutar programas a través de un intérprete en **Haskell**, manteniendo coherencia entre el modelo teórico y la implementación práctica.

Objetivos específicos

1. Definir formalmente la sintaxis léxica y libre de contexto del lenguaje, incluyendo operadores, estructuras de control y mecanismos de definición local.
2. Implementar un analizador léxico y un analizador sintáctico utilizando las herramientas **Alex** y **Happy**, respectivamente.
3. Diseñar la sintaxis abstracta usando dos niveles, un **ASA** con azúcar sintáctica y **AST** como núcleo del lenguaje.

4. Desarrollar un módulo de eliminación de azúcar sintáctica (**Desugar**) que traduzca expresiones superficiales a su representación mínima.
5. Implementar un intérprete funcional (**Interp**) basado en la semántica operacional, utilizando ambientes y **evaluación ansiosa** (eager evaluation).
6. Proveer una interfaz interactiva que permita ejecutar programas escritos en la sintaxis concreta de **MINILISP**.

1.3. Delimitación del Proyecto

El proyecto se reduce al diseño e implementación de un subconjunto de **Lisp** llamado **MINILISP**, con el propósito de estudiar los principios fundamentales de los lenguajes funcionales y su formalización. Por lo tanto:

- No se abordará la **gestión de tipos** ni el **análisis estático**.
- El sistema de evaluación se restringe a la **evaluación ansiosa** (eager evaluation).
- La semántica implementada se limita al **nivel estructural**, sin considerar aspectos de optimización, compilación ni concurrencia.
- El alcance del proyecto comprende la construcción del **intérprete**, no un compilador ni un entorno gráfico.

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* [1].

*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 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 expresion (por eso se llaman *delimitadores*).
- **Variables:** cualquier secuencia de caracteres de la forma $[a - z + A - Z][a - z A - Z 0 - 9]^*$.
- **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:** (`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[6]. 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`, `!=`, `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.

```

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 "add1"             { \_ -> TokenAdd1 }
14 "sub1"             { \_ -> 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

Según Hopcroft y Ullman es su libro *Introduction to Automata Theory, Languages, and Computation* [2], 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 símbolos terminales**.
- P es el conjunto finito de **reglas de producción**; cada producción es de la forma $A \rightarrow \alpha$, 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 $\langle \rangle$.
- 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>})
        | (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 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

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 establecido, 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 distinguen 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).

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

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)

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

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).

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

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

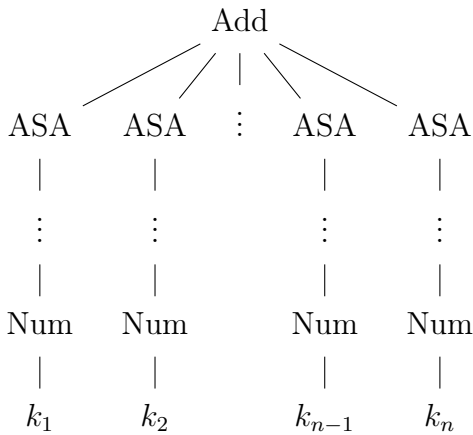
Como bien explicamos, dentro de nuestro lenguaje algunos de los operadores aritméticos se caracterizan por ser *variádicos*, por lo que su representación dentro del **ASA** se conforma de un árbol *n-ario*, mientras que el resto de los operadores aplican únicamente a un único argumento, es decir, son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

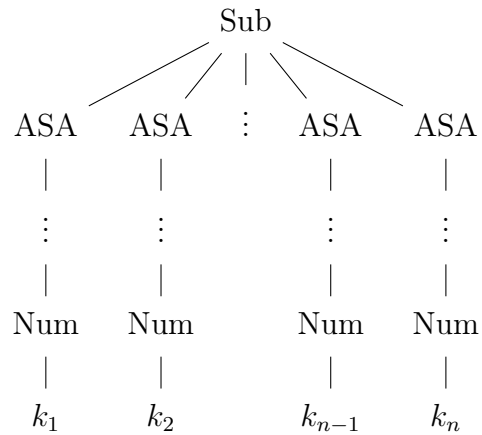
$Add(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**. Es un árbol *n-ario*.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Add(e_1, e_2, \dots, e_n) \text{ ASA}}$$



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

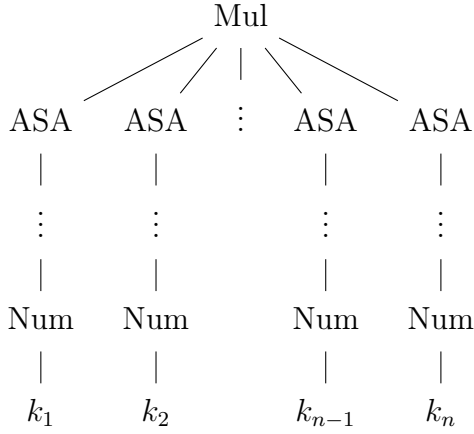
$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Sub(e_1, e_2, \dots, e_n) \text{ ASA}}$$



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

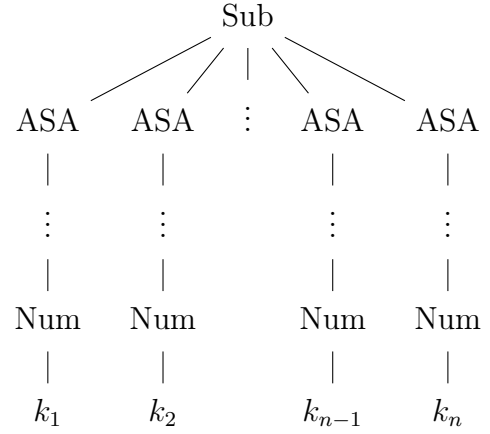
$Mul(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Mul(e_1, e_2, \dots, e_n) \text{ ASA}}$$



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

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Div(e_1, e_2, \dots, e_n) \text{ ASA}}$$

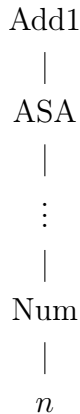


■ No variádicos

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*.

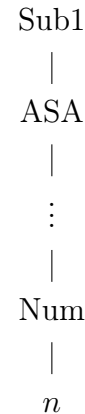
$Add1(e)$ es un **ASA** si e es un **ASA**.

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



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

$$\frac{e \text{ ASA}}{Sub1(e) \text{ ASA}}$$



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

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

$$\frac{e \text{ ASA}}{Sqrt(e) \text{ ASA}}$$

$$\frac{e \text{ ASA}}{Expt(e) \text{ ASA}}$$

Sqrt
|
ASA
|
⋮
|
Num
|
 n

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*.

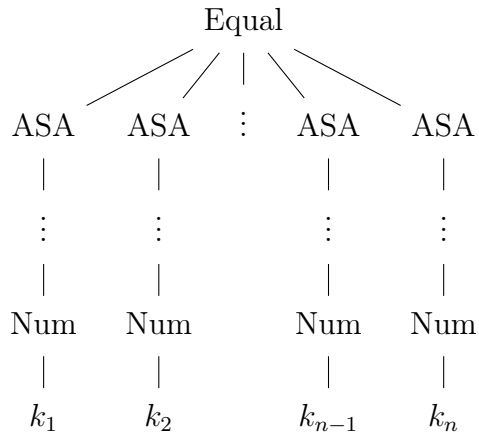
$$\frac{p \text{ ASA}}{Not(p) \text{ ASA}}$$

Not
|
ASA
|
⋮
|
Boolean
|
 n

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

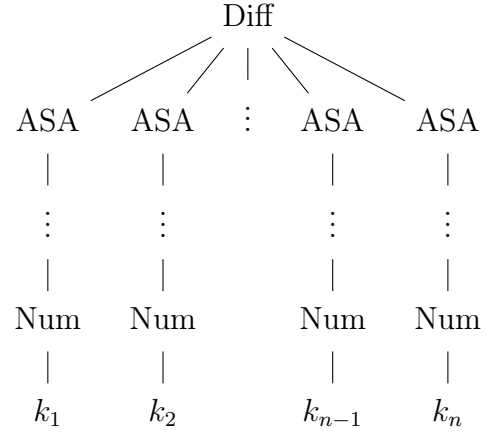
$Equal(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Equal(e_1, e_2, \dots, e_n) \text{ ASA}}$$



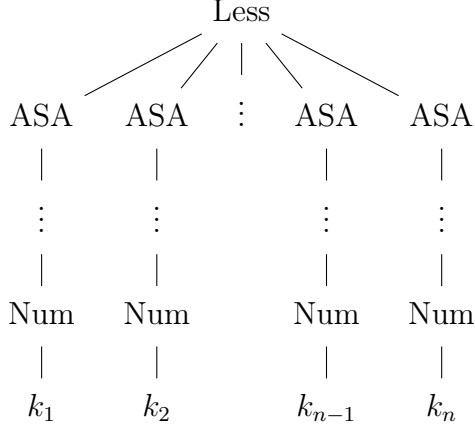
$Diff(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Diff(e_1, e_2, \dots, e_n) \text{ ASA}}$$



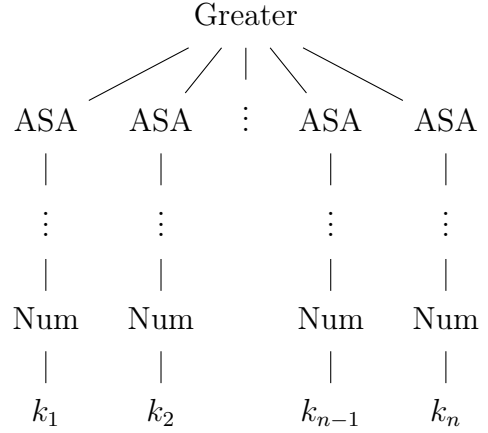
$Less(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Less(e_1, e_2, \dots, e_n) \text{ ASA}}$$



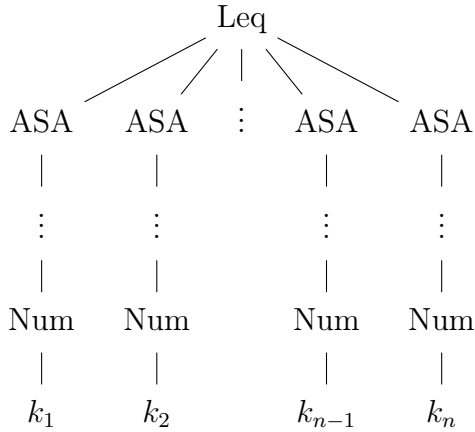
$Greater(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Greater(e_1, e_2, \dots, e_n) \text{ ASA}}$$



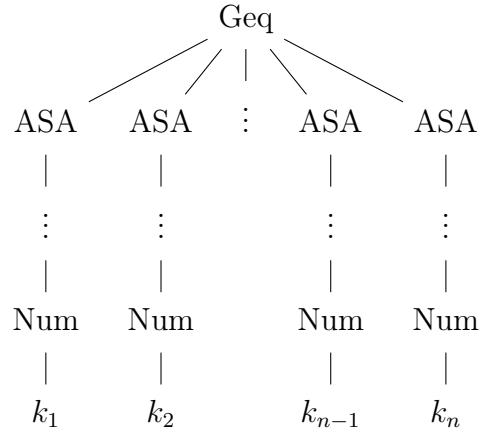
$Leq(e_1, e_2, \dots, e_n)$ es un **ASA** si cada e_i es un **ASA**.

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Leq(e_1, e_2, \dots, e_n) \text{ ASA}}$$



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

$$\frac{e_1, e_2, \dots, e_n \text{ ASA}}{Geq(e_1, e_2, \dots, e_n) \text{ ASA}}$$



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. De manera 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:

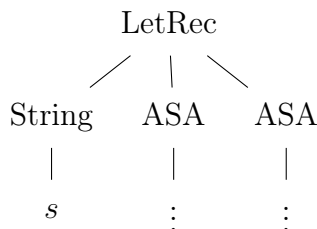
- **let** define un nuevo entorno donde las variables se evalúan en paralelo (**alcance estático**).
- **let*** permite una evaluación secuencial, donde las definiciones anteriores pueden ser utilizadas en las siguientes (**alcance dinámico**).
- **letrec** introduce definiciones recursivas, es decir, 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 identificador i es de tipo **String**, el valor v y el cuerpo b son todos **ASA**.

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

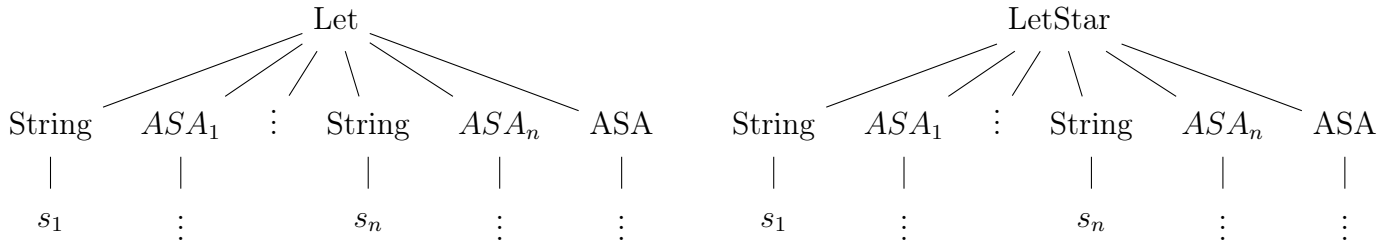


$Let((i_1, v_1), (i_2, v_2), \dots, (i_n, v_n), b)$ es un **ASA** si cada identificador i_j es de tipo **String** y cada valor v_j y cuerpo b son todos **ASA**.

$$\frac{i_1, i_2 \dots, i_n: \text{String } v_1, v_2, \dots, v_n, b \text{ ASA}}{Let((i_1, v_1), (i_2, v_2), \dots, (i_n, v_n), b) \text{ ASA}}$$

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

$$\frac{i_1, i_2 \dots, i_n: \text{String } v_1, v_2, \dots, v_n, b \text{ ASA}}{LetStar((i_1, v_1), (i_2, v_2), \dots, (i_n, v_n), b) \text{ ASA}}$$



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 siendo 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, \dots, e_n) , 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:

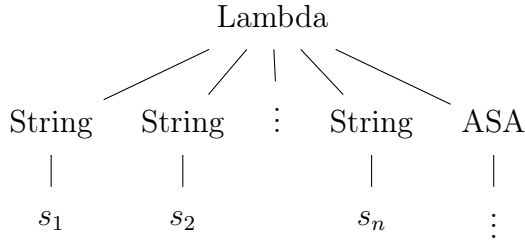
1. Como explicamos, la primer expresión representa la función que será aplicada (intuitivamente una expresión **Lambda** en nuestro lenguaje), esta no es variádica.
2. La segunda expresión corresponde a una lista de n argumentos sobre los cuales se aplicará la función anterior, y sí es variádica, ya 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:

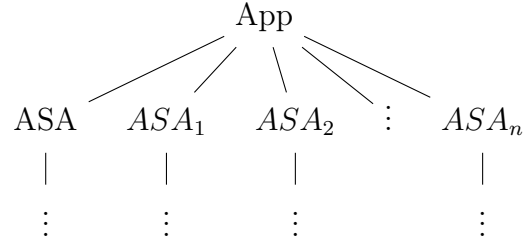
$\text{Lambda}(i_1, i_2, \dots, i_n, c)$ es un **ASA** si cada identificador i_j es de tipo **String** que representa el nombre de su parámetro y b es **ASA** que representa el cuerpo.

$$\frac{i_1, i_2, \dots, i_n: \text{String } b \text{ ASA}}{\text{Lambda}(i_1, i_2, \dots, i_n, b) \text{ ASA}}$$



$\text{App}(f, e_1, e_2, \dots, e_n)$ 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**.

$$\frac{f \text{ ASA } e_1, e_2, \dots, e_n \text{ ASA}}{\text{App}(e_0, e_1, e_2, \dots, e_n) \text{ ASA}}$$

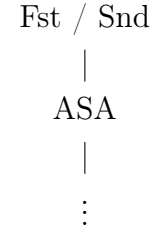
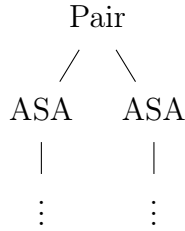


3.2.5. Pares ordenados y Proyecciones

Para los pares ordenados y sus proyecciones basta con tener en cuenta las siguientes reglas:
 $\text{Pair}(f, s)$ es un **ASA** si las expresiones f y s son **ASA**.
 $\text{Fst}(p)$ y $\text{Snd}(p)$ son **ASA** si la expresión p es **ASA**.

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

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

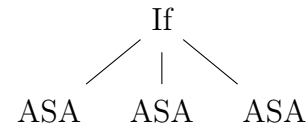
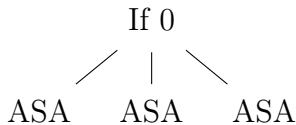


3.2.6. Condicionales

Tenemos dos condicionales **if0** e **if**, ambas son similares en cuanto syntaxis abstracta y por ende, en cuanto a su **ASA**:
 $\text{If0}(c, t, e)$ es un **ASA** si c, t y e son **ASA**.
 $\text{If}(c, t, e)$ es un **ASA** si c, t y e son **ASA**.

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

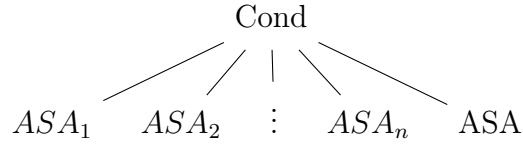
$$\frac{c \text{ ASA } t \text{ ASA } e \text{ ASA}}{\text{If}(c, t, e) \text{ ASA}}$$



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

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

$$\frac{c_1, t_1, c_2, t_2, \dots, c_n, t_n, e \text{ **ASA**}}{Cond((c_1, t_1), (c_2, t_2), \dots, (c_n, t_n), e) \text{ **ASA**}}$$

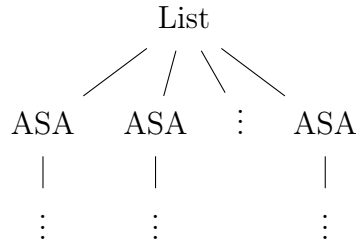


3.2.7. Listas

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

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

$$\frac{e_1, e_2, \dots, e_n \text{ **ASA**}}{List(e_1, e_2, \dots, e_n) \text{ **ASA**}}$$



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.

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

$$\frac{l \text{ **ASA**}}{Head(l) \text{ **ASA**}}$$

Head
|
ASA
|
vdots

$$\frac{l \text{ **ASA**}}{Tail(l) \text{ **ASA**}}$$

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:

```

1 data ASA
2   = Var String
3   | Num Int
4   | Boolean Bool
5   | Add [ASA]
6   | Sub [ASA]
7   | Mul [ASA]
8   | Div [ASA]
9   | Add1 ASA
10  | Sub1 ASA
11  | Sqrt ASA
12  | Expt ASA
13  | Not ASA
14  | Equal [ASA]
15  | Less [ASA]
16  | Greater [ASA]
17  | Diff [ASA]
18  | Leq [ASA]
19  | Geq [ASA]
20  | Pair ASA ASA
21  | Fst ASA
22  | Snd ASA
23  | Let [(String, ASA)] ASA
24  | LetRec String ASA ASA
25  | LetStar [(String, ASA)] ASA
26  | If0 ASA ASA ASA
27  | If ASA ASA ASA
28  | Lambda [String] ASA
29  | App ASA [ASA]
30  | List [ASA]
31  | Head ASA
32  | Tail ASA
33  | Cond [(ASA, ASA)] ASA
34  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:

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

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:

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

- **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:

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

$$(\text{cond } [(>x \ 0) \ 1] \ [(<x \ 0) \ 2] \ [\text{else } 3])$$

\Rightarrow

$$\text{Cond} \ [(\text{Greater} \ [\text{Var } \text{"x"}, \text{Num } 0], \text{Num } 1)(\text{Less} \ [\text{Var } \text{"x"}, \text{Num } 0], \text{Num } 2)](\text{Num } 3)$$

- **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:

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

$$((\text{lambda}(x \ y)(+ \ x \ y)) \ 2 \ 3)$$

\Rightarrow

$$\text{App}(\text{Lambda} \ [\text{"x"}, \text{"y"}](\text{Add} \ [\text{Var } \text{"x"}, \text{Var } \text{"y"}])[\text{Num } 2, \text{Num } 3])$$

- **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:

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

- **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:

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

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

3.4. Parser con Happy

Happy 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.

```
1 {  
2 module Grammar where  
3  
4 import Lexer  
5 import Token  
6 import ASA  
7 }  
8  
9 %name parse  
10 %tokentype { Token }  
11 %error { parseError }  
12  
13 ...  
14  
15 {  
16 parseError :: [Token] -> a  
17 parseError _ = error "Parser Error"  
18 }
```

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:

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

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*:

```

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

```

1 %%
2
3 ASA
4 : var                { Var $1 }
5 | num                { Num $1 }
6 | boolean            { Boolean $1 }
7 | '(' '+' opArgs ')' { Add (reverse $3) }
8 | '(' '-' opArgs ')' { Sub (reverse $3) }
9 | '(' '*' opArgs ')' { Mul (reverse $3) }
10 | '(' '/' opArgs ')' { Div (reverse $3) }
11 | '(' '=' opArgs ')' { Equal (reverse $3) }
12 | '(' '<' opArgs ')' { Less (reverse $3) }
13 | '(' '>' opArgs ')' { Greater (reverse $3) }
14 | '(' '!=' opArgs ')' { Diff (reverse $3) }
15 | '(' '<=' opArgs ')' { Leq (reverse $3) }
16 | '(' '>=' opArgs ')' { Geq (reverse $3) }
17 | '(' '++' ASA ')' { Add1 $3 }
18 | '(' '--' ASA ')' { Sub1 $3 }
19 | '(' "sqrt" ASA ')' { Sqrt $3 }
20 | '(' "==" ASA ')' { Expt $3 }
21 | '(' "not" ASA ')' { Not $3 }
22 | '(' ASA ',' ASA ')' { Pair $2 $4 }
23 | '(' "first" ASA ')' { Fst $3 }
24 | '(' "second" ASA ')' { Snd $3 }
25 | '(' "let" '(' ids ')' ASA ')' { Let (reverse $4) $6 }
26 | '(' "letrec" '(' var ASA ')' ASA ')' { LetRec $4 $5 $7 }
27 | '(' "let*" '(' ids ')' ASA ')' { LetStar (reverse $4) $6 }
28 | '(' "if0" ASA ASA ASA ')' { If0 $3 $4 $5 }
29 | '(' "if" ASA ASA ASA ')' { If $3 $4 $5 }
30 | '(' "lambda" '(' vars ')' ASA ')' { Lambda (reverse $4) $6 }
31 | '(' ASA appArgs ')' { App $2 (reverse $3) }
32 | '(' '[' listArgs ']' ')' { List (reverse $3) }
33 | '(' "head" ASA ')' { Head $3 }
34 | '(' "tail" ASA ')' { Tail $3 }
35 | '(' "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:

```

1 opArgs
2   : ASA ASA                { [$2, $1] }
3   | opArgs ASA             { $2 : $1 }
4
5 ids
6   : id                     { [$1] }
7   | ids id                 { $2 : $1 }
8
9 id
10  : '(' var ASA ')'        { ($2, $3) }
11
12 vars
13  : var                     { [$1] }
14  | vars var                { $2 : $1 }
15
16 appArgs
17  : ASA                     { [$1] }
18  | appArgs ASA             { $2 : $1 }
19
20 listArgs
21  : {- empty -}             { [] }
22  | ASA                     { [$1] }
23  | listArgs ',' ASA        { $3 : $1 }
24
25 condis
26  : condy                   { [$1] }
27  | condis condy            { $2 : $1 }
28
29 condy
30  : '[' 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 vacía 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 capítulo, 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

El término *azúcar sintáctica* (*syntactic sugar*) fue introducido por Peter J. Landin en 1964. La azúcar sintáctica consiste en construcciones del lenguaje que pueden ser sistemáticamente traducidas a formas más básicas sin alterar la semántica del programa. Es decir, no añaden nuevas capacidades expresivas, sino que facilitan la escritura o lectura del código.

Landin fue un pionero de la teoría de lenguajes de programación, utilizó el término para describir aquellas construcciones que, aunque convenientes para el programador, pueden eliminarse mediante una transformación mecánica sin modificar el significado del programa.

Landin uso este termino por primera vez en su trabajo sobre la correspondencia entre programación y cálculo lambda, específicamente en el artículo "*The Next 700 Programming Languages*" [3], aunque la idea aparece en escritos de 1964.

Continuando con nuestro proyecto. Una vez que hemos eliminado todos los elementos superficiales y auxiliares de la sintaxis concreta y léxica para quedarnos únicamente con la representación mínima y estructural del programa, es decir, el **Arbol de Sintaxis Abstracta** (ASA) logramos capturar la estructura esencial del programa. Pasamos a la siguiente fase, 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, hacer esto nos reduce en mayor medida las reglas que debemos implementar al momento de la evaluación, en nuestra implementación pasamos de ASA (con azúcar sintáctica) a AST (sin azúcar sintáctica).

Para explicar lo anterior con mas detalle tomaremos un 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*.

4.1. Sintaxis Abstracta sin azúcar en MINILISP

Necesitamos definir una función que realice el prodecimiento 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 encademiento 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 se $n \times n$:

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

- **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 encademiento de comparadores, pues al evaluar la comparación entre dos números, el resultado es de tipo **Bool**. Si 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 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**:

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

- **Lets:** Los **Let** son solo la azúcar de la aplicación de funciones donde la sustitución 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 sintáctica de **Let**, de modo que **LetStar** se reexpresa como **lets** anidados.

Por lo que el proceso de desazucarización sería similar a:

$$\begin{aligned} \text{desugar}(\text{Let}(id, value) \ body) &\Rightarrow \text{App } (body) \ (id, value) \\ \text{desugar}(\text{LetStar}(id_1, value_1) \ (id_2, value_2) \ \dots \ (id_n, value_n) \ body) &\Rightarrow \text{Let } ((id_1, \\ &\quad value_1) \ (\text{Let } (id_2, value_2) \ \dots \ (\text{Let}(id_n, value_n) \ body))) \end{aligned}$$

Pasando al caso específico de **letrec**, este nos permite definir funciones recursivas locales. Formalmente, la recursión puede desazucararse utilizando un operador de punto fijo. Para lenguajes de evaluación ansiosa se utiliza el operador **Z**, que implementa recursión sin necesidad de auto-referencia explícita.

$$\text{desugar}(\text{LetRec}(id, value) \ body) \Rightarrow \text{Let } ((id, \text{App } (\text{Lambda } (id) \ value)(\text{Lambda } (id) \ value))) \ body$$

En el trabajo *The Formal of Programming Languages*, el concepto de punto fijo es la herramienta matemática fundamental para dar un significado preciso a las construcciones recursivas de un lenguaje, como los bucles y las funciones que se llaman a sí mismas [4]

El **combinador de punto fijo Z** se define en cálculo lambda como:

$$Z = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$$

que satisface la propiedad:

$$Z F = F (Z F) \quad \text{para cualquier término } F$$

¿Por qué es necesario el combinador *Z* de punto fijo? En la implementación de nuestro miniLisp, hemos definido las funciones anónimas (λ) como un núcleo fundamental del lenguaje. Esta decisión se tomó con el fin de poder desazucarar muchas construcciones del lenguaje. Por ejemplo, *let* y *letstar* pueden ser traducidos de forma directa a simples aplicaciones de *lambda*.

Sin embargo, *letrec* (la construcción que permite definir funciones recursivas) presenta un desafío.

La recursión, se basa en nombres: una función se llama a sí misma por su propio nombre hasta llegar a un caso base. Pero esto crea una paradoja en nuestro sistema: si el núcleo solo entiende de funciones anónimas, ¿cómo puede una función sin nombre llamarse a sí misma?

Es aquí donde la teoría del punto fijo se vuelve clave.

En lugar de intentar que la función se llame a sí misma (lo cual es imposible sin un nombre), se crea una función "generadora" (un funcional), llamémosla *G*. Esta función *G* no es recursiva, sino que acepta un argumento. Dicho argumento, llamémoslo *g*, representa la suposición de la función recursiva.

Para solucionar esto es necesario el uso de una "máquina" que haga esto automáticamente. Esa máquina es un combinador de punto fijo (como el Combinador *Y* visto en clase).

- **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:

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

De esta manera, la estructura superficial de las listas en la sintaxis concreta, que aparenta ser n-aria se reexpresa en términos de una estructura binaria en el núcleo del lenguaje. En otras palabras, una lista como *[a,b,c]* no es realmente un nodo con tres hijos, sino una secuencia anidada de constructores *Cons*, cada uno tomando dos argumentos: el elemento y la referencia al resto de la lista.

Esto implica que algunas construcciones que originalmente se representan como árboles n-arios en el **ASA**, después de ser desazucarizadas corresponden a árboles binarios en el **AST**. *Head*, *Tail* ya no operan sobre listas **ASA**, sino directamente sobre la estructura *Nil* y *Cons*, que constituye el núcleo semántico para listas.

4.1.1. Desugar en Haskell

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

```

1 module AST where
2
3 -- ASA sin azucar (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 4.1: Tipo de dato ASA sin azúcar, AST

AST (*Abstract Syntaxis 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  {- Desazucaremos 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 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 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.

```

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 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 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 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 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 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.

```

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 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`

```

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 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:

```

1  desugar :: ASA -> AST
2  -- Lets
3  desugar (Let iv b) = desugarLet iv b
4  desugar (LetStar [] body) = desugar body
5  desugar (LetStar (iv:ivs) b) = desugar (Let [iv] (LetStar ivs b))
6
7  -- Funciones auxiliares para desazucarar let
8  desugarLet :: [(String, ASA)] -> ASA -> AST
9  desugarLet [] b = desugar b
10 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 tenemos el caso del `LetRec`:

```

1  desugar :: ASA -> AST
2  --- LetRec

```

```

3  desugar (LetRec i v b) = desugar (Let [(i, App (Var "Z") [Lambda [i] v])
    ] b)

```

Código 4.10: Desazucarización de LetRec

La construcción `letrec` permite la definición de funciones recursivas locales. Dado que el núcleo del lenguaje no incluye recursión como mecanismo primitivo, esta debe expresarse mediante azúcar sintáctica utilizando construcciones ya presentes, como funciones anónimas y `let*`.

La desazucarización transforma la expresión:

$$\text{letrec } i = v \text{ in } b$$

en la siguiente forma equivalente:

$$\text{let* } [(i, (\lambda i. v) (\lambda i. v))] b$$

Este patrón corresponde a la expansión operacional del combinador de punto fijo para lenguajes con evaluación estricta (combinador *Z* mencionado anteriormente). La función $\lambda i. v$ se aplica a sí misma, permitiendo que *i* haga referencia a su propia definición dentro de *v*, sin necesidad de introducir recursión directamente en el núcleo semántico del lenguaje.

De esta manera, `letrec` se reduce a una combinación de aplicación lambda y `let*`, asegurando que el análisis y la ejecución posterior puedan realizarse de acuerdo con las reglas ya definidas para el lenguaje base. Una vez realizada la transformación, se invoca nuevamente a `desugar` para continuar el proceso hasta obtener una expresión completamente desazucarizada en el árbol AST.

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:

```

1  desugar :: ASA -> AST
2  --Expresiones lambda
3  desugar (Lambda ps b) = desugarLmb ps b
4
5  --Funcion auxiliar para desazucarar las funciones lambda
6  desugarLmb :: [String] -> ASA -> AST
7  desugarLmb [] b = desugar b
8  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 desazucararlo en aplicaciones sucesivas de **AppC**.

```

1  desugar :: ASA -> AST
2  --Expresiones lambda
3  desugar (App f as) = desugarApp (desugar f) as
4
5  --Funcion auxiliar para desazucarar las aplicaciones de funcion
6  desugarApp :: AST -> [ASA] -> AST
7  desugarApp f [] = f
8  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 desazucarar, 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 llamada recursiva para que formen las aplicaciones sucesivas de **AppC**.

Finalmente tenemos **desugar** para listas. En un principio, la idea fue implementar las listas como encadenamiento de pares, no obstante, esto nos trajo problemas al momento de implementar la función que devuelve el resultado al usuario¹, 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.

```

1  desugar :: ASA -> AST
2  --Listas
3  desugar (List l) = desugarList l
4  desugar (Head l) = HeadC (desugar l)
5  desugar (Tail l) = TailC (desugar l)
6
7  --Funcion auxiliar para construir listas como cons y nil
8  desugarList :: [ASA] -> AST
9  desugarList [] = NiL
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 convertirlos a **AST**:

```

1  {- Desazucaramos los ASA -}
2  desugar :: ASA -> AST
3  -- Casos base

```

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

```

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)
39 ---
40 --Expresiones lambda
41 desugar (Lambda ps b) = desugarLmb ps b
42 desugar (App f as) = desugarApp (desugar f) as
43 --Listas
44 desugar (List l) = desugarList l
45 desugar (Head l) = HeadC (desugar l)
46 desugar (Tail l) = TailC (desugar l)
47
48 {- Funciones auxiliares para desugar -}
49 --Funcion auxiliar para desazucarar los operadores
50 desugarOps :: (AST -> AST -> AST) -> [ASA] -> AST
51 desugarOps _ [] = error "[desugarOps Error]: Lista vacia (no deberia
    suceder)"
52 desugarOps _ [x] = desugar x
53 desugarOps op (x:xs) = op (desugar x) (desugarOps op xs)
54

```

```

55  --Funcion auxiliar para desazucarar los comparadores
56  desugarComp :: (AST -> AST -> AST) -> [ASA] -> AST
57  desugarComp _ [] = BoolC True
58  desugarComp _ [_] = BoolC True
59  desugarComp op [i, d] = op (desugar i) (desugar d)
60  desugarComp op (i:d:is) = IfC (op (desugar i) (desugar d))
61  (desugarComp op (d:is))
62  (BoolC False)
63
64  --Funcion auxiliar para desazucarar el operador cond
65  desugarCond :: [(ASA, ASA)] -> ASA -> AST
66  desugarCond [] e = desugar e
67  desugarCond ((c, t):cs) e = IfC (desugar c) (desugar t) (desugarCond cs
68  e)
69
70  --Funciones auxiliares para desazucarar let
71  desugarLet :: [(String, ASA)] -> ASA -> AST
72  desugarLet [] b = desugar b
73  desugarLet ((p, v):ps) b = AppC (Func p (desugarLet ps b)) (desugar v)
74
75  --Funcion auxiliar para desazucarar las funciones lambda
76  desugarLmb :: [String] -> ASA -> AST
77  desugarLmb [] b = desugar b
78  desugarLmb (p:ps) b = Func p (desugarLmb ps b)
79
80  --Funcion auxiliar para desazucarar las aplicaciones de funcion
81  desugarApp :: AST -> [ASA] -> AST
82  desugarApp f [] = f
83  desugarApp f (a:as) = desugarApp (AppC f (desugar a)) as
84
85  --Funcion auxiliar para construir listas como cons y nil
86  desugarList :: [ASA] -> AST
87  desugarList [] = Nil
88  desugarList [x] = desugar x
89  desugarList (x:xs) = Cons (desugar x) (desugarList xs)

```

Código 4.14: Algoritmo para función desugar en Haskell completo

Concluimos así con la implementación del proceso de desazucarado de nuestros Árboles de Sintaxis Abstracta, obteniendo como resultado el núcleo fundamental de nuestro lenguaje MINILISP . Como vimos, este procedimiento es esencial, pues nos permite simplificar la estructura del lenguaje al eliminar construcciones sintácticas superficiales y reducirlas a formas más primitivas.

Este enfoque no solo disminuye la complejidad de las reglas necesarias para el intérprete, sino que también evita operaciones redundantes, facilitando el análisis, la evaluación y el mantenimiento del lenguaje. Además, garantiza el respeto absoluto a la intención del usuario, manteniendo intacta la semántica del programa original.

En resumen, la eliminación del azúcar sintáctica constituye una fase clave en el diseño de lenguajes funcionales, ya que nos permite trabajar con una base sólida, mínima y expresiva sobre la cual construir el resto de las capacidades del lenguaje, asegurando claridad, consistencia y elegancia en su implementación.

Capítulo 5

Semántica Operacional

Una vez que hemos establecido la sintaxis del lenguaje en sus formas núcleo, el siguiente paso es proporcionar significado formal a los programas: especificar cómo deben ejecutarse y cuál es el resultado de cada construcción del lenguaje. Para ello recurrimos a la *semántica operacional*, un enfoque formal que describe el comportamiento de los programas mediante reglas que modelan su ejecución paso a paso o de manera global.

La formalización de la semántica permite razonar rigurosamente sobre programas, verificar propiedades (por ejemplo corrección y seguridad), y relacionar la especificación formal con una implementación (intérprete o compilador). Existen distintos enfoques para dar semántica a un lenguaje (operacional, denotacional y axiomática). En este trabajo nos centraremos en la semántica operacional y, concretamente, en el estilo *small-step* (semántica estructural), que es el más conveniente para MINILISP por su claridad al exponer transformaciones intermedias.

5.1. ¿Qué es la semántica operacional?

La semántica operacional especifica el significado de los programas describiendo cómo estos evolucionan durante su ejecución. En lugar de asignar directamente un valor matemático al programa, se define un sistema de reglas de inferencia que muestran cómo un programa avanza desde un estado inicial hasta un estado final (o intermedio). Cada regla representa un paso válido de evaluación en el lenguaje. Este enfoque permite expresar la dinámica del programa de manera clara, estructural y verificable, lo que resulta especialmente útil para lenguajes funcionales como MINILISP.

Dos estilos comunes son:

- **Semántica Natural (Big-step)** La *semántica natural*, también llamada *big-step*, evalúa expresiones completas en una sola derivación, directamente desde el programa inicial hasta el valor final. Es decir, describe qué resultado obtiene un programa, sin enfocarse en cada uno de los pasos intermedios.

Fue introducida por Gordon Plotkin en 1981 como parte de su trabajo fundacional en semántica estructural. Describe la evaluación completa de los programas. [5]

- **Semántica Estructural (Small-step)** La *semántica estructural*, también conocida

como *small-step*, especifica la evaluación como una secuencia de pasos pequeños, donde cada regla transforma una expresión en otra más simple hasta llegar a un valor final.

Se abordará a detalle su historia mas adelante.

Elección del enfoque para MiniLisp Para nuestro lenguaje MINILISP utilizaremos la semántica operacional de estilo *small-step*, ya que este enfoque nos permite observar y controlar cada transformación intermedia del programa, lo cual es ideal para entender con precisión el proceso de evaluación y para implementar un intérprete paso a paso.

5.2. Semántica Estructural (Paso pequeño)

La **semántica estructural de paso pequeño** (*small-step operational semantics*) describe la ejecución de los programas como una secuencia de transiciones elementales. En este enfoque, el significado de un programa no se define únicamente por su resultado final, sino por la sucesión de configuraciones que atraviesa durante su evaluación. Cada paso representa una transformación mínima y formalmente justificada sobre una expresión, lo que permite capturar con precisión el proceso computacional.

Formalmente, esta semántica define una relación de transición:

$$e \longrightarrow e'$$

que indica que la expresión e evoluciona en un solo paso a la expresión e' . La relación de evaluación completa se obtiene mediante la clausura reflexivo-transitiva:

$$e \longrightarrow^* v$$

donde v es un valor.

Este enfoque fue sistematizado por Gordon Plotkin como parte del marco **Structural Operational Semantics (SOS)** [5], el cual enfatiza la correspondencia directa entre la estructura sintáctica del lenguaje y sus reglas de evaluación. La semántica de paso pequeño resulta especialmente útil en el estudio de lenguajes funcionales como MINILISP, donde el orden de evaluación, la sustitución y la estructura de funciones juegan un papel esencial en la ejecución.

Motivos para emplear la semántica de paso pequeño

Optar por este estilo semántico ofrece varias ventajas:

- **Claridad sobre el proceso de ejecución:** permite observar cada paso de la evaluación, en contraste con la semántica de *big-step*, que solo muestra el estado inicial y el resultado final.
- **Modelo fiel para intérpretes:** las reglas se traducen casi directamente en algoritmos, facilitando la implementación de un intérprete paso a paso.
- **Razonamiento formal detallado:** es más adecuada para estudiar propiedades como determinismo, progresión, preservación de tipos, y analizar comportamientos intermedios.

- **Adecuada para depuración y evaluación interactiva:** permite observar el flujo de ejecución e inspeccionar expresiones intermedias, lo cual es crucial para sistemas educativos e intérpretes interactivos.

En el contexto de MINILISP, la semántica de paso pequeño nos permite capturar explícitamente cómo las expresiones se reducen, aplicando reglas como la reducción β , la evaluación de condicionales, y la expansión de funciones recursivas. Esta formalización establece la base teórica para la implementación del intérprete desarrollado en este proyecto.

5.2.1. Sistema de transición

Para modelar formalmente la ejecución de nuestro lenguaje, utilizamos un **sistema de transición**. Este marco matemático permite describir cómo un programa evoluciona paso a paso mediante la aplicación de reglas de reducción.

Un **sistema de transición** es una tupla (E, \rightarrow, I, F) donde:

- E es el conjunto (posiblemente infinito) de **estados** del sistema.
- $\rightarrow \subseteq E \times E$ es la **relación de transición**, que indica cómo un estado puede transformarse en otro.
- $I \subseteq E$ es el conjunto de **estados iniciales**, desde los cuales comienza la ejecución.
- $F \subseteq E$ es el conjunto de **estados finales**, es decir, aquellos estados donde la ejecución se considera terminada.

La ejecución de un programa se modela entonces como una secuencia de transiciones:

$$e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$$

donde $e_0 \in I$, cada transición $e_i \rightarrow e_{i+1}$ está justificada por una regla de evaluación, y $e_n \in F$ representa un estado final.

En el contexto de MINILISP, usaremos este formalismo para describir cómo las expresiones del lenguaje se transforman gradualmente hasta producir un valor o alcanzar una condición de término. Los **estados finales** corresponderán a valores que ya no pueden seguir reduciéndose, tales como números, valores booleanos, funciones evaluadas y otras construcciones que definiremos formalmente más adelante.

Sobre la noción de cerradura (closure)

Un elemento fundamental en la semántica de lenguajes funcionales es la **cerradura** o *closure*. Una closure es una representación formal de una función junto con el entorno léxico en el cual fue definida. Esto significa que, además del cuerpo de la función, se almacena la información necesaria para interpretar correctamente sus variables libres.

$$\text{closure} = \langle \lambda x. e, \rho \rangle$$

donde:

- $\lambda x. e$ es una expresión lambda, es decir, la definición de la función.
- ρ es un **ambiente léxico** que mapea variables a valores.

Las closures permiten implementar correctamente *scoping léxico*, asegurando que las funciones mantengan acceso a las variables del contexto donde fueron creadas, aun cuando se utilicen fuera de dicho contexto. Aunque mencionamos este concepto aquí, su definición formal y su rol dentro de los estados finales se precisará en secciones posteriores.

5.2.2. Estados Finales en MINILISP

Definimos los estados finales para el lenguaje MINILISP -0.2cm:

$$F = \{ Num_V(n) \mid n \in \mathbb{Z} \} \cup \{ Boolean_V(b) \mid b \in \{True, False\} \} \cup \{ Pair_V(f, s) \mid f, s \in F \} \cup \{ Cons_V(h, t) \mid h, t \in C \} \cup \{ Nil_V \mid \text{es la lista vacía} \} \cup \{ Closure(f, \varepsilon) \mid f \text{ es una función y } \varepsilon \text{ es un ambiente léxico} \}$$

Intuitivamente implementamos los estados finales en nuestro lenguaje como un tipo de dato en Haskell:

```

1 module ASV where
2
3 -- ASA Values
4 data ASV
5   = VarV String
6   | NumV Int
7   | BoolV Bool
8   | NiV
9   | PairV ASV ASV
10  | ConV ASV ASV
11  | Closure String ASV [(String, ASV)]
12  deriving (Show, Eq)

```

Código 5.1: Tipo de dato ASV, representan los estados finales

El tipo de dato **ASV** serán con el que modelaremos los estados finales del lenguaje. Sin embargo, en nuestra implementación, es necesario agregar las demás estructuras que trabajan con estos estados finales -como lo son los operadores aritméticos o las operaciones sobre pares o listas- ya que **ASV** sigue modelando una estructura de tipo árbol la cual sigue el mismo principio que las estructuras **ASA**:

*Una expresión es **ASV** si y solo si sus hijos también son **ASV**.*

Por lo que debemos hacer que nuestros Para diferenciar entre los verdaderos estados finales y las demás estructuras de tipo **ASV** será que nos referimos a los finales como **valores canónicos**.¹ Lo que nos queda en la nueva definición del tipo de dato **ASV**:

¹Más adelante enfatizaremos en como los diferenciamos entre Valores **ASV** y expresiones **ASV** en Haskell.

```

1  -- ASA Values
2  data ASV
3    = VarV String
4    | NumV Int
5    | BoolV Bool
6    | NiV
7    | AddV ASV ASV
8    | SubV ASV ASV
9    | MulV ASV ASV
10   | Div ASV ASV
11   | SqrtV ASV
12   | NotV ASV
13   | EqualV ASV ASV
14   | LessV ASV ASV
15   | GreaterV ASV ASV
16   | DiffV ASV ASV
17   | LeqV ASV ASV
18   | GeqV ASV ASV
19   | PairV ASV ASV
20   | FstV ASV
21   | SndV ASV
22   | IfV ASV ASV ASV
23   | FunV String ASV
24   | AppV ASV ASV
25   | ConV ASV ASV
26   | HeadV ASV
27   | TailV ASV
28   | Closure String ASV [(String, ASV)]
29   deriving (Show, Eq)

```

Código 5.2: Tipo de dato ASV completo

Ya que hemos definido el tipo de dato ASV veamos como convertimos nuestras estructuras AST a estados finales ASV:

```

1  {- Convertimos los AST a estados finales ASV -}
2  toFinalState :: AST -> ASV
3  toFinalState (VarC x) = VarV x
4  toFinalState (NumC n) = NumV n
5  toFinalState (BoolC b) = BoolV b
6  toFinalState (AddC i d) = AddV (toFinalState i) (toFinalState d)
7  toFinalState (SubC i d) = SubV (toFinalState i) (toFinalState d)
8  toFinalState (MulC i d) = MulV (toFinalState i) (toFinalState d)
9  toFinalState (DivC i d) = DivV (toFinalState i) (toFinalState d)
10 toFinalState (SqrtC n) = SqrtV (toFinalState n)
11 toFinalState (NotC x) = NotV (toFinalState x)
12 toFinalState (EqualC i d) = EqualV (toFinalState i) (toFinalState d)
13 toFinalState (LessC i d) = LessV (toFinalState i) (toFinalState d)
14 toFinalState (GreaterC i d) = GreaterV (toFinalState i) (toFinalState d)
15 toFinalState (DiffC i d) = DiffV (toFinalState i) (toFinalState d)
16 toFinalState (LeqC i d) = LeqV (toFinalState i) (toFinalState d)
17 toFinalState (GeqC i d) = GeqV (toFinalState i) (toFinalState d)
18 toFinalState (PairC f s) = PairV (toFinalState f) (toFinalState s)
19 toFinalState (FstC p) = FstV (toFinalState p)

```

```

20  toFinalState (SndC p) = SndV (toFinalState p)
21  toFinalState (ConS f s) = ConV (toFinalState f) (toFinalState s)
22  toFinalState (HeadC p) = HeadV (toFinalState p)
23  toFinalState (TailC p) = TailV (toFinalState p)
24  toFinalState (IfC c t e) = IfV (toFinalState c) (toFinalState t) (
    toFinalState e)
25  toFinalState (Func p b) = FunV p (toFinalState b)
26  toFinalState (AppC f a) = AppV (toFinalState f) (toFinalState a)
27  toFinalState NiL = NiV

```

Código 5.3: Función `toFinalState` que transforma los núcleos AST a estados finales ASV

La función `toFinalState` transforma cada estructura del núcleo AST en su equivalente ASV. Aunque a primera vista parezca una simple correspondencia entre constructores, su propósito es fundamental: garantizar que todas las expresiones que se evalúan dentro del intérprete sean expresadas en términos de ASV, permitiendo así que la semántica del lenguaje opere únicamente sobre estructuras homogéneas y compatibles con los valores finales del lenguaje.

5.2.3. Reglas de evaluación

Las reglas de evaluación constituyen el núcleo de la semántica operacional estructural. Su propósito es describir formalmente cómo una expresión del lenguaje progresa paso a paso hasta convertirse en un valor. Cada regla establece una relación de transición entre estados del programa, donde un estado suele estar compuesto por una expresión y un ambiente (o entorno) de ejecución.

Estas reglas son necesarias porque proporcionan una base rigurosa para entender, implementar y razonar formalmente sobre el comportamiento de un lenguaje de programación. Al definir el significado de un programa mediante reglas inductivas, aseguramos que cada paso de cómputo esté claramente especificado y sea verificable. Además, permiten demostrar propiedades importantes como corrección, determinismo, o progresión de los programas.

Un componente fundamental en esta formalización es el ambiente (ε), que modela el contexto donde se evalúan las expresiones. El ambiente es una estructura que asocia identificadores del lenguaje con valores, y se actualiza o consulta durante la evaluación. En esencia, actúa como una memoria que mantiene las asignaciones vigentes en cada punto del cómputo. Las reglas operan entonces sobre pares $\langle \text{expresión}, \varepsilon \rangle$, indicando cómo se transforma la expresión manteniendo o modificando dicho ambiente.

- Expresiones atómicas:

- $\text{VarV}(i)$:

$$\frac{\text{lookup } i \ \varepsilon = v}{\langle \text{VarV}(i), \varepsilon \rangle \rightarrow \langle v, \varepsilon \rangle}$$

$$\frac{\text{lookup } i \ \varepsilon \text{ no está definido}}{\langle \text{VarV}(i), \varepsilon \rangle \rightarrow \text{Error en la ejecución de la evaluación}}$$

El operador `lookup` busca el valor asociado a un identificador dentro del ambiente ε . Si el identificador se encuentra definido, `lookup` devuelve su valor; en caso

contrario, la evaluación produce un error, ya que se intenta acceder a una variable no declarada o fuera de su alcance. De esta manera, `lookup` garantiza que toda referencia a variables se resuelva correctamente según el entorno léxico vigente.

- NumV(n):

$$\overline{\langle NumV(n), \varepsilon \rangle \rightarrow \langle NumV(n), \varepsilon \rangle}$$

- BoolV(b):

$$\overline{\langle BoolV(b), \varepsilon \rangle \rightarrow \langle BoolV(b), \varepsilon \rangle}$$

- NiV:

$$\overline{\langle NivV, \varepsilon \rangle \rightarrow \langle NiV, \varepsilon \rangle}$$

- AddV(i,d):

$$\begin{array}{c} \langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle \\ \hline \langle AddV(i, d), \varepsilon \rangle \rightarrow \langle AddV(i', d), \varepsilon \rangle \\ \langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle \\ \hline \langle AddV(NumV(n), d), \varepsilon \rangle \rightarrow \langle Add(NumV(n), d'), \varepsilon \rangle \\ \hline \langle AddV(NumV(n), NumV(m)), \varepsilon \rangle \rightarrow \langle NumV(n +_{\mathbb{Z}} m), \varepsilon \rangle \end{array}$$

- SubV(i,d):

$$\begin{array}{c} \langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle \\ \hline \langle SubV(i, d), \varepsilon \rangle \rightarrow \langle SubV(i', d), \varepsilon \rangle \\ \langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle \\ \hline \langle SubV(NumV(n), d), \varepsilon \rangle \rightarrow \langle SubV(NumV(n), d'), \varepsilon \rangle \\ \hline \langle SubV(NumV(n), NumV(m)), \varepsilon \rangle \rightarrow \langle NumV(n -_{\mathbb{Z}} m), \varepsilon \rangle \end{array}$$

- MulV(i,d):

$$\begin{array}{c} \langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle \\ \hline \langle MulV(i, d), \varepsilon \rangle \rightarrow \langle MulV(i', d), \varepsilon \rangle \\ \langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle \\ \hline \langle MulV(NumV(n), d), \varepsilon \rangle \rightarrow \langle MulV(NumV(n), d'), \varepsilon \rangle \\ \hline \langle MulV(NumV(n), NumV(m)), \varepsilon \rangle \rightarrow \langle NumV(n *_{\mathbb{Z}} m), \varepsilon \rangle \end{array}$$

- DiV(i,d):

$$\begin{array}{c} \langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle \\ \hline \langle DiV(i, d), \varepsilon \rangle \rightarrow \langle DiV(i', d), \varepsilon \rangle \\ \langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle \\ \hline \langle DiV(NumV(n), d), \varepsilon \rangle \rightarrow \langle DiV(NumV(n), d'), \varepsilon \rangle \\ \hline \langle DiV(NumV(n), NumV(0)), \varepsilon \rangle \rightarrow \text{Error en la ejecución de la evaluación} \\ \hline \langle DiV(NumV(n), NumV(m)), \varepsilon \rangle \rightarrow \langle NumV(n /_{\mathbb{Z}} m), \varepsilon \rangle \quad (m \neq 0) \end{array}$$

- SqrtV(n):

$$\begin{array}{c}
\frac{\langle n, \varepsilon \rangle \rightarrow \langle n', \varepsilon \rangle}{\langle \text{SqrtV}(n), \varepsilon \rangle \rightarrow \langle \text{SqrtV}(n'), \varepsilon \rangle} \\
\frac{n < 0}{\langle \text{SqrtV}(\text{NumV}(n)), \varepsilon \rangle \rightarrow \text{Error en la ejecución de la evaluación}} \\
\frac{}{\langle \text{SqrtV}(\text{NumV}(n)), \varepsilon \rangle \rightarrow \langle \text{NumV}(\sqrt{n_{\mathbb{N}}}), \varepsilon \rangle \quad (n \geq 0)}
\end{array}$$

- NotV(b):

$$\begin{array}{c}
\frac{\langle b, \varepsilon \rangle \rightarrow \langle n', \varepsilon \rangle}{\langle \text{NotV}(b), \varepsilon \rangle \rightarrow \langle \text{NotV}(b'), \varepsilon \rangle} \\
\frac{}{\langle \text{NotV}(\text{BoolV}(b)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(\neg_{\mathbb{P}} b), \varepsilon \rangle}
\end{array}$$

- EqualV(i,d):

$$\begin{array}{c}
\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle \text{EqualV}(i, d), \varepsilon \rangle \rightarrow \langle \text{EqualV}(i', d), \varepsilon \rangle} \\
\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle \text{EqualV}(\text{NumV}(n), d), \varepsilon \rangle \rightarrow \langle \text{EqualV}(\text{NumV}(n), d'), \varepsilon \rangle} \\
\frac{}{\langle \text{EqualV}(\text{NumV}(n), \text{NumV}(m)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n =_{\mathbb{Z}} m), \varepsilon \rangle}
\end{array}$$

- LessV(i,d)

$$\begin{array}{c}
\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle \text{LessV}(i, d), \varepsilon \rangle \rightarrow \langle \text{LessV}(i', d), \varepsilon \rangle} \\
\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle \text{LessV}(\text{NumV}(n), d), \varepsilon \rangle \rightarrow \langle \text{LessV}(\text{NumV}(n), d'), \varepsilon \rangle} \\
\frac{}{\langle \text{LessV}(\text{NumV}(n), \text{NumV}(m)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n <_{\mathbb{Z}} m), \varepsilon \rangle}
\end{array}$$

- GreaterV(i,d)

$$\begin{array}{c}
\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle \text{GreaterV}(i, d), \varepsilon \rangle \rightarrow \langle \text{GreaterV}(i', d), \varepsilon \rangle} \\
\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle \text{GreaterV}(\text{NumV}(n), d), \varepsilon \rangle \rightarrow \langle \text{GreaterV}(\text{NumV}(n), d'), \varepsilon \rangle} \\
\frac{}{\langle \text{GreaterV}(\text{NumV}(n), \text{NumV}(m)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n >_{\mathbb{Z}} m), \varepsilon \rangle}
\end{array}$$

- DiffV(i,d)

$$\begin{array}{c}
\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle \text{DiffV}(i, d), \varepsilon \rangle \rightarrow \langle \text{DiffV}(i', d), \varepsilon \rangle} \\
\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle \text{DiffV}(\text{NumV}(n), d), \varepsilon \rangle \rightarrow \langle \text{DiffV}(\text{NumV}(n), d'), \varepsilon \rangle} \\
\frac{}{\langle \text{DiffV}(\text{NumV}(n), \text{NumV}(m)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n \neq_{\mathbb{Z}} m), \varepsilon \rangle}
\end{array}$$

- $\text{LeqV}(i, d)$

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle \text{LeqV}(i, d), \varepsilon \rangle \rightarrow \langle \text{LeqV}(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle \text{LeqV}(\text{NumV}(n), d), \varepsilon \rangle \rightarrow \langle \text{LeqV}(\text{NumV}(n), d'), \varepsilon \rangle}$$

$$\frac{}{\langle \text{LeqV}(\text{NumV}(n), \text{NumV}(m)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n \leq_{\mathbb{Z}} m), \varepsilon \rangle}$$

- $\text{GeqV}(i, d)$

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle \text{GeqV}(i, d), \varepsilon \rangle \rightarrow \langle \text{GeqV}(i', d), \varepsilon \rangle}$$

$$\frac{\langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle \text{GeqV}(\text{NumV}(n), d), \varepsilon \rangle \rightarrow \langle \text{GeqV}(\text{NumV}(n), d'), \varepsilon \rangle}$$

$$\frac{}{\langle \text{GeqV}(\text{NumV}(n), \text{NumV}(m)), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n \geq_{\mathbb{Z}} m), \varepsilon \rangle}$$

Como se puede ver en las reglas para los comparadores, la regla final cuando ambas expresiones del comparador son el resultado deriva en

- $\text{PairV}(f, s)$:

$$\frac{\langle f, \varepsilon \rangle \rightarrow \langle f', \varepsilon \rangle}{\langle \text{PairV}(f, s), \varepsilon \rangle \rightarrow \langle \text{PairV}(f', s), \varepsilon \rangle}$$

$$\frac{\langle s, \varepsilon \rangle \rightarrow \langle s', \varepsilon \rangle}{\langle \text{PairV}(v_f, s), \varepsilon \rangle \rightarrow \langle \text{PairV}(v_f, s'), \varepsilon \rangle}$$

$$\frac{v_f, v_s \text{ son valores canónicos}}{\langle \text{PairV}(v_f, v_s), \varepsilon \rangle \rightarrow \langle \text{PairV}(v_f, v_s), \varepsilon \rangle}$$

- $\text{FstV}(p)$:

$$\frac{\langle p, \varepsilon \rangle \rightarrow \langle p', \varepsilon \rangle}{\langle \text{FstV}(p), \varepsilon \rangle \rightarrow \langle \text{FstV}(p'), \varepsilon \rangle}$$

$$\frac{v_1, v_2 \text{ son valores canónicos}}{\langle \text{FstV}(\text{PairV}(v_1, v_2)), \varepsilon \rangle \rightarrow \langle v_1, \varepsilon \rangle}$$

- $\text{SndV}(p)$:

$$\frac{\langle p, \varepsilon \rangle \rightarrow \langle p', \varepsilon \rangle}{\langle \text{SndV}(p), \varepsilon \rangle \rightarrow \langle \text{SndV}(p'), \varepsilon \rangle}$$

$$\frac{v_1, v_2 \text{ son valores canónicos}}{\langle \text{SndV}(\text{PairV}(v_1, v_2)), \varepsilon \rangle \rightarrow \langle v_2, \varepsilon \rangle}$$

- $\text{IfV}(c, t, e)$, IfV solo evalúa la condicional hasta llegar a un **BoolV** mas no evalúa el **then** o **else** dependiendo del resultado de la condición, solo se encarga de retornar alguno de los dos dependiendo del caso:

$$\frac{\langle c, \varepsilon \rangle \rightarrow \langle c', \varepsilon \rangle}{\langle \text{IfV}(c, t, e), \varepsilon \rangle \rightarrow \langle \text{IfV}(c', t, e), \varepsilon \rangle}$$

$$\frac{}{\langle IfV(BoolV(\#t), t, e), \varepsilon \rangle \rightarrow \langle t, \varepsilon \rangle}$$

$$\frac{}{\langle IfV(BoolV(\#f), t, e), \varepsilon \rangle \rightarrow \langle e, \varepsilon \rangle}$$

- ConV(i,d):

$$\frac{\langle i, \varepsilon \rangle \rightarrow \langle i', \varepsilon \rangle}{\langle ConV(i, d), \varepsilon \rangle \rightarrow \langle ConV(i', d), \varepsilon \rangle}$$

$$\frac{v_i \text{ es valor canónico } \langle d, \varepsilon \rangle \rightarrow \langle d', \varepsilon \rangle}{\langle ConV(v_i, d), \varepsilon \rangle \rightarrow \langle ConV(v_i, d'), \varepsilon \rangle}$$

$$\frac{v_i, v_d \text{ son valores canónicos}}{\langle ConV(v_i, v_d), \varepsilon \rangle \rightarrow \langle ConV(v_i, v_d), \varepsilon \rangle}$$

- HeadV(l):

$$\frac{\langle p, \varepsilon \rangle \rightarrow \langle p', \varepsilon \rangle}{\langle HeadV(l), \varepsilon \rangle \rightarrow \langle HeadV(l'), \varepsilon \rangle}$$

$$\frac{}{\langle HeadV(ConV(v_i, v_d)), \varepsilon \rangle \rightarrow \langle v_i, \varepsilon \rangle}$$

- TailV(l):

$$\frac{\langle l, \varepsilon \rangle \rightarrow \langle l', \varepsilon \rangle}{\langle TailV(l), \varepsilon \rangle \rightarrow \langle TailV(l'), \varepsilon \rangle}$$

$$\frac{v_d \text{ es valor pero } v_d = ConV(i, d) \quad v_d \rightarrow v'_d}{\langle TailV(ConV(v_i, v_d)), \varepsilon \rangle \rightarrow \langle TailV(v_i, v'_d), \varepsilon \rangle}$$

$$\frac{v_d \text{ es valor pero } v_d \neq ConV(i, d)}{\langle TailV(ConV(v_i, v_d)), \varepsilon \rangle \rightarrow \langle v_d, \varepsilon \rangle}$$

- FunV(p,c):

$$\frac{}{\langle FunV(p, c), \varepsilon \rangle \rightarrow \langle Closure(FunV(p, c), \varepsilon), \varepsilon \rangle}$$

- AppV(f,a):

$$\frac{\langle f, \varepsilon \rangle \rightarrow \langle f', \varepsilon \rangle}{\langle AppV(f, a), \varepsilon \rangle \rightarrow \langle AppV(f', a), \varepsilon \rangle}$$

$$\frac{\langle a, \varepsilon \rangle \rightarrow \langle a', \varepsilon \rangle}{\langle AppV(Closure(FunV(p, c), \varepsilon'), a), \varepsilon \rangle \rightarrow \langle AppV(Closure(FunV(p, c), \varepsilon'), a'), \varepsilon \rangle}$$

$$\frac{\text{es un valor canónico}}{\langle AppV(Closure(FunV(p, c), \varepsilon'),), \varepsilon \rangle \rightarrow \langle c, \varepsilon' [p \leftarrow] \rangle}$$

Conclusión

En resumen, las reglas de evaluación proporcionan el marco formal necesario para describir con precisión el proceso de ejecución de los programas dentro de un lenguaje. A través de estas reglas se establece, de manera inductiva, cómo las expresiones evolucionan paso a paso bajo un ambiente de evaluación, garantizando así un comportamiento bien definido y verificable. Este enfoque no solo facilita la implementación y el razonamiento formal del lenguaje, sino que también permite identificar errores semánticos de forma rigurosa, asegurando que cada etapa del cómputo sea consistente con la semántica especificada.

5.3. Intérprete para MINILISP

Una vez definida formalmente la Semántica Operacional Estructural para nuestro lenguaje, estamos en posición de implementar un intérprete que materialice dicha semántica en código. El propósito del intérprete es ejecutar programas escritos en nuestro MINILISP, aplicando las reglas de evaluación paso a paso hasta obtener un valor final o detectar un error semántico.

En otras palabras, el intérprete funge como puente entre la especificación teórica y el comportamiento observable del lenguaje: toma las expresiones del usuario, las transforma a la sintaxis abstracta sin azúcar sintáctica (ASV), y posteriormente dirige el proceso de evaluación conforme a las reglas formales previamente establecidas.

El intérprete opera manteniendo un ambiente de ejecución (ε) y evaluando cada expresión de acuerdo con la relación de transición definida en el sistema de semántica estructural. Cada paso de evaluación corresponde directamente a una instancia de alguna de nuestras reglas inferenciales, lo cual garantiza que la implementación permanezca fiel al modelo teórico.

De manera general, nuestro intérprete debe:

- Recibir una expresión en sintaxis concreta y traducirla a su representación en el ASA y posteriormente ASV.
- Inicializar un ambiente de ejecución vacío o con valores predefinidos.
- Aplicar las reglas de evaluación de manera iterativa (paso a paso), hasta alcanzar un valor final (estado final), una expresión bloqueada, o un error.
- Gestionar ambientes y cierres (*closures*) para implementar alcance léxico y funciones como valores de primera clase.
- Reportar resultados o errores de evaluación al usuario de forma clara.

Es importante destacar que, dado que nuestra semántica es de paso pequeño (*small-step*), el intérprete refleja explícitamente la evolución del programa a través de múltiples transiciones, lo cual facilita tanto la verificación formal como la depuración y el entendimiento del proceso de ejecución.

Con este componente, obtenemos un sistema completo: comenzamos con la especificación formal, definimos su semántica, y finalmente construimos un ejecutor funcional que sigue rigurosamente dichas reglas. Esto garantiza que el lenguaje MINILISP no solo está formalmente fundamentado, sino que también es operacionalmente realizable.

5.3.1. Función paso pequeño en MINILISP

Para nuestro lenguaje definimos la función de evaluación como sigue:

```

1 module ASV where
2
3 import AST
4
5 {--
```

```

6  Definimos la representacion del ambiente de ejecucion.
7  Un ambiente es una lista de pares (id, valor).
8  --}
9  type Env = [(String, ASV)]
10
11  {- - ASA Values - -}
12  data ASV
13    = VarV String
14    | NumV Int
15    | BoolV Bool
16    | NiV
17    | AddV ASV ASV
18    | SubV ASV ASV
19    | MulV ASV ASV
20    | DiV ASV ASV
21    | SqrtV ASV
22    | NotV ASV
23    | EqualV ASV ASV
24    | LessV ASV ASV
25    | GreaterV ASV ASV
26    | DiffV ASV ASV
27    | LeqV ASV ASV
28    | GeqV ASV ASV
29    | PairV ASV ASV
30    | FstV ASV
31    | SndV ASV
32    | IfV ASV ASV ASV
33    | FunV String ASV
34    | AppV ASV ASV
35    | ConV ASV ASV
36    | HeadV ASV
37    | TailV ASV
38    | Closure String ASV Env
39  deriving (Show, Eq)

```

Código 5.4: Tipo de dato ASV (Estados Finales) y ENV (ambiente de evaluación)

Definimos tanto los estados finales como un nuevo tipo llamado `Env`, que representa el ambiente de evaluación. Este ambiente no es más que una emulación de la pila donde guardamos los ambientes como pares ((id, valor) como si de `["id" → value]`), donde cada identificador está asociado con el valor que tiene en ese momento de la ejecución.

Ahora definimos la función principal `eval`, para el intérprete del lenguaje:

```

1  module Interprete where
2
3  import ASV
4
5  {- - Funcion principal del interprete - -}
6  {- -
7  Evaluamos una expresion paso a paso utilizando la funcion 'pasito' hasta
8  llegar a algun valor canonico.
9  --}
10 eval :: ASV -> Env -> ASV

```

```

10 eval asv env
11   | isValue asv = asv
12   | otherwise =
13     let (asv', env') = pasito asv env
14     in eval asv' env'
15
16 {-- Funcion que determina si una expresion es valor canonico --}
17 isValue :: ASV -> Bool
18 isValue (NumV _) = True
19 isValue (BoolV _) = True
20 isValue (Closure _ _ _) = True
21 isValue (PairV f s) = isValue f && isValue s
22 isValue (ConV f s) = isValue f && isValue s
23 isValue (NiV) = True
24 isValue _ = False

```

Código 5.5: Estados Finales

La función `eval` recibe un estado final con el ambiente de inicio el cuál es vacío al comienzo de la evaluación y devuelve un ASV que se al terminar será un valor canónico.

Utilizamos guardas con los que describimos que hacer. Si la expresión ya es un valor canónico (según `isValue`), no hay nada que evaluar y devolvemos el valor tal cual. En otro caso procedemos a evaluar la expresión paso a paso.

Con la función auxiliar `isValue` utilizando la casa de patrones definimos cuales estados son valores canónicos que ya establecimos y, en caso de serlo, devolvemos `True`, en otro caso no es canónico y devolvemos `False`.

Modelamos las reglas de evaluación con paso pequeño de MINILISP -0.2cm con la función `pasito`. Esta función `pasito` recibe el estado final con su ambiente a evaluar y regresa la configuración resultante:

```

1 {-- Funcion pasito que implementa la semantica operacional estructural
  del lenguaje --}
2 {--
3 'Avanza' un solo paso en la evaluacion de una expresion, devolviendo el
  resultado y el ambiente actualizado.
4 --}
5 pasito :: ASV -> Env -> (ASV, Env)
6 --Valores
7 pasito (VarV i) env = (mirarriba i env, env)
8 pasito (NumV n) env = (NumV n, env)
9 pasito (BoolV b) env = (BoolV b, env)
10 pasito (NiV) env = (NiV, env)
11
12
13 {-- Funcion mirarriba (lookup) para la busqueda de variables en el
  ambiente --}
14 mirarriba :: String -> Env -> ASV
15 mirarriba i [] = error ("Var '" ++ i ++ "' no definida")
16 mirarriba i ((j, v):e)
17   | i == j = v
18   | otherwise = mirarriba i e

```

Código 5.6: Función `pasito` para valores, casos base

Como se puede ver, `pasito` utiliza la casa de patrones de acuerdo con las reglas que establecimos previamente. `NumV n`, `BoolV b` y `NiV` solo devuelven la misma expresión con el mismo ambiente recibido.

Tenemos la mención especial de que para las variables `VarV` utilizamos `lookup`, que en nuestra implementación, con un arrebato increíble de originalidad lo nombramos como `mirarriba`; para buscar las varibales en el ambiente dado. La función implementa la búsqueda de una variable en el ambiente, toma el nombre de una variable y un ambiente `Env` y devuelve el `ASV` asociado.

Si el ambiente es vacío o llegamos al final de este y no encontramos la variable, lanzamos un error con un mensaje indicando que la variable no está definida, deteniendno la ejecución de todo el intérprete. En el caso recursivo, comprobamos que el identificador en el ambiente sea el mismo que el ambiente que se está buscando, en caso de que coincidan devolvemos el valor `v` asociado ya que hemos encontrado la variable. En otro caso, si no coinciden, continuamos la búsqueda recursivamente en la cola. Recorremos el ambiente hasta encontrar la variable o agotar la "pila".

```

1 pasito :: ASV -> Env -> (ASV, Env)
2 -- Operadores
3 pasito (AddV (NumV n) (NumV m)) env = (NumV (n + m), env)
4 pasito (AddV (NumV n) d) env       = let (d', env') = pasito d env
5                                     in (AddV (NumV n) d', env')
6 pasito (AddV i d) env              = let (i', env') = pasito i env
7                                     in (AddV i' d, env')
8 pasito (SubV (NumV n) (NumV m)) env = (NumV (n - m), env)
9 pasito (SubV (NumV n) d) env       = let (d', env') = pasito d env
10                                    in (SubV (NumV n) d', env')
11 pasito (SubV i d) env              = let (i', env') = pasito i env
12                                    in (SubV i' d, env')
13 pasito (MulV (NumV n) (NumV m)) env = (NumV (n * m), env)
14 pasito (MulV (NumV n) d) env       = let (d', env') = pasito d env
15                                    in (MulV (NumV n) d', env')
16 pasito (MulV i d) env              = let (i', env') = pasito i env
17                                    in (MulV i' d, env')
18 pasito (DivV (NumV n) (NumV m)) env
19   | m == 0 = error ("No se puede dividir entre 0")
20   | otherwise = (NumV (div n m), env)
21 pasito (DivV (NumV n) d) env = let (d', env') = pasito d env
22                               in (DivV (NumV n) d', env')
23 pasito (DivV i d) env       = let (i', env') = pasito i env
24                               in (DivV i' d, env')
25 pasito (SqrtV (NumV n)) env
26   | n < 0 = error ("No se puede obtener la raiz de un numero negativo")
27   | otherwise = (NumV (integerSquareRoot n), env)
28 pasito (SqrtV n) env       = let (n', env') = pasito n env
29                               in (SqrtV n', env')
30
31 {- Funcion auxiliar para SqrtV para calcular la raiz cuadrada de un

```

```

32 numero entero --}
integerSquareRoot :: Int -> Int
33 integerSquareRoot n = floor (sqrt (fromIntegral n :: Double))

```

Código 5.7: Función `pasito` para operadores

Para las siguientes reglas de evaluación definiremos primero la tercer regla de cada expresión donde los argumentos de estas ya son valores, ya que al usar *pattern matching* debemos implementarlo de este modo pues de otra forma caeríamos siempre en la primer regla donde solo evaluamos un argumento, lo que derivaría en un bucle infinito. Así, la regla para los operadores utiliza las reglas donde ambos argumentos son ya números para terminar la evaluación, donde usamos los operadores del lenguaje anfitrión respectivos a la expresión.

Los casos especiales son para la división y la raíz cuadrada, donde, una vez llegamos a un `NumV`, comprobamos que no sea cero, para la división, o que no sea negativo, para la raíz. En el caso de `SqrtV`, una vez comprobamos que el número es válido, nos apoyamos de la función auxiliar `integerSquareRoot` que convierte el entero a `Double`, calcula la raíz cuadrada y la trunca hacia abajo, devolviendo un `Int`.

De manera similar implementamos las reglas para los comparadores:

```

1 pasito :: ASV -> Env -> (ASV, Env)
2 --Not
3 pasito (NotV (BoolV b)) env = (BoolV (not b), env)
4 pasito (NotV e) env        = let (e', env') = pasito e env
5                             in (NotV e', env')
6 --Comparadores
7 pasito (EqualV (NumV n) (NumV m)) env = (BoolV (n == m), env)
8 pasito (EqualV (NumV n) d) env        = let (d', env') = pasito d env
9                                         in (EqualV (NumV n) d', env')
10 pasito (EqualV i d) env               = let (i', env') = pasito i env
11                                         in (EqualV i' d, env')
12 pasito (LessV (NumV n) (NumV m)) env = (BoolV (n < m), env)
13 pasito (LessV (NumV n) d) env        = let (d', env') = pasito d env
14                                         in (LessV (NumV n) d', env')
15 pasito (LessV i d) env               = let (i', env') = pasito i env
16                                         in (LessV i' d, env')
17 pasito (GreaterV (NumV n) (NumV m)) env = (BoolV (n > m), env)
18 pasito (GreaterV (NumV n) d) env        = let (d', env') = pasito d env
19                                         in (GreaterV (NumV n) d', env')
20 pasito (GreaterV i d) env             = let (i', env') = pasito i env
21                                         in (GreaterV i' d, env')
22 pasito (DiffV (NumV n) (NumV m)) env = (BoolV (n /= m), env)
23 pasito (DiffV (NumV n) d) env        = let (d', env') = pasito d env
24                                         in (DiffV (NumV n) d', env')
25 pasito (DiffV i d) env              = let (i', env') = pasito i env
26                                         in (DiffV i' d, env')
27 pasito (LeqV (NumV n) (NumV m)) env = (BoolV (n <= m), env)
28 pasito (LeqV (NumV n) d) env        = let (d', env') = pasito d env
29                                         in (LeqV (NumV n) d', env')
30 pasito (LeqV i d) env              = let (i', env') = pasito i env
31                                         in (LeqV i' d, env')
32 pasito (GeqV (NumV n) (NumV m)) env = (BoolV (n <= m), env)
33 pasito (GeqV (NumV n) d) env        = let (d', env') = pasito d env

```

```

34                                     in (GeqV (NumV n) d', env')
35 pasito (GeqV i d) env              = let (i', env') = pasito i env
36                                     in (GeqV i' d, env')

```

Código 5.8: Función pasito para comparadores

En el caso de `Not`, una vez que el argumento es `BoolV` retornamos la negación en Haskell de este `BoolV` como un nuevo `BoolV`. De la misma manera como fue con los operadores aritméticos, cuando ambos argumentos ya son números con los cuales comparar, regresamos la comparación como valor booleano `BoolV` con el resultado evaluado con los operadores de comparación en Haskell.

```

1 pasito :: ASV -> Env -> (ASV, Env)
2 --Pares
3 pasito (PairV f s) env
4   | isValue (PairV f s) = (PairV f s, env)
5   | isValue f = let (s', env') = pasito s env
6                 in (PairV f s', env')
7   | otherwise = let (f', env') = pasito f env
8                 in (PairV f' s, env')
9 pasito (FstV (PairV f s)) env
10  | isValue f && isValue s = (f, env)
11 pasito (FstV p) env = let (p', env') = pasito p env
12                       in (FstV p', env')
13 pasito (SndV (PairV f s)) env
14  | isValue f && isValue s = (s, env)
15 pasito (SndV p) env = let (p', env') = pasito p env
16                       in (SndV p', env')
17 ---Cons
18 pasito (ConV f s) env
19   | isValue (ConV f s) = (ConV f s, env)
20   | isValue f = let (s', env') = pasito s env
21                 in (ConV f s', env')
22   | otherwise = let (f', env') = pasito f env
23                 in (ConV f' s, env')
24 pasito (HeadV (ConV f s)) env
25  | isValue f && isValue s = (f, env)
26 pasito (HeadV p) env =
27   let (p', env') = pasito p env
28   in (HeadV p', env')
29 pasito (TailV (ConV f s)) env
30  | isValue f && isValue s && not (isConV s) = (s, env)
31  | otherwise = let (s', env') = pasito s env
32                in (TailV s', env')
33 pasito (TailV p) env =
34   let (p', env') = pasito p env
35   in (TailV p', env')
36
37
38 {- Funcion que nos dice si la estructura sigue siendo un ecadenamiento de
   ConV -}
39 isConV :: ASV -> Bool
40 isConV (ConV _ _) = True
41 isConV _ = False

```

Código 5.9: Función `pasito` para comparadores

Para el caso de `PairV` y `ConV`, en ambos, comprobamos que sean valores canónicos y aunque suene redundante, con la función `isValue`, ambos son valores si y sólo si sus dos argumentos también son valores. En otro caso continuamos evaluando ambas expresiones, primero con el primer argumento hasta llegar a un valor para poder seguir con la evaluación del segundo.

Para la evaluación de los operadores sobre pares, con `FstV`, como utilizamos evaluación global, debemos llegar a que el par recibido tenga ambas expresiones como valor, para así devolver el primer argumento del par. Lo mismo con `SndV` que, hasta tener los argumentos f y s como valores, devolvemos s , el segundo.

Para el caso de `HeadV` con `ConV` es análogo a `FstV` con los pares. Sin embargo, `TailV` no solo necesita que ambos argumentos ya sean valores, sino que el segundo valor no sea de tipo `ConV`, pues esto significa que hay más expresiones en el encadenamiento de `ConV`, por lo que necesitamos seguir con este segundo argumento hasta llegar a un valor canónico distinto de `ConV` -justo como en las reglas de evaluación para esta expresión-.

Para lograr esto, no apoyamos en la función `isConV`, que nos ayuda a determinar que la expresión dada sea otro `ConV` o si es otro valor canónico.

Para las reglas de la condicional `IfV` su implementación es más sencilla pues basta que, al llegar a un valor `BoolV` de la condición devolvemos la configuración respectiva: (t, env) en el caso de que la condición es verdad, y (e, env) en otro caso. De otro modo, continuamos evaluando la condición.

```

1 pasito :: ASV -> Env -> (ASV, Env)
2 --If
3 pasito (IfV (BoolV True) t e) env = (t, env)
4 pasito (IfV (BoolV False) t e) env = (e, env)
5 pasito (IfV c t e) env             = let (c', env') = pasito c env
6                                     in (IfV c' t e, env')
```

Código 5.10: Función `pasito` para comparadores

Por último tenemos las funciones y `FunV` las aplicaciones de funciones `AppV`, que quedan como sigue:

```

1 pasito :: ASV -> Env -> (ASV, Env)
2 --Funciones
3 pasito (FunV p c) env = (Closure p c env, env)
4 --Aplicacion de funciones
5 pasito (AppV (Closure p c env') a) env
6   | isValue a = (c, (p, a):env')
7   | otherwise =
8     let (a', env'') = pasito a env
9     in (AppV (Closure p c env') a', env'')
10 pasito (AppV f a) env = let (f', env') = pasito f env
11                        in (AppV f' a, env')
```

Código 5.11: Función `pasito` para comparadores

La evaluación de las funciones es simple, la función **FunV** se convierte en una cerradura con el mismo ambiente de la configuración. Para **AppV**, siguiendo las reglas de evaluación para esta expresión: si se tiene una cerradura con un argumento a aplicar, comprobamos que este argumento sea ya un valor canónico, de ser así, retornamos la configuración con la aplicación extendiendo el ambiente. En otro caso continuamos evaluando el argumento. De no ser una cerradura, entonces evaluamos esta función.

Así, terminamos la evaluación glotona con alcance estático usando paso pequeño con la función **pasito** en nuestro intérprete.

Sin embargo, esta implementación no nos sirve para la aplicación de funciones derivada de **LetRec**, pues veamos el resultado de evaluar **LetRec**:

```
[MiniLisp]> (letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum 3))
[Error]: Var 'Z' no definida
CallStack (from HasCallStack):
  error, called at ./Interprete.hs:154:18 in main:Interprete
```

Y es debido a que, como se menciona en el capítulo de **Azúcar Sintáctica**, necesitamos definir un *Combinador* para darle recursión. Más concretamente, usamos el combinador **Z**:

$$(\lambda f.((\lambda x.(f(\lambda v.((xx)v)))))(\lambda x.(f(\lambda v.((xx)v))))))$$

creo se puede mejorar la explicacion anterior

Y este combinador debemos agregarlo al ambiente de evaluación para que en el intérprete lo utilice, y asu vez, el ambiente debe estar evaluado:

$$[Z \rightarrow (\lambda f.((\lambda x.(f(\lambda v.((xx)v)))))(\lambda x.(f(\lambda v.((xx)v)))))]$$

No obstante, esto también presenta un error a la hora de ser evaluado:

```
[MiniLisp]> (letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum 3))
[Error]: Var 'v' no definida
CallStack (from HasCallStack):
  error, called at ./Interprete.hs:154:18 in main:Interprete
```

Este error ocurre porque el combinador **Z**, bajo un esquema de evaluación glotona, intenta evaluar inmediatamente el argumento que representa la función recursiva. En términos operacionales, la versión clásica del combinador de punto fijo asume evaluación perezosa (lazy evaluation): el argumento debe permanecer sin evaluar hasta que sea estrictamente necesario. En nuestro caso, la evaluación glotona fuerza la aplicación prematura de la función a sí misma, lo que conduce a buscar variables aún no ligadas en el ambiente, provocando fallos como la variable **v** no definida.

En otras palabras, la raíz del problema es que:

- el combinador **Z** está diseñado para lenguajes con evaluación perezosa,

- nuestro intérprete aplica evaluación glotona,
- y como consecuencia, la expansión recursiva ocurre antes de que el ambiente esté correctamente extendido.

Para soportar recursión en un lenguaje con evaluación estricta necesitamos, entonces, adaptar la construcción del combinador o modificar la semántica para permitir que las referencias recursivas se establezcan antes de evaluarse efectivamente. En la siguiente sección abordamos precisamente esta extensión.

Capítulo 6

Puntos Estrictos

Este es un capítulo corto pero esencial y crucial para la implementación de nuestro intérprete del lenguaje, ya que al meter recursión de la forma tradicional (como se explica en el capítulo 4 y 5), la manera en la que hemos planteado la evaluación hasta ahora necesita ser remodelada.

Para evitar el problema de las variables no definidas al utilizar recursion y combinadores, debemos volver a MINILISP perezoso, es decir, debemos reeplantear la estrategia de evaluación de MINILISP de glotón a perezoso.

Al usar evaluación perezosa, existen ciertos puntos dentro de la implementación en los que no puede postergarse la evaluación.

Explicar que es el alcance perezoso y lo que son puntos estrictos.

Para ello tenemos que definir una forma que nos permitiera aplicar los *puntos estrictos* de nuestro lenguaje. Esta forma la modelaremos mediante una función **strict**, que reduce la expresión hasta llegar a un **valor canónico** del lenguaje.

$$\mathbf{strict}(\langle e, \varepsilon \rangle) = \begin{cases} e, & \text{si } e \text{ es valor,} \\ \langle e', \varepsilon \rangle, & \text{si } \langle e, \varepsilon \rangle \rightarrow \langle e', \varepsilon' \rangle. \end{cases}$$

Para aplicar este mecanismo, sí o sí necesitamos nuestras reglas semánticas de modo que apliquen los *puntos estrictos* correspondientes usando **strict**. El resto de reglas se mantienen sin cambios, pero el cambio radica en que: aquellas expresiones donde detectamos la presencia de *puntos estrictos* y recibían cerraduras de expresión $\langle e, \varepsilon \rangle$, es donde llamamos **strict**.

6.1. Reglas semánticas perezosas con strict

- NumV(n) :

$$\langle \text{NumV}(n), \varepsilon \rangle \rightarrow \langle \text{NumV}(n), \varepsilon \rangle$$

- BoolV(b):

$$\langle \text{BoolV}(b), \varepsilon \rangle \rightarrow \langle \text{BoolV}(b), \varepsilon \rangle$$

- AddV(i,d):

$$\frac{\text{strict}(i, \varepsilon) = \text{NumV}(n) \quad \text{strict}(d, \varepsilon) = \text{NumV}(m)}{\langle \text{AddV}(i, d), \varepsilon \rangle \rightarrow \langle \text{NumV}(n + m), \varepsilon \rangle}$$

- SubV(i,d):

$$\frac{\text{strict}(i, \varepsilon) = \text{NumV}(n) \quad \text{strict}(d, \varepsilon) = \text{NumV}(m)}{\langle \text{SubV}(i, d), \varepsilon \rangle \rightarrow \langle \text{NumV}(n - m), \varepsilon \rangle}$$

- MulV(i,d):

$$\frac{\text{strict}(i, \varepsilon) = \text{NumV}(n) \quad \text{strict}(d, \varepsilon) = \text{NumV}(m)}{\langle \text{MulV}(i, d), \varepsilon \rangle \rightarrow \langle \text{NumV}(n * m), \varepsilon \rangle}$$

- DiV(i,d):

$$\frac{\text{strict}(i, \varepsilon) = \text{NumV}(n) \quad \text{strict}(d, \varepsilon) = \text{NumV}(m)}{\langle \text{DiV}(i, d), \varepsilon \rangle \rightarrow \langle \text{NumV}(n / m), \varepsilon \rangle}$$

$$\frac{\text{strict}(d) = \text{NumV}(0)}{\langle \text{DivV}(i, d), \varepsilon \rangle \rightarrow \text{Error}}$$

$$\frac{\text{strict}(i) = \text{NumV}(n) \quad \text{strict}(d) = \text{NumV}(m) \quad m \neq 0}{\langle \text{DivV}(i, d), \varepsilon \rangle \rightarrow \langle \text{NumV}(n/m), \varepsilon \rangle}$$

- SqrtV(e):

$$\frac{\text{strict}(e) = \text{NumV}(n) \quad n < 0}{\langle \text{SqrtV}(n), \varepsilon \rangle \rightarrow \text{Error}}$$

$$\frac{\text{strict}(e) = \text{NumV}(n) \quad n \geq 0}{\langle \text{SqrtV}(e), \varepsilon \rangle \rightarrow \langle \text{NumV}(\sqrt{n}), \varepsilon \rangle}$$

- NotV(e):

$$\frac{\text{strict}(e) = \text{BoolV}(b) \quad n \geq 0}{\langle \text{NotV}(e), \varepsilon \rangle \rightarrow \langle \text{BoolV}(\neg b), \varepsilon \rangle}$$

- EqualV(i,d)

$$\frac{\text{strict}(i) = \text{NumV}(n) \quad \text{strict}(d) = \text{NumV}(m)}{\langle \text{EqualV}(i, d), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n = m), \varepsilon \rangle}$$

- LessV(i,d):

$$\frac{\text{strict}(e_1) = \text{NumV}(n) \quad \text{strict}(e_2) = \text{NumV}(m)}{\langle \text{LessV}(e_1, e_2), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n < m), \varepsilon \rangle}$$

- DiffV(i,d):

$$\frac{\text{strict}(e_1) = \text{NumV}(n) \quad \text{strict}(e_2) = \text{NumV}(m)}{\langle \text{DiffV}(e_1, e_2), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n \neq m), \varepsilon \rangle}$$

- LeqV(i,d):

$$\frac{\text{strict}(e_1) = \text{NumV}(n) \quad \text{strict}(e_2) = \text{NumV}(m)}{\langle \text{LeqV}(e_1, e_2), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n \leq m), \varepsilon \rangle}$$

- $\text{GeqV}(i,d)$:

$$\frac{\text{strict}(e_1) = \text{NumV}(n) \quad \text{strict}(e_2) = \text{NumV}(m)}{\langle \text{GeqV}(e_1, e_2), \varepsilon \rangle \rightarrow \langle \text{BoolV}(n \geq m), \varepsilon \rangle}$$

- $\text{PairV}(f,s)$:

$$\langle \text{PairV}(f, s), \varepsilon \rangle \rightarrow \langle \text{PairV}(f, s), \varepsilon \rangle$$

- $\text{FstV}(p)$:

$$\frac{\text{strict}(p) = \text{PairV}(f, s)}{\langle \text{FstV}(p), \varepsilon \rangle \rightarrow \langle f, \varepsilon \rangle}$$

- $\text{SndV}(p)$:

$$\frac{\text{strict}(p) = \text{PairV}(f, s)}{\langle \text{SndV}(p), \varepsilon \rangle \rightarrow \langle s, \varepsilon \rangle}$$

- $\text{ConV}(i,d)$:

$$\langle \text{ConV}(i, d), \varepsilon \rangle \rightarrow \langle \text{ConV}(i, d), \varepsilon \rangle$$

- $\text{HeadV}(l)$:

$$\frac{\text{strict}(l) = \text{ConV}(i, d)}{\langle \text{HeadV}(l), \varepsilon \rangle \rightarrow \langle i, \varepsilon \rangle}$$

- $\text{TailV}(l)$:

$$\frac{\text{strict}(l) = \text{ConV}(i, d)}{\langle \text{TailV}(l), \varepsilon \rangle \rightarrow \langle d, \varepsilon \rangle}$$

- $\text{FunV}(p,c)$:

$$\langle \text{FunV}(p, c), \varepsilon \rangle \rightarrow \langle \text{Closure}(p, c, \varepsilon), \varepsilon \rangle$$

- $\text{AppV}(f,a)$:

$$\frac{\text{strict}(f) = \text{Closure}(p, c, \varepsilon')}{\langle \text{AppV}(f, a), \varepsilon \rangle \rightarrow \langle c, \varepsilon'[p \leftarrow a] \rangle}$$

Consultar la pagina 13-14 del ultimo pdf del profesor para ver contexto

Una vez definido todo lo anterior, veamos su implementación en Haskell y cómo se modifica el interprete de MINILISP -0.2cm.

6.2. Evaluación perezosa para MINILISP

Como se mostró, al cambiar la estrategia de evaluación de nuestro intérprete, debemos hacer muchos cambios en su implementación en Haskell.

Con este nuevo enfoque, nuestro tipo de dato **ASV** ya es más reducido donde solo están los valores canónicos:

```

1 module ASV where
2
3 import AST
4
5 {--
6 Definimos la representacion del ambiente de ejecucion.
7 Un ambiente es una lista de pares (id, valor).
8 --}
9 type Env = [(String, ASV)]
10
11 {-- ASA Values --}
12 data ASV
13   = NumV Int
14   | BoolV Bool
15   | NiV
16   | PairV ASV ASV
17   | ConV ASV ASV
18   | ClosureF String AST Env
19   | ExprV AST Env
20   deriving (Show, Eq)

```

Código 6.1: Tipo de dato ASV con valores canónicos

A diferencia de la implementación anterior, en este nuevo interprete ya no tenemos que usar la función `toFinalState` para convertir los **ASA** en estados finales, ya que los únicos estados finales son los valores canónicos (Ambas definiciones significan lo mismo pero recordemos que en la primer implementación hicimos el abuso de notación para marcar la diferencia entre ambos, de eso se encargará el nuevo intérprete, de convertir las expresion **AST** a estados finales (valores canónicos).

Definimos la función de evaluación perezosa con puntos estrictos como sigue:

```

1 module EvalStrict where
2
3 import AST
4 import ASV
5 import Interprete
6
7 {-- Funcion de evaluacion perezosa --}
8 evals :: AST -> Env -> ASV
9 --Valores
10 evals (VarC i) env = lookupS i env
11 evals (NumC n) _   = (NumV n)
12 evals (BoolC b) _  = (BoolV b)
13 evals NiL _        = NiV
14 --Operadores aritmeticos
15 evals (AddC i d) env =
16   let i' = strict (evals i env)
17       d' = strict (evals d env)
18   in NumV (numN i' + numN d')
19 evals (SubC i d) env =
20   let i' = strict (evals i env)
21       d' = strict (evals d env)

```

```

22   in NumV (numN i' - numN d')
23 evalS (MulC i d) env =
24   let i' = strict (evalS i env)
25       d' = strict (evalS d env)
26   in NumV (numN i' * numN d')
27 evalS (DivC i d) env =
28   let n = numN (strict (evalS i env))
29       m = numN (strict (evalS d env))
30   in if m == 0
31       then error "No se puede dividir entre 0"
32       else NumV (div n m)
33 evalS (SqrtC n) env =
34   let n' = numN (strict (evalS n env))
35   in if n' < 0
36       then error "No se puede obtener la raiz de un numero negativo"
37       else NumV (integerSquareRoot n')
38 --Not
39 evalS (NotC e) env =
40   let e' = strict (evalS e env)
41   in BoolV (not (boolN e'))
42 --Comparadores
43 evalS (EqualC i d) env =
44   let i' = strict (evalS i env)
45       d' = strict (evalS d env)
46   in BoolV (numN i' == numN d')
47 evalS (LessC i d) env =
48   let i' = strict (evalS i env)
49       d' = strict (evalS d env)
50   in BoolV (numN i' < numN d')
51 evalS (GreaterC i d) env =
52   let i' = strict (evalS i env)
53       d' = strict (evalS d env)
54   in BoolV (numN i' > numN d')
55 evalS (DiffC i d) env =
56   let i' = strict (evalS i env)
57       d' = strict (evalS d env)
58   in BoolV (numN i' /= numN d')
59 evalS (LeqC i d) env =
60   let i' = strict (evalS i env)
61       d' = strict (evalS d env)
62   in BoolV (numN i' <= numN d')
63 evalS (GeqC i d) env =
64   let i' = strict (evalS i env)
65       d' = strict (evalS d env)
66   in BoolV (numN i' >= numN d')
67 --Pares
68 evalS (PairC i d) env =
69   let i' = strict (evalS i env)
70       d' = strict (evalS d env)
71   in PairV i' d'
72 evalS (FstC p) env =
73   case strict (evalS p env) of
74     PairV f _ -> f
75     -         -> error "Fst espera un par"

```

```

76 evalS (SndC p) env =
77     case strict (evalS p env) of
78         PairV _ s -> s
79         _         -> error "Snd espera un par"
80 --Cons
81 evalS (ConS i d) env =
82     let i' = strict (evalS i env)
83         d' = strict (evalS d env)
84     in ConV i' d'
85 evalS (HeadC p) env =
86     case strict (evalS p env) of
87         ConV h _ -> h
88         _         -> error "Head espera una lista"
89 evalS (TailC p) env =
90     case strict (evalS p env) of
91         ConV _ t ->
92             let t' = strict t
93                 in if not (isConV t')
94                     then t'
95                     else tailDeep t'
96         _         -> error "Tail espera una lista"
97 --If
98 evalS (IfC c t e) env =
99     let cond = boolN (strict (evalS c env))
100     in if cond then evalS t env else evalS e env
101 --Funciones
102 evalS (FunC p c) env = ClosureF p c env
103 --Aplicacion de funciones
104 evalS (AppC f a) env =
105     let f' = evalS f env
106     funV = strict f'
107     in evalS (closureC funV) (((closureP funV), ExprV a env) : (closureE
108         funV))
109
110 {- Funcion strict para forzar la evaluacion de los puntos estrictos -}
111 strict :: ASV -> ASV
112 strict (NumV n) = NumV n
113 strict (BoolV b) = BoolV b
114 strict (PairV f s) = PairV (strict f) (strict s)
115 strict (ConV i d) = ConV (strict i) (strict d)
116 strict (ExprV a e) = strict (evalS a e)
117 strict (NiV) = NiV
118 strict (ClosureF p c e) = ClosureF p c e
119
120 {- Funcion auxiliar para devolver el numero de NumV -}
121 numN :: ASV -> Int
122 numN (NumV n) = n
123
124 {- Funcion auxiliar para devolver el booleano de BoolV -}
125 boolN :: ASV -> Bool
126 boolN (BoolV b) = b
127 boolN _ = False
128
129 {- Funcion auxiliar para devolver el parametro de la cerradura -}

```

```

129 closureP :: ASV -> String
130 closureP (ClosureF p _) = p
131
132 {- Funcion auxiliar para devolver el cuerpo de la cerradura -}
133 closureC :: ASV -> AST
134 closureC (ClosureF _ c _) = c
135
136 {- Funcion auxiliar para devolver el ambiente de la cerradura -}
137 closureE :: ASV -> Env
138 closureE (ClosureF _ _ e) = e
139
140 {- Funcion auxiliar para encontrar el ultimo elemento canonico de los
    ConV anidados -}
141 tailDeep :: ASV -> ASV
142 tailDeep (ConV _ rest) =
143     let rest' = strict rest
144     in if not (isConV rest')
145         then rest'
146         else tailDeep rest'
147 tailDeep v = v

```

Código 6.2: Evaluación perezosa con **strict** para MINILISP -0.2cm

La mayor diferencia a `eval` es que la función `evalS` ya no utiliza la evaluación de paso pequeño, sino que implementa una evaluación directa (*big-step*). Ahora utilizamos funciones auxiliares (`strict`, `numN`, `boolN`, etc.) para obtener valores “forzados”.

Además de este cambio, lo más notorio es `TailC` y la aplicación de funciones. La primer expresión ahora utiliza una función auxiliar `tailDeep` para seguir buscando el último elemento en la anidación de `ConS`, ya que usar `evalS` nos da una inconsistencia de tipos, pues al evaluar por primera vez el par, este se transforma en un `ASV` siendo que `evalS` recibe `AST`.

Para el caso de `AppC`:

1. Evalúa f en el ambiente actual, esto debería devolver una cerradura.
2. Fuerza f' con `strict`, asegurándose de que sea una cerradura concreta.
3. Extrae información de la cerradura:
 - `closureC funV`: el cuerpo de la función.
 - `closureP funV`: el nombre del parámetro formal.
 - `closureE funV`: el ambiente donde la función fue creada.
4. Crea un nuevo ambiente donde:
 - El parámetro formal (`closureP funV`) se asocia a la expresión real (`ExprV a env`);
 - Se añade al ambiente original de la cerradura (`closureE funV`).
5. Evalúa el cuerpo de la función (`closureC funV`) en ese nuevo ambiente extendido.

De este modo tenemos una evaluación adecuada para introducir recursión con `LetRec` en nuestro lenguaje MINILISP -0.2cm, pues como mencionamos, necesitamos meter al ambiente inicial la evaluación de nuestro combinador **Z**, el cuál se mostró que presentaba errores con la evaluación glotona.

6.3. Cierre: intérprete, recursión y evaluación glotona

En este capítulo hemos introducido la idea de *puntos estrictos* y la operación meta-semántica **strict** para mezclar evaluación perezosa con los contextos que requieren valores canónicos. Antes de terminar, conviene cerrar con tres observaciones importantes relacionadas con la representación de cierres en el intérprete, el tratamiento de la recursión y el contraste con la evaluación *glotona* (estricta).

6.3.1. Cierre del intérprete (closures)

Un intérprete funcional perezoso debe representar funciones como *closures* que capturan su ambiente léxico en el momento de la creación. Formalmente, una función se convierte en el valor

$$Closure(p, c, \varepsilon)$$

donde p es el parámetro (o patrón), c el cuerpo y ε el ambiente en el que la función fue creada. Esta representación garantiza que, al aplicar la función, las referencias libres del cuerpo se resuelvan respecto al ambiente léxico original.

En un intérprete real (implementación), un closure típicamente contiene:

- el código del cuerpo (o un puntero/índice a ese código);
- un mapa de identidades a valores diferidos (thunks) o valores ya evaluados;
- una semántica consistente para la sustitución o enlace del parámetro con el argumento.

6.3.2. Recursión y combinadores

La recursión puede introducirse de varias maneras:

1. **Ligada (letrec / definiciones recursivas)**: extiende el ambiente con una entrada que referencia una celda que puede actualizarse o auto-referenciarse.
2. **Combinadores de punto fijo (por ejemplo Y)**: expresan la recursión sin nombres mediante un combinador que produce un punto fijo de funciones.

Es importante destacar la diferencia de comportamiento del combinador Y según la estrategia de evaluación:

Estrategia perezosa. Con evaluación perezosa, el combinador Y puede ser útil porque el desempaqueado del punto fijo se hace cuando el cuerpo lo requiere, permitiendo definiciones recursivas sin evaluación inmediata que provoque bucles infinitos. Por eso, muchas construcciones recursivas clásicas basadas en Y funcionan en lenguajes perezosos.

Estrategia glotona (estricta). En evaluación estricta, el uso directo del combinador Y clásico provoca normalmente no-terminación: la autoinvocación se expande inmediatamente y nunca alcanza un punto en que se pueda seguir computando. Para obtener recursión en un lenguaje estricto se suele necesitar:

- **letrec** o **fix** implementado con celdas mutables (o un mecanismo de enlace que permita referencias a sí mismo), o
- transformar las definiciones mediante técnicas como *lambda-lifting* o introducir explícitamente una celda recursiva (thunk auto-referente) que posponga la evaluación.

Ejemplo informal

Sea la definición recursiva factorial, expresada con un combinador de punto fijo:

$$fact = Y(\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1)).$$

En evaluación perezosa, las llamadas a f se fuerzan sólo cuando es necesario, y Y puede comportarse adecuadamente. En evaluación estricta, la expansión de Y genera una evaluación infinita antes de que se alcance alguna condición base, salvo que se use una forma de **letrec** o una celda recursiva.

6.3.3. Evaluación glotona (estricta): reglas contrastantes

Para clarificar el contraste con la semántica perezosa con **strict**, presentamos una versión compacta de la regla de aplicación en evaluación estricta (call-by-value), que ilustra la diferencia con la regla lazy que usamos antes.

$$\frac{\langle f, \varepsilon \rangle \rightarrow^* \langle Closure(p, c, \varepsilon_f), \varepsilon \rangle \quad \langle a, \varepsilon \rangle \rightarrow^* \langle v_a, \varepsilon \rangle}{\langle AppV(f, a), \varepsilon \rangle \rightarrow \langle c, \varepsilon_f[p \leftarrow v_a] \rangle}$$

En palabras: en evaluación estricta evaluamos primero **la función** hasta obtener un closure y **el argumento** hasta obtener un valor canónico v_a ; sólo entonces realizamos la sustitución/enlace del parámetro. Esto contrasta con la regla lazy:

$$\frac{strict(f) = Closure(p, c, \varepsilon_f)}{\langle AppV(f, a), \varepsilon \rangle \rightarrow \langle c, \varepsilon_f[p \leftarrow a] \rangle}$$

donde en el segundo miembro *no* se evalúa a al aplicar: se guarda la expresión a (thunk) en el ambiente del closure.

6.3.4. Cierre

Con esto concluimos el tratamiento de los puntos estrictos y su integración en la semántica de MINILISP. Hemos mostrado cómo conservar la pereza por defecto mientras forzamos evaluación en contextos donde es necesaria, y cómo este diseño afecta la representación de cierres, la forma de implementar recursión y las diferencias prácticas frente a una semántica glotona. En capítulos posteriores podremos:

- mostrar la implementación concreta de **strict** en el intérprete (código y gestión de thunks),
- dar ejemplos paso a paso de evaluación de expresiones recursivas bajo ambas estrategias, y
- discutir optimizaciones (memoización de thunks, eliminación de thunks no necesarios, etc.).

Capítulo 7

Resultados

Bien, una vez hemos visto toda la teoría de nuestro MINILISP y además de que hemos mostrado el código mismo. En este capítulo presentamos los resultados obtenidos tras la implementación completa del lenguaje MINILISP. Se mostrará el funcionamiento de cada una de las etapas principales del lenguaje (*lexer*, *parser*, *desugar*, *interprete*), acompañadas de ejemplos que ilustran tanto su entrada como su salida.

Además, se presenta el *menú interactivo* del proyecto, que permite al usuario ejecutar comandos y evaluar expresiones sin hacer el proceso paso a paso. Este componente sirve como punto de unión entre todas las fases del lenguaje, permitiendo observar la interacción completa desde la lectura del código fuente hasta la obtención del resultado final.

En conjunto, buscamos que este capítulo tenga como propósito mostrar de forma integrada y funcional el resultado del trabajo desarrollado a lo largo de este reporte. Más que solo validar la implementación, buscamos evidenciar la coherencia entre el diseño teórico del lenguaje y su comportamiento práctico, demostrando que los principios formales de la sintaxis y la semántica pueden efectivamente traducirse en un sistema ejecutable, expresivo y consistente.

7.1. MINILISP

Primero mostramos los resultados individuales de cada fase del código implementado que le da vida a nuestro lenguaje MINILISP -0.2cm.

Utilizamos el intérprete interactivo de Haskell `GHCi` para compilar, cargar módulos y ejecutar las pruebas de de nuestra implementación. En el archivo `README.md` mostramos más a detalle como inicializar y compilar nuestro proyecto.

7.1.1. Lexer

Como se vió a inicios del reporte, el *analizador léxico* es el primer paso en la ejecución del lenguaje. Recibe el programa del usuario como cadena de caracteres y devuelve una lista de *Tokens* ya definido. Generamos el analizador léxico con `Alex`:

```
$ alex Lexer.x
```

Iniciamos el intérprete interactivo de Haskell y para no tener que escribir siempre el nombre completo del módulo `Lexer`, cargalo:

```
$ ghci
GHCi, version 9.4.5: https://www.haskell.org/ghc/  :? for help
ghci> :l Lexer.hs
```

Ya con esto podemos probar nuestra función `lexer`, que no es muy impresionante, como dijimos, va verificando la entrada detectando los *Tokens*:

■ Variables:

```
ghci> lexer "var12 #f 512 #t sum 90ba"
[TokenVar "var12",TokenBool False,TokenNum 512,
 TokenBool True,TokenVar "sum",TokenNum 90,TokenVar "ba"]
```

Podemos ver que el `lexer` separa adecuadamente entre variables, números y booleanos asignándolos a su respectivo `Token`. Incluso `var12` se guarda completo como `TokenVar "var12"` justo como lo definimos, donde las variables comienzan siempre con un caracter seguido de una combinación de caracteres o números. Se puede apreciar también con el vaso de `90ba` donde lo separa como dos *Token* distintos, pues los números no pueden tener caracteres ni las variables pueden comenzar con números.

■ Operadores:

Veamos la lista de operadores generada por una cadena de operaciones (para acortar la extensión de esta sección):

```
ghci> lexer "(expt (+ (- (* 10 3) (/ (add1 5) (sub1 4))) (sqrt 81)))"
[TokenPA,TokenExpt,
 TokenPA,TokenAdd,
 TokenPA,TokenSub,
 TokenPA,TokenMul,TokenNum 10,TokenNum 3,TokenPC,
 TokenPA,TokenDiv,
 TokenPA,TokenAdd1,TokenNum 5,TokenPC,
 TokenPA,TokenSub1,TokenNum 4,TokenPC,
 TokenPC,TokenPC,
 TokenPA,TokenSqrt,TokenNum 81,TokenPC,TokenPC,TokenPC]
```

Como se puede apreciar, los símbolos y palabras reservadas para operadores son detectadas correctamente por el `lexer`. No verificamos aún que los argumentos sean válidos en cantidad y tipo, pero si verificamos que los símbolos sean solo los definidos:

```
ghci> lexer "(+ 3)"
[TokenPA,TokenAdd,TokenNum 3,TokenPC]
ghci> lexer "(sqrt hola)"
[TokenPA,TokenSqrt,TokenVar "hola",TokenPC]
ghci> lexer "(% 3 5)"
[TokenPA,*** Exception: Lexical error:
  character no reconocido = "%" | codepoints = [37]
  CallStack (from HasCallStack):
    error, called at Lexer.hs:10816:24 in main:Lexer]
```

■ Comparadores:

De manera análoga, tenemos los comparadores:

```
ghci> lexer "(!= 0 9 9) (= 6 6 1)
          (> 1 1) (< 4 3) (>= 7 3 5) (<= 22 22 2) (not #t)"
[TokenPA,TokenNeq,TokenNum 0,TokenNum 9,TokenNum 9,TokenPC,
 TokenPA,TokenEq,TokenNum 6,TokenNum 6,TokenNum 1,TokenPC,
 TokenPA,TokenGt,TokenNum 1,TokenNum 1,TokenPC,
 TokenPA,TokenLt,TokenNum 4,TokenNum 3,TokenPC,
 TokenPA,TokenGeq,TokenNum 7,TokenNum 3,TokenNum 5,TokenPC,
 TokenPA,TokenLeq,TokenNum 22,TokenNum 22,TokenNum 2,TokenPC,
 TokenPA,TokenNot,TokenBool True,TokenPC]
```

Como se mencionó, en este punto no es relevante para el `lexer` los argumentos de cada comparador.

■ Condicionales:

```
ghci> lexer "(cond [(= (sqrt 1000) (expt 10)) -1]
                [(!= (sub1 9) (add1 8)) 1] [else 0])"
[TokenPA,TokenCond,
 TokenLI,TokenPA,TokenEq,TokenPA,TokenSqrt,TokenNum 1000,TokenPC,
 TokenPA,TokenExpt,TokenNum 10,TokenPC,TokenPC,TokenNum (-1),TokenLD,
 TokenLI,TokenPA,TokenNeq,TokenPA,TokenSub1,TokenNum 9,TokenPC,
 TokenPA,TokenAdd1,TokenNum 8,TokenPC,TokenPC,TokenNum 1,TokenLD,
 TokenLI,TokenElse,TokenNum 0,TokenLD,TokenPC]
```

■ Pares y Listas:

```
ghci> lexer "[(8,10),[],(#t,#f)]"
[TokenLI,
 TokenPA,TokenNum 8,TokenComma,TokenNum 10,TokenPC,TokenComma,
```

```
TokenLI,TokenLD,TokenComma,
TokenPA,TokenBool True,TokenComma,TokenBool False,TokenPC,
TokenLD]
```

■ Lets y Expresiones Lambda:

```
ghci> lexer "(let (x 10) (expt x))"
[TokenPA,TokenLet,TokenPA,TokenVar "x",TokenNum 10,TokenPC,
TokenPA,TokenExpt,TokenVar "x",TokenPC,TokenPC]
```

```
ghci> lexer "(let* ((x 2)) (+ x 3))"
[TokenPA,TokenLetStar,
TokenPA,TokenPA,TokenVar "x",TokenNum 2,TokenPC,TokenPC,
TokenPA,TokenAdd,TokenVar "x",TokenNum 3,TokenPC,TokenPC]
```

```
ghci> lexer "((lambda (x) (+ x 1)) 5)"
[TokenPA,TokenPA,TokenLambda,TokenPA,TokenVar "x",TokenPC,
TokenPA,TokenAdd,TokenVar "x",TokenNum 1,TokenPC,TokenPC,
TokenNum 5,TokenPC]
```

Así, aunque los resultados del `lexer` no son tan emocionantes como lo pudieran ser para el `desugar` o `eval`, hemos mostrado con estos ejemplos que hasta el momento, el análisis sintáctico para el lenguaje funciona.

A partir de este momento no mostraremos los resultados de aplicar las fases a únicamente variables ya que es redundante su procedimiento pues podemos ver su progreso en las fases del lenguaje a través de las demás expresiones.

7.1.2. Parser

En la fase del `Parser` la situación se vuelve más interesante pues es donde aplicamos la gramática del lenguaje y decidimos las estructuras del programa que son válidas. Veamos ejemplos para algunas expresiones donde son rechazados por el `Parser`:

```
ghci> tokens = lexer "(+ 3)"
ghci> parse tokens
*** Exception: Error al Parsear los Tokens
CallStack (from HasCallStack):
  error, called at ./Grammar.hs:1265:16 in main:Grammar
```

Falla porque la suma es un operador variádico que requiere de al menos dos elementos, por eso hay error en el **Parser**. De manera similar, fallan los operadores de resta, multiplicación y división con un solo argumentos, pues, requieren también de dos argumentos como mínimo. Los ejemplos válidos serían:

```
ghci> tokens = lexer "(+ 52 34 42)"
ghci> parse tokens
Add [Num 52,Num 34,Num 42]
ghci> tokens = lexer "(- 22 -11 7)"
ghci> parse tokens
Sub [Num 22,Num (-11),Num 7]
ghci> tokens = lexer "(* 2 200)"
ghci> parse tokens
Mul [Num 2,Num 200]
ghci> tokens = lexer "(/ 21 0)"
ghci> parse tokens
Div [Num 21,Num 0]
```

Nótese que en la división no marcamos error al dividir por cero, pues recordemos que eso es trabajo de la semántica, estamos en el *análisis sintáctico*.

Otro caso son los operadores unarios:

```
ghci> tokens = lexer "(sqrt 33 81)"
ghci> parse tokens
*** Exception: Error al Parsear los Tokens
CallStack (from HasCallStack):
  error, called at ./Grammar.hs:1265:16 in main:Grammar
ghci> tokens = lexer "(expt 21 1 3)"
ghci> parse tokens
*** Exception: Error al Parsear los Tokens
CallStack (from HasCallStack):
  error, called at ./Grammar.hs:1265:16 in main:Grammar
```

Sqrt y Expt fallan porque son operadores unarios, la gramática rechaza tener más de uno:

```
ghci> tokens = lexer "(sqrt 81)"
ghci> parse tokens
Sqrt (Num 81)
ghci> tokens = lexer "(expt 16)"
ghci> parse tokens
Expt (Num 16)
```

Las demás expresiones funcionan igual, dan error en el **Parser** con una gramática inválida, por ello veamos como queda el resultado de parsear programas válidos:

■ Comparadores:

```
ghci> tokens = lexer "(= (+ 2 3) (* 5 1))"  
ghci> parse tokens  
Equal [Add [Num 2,Num 3],Mul [Num 5,Num 1]]
```

```
ghci> tokens = lexer "< (+ 1 2) (* 2 3))"  
ghci> parse tokens  
Less [Add [Num 1,Num 2],Mul [Num 2,Num 3]]
```

```
ghci> tokens = lexer "> (* 3 3) (+ 4 2))"  
ghci> parse tokens  
Greater [Mul [Num 3,Num 3],Add [Num 4,Num 2]]
```

```
ghci> tokens = lexer "(!= (* 2 3) (+ 3 3))"  
ghci> parse tokens  
Diff [Mul [Num 2,Num 3],Add [Num 3,Num 3]]
```

```
ghci> tokens = lexer "<= (+ 2 3) (* 2 3))"  
ghci> parse tokens  
Leq [Add [Num 2,Num 3],Mul [Num 2,Num 3]]
```

```
ghci> tokens = lexer ">= (* 3 3) (+ 4 2))"  
ghci> parse tokens  
Geq [Mul [Num 3,Num 3],Add [Num 4,Num 2]]
```

Notemos que las estructuras se muestran como se deben, con sus argumento guardados como listas y sus etiquetas respectivas a sus *Tokens*.

■ Condicionales:

```
ghci> tokens = lexer "(if (> 3 2) (+ 1 2) (* 2 2))"  
ghci> parse tokens  
If (Greater [Num 3,Num 2]) (Add [Num 1,Num 2]) (Mul [Num 2,Num 2])
```

```
ghci> tokens = lexer "(if0 (- 3 3) (+ 1 2) (* 3 3))"
ghci> parse tokens
If0 (Sub [Num 3,Num 3]) (Add [Num 1,Num 2]) (Mul [Num 3,Num 3])
```

```
ghci> tokens = lexer "(cond [(< x 0) (- 0 x)] [(= x 0) 0]
                        [else (+ x 1)])"
ghci> parse tokens
Cond [(Less [Var "x",Num 0],Sub [Num 0,Var "x"]),
      (Equal [Var "x",Num 0],Num 0)] (Add [Var "x",Num 1])
```

■ Pares y Listas:

```
ghci> tokens = lexer "((+ 1 2), (* 3 4))"
ghci> parse tokens
Pair (Add [Num 1,Num 2]) (Mul [Num 3,Num 4])
```

```
ghci> tokens = lexer "(fst ((+ 1 2), (sqrt 9)))"
ghci> parse tokens
Fst (Pair (Add [Num 1,Num 2]) (Sqrt (Num 9)))
```

```
ghci> tokens = lexer "(snd ((sqrt 16), (+ 3 5)))"
ghci> parse tokens
Snd (Pair (Sqrt (Num 16)) (Add [Num 3,Num 5]))
```

```
ghci> tokens = lexer "[[1, 2, (3, 4)], #t, (+ 1 2)]"
ghci> parse tokens
List [List [Num 1,Num 2,Pair (Num 3) (Num 4)],
      Boolean True,Add [Num 1,Num 2]]
```

```
ghci> tokens = lexer "(head [[1, 2], (+ 3 4), #f])"
ghci> parse tokens
Head (List [List [Num 1,Num 2],Add [Num 3,Num 4],Boolean False])
```

```
ghci> tokens = lexer "(tail [(+ 1 2)], (* 3 4), #t)"
ghci> parse tokens
Tail (List [List [Add [Num 1,Num 2]],Mul [Num 3,Num 4],Boolean True])
```

■ Lets y Expresiones Lambda:

```
ghci> tokens = lexer "(let ((x 2) (y (* x 3))) (+ x y))"
ghci> parse tokens
Let [("x",Num 2),("y",Mul [Var "x",Num 3])] (Add [Var "x",Var "y"])
```

```
ghci> tokens = lexer "(let* ((x 2) (y (+ x 3)) (z (* y 2))) (+ x y z))"
ghci> parse tokens
LetStar [("x",Num 2),("y",Add [Var "x",Num 3]),
        ("z",Mul [Var "y",Num 2])] (Add [Var "x",Var "y",Var "z"])
```

```
ghci> tokens =
lexer "(letrec (fact
              (lambda (n) (if0 n 1 (* n (fact (sub1 n)))))) (fact 5))"
ghci> parse tokens
LetRec "fact" (Lambda ["n"] (If0 (Var "n") (Num 1)
  (Mul [Var "n",App (Var "fact") [Sub1 (Var "n")]])))
(App (Var "fact") [Num 5])
```

```
ghci> tokens = lexer "(lambda (x y) (if (> x y) (- x y) (+ x y)))"
ghci> parse tokens
Lambda ["x","y"] (If (Greater [Var "x",Var "y"])
  (Sub [Var "x",Var "y"]) (Add [Var "x",Var "y"])
```

```
ghci> tokens = lexer "((lambda (f x) (f x)) (lambda (y) (* y y)) 4)"
ghci> parse tokens
App (Lambda ["f","x"] (App (Var "f") [Var "x"]))
  [Lambda ["y"] (Mul [Var "y",Var "y"]),Num 4]
```

Podemos observar que las salidas generadas por el parser corresponden correctamente a la estructura del *Árbol de Sintaxis Abstracta* definido para MINILISP. Cada expresión se traduce en una construcción interna con sus operadores y argumentos organizados en listas.

En los ejemplos de comparadores y condicionales, se refleja cómo las expresiones se agrupan jerárquicamente, respetando el orden y los paréntesis del código fuente. Las secciones de pares y listas muestran la correcta interpretación de estructuras anidadas y de funciones de acceso como `fst`, `snd`, `head` y `tail`. Por último, las construcciones de `let`, `let*`, `letrec` y `lambda` evidencian el manejo de entornos locales y funciones como valores, preparando el terreno para su posterior desazucarización y evaluación semántica.

En conjunto, estos resultados confirman que la gramática y el parser generan correctamente los ASA esperados para cada tipo de expresión del lenguaje MiniLisp. Nótese que además que estas estructuras son **ASA** con azúcar, pues se puede apreciar por ejemplo, el uso listas en Haskell para ciertas estructuras. Aún nos falta la fase de desazucarización.

7.1.3. Desugar

Veremos los resultados interesantes del proceso de desazucarización:

```
ghci> tokens = lexer "(add1 (* 2 3))"
ghci> asa = parse tokens
ghci> desugar asa
AddC (MulC (NumC 2) (NumC 3)) (NumC 1)
```

```
ghci> tokens = lexer "(sub1 (+ 4 (* 2 3)))"
ghci> asa = parse tokens
ghci> desugar asa
SubC (AddC (NumC 4) (MulC (NumC 2) (NumC 3))) (NumC 1)
```

```
ghci> tokens = lexer "(expt (add1 (* 2 2)))"
ghci> asa = parse tokens
ghci> desugar asa
MulC (AddC (MulC (NumC 2) (NumC 2)) (NumC 1))
      (AddC (MulC (NumC 2) (NumC 2)) (NumC 1))
```

Como se puede ver, `add1`, `sub1` y `expt`, se convierten en suma, resta y multiplicación respectivamente.

```
ghci> tokens = lexer "(>= (* 3 3) (+ 4 2))"
ghci> asa = parse tokens
ghci> desugar asa
GeqC (MulC (NumC 3) (NumC 3)) (AddC (NumC 4) (NumC 2))
```

```
ghci> tokens = lexer "(= 4 0 (+ 9 3))"
ghci> asa = parse tokens
```

```
ghci> desugar asa
IfC (EqualC (NumC 4) (NumC 0)) (EqualC (NumC 0) (AddC (NumC 9) (NumC 3)))
  (BoolC False)
```

```
ghci> tokens = lexer "(!= 1 1 (- 7 7))"
ghci> asa = parse tokens
ghci> desugar asa
IfC (DiffC (NumC 1) (NumC 1)) (DiffC (NumC 1) (SubC (NumC 7) (NumC 7)))
  (BoolC False)
```

De igual manera, los comparadores ya no son una lista de comparaciones, sino un enca-
denamiento de condicionales. De manera similar con las condiciones If0 y Cond que pasan a
IfC.

```
ghci> tokens = lexer "(if0 (- 3 3) (+ 1 2) (* 3 3))"
ghci> asa = parse tokens
ghci> desugar asa
IfC (EqualC (SubC (NumC 3) (NumC 3)) (NumC 0)) (AddC (NumC 1) (NumC 2))
  (MulC (NumC 3) (NumC 3))
```

```
ghci> tokens = lexer "(cond [(< x 0) (- 0 x)] [(= x 0) 0] [else (+ x 1)])"
ghci> asa = parse tokens
ghci> desugar asa
IfC (LessC (VarC "x") (NumC 0)) (SubC (NumC 0) (VarC "x"))
  (IfC (EqualC (VarC "x") (NumC 0)) (NumC 0) (AddC (VarC "x") (NumC 1)))
```

En el caso de las listas notemos que se han convertido en una encadenación de *cons*.

```
ghci> tokens = lexer "[1, 2, 3]"
ghci> asa = parse tokens
ghci> desugar asa
ConS (NumC 1) (ConS (NumC 2) (NumC 3))
ghci> tokens = lexer "[[1, 2, (3, 4)], #t, (+ 1 2)]"
ghci> asa = parse tokens
ghci> desugar asa
ConS (ConS (NumC 1) (ConS (NumC 2) (PairC (NumC 3) (NumC 4))))
  (ConS (BoolC True) (AddC (NumC 1) (NumC 2)))
```

Para los lets, como se puede apreciar a continuación, se han convertido en aplicaciones
de funciones.

```
ghci> tokens = lexer "(let ((x 5)) (+ x 1))"
ghci> asa = parse tokens
```

```
ghci> desugar asa
AppC (FunC "x" (AddC (VarC "x") (NumC 1))) (NumC 5)
ghci> tokens = lexer "(let ((x 2) (y (* x 3))) (+ x y))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "x" (AppC (FunC "y" (AddC (VarC "x") (VarC "y"))))
      (MulC (VarC "x") (NumC 3)))) (NumC 2)
```

```
ghci> tokens = lexer "(let* ((x 2) (y (+ x 3)) (z (* y 2))) (+ x y z))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "x" (AppC (FunC "y" (AppC (FunC "z" (AddC (VarC "x")
      (AddC (VarC "y") (VarC "z")))) (MulC (VarC "y") (NumC 2))))
      (AddC (VarC "x") (NumC 3)))) (NumC 2)
```

```
ghci> tokens = lexer "(letrec (fact (lambda (n)
      (if0 n 1 (* n (fact (sub1 n)))))) (fact 5))"
ghci> asa = parse tokens
ghci> desugar asa
AppC (FunC "fact" (AppC (VarC "fact") (NumC 5))) (AppC (VarC "Z")
      (FunC "fact" (FunC "n" (IfC (EqualC (VarC "n") (NumC 0))
      (NumC 1) (MulC (VarC "n") (AppC (VarC "fact") (SubC (VarC "n") (NumC 1)))
      )))))
```

7.2. Menú interactivo

Para mostrar los resultados del intérprete, utilizaremos un menú interactivo que tendrá la función de recibir la entrada del usuario, para procesarla a través de todas las fases de Lexer, Parser y Desugar para eventualmente realizar el proceso de evaluación semántica.

El menú interactivo queda definido en el archivo `MiniLisp.hs` como sigue:

```
1 module MiniLisp where
2 import Token
3 import ASA
4 import AST
5 import ASV
6 import Lexer
7 import Grammar
8 import Desugar
9 import Interprete
10 import EvalStrict
11 import Saca
12 import Control.Exception (catch, SomeException)
```

```

13 -- Combinador Z
14 combZ :: String
15 combZ = "(lambda (f) ((lambda (x) (f (lambda (v) ((x x) v)))) (lambda (x)
    (f (lambda (v) ((x x) v))))))"
16 -- Evaluamos el combinador Z
17 z :: ASV
18 z = evalS (desugar $ parse $ lexer combZ) []
19 -- Punto de entrada principal
20 main :: IO ()
21 main =
22     do
23         putStrLn "\nBienvenido a MiniLisp. "
24         putStrLn "Escriba (exit) para salir."
25         minilisp
26 -- Bucle principal del interprete
27 minilisp =
28     do
29         putStr "[MiniLisp]> "
30         str <- getLine
31         if str == ""
32             then minilisp
33             else if str == ":q"
34                 then putStrLn "Bye :)"
35                 else do
36                     run str
37                     minilisp
38 -- Envuelve la evaluacion con manejo de errores
39 run :: String -> IO ()
40 run input =
41     catch
42         (do
43             let tokens = lexer input
44             let asa = parse tokens
45             let ast = desugar asa
46             let asv = evalS (ast) [("Z", z)]
47             putStrLn (saca asv))
48         errors
49 -- Manejador de errores
50 errors :: SomeException -> IO ()
51 errors e = putStrLn $ "[Error]: " ++ show e

```

Código 7.1: Menú interactivo de MINILISP

Importamos todos los módulos a usar en el proyecto. Definimos el combinador **Z** y aplicamos su evaluación como es requerido para la recursión en MINILISP -0.2cm.

El punto de entrada principal del lenguaje es la función **main**, donde damos la bienvenida al usuario y comnzamos el intérprete interactivo de MINILISP -0.2cm. De este modo hacemos lo siguiente para correr el proyecto:

```

$ ghci
GHCi, version 9.4.5: https://www.haskell.org/ghc/  :? for help
ghci> :l MiniLisp.hs
ghci> main

```

```

Bienvenido a MiniLisp.
Escriba (exit) para salir.
[MiniLisp]>

```

Como primer instancia, definimos y evaluamos el **combinador Z**. Este combinador es una herramienta fundamental para implementar la recursión en MINILISP, ya que el lenguaje carece de recursión directa a nivel del núcleo. El resultado de evaluar el combinador se guarda en la variable **z**, que se introduce en el ambiente inicial de evaluación. Así, los programas que usan **letrec** pueden implementar funciones recursivas correctamente.

El bucle principal del intérprete que, en un arrebato increíble de originalidad lo llamamos **minilisp**, es donde leemos la entrada del usuario, y la pasamos a una función **run** para que sea procesada. En este bucle antes de mandar la cadena a ser evaluada comprobamos que si es vacía o la cadena reservada para salir del intérprete.

La función **run** es la que coordina todas las fases de análisis, evaluación y muestreo al usuario. Desde iniciar el proceso pasando como argumento la cadena recibida al **Lexer**, como su evaluación en AST (con **EvalS**) dentro de un ambiente inicial que contiene la definición del combinador **Z**, necesario para la recursión. Esta función mete de inicio a ese ambiente, la variable **Z** con su respectiva evaluación. **run** se ejecuta dentro de un bloque **catch** permite capturar cualquier excepción que ocurra durante el análisis o la evaluación, evitando que el intérprete se detenga ante un error. En su lugar, se muestra un mensaje informativo en pantalla y el programa continúa su ejecución de forma segura.

Finalmente, para la visualización de los resultados, nos hace falta explicar la función **saca**. Esta función es una función auxiliar que se encarga de mostrar el resultado de la evaluación al usuario. Esto es necesario porque los valores evaluados en MINILISP se representan mediante constructores internos del tipo **ASV**, los cuales no son legibles directamente. La función **saca** transforma estos valores en una representación textual clara y amigable para el usuario.

La función **saca** queda implementada en el archivo **Saca.hs** como sigue:

```

1 module Saca where
2
3 import ASV
4
5 --Funcion para obtener el resultado e imprimirlo como cadena y no como
   tipo de dato ASV
6 saca :: ASV -> String
7 saca (NiV) = "[]"
8 saca (NumV n) = show n
9 saca (BoolV b)
10   | b == True = "#t"
11   | otherwise = "#f"
12 saca (ClosureF p c e) = "#<procedure>"
13 saca (ConV f s) = "[" ++ sacaElems (ConV f s) ++ "]"
14 saca (PairV f s) = "(" ++ saca f ++ "," ++ saca s ++ ")"
15 saca _ = "#<unknown>"
16

```

```

17 -- Funcion auxiliar para recorrer ConV
18 sacaElems :: ASV -> String
19 sacaElems NiV = "[]"
20 sacaElems (ConV x xs) = saca x ++ ", " ++ sacaElems xs
21 sacaElems x = saca x

```

Código 7.2: Procedimiento `saca` para mostrar el resultado al usuario

Nos apoyamos de *pattern matching* para proceder con los casos correspondientes, estos son solo los valores canónicos:

- `NiV`: devolvemos la lista vacía y nada más.
- `NumV`: devolvemos el valor n que guarda `NumV`, este ya es un número en Haskell y utilizamos `show` para mostrarlo en su representación de salida.
- `BoolV`: comprobamos que valor tiene b . Si es `True` devolvemos la cadena `"#t"`, en otro caso `"#f"` para `False`.
- `ClosureF`: si el valor es una una cerradura, no se imprime su contenido interno sino que se representa como `#<procedure>`. Esto es un indicador textual, no un valor de lenguaje; se usa para que el usuario sepa que el valor es una función, pero no puede imprimirse directamente, no es un valor real del lenguaje, sino una representación simbólica para el usuario.
- `ConV`: cuando caemos en este valor, quiere decir que debemos representarlo como listas. Para ello nos auxiliaremos en la función `sacaElems` para formar una cadena con el formato adecuado y así devolver una representación fiel de los elementos.

Esta función `sacaElems` recorre el encadenamiento de `ConV`, si hay dos elementos representamos estos recursivamente con la función `saca` separados por comas, cuando hemos llegado al último elemento, lo regresamos tal cual.

- `PairV`: es más simple que `ConV`, solo es representar dos elementos recursivamente con `saca` entre paréntesis y separados por una coma.
- Si el valor canónico `ASV` recibido no es ninguno de los anteriores, entonces este no es válido, cosa que no debería ni puede suceder pero lo ponemos por completitud.

Ya con esto, veamos los resultados finales del lenguaje MINILISP -0.2cm:

```

[MiniLisp]> (+ (* 2 3 -1) (- 10 4 9 -22) (sqrt 16) (expt 36) (/ 1 2))
1331

```

Comprobemos arugmento por argumento que la expresion anterior es correcta:

```
[MiniLisp]> (* 2 3 -1)
-6
[MiniLisp]> (- 10 4 9 -22)
37
[MiniLisp]> (sqrt 16)
4
[MiniLisp]> (expt 36)
1296
[MiniLisp]> (/ 1 2)
0
[MiniLisp]>
```

Al sumar los resultado se puede apreciar que los valores coinciden.

```
[MiniLisp]> (+ -6 37 4 1296 0)
1331
```

Veamos otro ejemplo:

```
[MiniLisp]> (let ((y 31) (x -22)) (cond [(< x 0 y) (- 0 x y)] [(= x 0) 0]
                                         [else (+ x 1)]))
53
```

Aquí, al resolver paso por paso, tenemos que se asigna 31 a y y -22 a x , por lo que la primer comprobación debería ser correcta al sustituir las variables:

```
[MiniLisp]> (< -22 0 31)
#t
```

Por lo que se resuelve su cláusula:

```
[MiniLisp]> (- 0 -22 31)
53
```

Por último veamos un ejemplo de evaluación para listas, se espera que cada elemento de la lista se evalúe, regresando una lista con valores evaluados. Hacemos el recordatorio también de que nuestras listas son heterogéneas:

```
[MiniLisp]> [[1, 2], (3, 4), (not #t), (+ 1 2 3 4 5 6), (!= 9 7 3 5),
             [], (/ 2 4), (not (not (= 9 9))), (tail [1, 2, 3]),
             ((lambda (f x) (f x)) (lambda (y) (* y y)) 4)]
[[1, 2], (3,4), #f, 21, #t, [], 0, #t, 3, 16]
```

Al evaluar cada elemento por separado, se puede apreciar que estos resultados coinciden:

```

[MiniLisp]> [1, 2]
[1, 2]
[MiniLisp]> (3, 4)
(3,4)
[MiniLisp]> (not #t)
#f
[MiniLisp]> (+ 1 2 3 4 5 6)
21
[MiniLisp]> (!= 9 7 3 5)
#t
[MiniLisp]> []
[]
[MiniLisp]> (/ 2 4)
0
[MiniLisp]> (not (not (= 9 9)))
#t
[MiniLisp]> (tail [1, 2, 3])
3
[MiniLisp]> ((lambda (f x) (f x)) (lambda (y) (* y y)) 4)
16

```

En el archivo `README.md` se incluyen ejemplos más detallados para cada expresión.

7.3. Funciones de prueba

En esta sección implementamos tres funciones especiales: la suma de los primeros n números naturales, el factorial de un número y el n -ésimo número de Fibonacci. Estas funciones nos sirven como ejemplos prácticos para probar las capacidades del lenguaje MINILISP -0.2cm y en especial, de su capacidades recursivas.

Debido a que cada una de estas funciones se define naturalmente mediante recursión, nos auxiliamos de la expresión `letrec`, con la cual definimos funciones recursivas dentro del lenguaje.

Para poder ejecutar estas funciones directamente desde el menú interactivo, modificamos la función `run` de nuestro archivo `MiniLisp.hs`. Agregamos lógica que detecta cuando la entrada del usuario comienza con alguna palabra clave: `fact`, `sum` o `fibo`.

```
fact(Int)  sum(Int)  fibo(Int)
```

Cuando se detecta uno de estos comandos, se genera dinámicamente una expresión MiniLisp equivalente que utiliza `letrec` y luego se evalúa normalmente.

```

1  -- Envuelve la evaluacion con manejo de errores
2  run :: String -> IO ()
3  run input =
4      catch
5          (do
6              expr <-

```

```

7      if "fact" `isPrefixOf` input
8      then return $ fact (read (last (words input)))
9      else if "sum" `isPrefixOf` input
10     then return $ sumSum (read (last (words input)))
11     else if "fibonacci" `isPrefixOf` input
12     then return $ fibo (read (last (words input)))
13     else return input
14
15     let tokens = lexer expr
16     let asa = parse tokens
17     let ast = desugar asa
18     let asv = evalS (ast) [("Z", z)]
19     putStrLn (saca asv))
20 errors

```

Código 7.3: Implementación de las funciones especiales de MINILISP

La función `isPrefixOf` de Haskell, nos permite verificar si la cadena introducida por el usuario comienza con un determinado prefijo. De esta forma, si el usuario escribe `fact 5`, el intérprete genera internamente el código `MiniLisp` equivalente y lo evalúa como si el usuario lo hubiese escrito explícitamente.

7.3.1. Suma primeros n números naturales

Para calcular la suma de los primeros n números naturales se utiliza una definición recursiva simple:

$$\text{sum}(n) = \begin{cases} 0, & \text{si } n=0, \\ n+\text{sum}(n-1) & \text{en otro caso} \end{cases}$$

La implementación en Haskell genera una expresión en MINILISP con `letrec` que define y ejecuta esta función:

```

1  --Generamos la suma de los primeros n numeros con letrec
2  sumSum :: Int -> String
3  sumSum n = "(letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum
    " ++ show n ++ "))"

```

Código 7.4: Función para obtener el factorial de un número con `letrec`

Algunos ejemplos de ejecución son:

```

[MiniLisp]> sum 15
120
[MiniLisp]> sum 5
15
[MiniLisp]> sum 50
1275
[MiniLisp]> sum 3
6

```

El intérprete convierte internamente esta entrada en la expresión:

```
(letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum 5))
```

7.3.2. Factorial

De manera análoga, para el factorial usamos la definición recursiva clásica:

$$\text{fact}(n) = \begin{cases} 1, & \text{si } n=0, \\ n \times \text{fact}(n-1) & \text{en otro caso} \end{cases}$$

La función generadora en Haskell construye la expresión correspondiente:

```
1 --Generamos la funcion factorial con letrec
2 fact :: Int -> String
3 fact n = "(letrec (fact (lambda (n) (if0 n 1 (* n (fact (- n 1)))))) (fact
  " ++ show n ++ "))"
```

Código 7.5: Función para obtener la suma de los primeros n números naturales con `letrec`

Algunos ejemplos de ejecución serían:

```
[MiniLisp]> fact 5
120
[MiniLisp]> fact 8
40320
[MiniLisp]> fact 3
6
[MiniLisp]> fact 7
5040
```

Donde la entrada `fact 7` por ejemplo, se traduce internamente como:

```
(letrec (fact (lambda (n) (if0 n 1 (* n (fact (- n 1)))))) (fact 5))
```

7.3.3. Fibonacci

La sucesión de Fibonacci se define recursivamente como:

$$\text{fibonacci}(n) = \begin{cases} 0, & \text{si } n=0, \\ 1, & \text{si } n=1, \\ \text{fibonacci}(n-1)+\text{fibonacci}(n-2) & \text{en otro caso} \end{cases}$$

Su implementación en Haskell genera el código MINILISP correspondiente con `letrec`:

```

1  --Generamos el n-esimo numero de Fibonacci con letrec
2  fibo :: Int -> String
3  fibo n = "(letrec (fib (lambda (n) (if0 n 0 (if0 (- n 1) 1 (+ (fib (- n 1))
    ) (fib (- n 2)))))) (fib " ++ show n ++ "))"
```

Código 7.6: Función para obtener el n-ésimo número de la sucesión de Fibonacci con con `letrec`

Como ejemplos de ejecución tenemos:

```

[MiniLisp]> fibo 12
144
[MiniLisp]> fibo 18
2584
[MiniLisp]> fibo 7
13
[MiniLisp]> fibo 2
1
```

El intérprete genera internamente la siguiente definición:

```
(letrec (fib (lambda (n)
```

```
(if0 n 0 (if0 (- n 1) 1 (+ (fib (- n 1)) (fib (- n 2)))))) (fib 7))
```

Estas funciones de prueba no sólo demuestran la expresividad del lenguaje donde el sistema de evaluación de MINILISP maneja correctamente la recursión, sino también que la implementación del combinador **Z** y las expresiones `letrec` permiten definir y evaluar funciones complejas sin necesidad de estructuras externas. Además, muestran cómo el menú interactivo puede extenderse para admitir comandos personalizados que simplifican la interacción del usuario con el intérprete.

Capítulo 8

Conclusiones

Reflexión

El desarrollo de **MINILISP** ha sido un ejercicio riguroso de diseño e implementación de un lenguaje de programación, cubriendo las etapas fundamentales de la teoría de lenguajes: desde la **Sintaxis Concreta** hasta la **Semántica Operacional**. La elección de **Haskell** y sus herramientas (**Alex** y **Happy**) fue muy útil, ya que la naturaleza funcional e inmutable del lenguaje anfitrión facilitó la construcción clara del modelo y la adaptación de conceptos abstractos, como los **Árboles de Sintaxis Abstracta (ASA)** y los estados finales.

El proyecto se caracteriza por un proceso de diseño bien estructurado que priorizó la claridad:

- **Análisis Formal:** La definición de la **Sintaxis Concreta** mediante la gramática **EBNF** sentó una base sólida para el análisis léxico y sintáctico.
- **Gestión de la Complejidad (Azúcar Sintáctica):** La etapa de **desazucarización** fue crucial, transformando construcciones expresivas pero complejas (como operadores variádicos, **let***, **cond** y la aplicación de funciones n-arias) en un núcleo (**AST**) más simple y homogéneo. Esto redujo drásticamente el número de reglas necesarias para la semántica, haciendo el intérprete más limpio y fácil de verificar. La correcta desazucarización de operadores variádicos y comparadores a estructuras binarias anidadas, manteniendo la consistencia de tipos, fue un punto crítico.
- **Estrategia Semántica:** La transición inicial de la **Semántica Estructural (Paso Pequeño)** a la implementación de **evaluación perezosa (Paso Grande o Big-Step)** con **Puntos Estrictos** fue esencial para manejar la recursión de forma correcta a través del constructo **letrec** y el **Combinador Z**, superando el problema de la evaluación glotona inicial.

Limitaciones Encontradas

A pesar de los logros, se identificaron áreas con limitaciones:

- **Tipado y Verificación:** El lenguaje **carece de un sistema de tipos explícito** que verifique la corrección de los programas antes de la ejecución. Errores de tipo sólo se

detectan durante la evaluación semántica (por ejemplo, intentar operar con un valor no numérico).

- **Pares y Listas:** Aunque se depuró la ambigüedad en la sintaxis de listas utilizando **Cons** y **Nil** frente a un simple encadenamiento de pares, la implementación del **TailV** en el intérprete requirió una función auxiliar (**tailDeep**) para forzar la evaluación perezosa en el segundo elemento de la lista anidada, lo que añade complejidad.
- **Ausencia del Combinador Z Explícito:** Si bien el concepto del **Combinador Z** fue fundamental para habilitar **letrec**, la solución final lo integra de manera implícita en la regla de **letrec** para la evaluación perezosa, sin una representación formal completamente visible en el ambiente como un valor canónico evaluado.

Posibles Extensiones Futuras

El diseño modular de MINILISP permite varias extensiones:

- **Sistema de Tipos Estático (Type Checker):** Implementar un **Type Checker** para verificar la coherencia del programa antes de la evaluación. Esto podría basarse en reglas de inferencia para un subconjunto del lenguaje, mejorando la robustez y la detección temprana de errores.
- **Manejo de Cadenas (Strings):** Extender el alfabeto y los tokens para soportar operaciones con cadenas de caracteres.

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_1.htm

Bibliografía

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

Bibliografía

- [1] *Especificación Formal de los Lenguajes de Programación. Sintaxis Concreta*, (M.Soto, lenguajes de programación, 2025).
- [2] *Introduction to Automata Theory, Languages, and Computation*, (Hopcroft y Ullman).
- [3] Landin.P.J.(1965) *The Next 700 Programming Languages*.Univac Division of Sperry Rand Corp,9(3),157-166.
- [4] Winske.G.(1993) *The Formal Semantics of Programming Languages*, Massachusetts Institute of Technology.
- [5] Plotkin.G.D.(1981) *A Structural Approach to Operational Semantics*, University of Aarhus.
https://lambdasspace.github.io/LDP/notas/ldp_n14.pdf