Universidad Nacional Autónoma de México

Facultad de Ciencias

Lenguajes de Programación

Proyecto 1:MiniLisp

Integrantes:

Elizalde Maza Jesús Eduardo
Navarro Fierro Michelle Alanis
Peredo López Citlali Abigail

Profesor: Manuel Soto Romero

Diego Méndez Medina
José Alejandro Pérez Márquez
Erick Daniel Arroyo Martínez
Mauro Emiliano Chávez Zamora

Fecha: 5 de noviembre de 2025

CDMX

Índice

1. Introducción

En la actualidad, los lenguajes de programación constituyen una de las herramientas más fundamentales en el campo de las Ciencias de la Computación, ya que no solo permiten escribir código, sino que representan el medio mediante el cual los seres humanos comunican instrucciones complejas a las computadoras. A través de ellos es posible expresar algoritmos e ideas de forma precisa y estructurada, impulsando el desarrollo de sistemas operativos, aplicaciones móviles, inteligencia artificial y una amplia gama de tecnologías que forman parte esencial de la vida cotidiana. En términos generales, un lenguaje de programación puede definirse como un conjunto de reglas y símbolos que posibilitan la escritura de instrucciones comprensibles para la máquina, permitiendo especificar tareas y procesos de manera eficiente.

La historia de los lenguajes de programación refleja la búsqueda constante por lograr una mayor abstracción, eficiencia y correctitud en la comunicación entre humanos y computadoras. En sus inicios, la programación se realizaba mediante lenguajes de bajo nivel, como el código de máquina y el ensamblador, los cuales resultaban difíciles de leer y propensos a errores. La aparición de lenguajes de alto nivel como FORTRAN y COBOL marcó un punto de inflexión, al permitir a los programadores centrarse en la lógica del problema en lugar de los detalles de la arquitectura del hardware.

Posteriormente, la evolución de los lenguajes dio lugar a diversos paradigmas que transformaron la manera de desarrollar software. La programación estructurada, con lenguajes como Pascal y C, promovió la organización del código en bloques lógicos, reduciendo la complejidad y mejorando la legibilidad. Más adelante, la programación orientada a objetos, representada por C++, Java y Smalltalk, introdujo conceptos como la encapsulación, la herencia y el polimorfismo, permitiendo modelar problemas del mundo real de forma más intuitiva y fomentando la creación de sistemas modulares y reutilizables. En paralelo, la programación funcional, impulsada por lenguajes como Lisp, Haskell y Scala, cobró relevancia por su capacidad para manejar la concurrencia y el procesamiento de datos, gracias a principios como la inmutabilidad y las funciones puras, que facilitan el razonamiento y las pruebas del código.

En la actualidad, la disciplina avanza hacia entornos cada vez más paralelos y concurrentes. Lenguajes contemporáneos como Go y Rust han sido diseñados con la seguridad y la eficiencia como ejes centrales:

- Go simplifica la concurrencia mediante goroutines y canales.
- Rust garantiza la seguridad de la memoria sin necesidad de un recolector de basura, consolidándose como una opción robusta para sistemas de alto rendimiento.

Esta evolución constante demuestra que los lenguajes de programación no solo han acompañado el progreso tecnológico, sino que han sido motores fundamentales en la transformación de la informática moderna, respondiendo a la necesidad de abordar problemas cada vez más complejos y especializados.

En este contexto, el presente proyecto se enmarca en el estudio formal de los lenguajes de programación mediante la extensión de un lenguaje de estilo Lisp denominado MiniLisp, implementado en Haskell con apoyo de la herramienta Happy. Este trabajo busca recorrer las etapas esenciales de la formalización de un lenguaje, desde la definición de su sintaxis léxica y libre de contexto hasta la especificación de su semántica operacional, con el propósito de integrar la teoría y la práctica en el diseño de intérpretes. De esta manera, el proyecto MiniLisp constituye una oportunidad para aplicar los fundamentos teóricos de

los lenguajes de programación en un entorno funcional real, fortaleciendo la comprensión de los principios que rigen su construcción, análisis e implementación.

2. Objetivos

El presente proyecto tiene como propósito definir formalmente la sintaxis léxica y libre de contexto del lenguaje MiniLisp, por lo que se propone el lenguaje Lambdae que es una extensión del mismo. Asimismo se formula su representación en sintaxis abstracta e incorporando mecanismos de azúcar sintáctica y su correspondiente eliminación hacia un núcleo mínimo. Por otra parte, se busca diseñar y especificar la semántica operacional estructural del lenguaje mediante reglas de inferencia y derivaciones que describan con precisión su comportamiento durante la evaluación, al tiempo que se modelan ambientes de evaluación y bindings que garanticen la correcta gestión de alcances, valores y consistencia semántica.

Con el fin de ampliar la expresividad del lenguaje, se propone incorporar nuevos operadores aritméticos, predicados, estructuras de datos, listas, condicionales y funciones variádicas, sustentando formalmente cada decisión de diseño. De igual manera, se plantea la implementación de un intérprete funcional completo de MiniLisp en Haskell, estructurado a partir de un pipeline modular que abarque las fases de análisis léxico, sintáctico, desazucarización y evaluación, asegurando una correspondencia directa entre la teoría formal y la ejecución práctica.

Finalmente, se busca demostrar la validez del modelo propuesto mediante casos de prueba y ejemplos representativos que evidencien la relación entre la formalización teórica y el comportamiento del sistema implementado, así como integrar y documentar los resultados en un informe académico que articule los fundamentos teóricos, las decisiones de diseño y los resultados experimentales, promoviendo una comprensión crítica del vínculo entre teoría y práctica en el estudio de los lenguajes de programación.

3. Sintaxis de un lenguaje de programación

En el diseño de los lenguajes de programación, establecer reglas claras que indiquen cómo deben escribirse y estructurarse los programas es un aspecto esencial. Estas reglas no sólo permiten que las máquinas comprendan las instrucciones, sino que también aseguran que los desarrolladores puedan comunicarse de manera precisa y coherente con ellas. Sin una estructura formal que determine qué expresiones son válidas y cómo deben organizarse, sería imposible garantizar que un programa sea interpretado de la misma manera por distintos compiladores o intérpretes. Además, esta estructura facilita la detección de errores, la lectura del código y la creación de herramientas de análisis y depuración. En este contexto surge la necesidad de definir con rigor la forma en que los programas deben escribirse, dando origen al concepto fundamental de la sintaxis en los lenguajes de programación.

Definición: Sintaxis

Definimos, de manera informal, la sintaxis de un lenguaje de programación como el conjunto de reglas y estructuras que especifican cómo deben escribirse los símbolos y palabras reservadas de un programa para que sea aceptado por el traductor del lenguaje.

3.1. Sintaxis concreta

La sintaxis concreta de un lenguaje de programación define la apariencia y estructura exacta con la que deben escribirse los programas, abarcando todos los elementos visibles como palabras clave, operadores, signos de puntuación y la forma en que estos se organizan.Por lo que podeso definir a la sintaxis concreta de la sigueinte fora:

Definición: Sintaxis concreta

La sintaxis concreta se puede definir como un par **(L, G)** donde:

- **L** es la definición léxica representada por el conjunto de expresiones regulares **R**.
- **G** es la gramática libre de contexto **(N, , P, S)**.

Dicha sintaxis se especifica mediante una gramática formal que determina, qué combinaciones de símbolos son válidas dentro del lenguaje.

Esta descripción se divide normalmente en dos niveles:

- Sintaxis léxica.
- Sintaxis estructural o gramática libre de contexto.

A continuación abordaremos un poco más a fondo cada uno de estos conceptos.

3.1.1. Sintaxis Léxica

Los elementos léxicos de un lenguaje de programación conforman el conjunto de símbolos, palabras reservadas y reglas de formación que determinan la estructura básica

mediante la cual se construyen las expresiones y sentencias del lenguaje. Este conjunto de caracteres reconocidos por el compilador o intérprete recibe el nombre de tokens, los cuales representan las unidades léxicas fundamentales del lenguaje. Dichos tokens pueden corresponder a símbolos de puntuación, operadores, representaciones numéricas, cadenas de texto o identificadores de variables.

Desde un punto de vista formal, la descripción léxica de un lenguaje se fundamenta en ciertos elementos esenciales:

- **Alfabeto (Σ):** se refiere al conjunto limitado de símbolos que pueden utilizarse en el lenguaje, como letras, números y signos especiales.
- **Tokens:** son secuencias de caracteres derivadas del alfabeto que representan las unidades sintácticas más simples. Cada clase de token se especifica mediante una expresión regular que determina su estructura o patrón.

Definición: Expresiones regulares

Una expresión regular E, definida sobre un alfabeto Σ, se construye de manera recursiva siguiendo las reglas siguientes:

- ε, que representa la cadena vacía, es considerada una expresión regular.
- Para cualquier símbolo $a \in \Sigma$, dicho símbolo constituye por sí mismo una expresión regular que representa una cadena formada únicamente por a.
- Si E_1 y E_2 son expresiones regulares, entonces su concatenación E_1E_2 también forma una expresión regular.
- De igual manera, si E_1 y E_2 son expresiones regulares, su unión $E_1 + E_2$ se considera una expresión regular.
- Si E es una expresión regular, su cerradura de Kleene E^* también lo es, y describe cualquier número de repeticiones de E, incluyendo la posibilidad de que no aparezca.

En consecuencia, para conformar el conjunto de tokens de un lenguaje, el analizador léxico recibe una secuencia de cadenas de entrada y las transforma en un conjunto de tokens válidos conforme a las reglas léxicas previamente definidas para dicho lenguaje.

El conjunto de tokens aceptados por nuestro lenguaje puede entenderse como una lista de unidades léxicas generada a partir de la lectura secuencial del programa de entrada, carácter por carácter. Durante este proceso, el analizador léxico identifica secuencias de caracteres que coinciden con alguno de los patrones o tokens previamente definidos en el lexer, y cada vez que se reconoce una coincidencia, se genera un nuevo token que se añade a la lista. De esta manera, al finalizar la lectura completa del programa, se obtiene un conjunto estructurado de tokens que representa de forma simbólica el contenido del código fuente.

A continuación, presentamos la especificación léxica del lenguaje LAMBDAE, definiendo formalmente su alfabeto y los principales tipos de tokens utilizados.

Sintaxis léxica de LAMBDAE

Para comenzar a definir la sintaxis léxica comenzaremos definiendo el alfabeto de nuestro lenguaje, el cual está conformado por lo siguiente:

Σ = { *True* , *Fasle*, +, - , * , / , *sqrt,expt,not*, (,) , [,] , = , >, <, >= ,<= ,!= , if , fst , *snd* , *let* , *let** , *letrec* , *lamdba* , *head* , *tail* , *pair* , *cond* , *add1* , *sub1* }

En nustra gramática también tenemos definidos a los enteros y a las variables. A las varibales las identificaremos como *id* y a los números como *int*, a ambos los podemos definir de la siguiente manera:
id ::= [A-Za-z_][A-Za-z0-9_]*
int ::= 0 | [1-9][0-9]*
Paréntesis
(+)
Se utilizan para agrupar subexpresiones lógicas. Al igual que en lenguajes de notación matemática o lógica simbólica, permiten definir la precedencia de los operadores.

En este punto, los tokens del lenguaje representan las unidades léxicas construidas a partir del alfabeto definido. Para esta versión del lenguaje, se incluyen los siguientes tipos de tokens, junto con las expresiones regulares que especifican la estructura de cada uno.

Tokens en LAMBDAE

```
int             { TokenNum $$ }
bool            { TokenBool $$ }
var             { TokenVar $$ }
'+'             { TokenSuma }
'-'             { TokenResta }
'*'             { TokenMult }
'/'             { TokenDiv }
"sqrt"          { TokenSqrt }
"expt"          { TokenExpt }
"not"           { TokenNot }
'('             { TokenPA }
')'             { TokenPC }
'['             { TokenCA }
']'             { TokenCC }
','             { TokenComa }
"in"            { TokenIn }
'='             { TokenEq }
'<'             { TokenMenor }
'>'             { TokenMayor }
'<='            { TokenMenorIgual }
'>='            { TokenMayorIgual }
'!='            { TokenDistinto }
"if"            { TokenIf }
"fst"           { TokenFst }
"snd"           { TokenSnd }
"let"           { TokenLet }
"let*"          { TokenLetStar }
"letrec"        { TokenLetRec }
"lambda"        { TokenLambda }
"head"          { TokenHead }
"tail"          { TokenTail }
"pair"          { TokenPair }
"cond"          { TokenCond }
"add1"          { TokenAdd1 }
"sub1"          { TokenSub1 }
```

Es por esto que necesitamos de herramientas formales y notaciones que nos permitan especificar la sintaxis eliminando ambigüedades. Algunas de éstas son:

- Gramáticas fromales (GLC)
- Árboles de sintaxis
- Analizadores léxicos y sintácticos

3.2. Gramática Formal: gramática libre de contexto (GLC)

Para iniciar, es importante definir todos los conceptos. En el ámbito de la teoría de los lenguajes de programación y de los lenguajes formales, la sintaxis concreta hace

referencia a la estructura específica de un lenguaje de programación, la cual determina la forma exacta en que deben escribirse los programas. Desde un punto de vista matemático, dicha estructura se describe mediante una gramática libre de contexto, la cual se define de la siguiente manera:

Definición: Gramática libre de contexto

Formalmente, una **Gramática Libre de Contexto (GLC)** se define como una tupla

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

donde:

- N es un conjunto finito de símbolos no terminales. Estos representan categorías sintácticas intermedias y permiten construir la jerarquía estructural del lenguaje.
- Σ es un conjunto finito de símbolos terminales, que corresponden a los tokens generados por el análisis léxico.
- P es un conjunto finito de producciones de la forma $A \to \alpha$, donde $A \in N$ y $\alpha \in (N \cup \Sigma)^*$.
- $S \in N$ es el símbolo inicial, desde el cual se derivan todas las cadenas válidas del lenguaje.

Cada uno de estos componentes tiene un papel específico en la descripción de la sintaxis:

1. **Terminales:** representan los elementos básicos del lenguaje que no pueden ser descompuestos más allá dentro de la gramática.En nuestro caos los terminales son los valores booleanos (*bool*), enteros (*int*), pares ordenados (*pair*) y clousures (*Clousure*).
2. **No terminales:** estos corresponden a variables sintácticas que describen cómo se organizan y combinan los diferentes componentes del lenguaje. En nuestra gramática, los no terminales principales serán expresiones que se pueden generar a partir de los elementos definidos ene el alfabeto de nuestra gramática, pueden ser binarias y variadicos como lo son: (*Add,Sub,Mul,Div,..*), funciones(*App, Fun, lambda,...*), condicionales(*if*), operaciones sobre pares ordenados(*Fst,Snd*), listas (*head, tail*), aplicaciones *let*, comparaciones, entre otras.
3. **Símbolo inicial:** representa el elemento desde el cual se generan todas las expresiones válidas del lenguaje.
4. **Producciones:** son las reglas mediante las cuales se obtienen expresiones más complejas a partir de otras más simples. Estas permiten describir de forma recursiva cómo se combinan operadores, constantes y subexpresiones, dando origen a la sintaxis del lenguaje.

GLC de LAMBDAE

```
< expr > ::= consB
        |  consE
        |  consV
        | (¬ < expr >)
        | (< binop > < expr > < expr >)
        | (<unop> <expr>)

<binop> ::= +
         | -
         | *
         | /
         | =
         | <
         | >
         | <=
         | >=
         | pair

<unop>  ::= not
         | null?
         | sqrt
         | expt
         | fst
         | snd
         | head
         | tail
         | cond
         | add1
         | sub1
Construcción que representa las constantes booleanas:
<consB > ::= True
        | False
Construcción que representa las constantes numéricas:
< consE > ::= 1 | 2 | 3| ...
Construcción que representa las constantes variádicas:
< consV > ::= a|b|c|...| z
            |identificadores alfanuméricos
```

Utilizamos la construcción *unop* para expresiones que tienen un solo argumento.
Construcciones especiales:

$$\langle lambda \rangle ::= \text{lambda}$$
$$\langle condicional \rangle ::= \text{if}$$
$$\langle let \rangle ::= \text{let}$$
$$\langle letRec \rangle ::= \text{letRec}$$
$$\langle lista \rangle ::= [\,]$$
$$\langle aplicacion \rangle ::= \text{app}$$

Es importane resaltar que en neustro lengauje LAMBDAE utilizamos notación prefija, las operaciones se evaluan de forma binaria apesar de que se puedan tomar de forma n-aria.

Clasificación de Construcciones

- **Operadores unarios:** Operaciones que reciben exactamente un argumento: `not`, `sqrt`, `fst`, `snd`, `head`, `tail`, `add1`, `sub1`, `null?`
- **Operadores binarios:** Operaciones que reciben dos o más argumentos (con asociatividad izquierda para múltiples argumentos): `+`, `-`, `*`, `/`, `=`, `<`, `>`, `<=`, `>=`, `!=`, `pair`
- **Abstracciones (lambda):** Construcción que crea funciones anónimas. Recibe un parámetro (variable) y un cuerpo (expresión): `(lambda <var><expr>)`
- **Condicionales:** Construcción de control de flujo que evalúa una condición y selecciona entre dos ramas: `(if <cond><then><else>)`
- **Vinculaciones locales (let):** Construcciones que introducen variables locales en un ámbito:
  - `let`: Vinculación simple no recursiva
  - `letRec`: Vinculación recursiva para definir funciones recursivas
- **Listas:** Construcción de estructuras de datos secuenciales mediante corchetes: `[e1 e2 ... en]`
- **Aplicación de funciones:** Construcción que aplica una función (que puede ser una lambda, variable, o expresión que evalúa a función) a uno o más argumentos: `(f arg1 arg2 ... argn)`

Notación

- $\langle expr \rangle^*$ indica cero o más repeticiones de $\langle expr \rangle$
- $\langle expr \rangle^+$ indica una o más repeticiones de $\langle expr \rangle$

Para definir la gramática formal utilizamos una gramática libre de contexto que siga la Forma Normal Extendida de Backus-Naur(en inglés Extended Backus–Naur Form o EBNF) que es una notación formal que se utiliza para describir la gramática de un lenguaje de programación (o cualquier lenguaje formal). Para esto comencemos dando un breve contexto.

Notación BNF

Antes de que tuvieramos la notación EBNF, se tenía la BNF(Backus-Naur Form) propuesta por John W. Backus en 1959, la cual surge ante la necesidad de tener una notación formal para describir la sintaxis de lenguajes de programación.Una de sus características principales es su capacidad de definir alternativas mediante el uso del operador |.

La forma BNF introdujo una forma clara y formal de especificar la sintaxis de un lenguaje (producciones, no terminales, terminales, alternancias). Esto permitió que las especificaciones fueran menos ambiguas y más aptas para que herramientas automáticas (analizadores sintácticos / parsers) las usaran.

En nuestro caso no haremos tanta énfasis en este tipo denotación pues no la utililizamos para definir nuestra gramática.

Componentes básicos de BNF

En su forma básica, una regla en BNF se representa mediante una producción con el símbolo ::= y puede incluir alternativas mediante el operador |. Las variables (o símbolos no terminales) suelen escribirse entre signos de mayor y menor (<>), mientras que los símbolos terminales (como palabras reservadas u operadores) se escriben tal cual aparecen en el lenguaje.

3.2.1. Notación EBNF

En 1970 Niklaus Wirth propuso notaciones y promovió una forma de extender BNF con construcciones para iteración, opcionalidad y agrupamiento, que se conocen comúnmente como EBNF (Extended Backus–Naur Form). El trabajo de Wirth de 1977 es un hito en la unificación/práctica de una notación extendida y legible.

Fue desarrollada con el propósito de ofrecer una forma más concisa, legible y expresiva de describir la sintaxis de lenguajes formales, incorporando mecanismos para representar repeticiones, secuencias opcionales y agrupaciones. Estas capacidades permiten expresar reglas sintácticas complejas de forma más clara, mejorando tanto la comprensión como la implementación de analizadores sintácticos. Formalmente, la notación EBNF introduce los siguientes elementos adicionales:

- **Repetición:** se expresa mediante llaves , indicando que una secuencia puede repetirse cero o más veces.
- **Opcionalidad:** se representa con corchetes [], permitiendo indicar partes opcionales de una producción.
- **Agrupación:** se realiza mediante paréntesis (), útiles para controlar la precedencia o delimitar alternativas.
- **Terminales:** se suelen escribir entre comillas para distinguirlos claramente de los no terminales.
- **Alternativas:** se siguen expresando con el operador |, heredado de BNF.

Sin embargo, al analizar la gramática del lenguaje MiniB presentada anteriormente, observamos que ninguna de estas construcciones adicionales es necesaria. La gramática se compone únicamente de combinaciones secuenciales y alternativas simples, sin necesidad de representar repeticiones, elementos opcionales ni agrupaciones complejas. Por tanto, al transformarla a EBNF, su estructura permanece idéntica.

Esta coincidencia no implica una limitación, sino que evidencia que la definición actual de MiniB es suficientemente simple como para que la notación BNF ya proporcione una representación clara y completa. En consecuencia, la forma EBNF de nuestra gramática resulta ser estructuralmente equivalente a la forma BNF previamente presentada.

Gramática en EBNF de LAMBDAE

```
<S > := < Expr >
<Expr > ::= <int >
      | <bool>
      | <var>
      | (not <Expr>)
      | (+ <Expr> <Expr> {<Expr>})
      | (- <Expr> <Expr> {<Expr>})
      | (/ <Expr> <Expr> {<Expr>})
      | (* <Expr> <Expr> {<Expr>})
      | (= <Expr> <Expr> {<Expr>})
      | (< <Expr> <Expr> {<Expr>})
      | (> <Expr> <Expr> {<Expr>})
      | (<= <Expr> <Expr> {<Expr>})
      | (>= <Expr> <Expr> {<Expr>})
      | (!= <Expr> <Expr> {<Expr>})
      | (lambda <Expr> <Expr>)
      | (sqrt <Expr>)
      | (expt <Expr>)
      | (pair <Expr> <Expr>)
      | (fst  <Expr>)
      | (snd <Expr>)
      | (if <Expr> <Expr>  <Expr>)
      | (let <var> <Expr>)
      | (let* <var> <Expr> {<Expr>})
      | [{ Expr }]
      | (letRec )
      | (head <Expr>)
      | (tail <Expr>)
      | (cond <Expr>)*
      | (<Expr> [<Expr> <Expr>(,)*])
      | (add1 <Expr>)
      | (sub1 <Expr>)

<int > := <N >
     | -<M >
<D > := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<N > := 0 | <D >{ <N >}
<M > := <D >{ <N >}

<bool> := <#t, #f>
<T> := True
<F> := False

<var> :=<X>
<X> := <X> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
       n | o | p | q | r | s | t | u | v | w | x | y | z
```

Estilo de la gramática EBNF sacado de

Todos los operadores asocian hacia la izquierda, lo que permite omitir paréntesis en muchos casos sin ambigüedad. Estas convenciones simplifican tanto la escritura como el procesamiento de expresiones regulares en analizadores léxicos.

Semántica del Patrón de Repetición

Notación EBNF utilizada:

- `{...}` indica repetición cero o más veces
- `(...)+` indica repetición una o más veces

Evaluación de operadores binarios con repetición:

Para los operadores aritméticos y de comparación que aceptan múltiples argumentos mediante `{<Expr>}`, se utiliza **evaluación izquierda-asociativa**:

- `(+ e1 e2 e3 ... en)` se evalúa como `(+ (+ (+ e1 e2) e3) ... en)`
- `(- e1 e2 e3)` se evalúa como `(- (- e1 e2) e3)`
- `(* e1 e2 e3)` se evalúa como `(* (* e1 e2) e3)`
- `(/ e1 e2 e3)` se evalúa como `(/ (/ e1 e2) e3)`

Para los operadores de comparación:

- `(= e1 e2 e3 ... en)` verifica que `e1 = e2 AND e2 = e3 AND ... en-1 = en`
- Similar para `<`, `>`, `<=`, `>=`, `!=`

Listas:

- `[e1 e2 ... en]` construye una lista mediante reducción por cons: `(cons e1 (cons e2 ... (cons en nil)))`

Aplicación de funciones:

- `(f arg1 arg2 ... argn)` aplica la función `f` a los argumentos de izquierda a derecha

Let* secuencial:

- `(let* ((x1 e1) (x2 e2) ... (xn en)) body)` vincula las variables secuencialmente, donde cada expresión `ei` puede usar las variables `x1, ..., xi-1` previamente definidas

4. Sintaxis Abstracta

En la sintaxis concreta se especifican de forma minuciosa todos los elementos necesarios para construir las expresiones válidas del lenguaje. No obstante, esta descripción no es suficiente para capturar toda la organización y el comportamiento lógico de un programa. La sintaxis concreta incluye una gran cantidad de detalles, como símbolos de separación, espacios y paréntesis,que son indispensables para interpretar el código, pero que no aportan directamente a la comprensión de su estructura conceptual. De hecho, dichos elementos pueden volver más complejo el análisis y la manipulación del código, especialmente cuando se llevan a cabo procesos de optimización durante la traducción.

La sintaxis abstracta a diferencia de la anterior, esta ofrece una representación simplificada y estructural del código fuente, enfocada en capturar la lógica y la jerarquía del programa sin incluir los detalles superficiales propios de la sintaxis concreta. La forma más común de representarla es mediante un Árbol de Sintaxis Abstracta (ASA), el cual facilita tanto el análisis como la manipulación del código en las fases de compilación o interpretación.

Todos estos elementos que nos son relevantes o que no aportan directamente algo a la comprensión conceptual los podemos simplificar utilizando *azúcar sintáctica* la cual se encargará de eliminar estos componentes que no son necesarios y estandarizar todas las expreisones quqe sean admitidas por nuestro lenaguaje a una misma.

4.1. Árboles de sintaxis

Los árboles de sintaxis constituyen una herramienta fundamental en la representación estructurada de un programa, ya que permiten visualizar cómo se organizan y combinan los distintos elementos de acuerdo con las reglas del lenguaje. Mediante esta estructura jerárquica, es posible determinar la precedencia de los operadores y comprender de manera precisa cómo deben evaluarse las expresiones.

Una ventaja clave de estos árboles es que ayudan a eliminar posibles ambigüedades al interpretar el código. Si una gramática permite construir más de un árbol sintáctico para la misma cadena, se generan múltiples significados para un mismo programa, lo cual resulta problemático en un lenguaje de programación. Por ello, los árboles sintácticos sirven como evidencia de la claridad o ambigüedad de la gramática que define al lenguaje.

En los compiladores e intérpretes, estos árboles son el resultado directo de la fase de análisis sintáctico (parsing), en la cual el código fuente se transforma en una representación estructurada y más manejable. Esta representación se convierte en la base sobre la cual se realizan posteriores procesos como el análisis semántico y la traducción a código ejecutable.

Además, cuando se formaliza la sintaxis abstracta de un lenguaje como LAMBDAE, se logra una versión aún más depurada de esta estructura. La sintaxis abstracta prescinde de detalles decorativos de la notación concreta —como paréntesis o delimitadores— y se enfoca en capturar únicamente la lógica y las relaciones semánticas entre los componentes del programa. Gracias a ello, se facilita el desarrollo de herramientas como analizadores, traductores y optimizadores, que pueden operar sobre una representación más simple, eficiente y extensible.

En conjunto, los árboles de sintaxis y la sintaxis abstracta aportan una comprensión conceptual más clara del lenguaje y proporcionan los cimientos necesarios para la construcción de un sistema de compilación sólido, coherente y capaz de evolucionar con el

tiempo.

4.1.1. Árboles de sintaxis concreta

Un árbol de sintaxis concreta (ASC) representa visualmente cómo el símbolo inicial de una gramática puede generar una cadena del lenguaje. Por ejemplo, si un no terminal A posee una producción $A \to XYZ$, entonces en un ASC aparecerá un nodo interno marcado con A, del cual se desprenden tres nodos hijos etiquetados como X, Y y Z en ese orden.

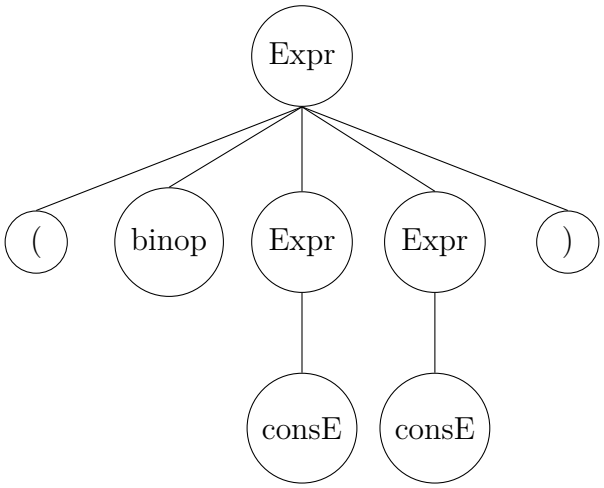
Desde una perspectiva formal, y considerando una gramática libre de contexto, un ASC asociado a dicha gramática es un árbol que cumple con las características siguientes: De manera formal, cuando se trabaja con una gramática libre de contexto, un árbol de sintaxis concreta (ASC) asociado a ella debe cumplir con las siguientes condiciones:

- La raíz del árbol lleva como etiqueta el símbolo inicial de la gramática.
- Todas las hojas están etiquetadas con símbolos terminales o, en su defecto, con ε (la cadena vacía).
- Cada nodo interno corresponde a un símbolo no terminal.
- Si un nodo interno está marcado con el no terminal A y sus hijos, de izquierda a derecha, están etiquetados como $X_1, X_2, \ldots, X_n$, entonces la gramática debe contener una producción de la forma
$$A \to X_1 X_2 \ldots X_n.$$

En este modelo, un programa puede representarse mediante un ASC que conserva todos los elementos presentes en el código fuente (como operadores, paréntesis y demás detalles sintácticos), reflejando paso a paso la derivación completa definida por la gramática.

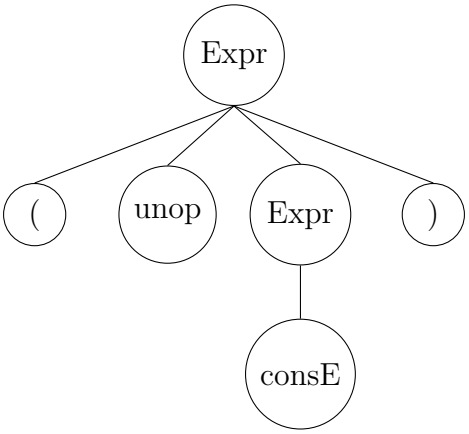
Consideremos el siguiente árbol de sintaxis concreta (ASC) del lenguaje LAMBDAE que represneta las expresiones binarias:

Esta expresión se deriva usando la gramática presentada. El ASC ilustra la derivación de la expresión, mostrando de forma explícita cada símbolo no terminal y terminal (paréntesis, operador y valores):



Los árboles de sintaxis concreta de -, * , / , = , <, >, <= , >= , != son análogos a este. La cantidad de hijos de nuestro árbol dependerá de la cantidad de argumentos de entrada que tegamos.

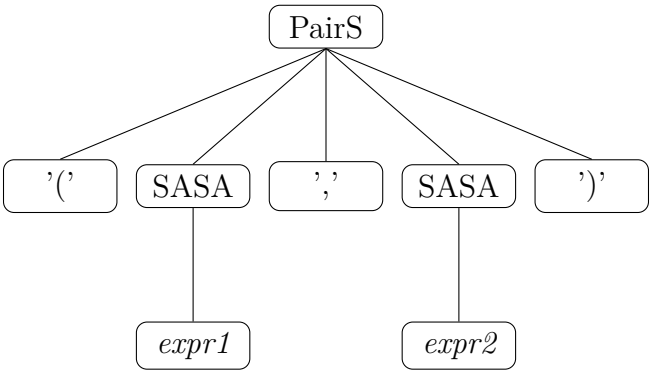
Ahora pasemos con los ASC que son unarios(*unop*), los cuales serían de la siguiente forma.



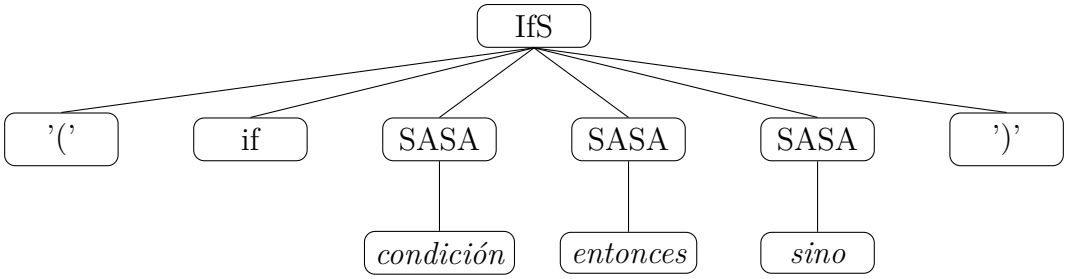
Con este tipo de árbol estamos representando a las expresiones que sepueden generar con *null , sqrt ,expt ,fst ,scnd ,cond ,head ,tail ,add1 , sub1* ,por lo que son análogos.

Ahora veamos casos más concretos:

- **ASC de par con coma:**



- **ASC de if:**



En los árboles obtenidos podemos observar que:

- La raíz está etiquetada como expr, derivando la estructura completa.
- Se incluyen todos los elementos sintácticos, como los paréntesis, operadores y .
- Cada nodo refleja una regla aplicada de la gramática, de forma que la estructura del árbol corresponde a la derivación concreta de la expresión.

A partir de estas bases teóricas, procederemos a definir los Árboles de Sintaxis Concreta correspondientes a las distintas construcciones que forman parte de nuestro lenguaje de programación. Para ello, presentaremos ejemplos representativos de expresiones válidas dentro del lenguaje y su respectiva interpretación mediante un ASC. Este enfoque nos permitirá ilustrar cómo las reglas gramaticales se reflejan de forma explícita en la estructura jerárquica del árbol, mostrando con claridad la combinación y precedencia de los elementos sintácticos que intervienen en cada expresión. De esta manera, se establecerá una correspondencia precisa entre la especificación formal del lenguaje y su representación estructurada, lo cual será fundamental para etapas posteriores del análisis y procesamiento del código.

Con esta base, es posible avanzar hacia un nivel más abstracto de especificación: la sintaxis abstracta. En esta etapa, se eliminan los elementos puramente sintácticos que no afectan el significado semántico del programa, con el fin de modelar únicamente su estructura lógica esencial.

4.1.2. Árboles de sintaxis abstracta (ASA)

Un árbol de sintaxis abstracta (AST) es una estructura de datos de tipo árbol que representa la estructura sintáctica abstracta de un fragmento de código fuente en un lenguaje de programación.

Veamos los términos clave:

"Abstracta" significa que no incluye todos los detalles del texto fuente tal como estaban escritos, sino sólo los elementos relevantes para la estructura del programa (por ejemplo, se omiten paréntesis redundantes, llaves, puntos y comas cuando no cambian la estructura semántica).

Representa de forma jerárquica los constructores del lenguaje (como expresiones, sentencias, declaraciones) como nodos conectados. Por ejemplo, una operación binaria "a + b" se representa por un nodo "+" con dos hijos: "a" y "b".

Es el resultado habitual de la fase de análisis sintáctico (parser) en la construcción de compiladores o intérpretes, que transforma el código fuente en una estructura más manejable para el análisis posterior.

Definición: Árboles de sintaxis abstracta

Un árbol de sintaxis abstracta es un árbol ordenado $A = (N, A, R)$ donde:

- N es un conjunto finito de nodos que representan las construcciones del lenguaje mediante etiquetas, y las hojas representan sus respectivos valores.
- $A \subseteq N \times N$ es un conjunto de aristas dirigidas que conectan a los nodos.
- $R \in N$ es la raíz del árbol.

Así pues, el AST captura lo esencial de la estructura del programa, dejando de lado detalles de la representación textual que no afectan su significado.

En el caso del lenguaje LAMBDAE, los árboles de sintaxix abstracta serán los que se generarán una vez que quitemos el azúcar sintáctica de todas las expresiones de nuestro lenguaje. Podemos decir que los ASA que tendremos en nuestro lenguaje serán de la sigueinte forma:

La conexión entre la sintaxis concreta y la sintaxis abstracta puede describirse de forma rigurosa mediante la composición de los procesos de análisis léxico y análisis sintáctico, ya mencionados previamente en clase. En este proceso, el analizador léxico (*lexer*) recibe como entrada el código fuente y lo convierte en una lista de tokens, mientras que el analizador sintáctico (*parser*) toma dicha lista de tokens y genera una representación estructurada, como lo es un Árbol de Sintaxis Abstracta (ASA).

Descripción formal:

Sea S una cadena de texto proveniente del programa fuente. El *lexer* se encarga de producir una lista de tokens $[Token]$ aplicando una función de transformación definida como:

$$S \to [Token].$$

Cada token corresponde a una unidad léxica relevante del lenguaje: identificadores, operadores, palabras clave, entre otros.

A continuación, el *parser* utiliza esa lista $[Token]$ para construir una estructura sintáctica jerárquica —típicamente un ASA— mediante una función definida como:

$$[Token] \to ASA.$$

En consecuencia, la relación entre la sintaxis concreta y la sintaxis abstracta, que denotamos mediante φ, se define como la composición de ambas funciones:

$$\varphi = parser \circ lexer : S \to ASA.$$

5. Azúcar sintáctica

subsectionContexto Histórico

El azúcar sintáctica es un concepto fundamental en el diseño de lenguajes de programación que data de 1964, cuando el científico de la computación Peter J. Landin acuñó este término para describir la sintaxis superficial de un lenguaje similar a ALGOL [**?**]. Originalmente, Landin utilizó este concepto para referirse a características sintácticas que podían ser definidas semánticamente en términos de expresiones lambda más simples. Con el tiempo, lenguajes como CLU, ML y Scheme extendieron el término para referirse a cualquier sintaxis dentro de un lenguaje que pudiera ser definida en términos de un núcleo de construcciones esenciales.

Definición: Azúcar Sintáctica

En ciencias de la computación, el azúcar sintáctica se refiere a la sintaxis dentro de un lenguaje de programación que está diseñada para hacer las cosas más fáciles de leer o expresar [**?**]. Esta hace que el lenguaje sea "más dulce" para el uso humano: las cosas pueden expresarse con mayor claridad, de manera más concisa, o en un estilo alternativo que algunos programadores pueden preferir.
De manera más formal, el azúcar sintáctica se define como aquellas características opcionales en los lenguajes de programación que proporcionan múltiples formas de realizar tareas similares [**?**].
Son construcciones sintácticas cuyos efectos están definidos en términos de otras construcciones sintácticas ya existentes en el lenguaje. Los procesadores de lenguaje, incluyendo compiladores y analizadores estáticos, frecuentemente expanden estas construcciones "azucaradas" en sus equivalentes más verbosos antes de procesarlas, un proceso denominado "desazucarado" o *desugaring*.

Importancia

El azúcar sintáctica desempeña un papel crucial en el desarrollo moderno de software por diversas razones fundamentales:

Mejora de la legibilidad y expresividad. El azúcar sintáctica permite que el código se escriba de manera más natural e intuitiva, facilitando que los programadores comprendan rápidamente el propósito y el flujo del código [**?**]. Esto es particularmente valioso en proyectos grandes donde múltiples desarrolladores deben colaborar y entender código escrito por otros.

Reducción del código repetitivo. Al proporcionar formas más concisas de expresar operaciones comunes, el azúcar sintáctica elimina la necesidad de código repetitivo (*boilerplate*) y expresiones idiomáticas difíciles de descifrar. Esto no solo ahorra tiempo de desarrollo, sino que también reduce las oportunidades de introducir errores.

Abstracción de alto nivel. El azúcar sintáctica permite a los programadores trabajar en un nivel de abstracción más elevado, enfocándose en la lógica del problema en lugar de los detalles de implementación de bajo nivel. Esto facilita la escritura de código que se asemeja más al lenguaje del dominio del problema que al lenguaje de la máquina [**?**].

Simplificación de la implementación del compilador. Al implementar características del lenguaje en el *frontend* mediante desazucarado, se reduce la complejidad del *backend* del compilador, ya que no es necesario agregar nuevos tipos de nodos al árbol de sintaxis abstracta (AST) para cada nueva característica sintáctica.

Sin embargo, es importante reconocer que el azúcar sintáctica también presenta desafíos. La inclusión de múltiples características opcionales aumenta la complejidad de la especificación del lenguaje y puede causar problemas a medida que los programas crecen en tamaño y complejidad [**?**]. Algunos programadores, particularmente en la comunidad Lisp, consideran que estas características de usabilidad sintáctica son innecesarias o incluso frívolas, como lo expresó Alan Perlis en su famosa frase: "El azúcar sintáctica causa cáncer de los punto y coma".

A pesar de estas críticas, el azúcar sintáctica continúa siendo una herramienta valiosa en el diseño de lenguajes de programación modernos, equilibrando la necesidad de expresividad y facilidad de uso con los requisitos de simplicidad y uniformidad del lenguaje.

En el caso de LAMBDAE pasaremos de tener los SASA que son los definidos en la gramática, a tener los ASA.

5.1. Eliminación de azúcar sintáctica: el Proceso de Desugaring en LAMBDAE

Justificación Teórica del Desugaring

El proceso de desugaring es una técnica fundamental en el diseño e implementación de lenguajes de programación modernos. Como establece la literatura, los procesadores de lenguaje, incluyendo compiladores y analizadores estáticos, frecuentemente expanden las construcciones azucaradas en sus equivalentes más verbosos antes de procesarlas [**?**]. Este enfoque arquitectónico permite separar claramente la sintaxis superficial del lenguaje de su semántica esencial.

El desugaring es esencialmente una transformación de sintaxis a sintaxis: una función que mapea expresiones escritas en una sintaxis rica y conveniente hacia un conjunto reducido de construcciones primitivas [**?**]. Esta estrategia ofrece beneficios arquitectónicos significativos: al reducir el número de construcciones que deben ser procesadas por las etapas posteriores del compilador (evaluación, análisis de tipos, optimización), se simplifica dramáticamente la implementación de estos componentes [**?**].

En el contexto de nuestro lenguaje LAMBDAE, el proceso de desugaring se manifiesta como una transformación explícita entre dos representaciones de árbol sintáctico abstracto (AST): **SASA** (Syntactic Abstract Syntax tree with Sugar Added) y **ASA** (Abstract Syntax tree After desugaring). Esta distinción arquitectónica refleja el principio de que las características sintácticas convenientes deben ser eliminadas antes del procesamiento semántico [**?**].

SASA y ASA

La arquitectura de LAMBDAE implementa una separación clara entre la sintaxis superficial (SASA) y el lenguaje núcleo (ASA). Esta separación es consistente con las prácticas establecidas en el diseño de lenguajes funcionales, donde el azúcar sintáctica se define en términos de un núcleo de construcciones esenciales [**?**].

5.1.1. SASA: Sintaxis Rica con Azúcar Sintáctica

El árbol SASA es producido directamente por el parser y contiene todas las construcciones sintácticas convenientes que el lenguaje ofrece al programador. Estas construcciones incluyen:

1. **Operaciones n-arias**: Expresiones como `(+ e1 e2 ... en)` permiten operaciones sobre múltiples argumentos, eliminando la necesidad de anidar operaciones binarias.

2. **Funciones lambda con múltiples parámetros**: La sintaxis `(lambda (x y z) body)` es más natural que las funciones unarias anidadas.

3. **Construcciones let especializadas**: LAMBDAE distingue entre `let`, `let*` y `letrec`, cada una con semántica diferente para el alcance de variables.

4. **Azúcar sintáctica de conveniencia**: Expresiones como `(add1 e)` y `(sub1 e)` que son abreviaciones de operaciones más complejas.

5. **Expresiones condicionales múltiples**: La construcción `cond` que maneja múltiples casos de manera más elegante que `if` anidados.

Como observa Rathi en su análisis de compiladores, estas abstracciones sintácticas "hacen la programación más dulce" al permitir código más legible y conciso, pero complican las etapas posteriores del compilador si no se eliminan [**?**].

5.1.2. ASA: Lenguaje Núcleo Desugared

El árbol ASA representa el lenguaje núcleo de LAMBDAE, conteniendo únicamente las construcciones primitivas necesarias para la evaluación. Este núcleo se caracteriza por:

1. **Operaciones binarias exclusivamente**: Todas las operaciones n-arias se reducen a composiciones de operaciones binarias mediante `foldl1` o `foldr1`.

2. **Funciones lambda unarias**: Toda función lambda acepta exactamente un parámetro. Las funciones multi-parámetro se representan mediante *currying*.

3. **Aplicación de función binaria**: La aplicación de función es estrictamente binaria, siguiendo el modelo del cálculo lambda.

4. **Eliminación de construcciones derivadas**: Expresiones como `let`, `let*` y `cond` se traducen a combinaciones de `lambda`, aplicación de función, e `if`.

Esta reducción a un núcleo mínimo es consistente con el principio de que "el azúcar sintáctica se puede definir en términos de un conjunto núcleo de construcciones esenciales" [**?**].

5.2. Transformaciones de Desugaring en LAMBDAE

El módulo `Desugar` de LAMBDAE implementa la función central `desugar :: SASA ->ASA` que realiza la transformación sintáctica. A continuación, analizamos las transformaciones más significativas:

5.2.1. Desugaring de Operaciones N-arias

Las operaciones aritméticas y relacionales que aceptan múltiples argumentos se transforman en cadenas de operaciones binarias:

Listing 1: Desugaring de operaciones n-arias

```
desugar (AddListS es) = foldl1 (BinOp AddOp) (map desugar es)
desugar (SubListS es) = foldl1 (BinOp SubOp) (map desugar es)
desugar (MulListS es) = foldl1 (BinOp MulOp) (map desugar es)
```

Por ejemplo, la expresión SASA `(+ 1 2 3 4)` se desugarea a:

```
BinOp AddOp (BinOp AddOp (BinOp AddOp (Num 1) (Num 2)) (Num 3)) (Num 4)
```

Esta transformación refleja el principio de que operaciones complejas pueden expresarse mediante composición de operaciones más simples [**?**].

5.2.2. Desugaring de Funciones Lambda Multi-parámetro

Las funciones lambda con múltiples parámetros se transforman en funciones unarias anidadas mediante currying:

Listing 2: Desugaring de funciones lambda

```
desugar (FunListS [] body)     = desugar body
desugar (FunListS (x:xs) body) = Fun x (desugar (FunListS xs body))
```

Una expresión SASA como `(lambda (x y z) (+ x y z))` se convierte en:

```
Fun "x"(Fun z"(Fun "z"(BinOp AddOp (BinOp AddOp (Id "x") (Id z")) (Id "z"))))
```

Esta transformación implementa el concepto clásico de currying, donde funciones multi-argumento se representan como secuencias de funciones unarias [**?**].

5.2.3. Desugaring de Construcciones Let

El desugaring de las construcciones `let` demuestra uno de los principios más elegantes del diseño de lenguajes funcionales: `let` es simplemente azúcar sintáctica para la aplicación inmediata de una función lambda.

Listing 3: Desugaring de let

```
desugar (LetS [(x,e)] body) = App (Fun x (desugar body)) (desugar e)
```

Esta transformación implementa la equivalencia:

$$\texttt{(let ((x e)) body)} \equiv \texttt{((lambda (x) body) e)}$$

Como señala el material de Brown University, esta es una de las transformaciones de desugaring más fundamentales en lenguajes funcionales, demostrando que el enlace de variables puede expresarse completamente en términos de abstracción y aplicación [**?**].

5.2.4. Desugaring de Let*

La construcción `let*`, que permite referencias secuenciales entre bindings, se desugarea recursivamente en `let` anidados:

Listing 4: Desugaring de let*

```
desugar (LetStarS [] body) = desugar body
desugar (LetStarS ((x,e):rest) body) =
  desugar (LetS [(x,e)] (LetStarS rest body))
```

Esta transformación ilustra cómo el azúcar sintáctica puede construirse en capas: `let*` se reduce a `let`, que a su vez se reduce a `lambda` y aplicación [**?**].

5.2.5. Desugaring de Expresiones Condicionales Múltiples

La construcción `cond` se desugarea recursivamente en expresiones `if` anidadas:

Listing 5: Desugaring de cond

```
desugarCond :: [(SASA, SASA)] -> ASA
desugarCond [] = error "cond sin clausulas"
desugarCond [(g,e)] = desugar e
desugarCond ((g,e):rest) =
  If (desugar g) (desugar e) (desugarCond rest)
```

Una expresión como:

```
(cond
  [(< x 0) "negativo"]
  [(= x 0) "cero"]
  [else "positivo"])
```

Se transforma en:

```
If (BinOp LtOp x 0) "negativo"(If (BinOp EqOp x 0) ceropositivo")
```

5.2.6. Desugaring de Azúcar de Conveniencia

LAMBDAE incluye construcciones de conveniencia como `add1` y `sub1` que se desugarean directamente en el parser:

Listing 6: Azúcar sintáctica de conveniencia

```
| '(' "add1" SASA ')' { AddListS [$3, NumS 1] }
| '(' "sub1" SASA ')' { SubListS [$3, NumS 1] }
```

Estas expresiones ilustran el principio de que incluso operaciones simples pueden beneficiarse del azúcar sintáctica para mejorar la legibilidad del código [**?**].

5.3. Beneficios Arquitectónicos del Desugaring en LAMBDAE

La implementación de desugaring en LAMBDAE proporciona múltiples ventajas arquitectónicas:

5.3.1. Simplificación del Evaluador

Al reducir SASA a ASA antes de la evaluación, el intérprete de LAMBDAE solo necesita manejar un conjunto reducido de construcciones. Esto reduce significativamente la complejidad del evaluador y facilita la verificación de su correctitud [**?**].

5.3.2. Reutilización de Código y Demostraciones

Como observa el material de la Universidad de Tübingen, si se han escrito algoritmos o demostrado propiedades sobre el lenguaje núcleo (ASA), estos automáticamente se aplican a todas las construcciones del lenguaje superficial (SASA) después del desugaring [**?**]. Esto representa una economía significativa en el esfuerzo de desarrollo y verificación.

5.3.3. Extensibilidad del Lenguaje

La arquitectura de dos niveles facilita la adición de nuevas características al lenguaje. Nuevas construcciones sintácticas pueden agregarse al nivel SASA siempre que puedan expresarse en términos del núcleo ASA, sin necesidad de modificar el evaluador [**?**].

5.3.4. Claridad Semántica

El desugaring hace explícita la semántica de cada construcción sintáctica al mostrar exactamente cómo se traduce en términos del núcleo. Esto proporciona una especificación formal del significado de cada característica del lenguaje [**?**].

6. Semántica operacional

La semántica operacional describe cómo se ejecutan los programas de un lenguaje paso a paso. A diferencia de la semántica denotacional (que define el significado de un programa como funciones matemáticas) o la axiomatica (que define propiedades lógicas de los programas), la semántica operacional se centra en la evolución del estado del programa durante su ejecución.

Se puede ver como "un conjunto de reglas que explican cómo cada instrucción afecta al estado del programa".

SmallStep Semantics con Entornos

A continuación se muestran las reglas de paso pequeño (*small-step semantics*) para el lenguaje, considerando entornos explícitos ρ.

Relación de evaluación: $\langle e, \rho\rangle \longrightarrow \langle e', \rho'\rangle$

Reglas Básicas

$$\frac{}{\langle Id(\text{"nil"}), \rho\rangle \to \langle Id(\text{"nil"}), \rho\rangle} \qquad \frac{}{\langle Num(n), \rho\rangle \to \langle Num(n), \rho\rangle}$$

$$\frac{}{\langle Boolean(b), \rho\rangle \to \langle Boolean(b), \rho\rangle} \qquad \frac{lookupEnv(i, \rho) = v}{\langle Id(i), \rho\rangle \to \langle v, \rho\rangle}$$

Operaciones Binarias y Unarias

$$\frac{isValue(v_1) \wedge isValue(v_2)}{\langle BinOp(op, v_1, v_2), \rho\rangle \to \langle evalBinOp(op, v_1, v_2), \rho\rangle}$$

$$\frac{isValue(v_1) \quad \langle e_2, \rho\rangle \to \langle e_2', \rho'\rangle}{\langle BinOp(op, v_1, e_2), \rho\rangle \to \langle BinOp(op, v_1, e_2'), \rho'\rangle}$$

$$\frac{\langle e_1, \rho\rangle \to \langle e_1', \rho'\rangle}{\langle BinOp(op, e_1, e_2), \rho\rangle \to \langle BinOp(op, e_1', e_2), \rho'\rangle}$$

$$\frac{}{\langle Sqrt(Num(n)), \rho\rangle \to \langle Num(\lfloor\sqrt{n}\rfloor), \rho\rangle} \qquad \frac{\langle e, \rho\rangle \to \langle e', \rho'\rangle}{\langle Sqrt(e), \rho\rangle \to \langle Sqrt(e'), \rho'\rangle}$$

$$\frac{}{\langle Not(Boolean(b)), \rho\rangle \to \langle Boolean(\neg b), \rho\rangle} \qquad \frac{\langle e, \rho\rangle \to \langle e', \rho'\rangle}{\langle Not(e), \rho\rangle \to \langle Not(e'), \rho'\rangle}$$

Listas y Pares

$$\frac{e = Id(\text{"}nil\text{"})}{\langle UnOp(NullOp, e), \rho\rangle \to \langle Boolean(True), \rho\rangle}$$

$$\frac{e = Pair(v_1, v_2)}{\langle UnOp(NullOp, e), \rho\rangle \to \langle Boolean(False), \rho\rangle}$$

$$\frac{\langle e, \rho\rangle \to \langle e', \rho'\rangle}{\langle UnOp(NullOp, e), \rho\rangle \to \langle UnOp(NullOp, e'), \rho'\rangle} \qquad \frac{isValue(v_1) \wedge isValue(v_2)}{\langle Pair(v_1, v_2), \rho\rangle \to \langle Pair(v_1, v_2), \rho\rangle}$$

$$\frac{isValue(v_1) \quad \langle e_2, \rho\rangle \to \langle e_2', \rho'\rangle}{\langle Pair(v_1, e_2), \rho\rangle \to \langle Pair(v_1, e_2'), \rho'\rangle} \qquad \frac{\langle e_1, \rho\rangle \to \langle e_1', \rho'\rangle}{\langle Pair(e_1, e_2), \rho\rangle \to \langle Pair(e_1', e_2), \rho'\rangle}$$

$$\frac{isValue(v_1) \wedge isValue(v_2)}{\langle Fst(Pair(v_1, v_2)), \rho\rangle \to \langle v_1, \rho\rangle} \qquad \frac{\langle e, \rho\rangle \to \langle e', \rho'\rangle}{\langle Fst(e), \rho\rangle \to \langle Fst(e'), \rho'\rangle}$$

$$\frac{isValue(v_1) \wedge isValue(v_2)}{\langle Snd(Pair(v_1, v_2)), \rho\rangle \to \langle v_2, \rho\rangle} \qquad \frac{\langle e, \rho\rangle \to \langle e', \rho'\rangle}{\langle Snd(e), \rho\rangle \to \langle Snd(e'), \rho'\rangle}$$

Condicionales y Funciones

$$\frac{}{\langle If(Boolean(True), e_t, e_f), \rho\rangle \to \langle e_t, \rho\rangle} \qquad \frac{}{\langle If(Boolean(False), e_t, e_f), \rho\rangle \to \langle e_f, \rho\rangle}$$

$$\frac{\langle cond, \rho\rangle \to \langle cond', \rho'\rangle}{\langle If(cond, e_t, e_f), \rho\rangle \to \langle If(cond', e_t, e_f), \rho'\rangle}$$

$$\frac{}{\langle Fun(p, body), \rho\rangle \to \langle Closure(p, body, \rho), \rho\rangle}$$

$$\frac{e_1 = Closure(p, body, \rho') \quad isValue(e_2)}{\langle App(e_1, e_2), \rho\rangle \to \langle Expr(body, (p, e_2) : \rho'), \rho\rangle} \qquad \frac{isValue(e_1) \quad \langle e_2, \rho\rangle \to \langle e_2', \rho'\rangle}{\langle App(e_1, e_2), \rho\rangle \to \langle App(e_1, e_2'), \rho'\rangle}$$

$$\frac{\langle e_1, \rho\rangle \to \langle e_1', \rho'\rangle}{\langle App(e_1, e_2), \rho\rangle \to \langle App(e_1', e_2), \rho'\rangle} \qquad \frac{isValue(e)}{\langle Expr(e, \rho_{in}), \rho_{out}\rangle \to \langle e, \rho_{out}\rangle}$$

$$\frac{\langle e, \rho_{in}\rangle \to \langle e', \rho_{in}'\rangle}{\langle Expr(e, \rho_{in}), \rho_{out}\rangle \to \langle Expr(e', \rho_{in}'), \rho_{out}\rangle}$$

7. Conclusiones

La implementación del proceso de desugaring en LAMBDAE, mediante la transformación de SASA a ASA, representa una aplicación rigurosa de principios establecidos en el diseño de lenguajes de programación. Al separar explícitamente la sintaxis superficial del núcleo semántico, LAMBDAE logra un balance entre expresividad para el programador y simplicidad para el implementador. Esta arquitectura no solo simplifica la implementación del intérprete, sino que también proporciona una base sólida para razonar formalmente sobre las propiedades del lenguaje, facilitando su extensión y mantenimiento futuro. El proceso de quitar elementos sintácticos no esenciales como paréntesis redundantes y representar todas las expresiones en formas canónicas (operaciones binarias, funciones unarias, aplicación binaria) demuestra el poder del desugaring como herramienta de diseño de lenguajes. Como establece la literatura, esta técnica es fundamental no solo para la implementación práctica de lenguajes, sino también para su estudio teórico y su enseñanza [?].

8. Referencias

Referencias

[1] LambdaSpace. (s. f.). *Notas de la asignatura Lenguajes de Programación (n.º 04)* [PDF]. Recuperado de https://lambdasspace.github.io/LDP/notas/ldp$_n$04.*pdf*

[2] LambdaSpace. (s. f.). *Notas de la asignatura Lenguajes de Programación (n.º 05)* [PDF]. Recuperado de https://lambdasspace.github.io/LDP/notas/ldp$_n$05.*pdf*

[3] Universidad Nacional Autónoma de México, Facultad de Ciencias. (s. f.). *Material de Lenguajes de Programación*. Recuperado de https://sites.google.com/ciencias.unam.mx/lengprog/material?authuser=0

[4] Universidad Nacional Autónoma de México. (2025 jul-sep). *El Libro del Dragón. Compiladores: Principios, Técnicas y Herramientas*. Tesis / Documento DGB-UNAM. Recuperado de https://tesiunamdocumentos.dgb.unam.mx/ptd2025/jul$_s$*ep*/0877653/*Index.html*

[5] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA: MIT Press.

[6] Scott, M. L. (2007). *Programming Language Pragmatics*. Amsterdam/Boston et al.: Morgan Kaufmann.

[7] Krishnamurthi, S. (2007). *Programming Languages: Application and Interpretation*. Brown University. Recuperado de https://cs.brown.edu/ sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf

[8] Harper, R. (s. f.). *Practical Foundations for Programming Languages*. (Nota: Verificar año de publicación y editorial).

[9] International Organization for Standardization & International Electrotechnical Commission. (1996). *ISO/IEC 14977:1996 – Information technology — Syntactic metalanguage — Extended BNF* (Edición 1). Ginebra: ISO.

[10] Krishnamurthi, S. (2015). *Desugaring in Practice: Opportunities and Challenges*. PEPM 2015. Recuperado de https://dl.acm.org/doi/10.1145/2678015.2678016

[11] Krishnamurthi, S. (2007). *Programming Languages: Application and Interpretation* (2.ª ed.). Brown University. Recuperado de https://cs.brown.edu/courses/cs173/2012/book/

[12] Rathi, M. (2020, 1 de julio). *Desugaring – taking our high-level language and simplifying it!*. Recuperado de https://mukulrathi.com/create-your-own-programming-language/lower-language-constructs-to-llvm/

[13] Serokell. (2020, 11 de agosto). *Understanding Haskell Features Through Their Desugaring*. Recuperado de https://serokell.io/blog/haskell-to-core

[14] Syntactic sugar. (s. f.). En Wikipedia. Recuperado de https://en.wikipedia.org/wiki/Syntactic$_s$*ugar*