Trabajo Práctico Transversal

Teoría de la Computación II

2025

Licenciatura en Análisis de Sistemas

Universidad Nacional de Salta

Nombres: Romero, Martin / Romero, Joaquin / Huebra, Maximiliano / Tolava, Axel

Aleph: Diseño e implementación de un lenguaje interpretado para conjuntos y listas

1. Introducción

1.1. Objetivo

El objetivo de este proyecto es documentar el análisis, diseñar e implementar un lenguaje de programación interpretado, enfocado en la manipulación de conjuntos y listas con distintos niveles de anidamiento incluyendo las operaciones que se pueden realizar entre estos y sus tipo de datos átomos. El lenguaje a su vez incluye estructuras de control, repetición y selección.

1.2. Sobre el nombre del lenguaje

El lenguaje se denominará Aleph, en alusión a la Teoría de Conjuntos. “Aleph” proviene de la primera letra del alfabeto hebreo y simboliza un origen o comienzo.

Principios de diseño para Aleph

1. Scripting e imperativo.
2. Fuerte apoyo a algoritmos matemáticos.
3. Alta legibilidad con sintaxis simple.
4. Tipado fuerte y evaluación interpretada.
5. Abstracción del manejo de memoria.

1.3. Sobre las herramientas a utilizar en el desarrollo

Para el diseño y desarrollo de este lenguaje interpretado utilizaremos dos herramientas: Flex y Bison. Los antecesores de tales programas son Lex y Yacc los cuales fueron creados aproximadamente en los 70 's en los laboratorios **Bell Labs** junto al desarrollo de UNIX. Una década después surgieron estas herramientas:

Flex (Fast Lexical Analyzer): Creado en 1987 por Vern Paxson en el Lawrence Berkeley Laboratory, este nació como reemplazo libre y más rápido de Lex, manteniendo compatibilidad casi total con su sintaxis y se liberó bajo licencia BSD/GPL y rápidamente se convirtió en el estándar en sistemas *open source*. El objetivo de Flex es generar un analizador léxico que lea un archivo de entrada (generalmente código fuente) y lo divida en tokens (símbolos con significado, como números, identificadores, operadores).

Bison: Publicado por la Free Software Foundation (FSF) en 1985, es un reemplazo libre de Yacc, completamente compatible con sus especificaciones y con el tiempo, añadió mejoras como soporte para múltiples lenguajes de salida, mejor manejo de conflictos y generación de código más eficiente. El

objetivo de Bison es generar un analizador sintáctico que, usando tokens obtenidos de Flex, compruebe si la secuencia corresponde a la gramática de un lenguaje (por ejemplo, las reglas de un lenguaje de programación).

En conjunto, Flex + Bison se utilizan para construir compiladores, intérpretes y procesadores de lenguajes. Flex hace la "fase de escaneo" y Bison la "fase de parsing".

Interacción con las herramientas y sus archivos:

En Flex se trabaja con un archivo de especificación que posee la extensión ".l" donde el archivo se dividirá en tres secciones separadas por "%%": La primera sección contiene declaraciones y configuraciones de opciones. La segunda sección contiene reglas léxicas en las cuales cada expresión regular está asociada a una acción en C, y la tercera sección es código C que se copia al escáner generado, generalmente pequeñas rutinas relacionadas con el código de las acciones en una función Main().

En cambio en Bison se tienen las mismas tres secciones mencionadas anteriormente pero están contenidas en un archivo .y, no por casualidad se repiten estas secciones, por primera parte las declaraciones incluyen código en C que será copiado al inicio del analizador generado (dentro de %{ and %}). Aquí se declaran los tokens con %token, que representan los símbolos que provienen del analizador léxico (por convención, los nombres de tokens suelen escribirse en mayúsculas. Cualquier símbolo que no se declare como token debe aparecer en el lado izquierdo de al menos una regla gramatical. Si un símbolo no es token ni aparece en reglas, es como una variable no usada en C: no causa error, pero probablemente es un descuido. En la segunda sección tenemos las reglas gramaticales las cuales se escriben en una notación similar a BNF (forma normal de Backus–Naur). Bison usa : en lugar de ::= . Las reglas terminan con punto y coma, ya que los saltos de línea no son significativos. Por ultimo tenemos la sección de Código en C del usuario en la que al igual que en Flex, aquí se escriben funciones adicionales, como main() o yyerror().

Cómo se conectan:

Flex (.l) detecta secuencias de caracteres y devuelve tokens (números, palabras clave, símbolos, etc.). Estos tokens se declaran en el archivo .y de Bison con %token.

Bison (.y) recibe los tokens desde la función yylex() (generada por Flex) y los organiza según las reglas gramaticales.

En código: Flex devuelve return NUMBER; cuando ve un número mientras que Bison tiene %token NUMBER y reglas como: expr: expr '+' expr | NUMBER ;

2. Análisis de requerimientos

2.1. Un caso de estudio: Los Autómatas Finitos y sus algoritmos

A es un autómata finito si A es una cinco-upla (Q,Σ,δ, q0, F) donde:

Q: es un conjunto finito, no vacío de estados.

Σ: es un conjunto finito, no vacío de símbolos, denominado alfabeto.

δ: es una función de transición, $\delta : Q \times \Sigma \rightarrow Q$, en la que cada transición se denota $\delta(q, a) = p$, donde q y p son estados, y a un símbolo del alfabeto

q0: es el estado inicial del autómata.

F: es un conjunto finito de estados de aceptación, con $F \subseteq Q$.

La diferencia entre un autómata determinista y no determinista se ve reflejada en la transición, pues:

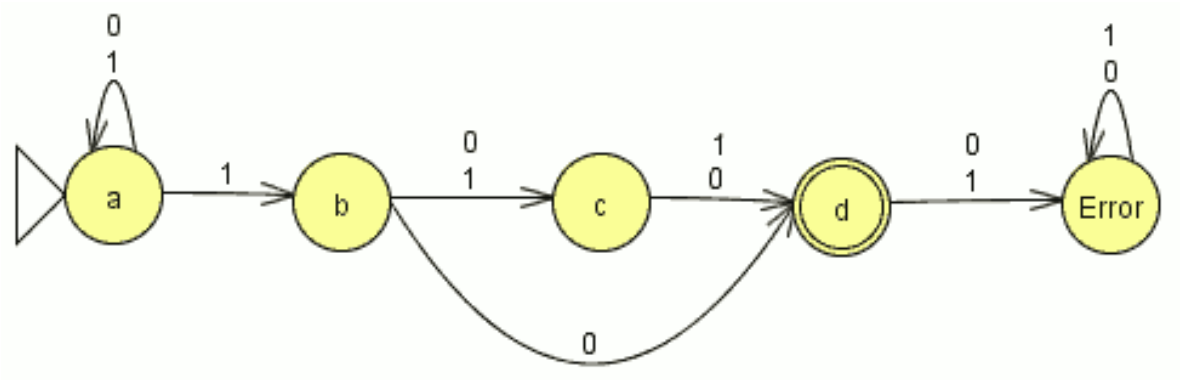
Si $\delta : Q \times \Sigma \rightarrow Q$ para toda transición, el autómata se llama determinista.

Si $\delta : Q \times \Sigma \rightarrow P(Q)$ (donde P es el conjunto partes) se denomina autómata no determinista.

En este caso δ(q, a) puede resultar en un conjunto de estados, pensando en δ como una relación sobre $Q \times \Sigma \times Q$: $\delta(q, a) = \{p \in Q \mid (q, a, p) \in \delta\}$. Además nos resultará importante entonces la siguiente definición: $\delta(P, a) = \cup_{q \in P} \delta(q, a)$, donde $P \subseteq Q$.

Q\Σ	**0**	**1**
->a	**{a}**	**{a,b}**
b	**{c,d}**	**{c}**
c	**{d}**	**{d}**
***d**	**{Error}**	**{Error}**
Error	**{Error}**	**{Error}**

Ejemplo de autómata y su representación en DTE y tabla de transiciones:



Autómata por extensión:

A = ({a,b,c,d,Error},{0,1},{[a,0,{a}],[a,1,{a,b}],[b,0,{c,d}],[b,1,{c},][c,0,{d}],[c,1,{d}],[d,0,{Error}],[d,1,{Error}],[Error,0,{Error}],[Error,1,{Error}]},a,{d}**)**

Conversión de AFND a AFD:

Teorema. Dado A = (Qa, Σ, δa, q0a, Fa) un AFND, existe B = (Qb, Σ, δb, q0b, Fb) un AFD tal que L(A) = L(B).

- Σb = Σa
- q0b = {q0a}
- Fb = { S ∈ Qb | S ∩ Fa ≠ ∅ }
- δb(S, a) = ∪_{q ∈ S} δa(q, a)
- Qb ⊆ P(Qa)

Pseudocódigo de la conversión:

```
 afnd2afd(A) =
sea A = (Q, Σ, δ, q0, F)
QB = {{q0}}
mientras ∃R ∈ QB tal que δB(R, a) esté indefinido ∀a ∈ Σ
    ∀a ∈ Σ
        δB(R, a) = ∪_{q ∈ R} δ(q, a)
    QB = QB ∪ {δB(R, a)}
FB = {S ∈ QB | S ∩ F ≠ ∅}
```

```
contestar (QB, Σ, δB, {q0}, FB)
```

Dificultades en la implementación:

El principal problema que hubo a la hora de desarrollar este algoritmo, fue que la implementación se volvió muy extensa debido a que no se contó con una base preparada para el manejo adecuado de estas listas y conjuntos, lo cual resultó en problemas de encapsulamiento, legibilidad y extensión del código.

El desarrollo se vio dificultado por la necesidad de escribir un número amplio de funciones auxiliares, para ser capaces de movernos cómodamente por la estructura propuesta. Por tal motivo, el resultado fue poco legible y con poca capacidad de escalado.

La implementación directa de estas transformaciones puede tornarse extensa si no se dispone de estructuras adecuadas para conjuntos y listas, afectando encapsulamiento, legibilidad y escalabilidad. Es recomendable abstraer operaciones de conjuntos y proveer utilitarios para iteración, unión, intersección y diferencia.

2.2. Conceptos sobre lenguajes de programación

La implementación de los lenguajes de programación, como todo desarrollo de software, comienzan por el análisis del problema que requiere una solución informática y el diseño de una respuesta adecuada. Un lenguaje de programación puede entenderse como el medio de comunicación entre programador y computadora. Tal como ocurre con el lenguaje humano, que Sapir define como “un método puramente humano y no instintivo de comunicar ideas, emociones y deseos por medio de un sistema de símbolos producidos voluntariamente”, los lenguajes de programación también se basan en símbolos y reglas, pero orientados a expresar procesos computacionales.

En la historia de la informática, pasamos de programar con cableados físicos a usar símbolos y localizaciones de memoria (lenguaje ensamblador). Con el tiempo surgieron lenguajes de más alto nivel, independientes de la máquina, que utilizan notaciones más cercanas al pensamiento humano.

Así, retomando la definición de Louden:

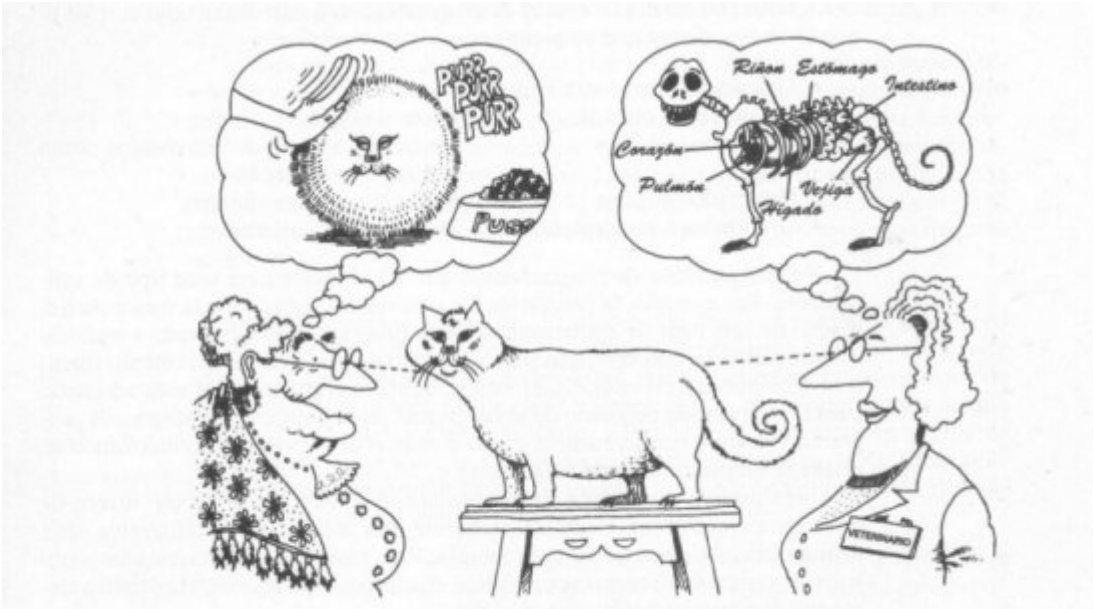
“Un lenguaje de programación es un sistema notacional para describir computaciones en una forma legible tanto para la máquina como para el ser humano.”

Los lenguajes de programación no solo son herramientas técnicas, sino también abstracciones cognitivas y culturales. Esto significa que, más allá de permitirle al programador dar instrucciones a una máquina, cada lenguaje modela una forma particular de pensar sobre los problemas y soluciones.

2.3. Abstracciones

La abstracción es uno de los principios fundamentales del diseño de lenguajes de programación y del desarrollo de software. Su objetivo central es permitir que el programador trabaje con conceptos de alto nivel sin necesidad de manejar detalles de implementación o representación interna. Tanto Louden como Sebesta coinciden en que la abstracción es la herramienta esencial para enfrentar la complejidad creciente de los programas, y que gracias a ella es posible construir sistemas extensos, robustos y mantenibles.

En términos generales, la abstracción consiste en generar una visión simplificada o conceptual de una entidad, conservando solamente sus atributos esenciales y omitiendo aquellos que no resultan necesarios para su uso. Sebesta utiliza un ejemplo taxonómico: si se define "ave" por un conjunto de características comunes (dos alas, dos patas, cola y plumas), entonces al clasificar un cuervo como ave no es necesario reiterar aquellas características compartidas; basta describir las propiedades particulares del cuervo. Este mecanismo de omitir lo uniformemente común y destacar lo relevante es precisamente la esencia de la abstracción.



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Louden explica que la abstracción es el principio rector que permite al programador evitar verse abrumado por detalles de bajo nivel, delegando muchos de ellos a capas inferiores de implementación.

Cada capa del sistema puede considerarse una máquina abstracta que oculta complejidades mediante interfaces claras.

En el mundo de los lenguajes de programación, la abstracción no sólo es útil para los programadores que escriben código, sino también para quienes diseñan e implementan el lenguaje mismo. La creación de un lenguaje implica modelar conceptos complejos de computación de manera que sean consistentes, seguros y reutilizables. Para lograrlo, los diseñadores usan la abstracción para separar niveles de detalle internos y organizar la implementación en módulos claros.

Tipos de abstracción:

- Abstracción de control: La abstracción de control consiste en encapsular un conjunto de pasos o algoritmos dentro de un subprograma, función, procedimiento o método. El usuario del subprograma solo necesita conocer su interfaz: nombre, parámetros y efecto. No requiere conocer la lógica interna.

  Ejemplos típicos son: Un procedimiento de ordenamiento: la llamada al subprograma es una abstracción; su implementación interna puede variar entre diferentes algoritmos.

  Estructuras de control como for, while o repeat: abstraen patrones de iteración sin exponer el control detallado del flujo.

  Funciones de orden superior: permiten abstraer patrones de aplicación, filtrado o reducción sin especificar cada paso.

  Herramientas concurrentes: hilos, procesos, monitores y regiones críticas abstraen la sincronización y el control de ejecución paralela.

  Sebesta destaca que la abstracción de procesos es históricamente la más antigua.

- Abstracción de datos: La abstracción de datos consiste en ocultar la representación interna del dato y exponer únicamente sus operaciones válidas. Las estructuras de datos avanzadas y los tipos abstractos de datos son los ejemplos clásicos. Sus objetivos principales son: Ocultamiento de información, definir interfaces claras y coherentes, evitar que clientes manipulen o dependan de detalles internos, permitir reemplazar la implementación sin modificar el código usuario, etc.

Los lenguajes orientados a objetos extienden este concepto con clases, objetos, herencia y polimorfismo, pero el fundamento sigue siendo la abstracción de datos.

Estas abstracciones internas permiten que la implementación del lenguaje sea modular, mantenible y extensible. Cada módulo puede concentrarse en un aspecto particular —por ejemplo, la

evaluación de expresiones, el manejo de memoria o la verificación de tipos— sin preocuparse por los detalles de los demás módulos. Esto reduce la complejidad del desarrollo del propio lenguaje y facilita la incorporación de nuevas características, optimizaciones o soporte para diferentes plataformas de hardware.

En resumen, la abstracción en la implementación de un lenguaje de programación permite organizar la complejidad del lenguaje mismo, haciendo que tanto la construcción como la evolución del lenguaje sean manejables, mientras que el programador final se beneficia de una interfaz más sencilla y coherente.

Las abstracciones también se agrupan en **niveles**, los cuales son definidos según la cantidad de información contenida en la abstracción.

1) Abstracciones de datos:

- **Abstracciones básicas**: resumen la representación interna de valores comunes en la computadora. Ejemplo: un número entero se abstrae de su almacenamiento en memoria (bits, complemento a dos) y se usa directamente como un valor.
- **Abstracciones estructuradas**: permiten agrupar varios valores relacionados en una unidad mayor. Ejemplo: un registro de empleado con nombre, dirección, teléfono y salario.
- **Abstracciones unitarias:** elevan aún más el nivel, combinando datos con las operaciones que actúan sobre ellos. Incluyen encapsulación, ocultación de información y reutilización. Ejemplo: un paquete en Java o un tipo abstracto de datos (TAD).

2) Abstracciones de control:

- **Abstracciones básicas**: son las sentencias fundamentales que permiten expresar acciones. Ejemplo: la asignación abstrae instrucciones de máquina sobre direcciones de memoria y tipos.
- **Abstracciones estructuradas**: agrupan sentencias básicas en estructuras con significado semántico más claro. Ejemplo: *if*, *while*, *for*, así como funciones y procedimientos que encapsulan secuencias de acciones.
- **Abstracciones unitarias**: representan un nivel aún mayor, en el cual lo importante no es cómo se implementa un procedimiento, clase o módulo, sino **cómo se utiliza a través de su interfaz pública**. Ejemplo: servicios, bibliotecas, paquetes, APIs.

Aplicación de la Abstracción en TREE-SL

TREE-SL, como lenguaje interpretado orientado al manejo de listas y conjuntos, incorpora abstracciones de datos y de control diseñadas para facilitar la manipulación de estas colecciones y permitir la construcción clara de algoritmos sobre ellas. Proporciona a los usuarios una vista conceptual de sus estructuras sin exponer sus detalles de implementación internos, lo que constituye un ejemplo de abstracción semántica. Las estructuras operan mediante tipos y operaciones de alto nivel y se encuentran respaldadas por un intérprete que actúa como intermediario entre el programa del usuario y la representación interna de datos.

En **TREE-SL**, ambos tipos de abstracción se concretan de la siguiente forma:

Abstracciones de datos en Aleph:

Se incluyen dos estructuras principales: listas y conjuntos. Ambas constituyen abstracciones de datos de alto nivel, pues el usuario accede a ellas como entidades lógicas, sin conocer cómo se implementan internamente.

Listas en Aleph: Las listas son colecciones ordenadas de datos que no necesariamente son del mismo tipo. Constituyen una abstracción de datos estructurada, similar a un tipo abstracto de datos.

- Tipos de elementos permitidos: Enteros,doubles, booleanos, átomos (símbolos como q0, q1, usados para representar elementos de autómatas), otras listas o conjuntos anidados.
- Operaciones provistas por el lenguaje: Inserción, eliminación, concatenación, recorrido mediante estructuras de control, tomar elemento por posición y añadir elementos.

Cada operación constituye una abstracción de proceso, dado que el usuario invoca la operación sin necesidad de conocer su implementación. La lista en Aleph cumple con las características de un tipo abstracto de datos en el sentido de Sebesta: define un conjunto de valores posibles (colecciones ordenadas de elementos permitidos) y un conjunto de operaciones, ocultando completamente su representación interna.

Conjuntos en Aleph: Los conjuntos representan colecciones no ordenadas sin elementos repetidos. Su diseño sigue el modelo matemático tradicional de conjunto y constituye una abstracción de alto nivel que oculta la representación interna de la estructura.

- Tipos de elementos permitidos: Enteros, doubles, booleanos, átomos e inclusive otras listas y conjuntos anidados.
- Operaciones definidas por el lenguaje: Unión, intersección, diferencia, pertenencia e inclusión.

Estas operaciones son abstracciones semánticas que representan conceptos matemáticos directamente, sin exponer procedimientos ni estructuras internas. Los conjuntos también constituyen un tipo abstracto de datos: su interfaz se expresa por medio de las operaciones admitidas, y su representación queda completamente oculta al usuario.

Abstracción de control en Aleph

El flujo de ejecución se organiza a través de estructuras declarativas, donde se abstraen los detalles de iteración o control explícito.

- If: La instrucción implementa una estructura de selección. Constituye una abstracción de control en la que el programador especifica una condición booleana y dos posibles flujos de ejecución. Oculta la necesidad de operaciones de salto explícitas y permite expresar decisiones de forma estructurada.
- While: La instrucción while constituye una repetición condicional clásica. Representa una abstracción de control de iteración basada en una condición booleana. En lugar de operar con saltos incondicionales y verificaciones manuales, el programador utiliza una forma estructurada que expresa la idea de repetir mientras una condición sea verdadera.
- Forall: Es una abstracción de control de alto nivel diseñada para recorrer los elementos de una lista o conjunto. Se trata de una abstracción de iteración dependiente de datos, ya que el control del ciclo no depende de contadores ni índices, sino del conjunto de elementos existentes en la colección. Forall expresa la operación conceptual "para todo elemento de la colección", eliminando detalles de recorrido, indexado y finalización. Este tipo de abstracción coincide con las abstracciones de control semánticas descritas por Louden y Sebesta, en las que una estructura de control representa una operación cuyo significado proviene del dominio del problema, no de la estructura de bajo nivel de la máquina.
- Forany: Es una estructura de control complementaria a forall. Representa el concepto "existe un elemento en la colección tal que...". Constituye una abstracción de control condicional basada en cuantificación existencial. Su función es recorrer elementos de una colección hasta encontrar uno que satisfaga una condición. Forany es una abstracción semántica, ya que su intención proviene del lenguaje matemático de los conjuntos y no de un mecanismo imperativo tradicional. Su existencia en Aleph permite trabajar en un nivel conceptual alto sin exponer detalles sobre recorridos o estructuras internas.

Abstracciones internas del lenguaje:

Además de las abstracciones accesibles directamente al usuario, Aleph incorpora varias abstracciones internas que permiten separar el diseño lógico del lenguaje de su implementación real. Estas incluyen:

- Manejo automático de memoria y almacenamiento dinámico de listas y conjuntos
- Implementación interna de estructuras de recorrido
- Verificación de tipos durante la ejecución
- Representación interna de átomos
- Gestión de operaciones de conjunto sin intervención del usuario
- Interpretación de estructuras de control de alto nivel

Todas estas características constituyen abstracciones semánticas y de implementación. El intérprete proporciona un modelo de ejecución idealizado que oculta detalles como modelos de memoria, punteros o estructuras físicas. El usuario opera únicamente sobre las abstracciones definidas por el lenguaje.

En conjunto, estas características hacen que Tree-SL use la abstracción como herramienta contra la complejidad: los datos se expresan con listas y conjuntos de alto nivel, mientras que el control se gestiona con estructuras declarativas como forall, simplificando tanto la lectura como la escritura de programas.

2.4. Dominio de programación de Tree-SL

Tree-SL presenta dos tipos de dominios. Por un lado tenemos el dominio de Aplicaciones Científicas y el dominio de la Inteligencia Artificial.

<u>Dominio: Aplicaciones Científicas:</u>

Las tareas que lleva a cabo un lenguaje dentro de este dominio son el análisis numérico, la visualización de datos, la manipulación de gran información y la precisión a la hora de representar cálculos. Esto incluye la capacidad de modelar estructuras matemáticas complejas, realizar operaciones sobre conjuntos y listas, y representar de manera eficiente las relaciones entre datos.

Es precisamente en esto último mencionado en donde busca destacar Tree-SL, la base forjada para el trabajo de listas y conjuntos permite al usuario del lenguaje, la creación de algoritmos complejos, sin perder legibilidad ni control sobre la estructura de datos.

Además, Tree-SL está diseñado con una capacidad de expansión considerable. Aunque actualmente Tree-SL no está centrado en la manipulación directa de números, la arquitectura del lenguaje permite que se pueda integrar un soporte completo para cálculos numéricos, modelado simbólico, etc; Garantizando que la base de listas y conjuntos siga siendo el núcleo para la organización y manipulación de datos.

Ejemplo de tareas en el dominio científico: modelado de estructuras matemáticas complejas (grafos, matrices, relaciones)

Con esta aproximación, Tree-SL combina claridad, precisión y flexibilidad, ofreciendo una herramienta potente para quienes necesiten trabajar con estructuras de datos complejas y problemas científicos. manteniendo siempre la simplicidad y legibilidad del código.

Dominio: Inteligencia Artificial:

Tree-SL en su núcleo se orienta principalmente al procesamiento simbólico y la manipulación de estructuras de datos no numéricas, utilizando listas y conjuntos como herramientas centrales. Esto permite representar problemas complejos de forma abstracta, como autómatas, gramáticas, árboles de decisión y otros modelos simbólicos, sin depender de cálculos numéricos.

La fuerza que posee Tree-SL en este dominio reside en su capacidad para organizar, combinar y transformar conjuntos de datos simbólicos de forma clara y eficiente, proporcionando al programador un lenguaje para tareas típicas de la inteligencia artificial, como el modelado lógico, el análisis combinatorio y la manipulación de relaciones simbólicas.

Ejemplo de tareas en el dominio de inteligencia artificial: operaciones de cálculo simbólico.

3. Diseño

3.1. Sentencias

Una sentencia es una unidad sintáctica completa que expresa una acción o instrucción que el programa debe ejecutar. Cada sentencia representa un paso del flujo de control o una operación que modifica el estado del programa.

3.1.1. Un primer diseño de sentencias de Aleph

```
def function afnd2afd(A):
 # Inicialización de los autómatas
```

```
let A => (QA, SigmaA, DA, Q0A, FA)
B = [{}, SigmaA, {}, {Q0A}, {}]
let B => (QB, SigmaB, DB, Q0B, FB)
QB = QB union {Q0B}
while (nuevos != {}) do
        temp = {}
        forall (R in nuevos) do
                forall (a in SigmaB) do
                        Response = {}
                        forall (q in R) do
                                forall (T in DA) do
                                        if (T[1] == q and T[2] == a)
                                                Response = Response union {T[3]}
                                        endif
                                end
                        end
                        if (not (Response in QB))
                                QB = QB union {Response}
                                temp = temp union {Response}
                        endif
                        DB = DB union {[R, a, Response]}
                end
        end
        nuevos = temp
end
forall (S in QB | (S inter FA) != {}) do
        FB = FB union {S}
```

end

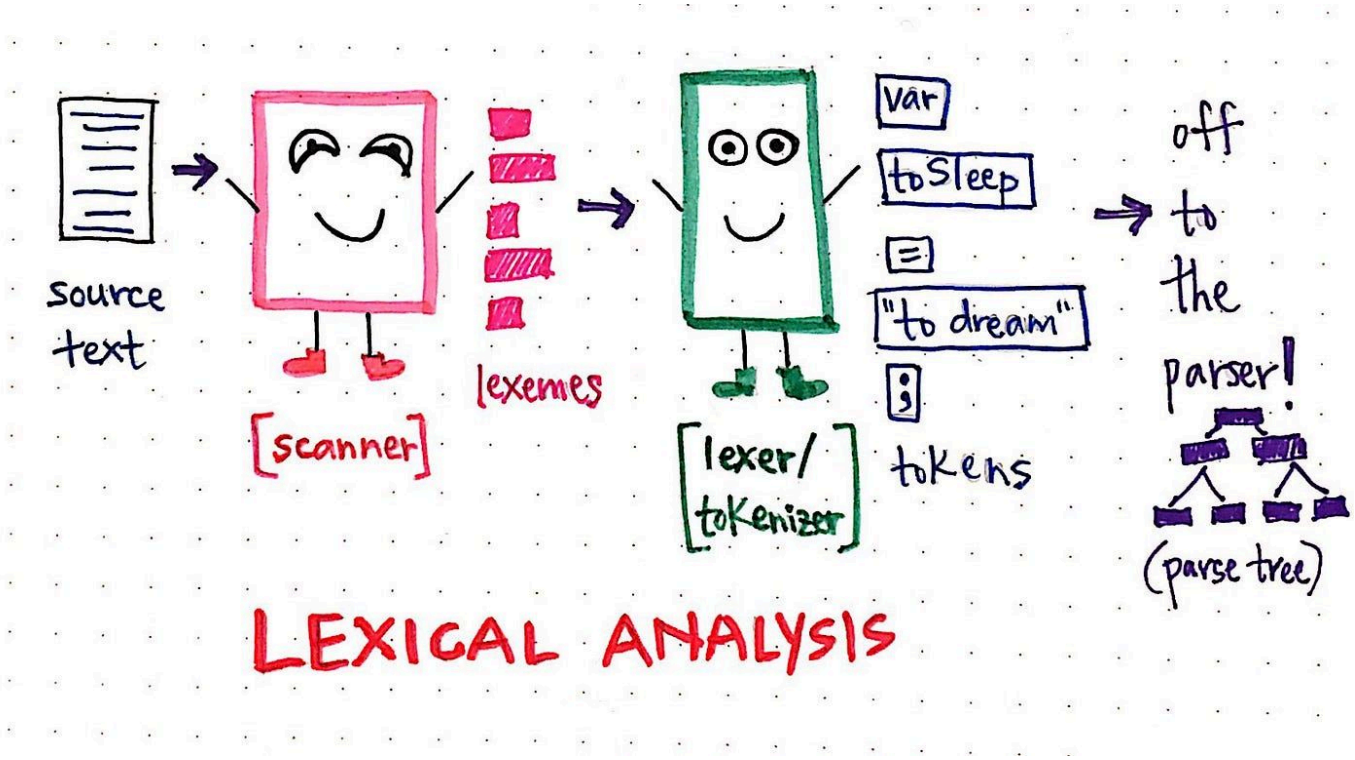
return B;

3.2. Reconocimiento de patrones. Tokens y Lexemas

Tanto en fuentes como Sebesta, Louden y John-Levine se llega a la misma definición de estos dos conceptos:

Los lexemas son las secuencias de caracteres que coinciden con los patrones de los tokens reconocidos por el analizador léxico. Un token es un par que consiste en un nombre de token y un valor opcional del lexema.

Véase esta imagen que nos pareció interesante de exponer:



Se pueden apreciar los tokens y lexemas utilizados en el 2.1 (algoritmo de AFND a AFD en ALEPH):

3.2.1. Tokens y lexemas de Aleph

Lexemas	Tokens
Palabras reservadas: let,while,forall,if,return.	LET,WHILE,FORALL,IF,RETURN.
Identificadores: A,QA,SigmaA,DA,Q0A,FA,B,QB,SigmaB,DB,Q0B,FB ,continuar,R,a,incompleto,encontrado,Response,q ,T,S	ID.
Operadores: =,==,! =,union,in,not,intersection.	ASIGNACION, IGUALDAD, DISTINTO, UNION, IN, NOT, INTERSECCION.

Delimitadores y agrupadores: : , ; , { , } , (,) , [,] , ,	RENOMBRAMIENTO, PUNTO_COMA, LLAVE_ABIERTA, LLAVE_CERRADA, PARENTESIS_ABIERTA, PARENTESIS_CERRADA, CORCHETE_ABIERTO, CORCHETE_CERRADO, COMA.
Lógicos: true, false.	BOOLEAN.

Palabras reservadas y simbolos de Aleph:

Declaración / Asignación

- let → asignación múltiple
- return → devolver un valor

Control de flujo:

- if
- else
- endif
- while
- forall
- forany
- do
- end

Tipos de datos y operaciones sobre conjuntos/listas:

- union
- inter
- diff
- in
- contains
- concat
- take
- kick
- add
- | | → cardinalidad

Literales booleanos:

- true
- false

Operadores aritméticos y de comparación:

- +,- ,*, /, %
- ==, !=, <, >, <=, >=

Otros símbolos importantes:

- = → asignación simple
- => → asignación múltiple (let A => (a,b,c))
- [ ] → listas
- { } → conjuntos
- ( ) → agrupación y parámetros de funciones
- , → separador de elementos
- / → usado en forany para separar condición
- | → cardinalidad

3.3. Sintaxis: Métodos Formales

Los lenguajes de programación requieren una interpretación de sus sentencias sin ambigüedades, su descripción, a los fines de comunicar su funcionamiento, tanto a los usuarios de los mismos como a quienes realizan su implementación, requiere herramientas formales.

Definición de Sintaxis de los lenguajes de programación:

Describe la estructura de un programa. Es la descripción de las maneras en las que distintas partes de un lenguaje pueden ser combinadas para formar otras partes. Las reglas de sintaxis establecen qué cadenas formadas con caracteres del alfabeto del Lenguaje de Programación forman parte del Lenguaje. La sintaxis de estos, se expresa mediante Gramáticas Libres de Contexto.

Definición de GLC

Una gramática libre de contexto (GLC) G = (N, T, S, P):

N: conjunto finito de variables sintácticas

T: conjunto finito de terminales

S: variable inicial (es elemento de N)

P: conjunto finito de producciones

Es una colección de reglas (producciones) que describen cómo formar cadenas válidas en un lenguaje. Cada regla tiene un símbolo no terminal en el lado izquierdo y una secuencia de terminales y/o no terminales en el lado derecho.

Se le dice libre de contexto porque las reglas de producción pueden aplicarse sin importar que variable o símbolo terminal tenga alrededor

Definición de BNF:

John Backus adoptó las reglas generativas de Chomsky para describir la sintaxis del lenguaje de programación ALGOL 58 (1959), presentando en el primer Congreso de Computación Mundial (World Computer Congress) el artículo «The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference».

Peter Naur, en su reporte sobre ALGOL 60 de 1963, identificó la notación de Backus como la Forma Normal de Backus (Backus Normal Form), y la simplificó para usar un conjunto de símbolos menor.

- BNF es una *notación natural para describir la sintaxis*. Moam+2studylib.es+2
- BNF es un metalenguaje para lenguajes de programación: es un lenguaje que sirve para describir otro lenguaje. studylib.es+2Moam+2

En BNF, las abstracciones sintácticas (no terminales) se delimitan generalmente con corchetes angulares (por ejemplo <assign>). Una regla (o producción) en BNF tiene un lado izquierdo (LHS) que es el símbolo no terminal que se define, y un lado derecho (RHS) que es una combinación de tokens, lexemas u otras abstracciones. Los símbolos no terminales en BNF pueden tener múltiples definiciones; las diferentes opciones se separan con el símbolo I (OR lógico)

BNF no incluye la notación de "…" (elipsis) para listas variables; para describir listas (elementos repetibles) se usa recursión con reglas donde el no terminal aparece de nuevo en su propia definición.

Definición de EBNF

EBNF es una extensión de BNF que no aumenta su poder descriptivo, pero mejora la legibilidad y la facilidad de escritura de las gramáticas

- Tres extensiones comunes se incluyen en varias versiones de EBNF:

La primera de éstas denota una parte opcional de un RHS (lado derecho), la cual se delimita con corchetes Por ejemplo, una sentencia if-else puede describirse como:

<if_stmt> ⟶ if <expression> <statement> [else <statement>]

Sin los corchetes, la descripción sintáctica de esta sentencia requeriría las dos reglas siguientes:

<if_stmt> ⟶ if <expression> <statement> | if <expression> <statement> else <statement>

La segunda extensión es el **uso de llaves en un RHS** para indicar que la parte encerrada puede repetirse o quedar fuera por completo. Esta extensión permite que las listas se construyan con una sola regla, en lugar de usar recursión y reglas múltiples. Por ejemplo, listas de identificadores separadas por comas pueden describirse con la siguiente regla:

<ident_list> → <identifier> {, <identifier>}

Esto reemplaza la recursión, la parte encerrada en llaves puede repetirse cualquier número de veces.

La tercera extensión común trata con **opciones múltiples**. Cuando un único elemento debe elegirse de un grupo, las opciones se colocan entre paréntesis y se separan con el operador OR |. Por ejemplo:

<term> → <term> (+ | - | *) <factor>

En BNF, una descripción de este <term> requeriría las tres reglas siguientes:

<term> → <term> + <factor>

| <term> - <factor>

| <term> * <factor>

Véase el ejemplo de la siguiente imagen extraída de Programming Languages, Robert W. Sebesta, capítulo 3:

EXAMPLE 3.5 BNF and EBNF Versions of an Expression Grammar

BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → <exp> ** <factor>
         <exp>
<exp> → (<expr>)
      | id
```

EBNF:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
      | id
```

3.3.1. Primera descripción formal de Tree-SL

<programa> → <modulo> | <expr> | <lista_sentencia>

<lista_sentencia> → <sentencia> <lista_sentencia> | <sentencia>

3.3.2. Descripción formal de la sintaxis Aleph

Las instrucciones que conforman todo lenguaje se pueden agrupar en instrucciones de asignación y estructuras de control de flujo. Las instrucciones de asignación también se conocen como transferencia de datos, pues asignan valores de una dirección a otra o de una expresión a una dirección. Las expresiones que son asignadas o evaluadas manipulan datos y retornan un valor determinado. La evaluación de expresiones puede servir entonces tanto para ser asignada a una dirección de memoria (nominada mediante un identificador) o para determinar bifurcaciones en el flujo de ejecución del programa.

Se presentan a continuación las BNF y EBNFs de las diferentes instrucciones que conforman Tree-SL.

Asignación y Asignación Múltiple

Las asignaciones son parte fundamental de un lenguaje de programación imperativo, sin ellas no sería posible relacionar valores con variables para un uso posterior.

Asignación: La operación de asignación en los LP provee de un mecanismo mediante el cual el programador puede modificar de manera dinámica la ligadura de valores con variables (lugar de memoria). Comúnmente los lenguajes de programación optaron por el uso del operador **"="** para la operación de asignación, y por ello utilizan un símbolo distinto a la hora de realizar la operación booleana de igualdad.

Si bien otros lenguajes utilizan otro símbolo para la operación, se decidió por un diseño más tradicional y simplemente hacer que la sentencia haga uso de solamente '=' como operador.

sintaxis:

```
<asign_simp> ::= id = <expr>
```

Ejemplo de uso: A = {1, 2, 3}

Asignación Múltiple: La sentencia de asignación múltiple en Aleph permite extraer los elementos de una estructura (como una lista o conjunto) y asignarlos simultáneamente a varias variables. Se utiliza para desempaquetar componentes de una estructura de datos en una sola operación, facilitando la legibilidad y el trabajo con modelos matemáticos compuestos.

Sintaxis

```
<asig_mult> ::= let id => ( list_ID )
```

Ejemplo de uso: let B => (x, y, z)

Para transformar valores y observar relaciones entre ellos debemos contar con expresiones que puedan evaluarse. Se presentan a continuación las expresiones de Aleph.

Asociatividad y Precedencia de operadores

Una expresión aritmética consiste en operadores, operandos, paréntesis y llamadas a funciones específicas. Los operandos pueden ser unarios, binarios o ternarios, dependiendo de la cantidad de operandos con la que la operación está definida. Además los operandos pueden clasificarse según la posición en la que aparecen en la expresión en relación a sus operandos, estos son infijos, prefijo y sufijo.

La misma especifica la computación aritmética que consiste en dos acciones buscar los valores de los operandos y ejecutar la operación aritmética. Como las expresiones son sumamente ricas en cuanto significado, para toda expresión se define reglas que definen el orden de evaluación de una expresión.

Las reglas de precedencia son definidas por las jerarquías de los operadores, permiten definir el orden de evaluación para expresiones adyacentes. En caso de que dos expresiones adyacentes tengan el mismo nivel de precedencia es necesario definir la asociatividad de los operadores. Otras características de las expresiones aritméticas son los paréntesis que alteran el nivel de precedencia de una expresión, donde una expresión con paréntesis tiene mayor precedencia que una expresión sin paréntesis**.**

```
<exp>::= number
        | <exp> + <exp>
        | <exp> - <exp>
        | <exp> * <exp>
        | <exp> / <exp>
        | | <exp> |
        | - <exp>
```

Un árbol de derivación es una representación jerárquica de la estructura de una cadena perteneciente a un Lenguaje Libre de Contexto. Donde queda reflejada la derivación usada para dicha cadena.

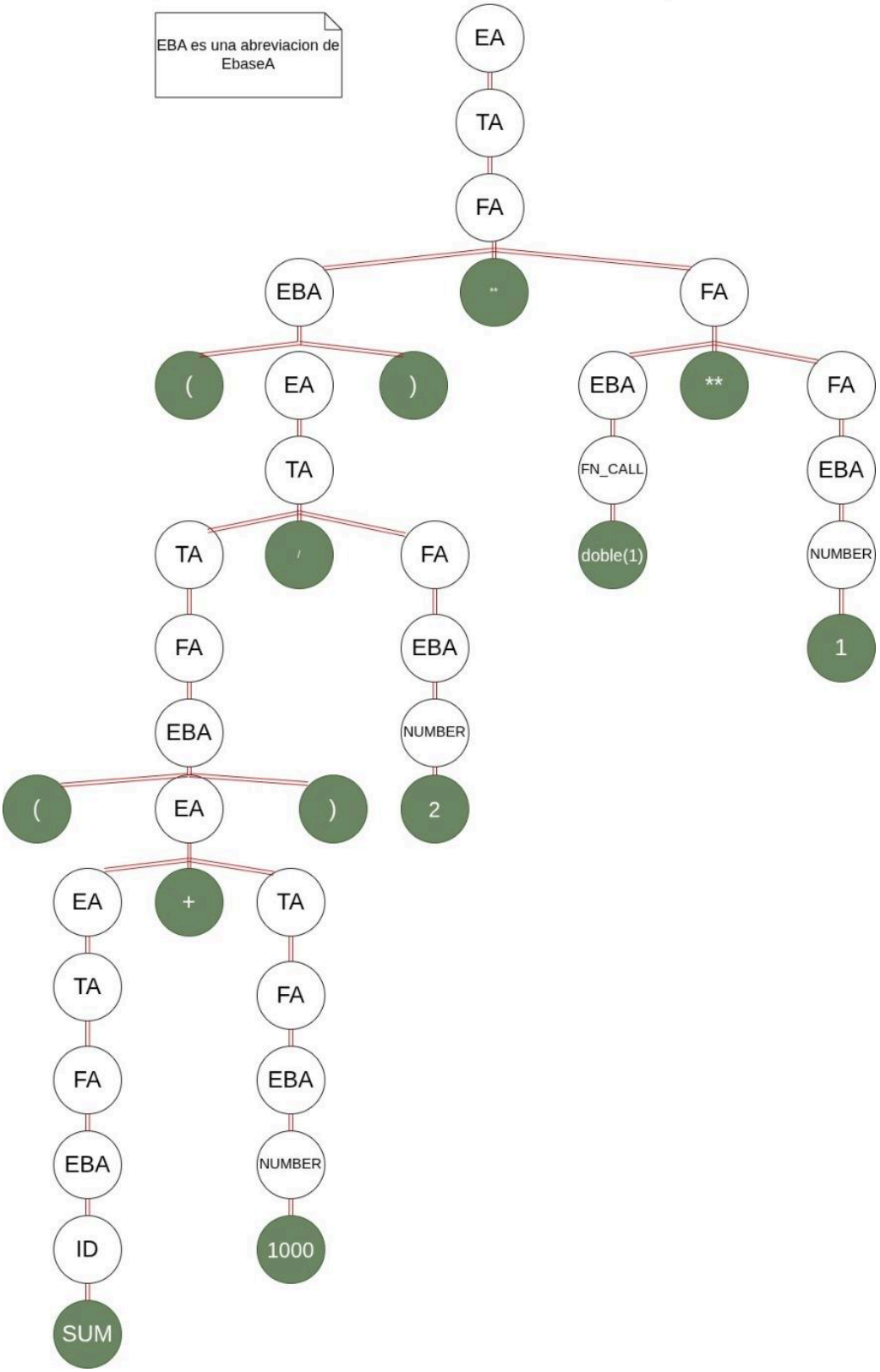
Una derivación es la aplicación de cero o más derivaciones desde el símbolo inicial a una cadena de terminales. Donde una derivación es la aplicación de una regla definida en la gramática.

En el contexto de lenguajes de programación el símbolo inicial es la abstracción de todas las cadenas del lenguaje, el símbolo <programa>. Donde las posibles derivaciones representan las construcciones aceptadas por el lenguaje, de misma forma sus combinaciones representan la cantidad infinita de programas válidos. Por último el alfabeto consiste en el conjunto de tokens definidos para el lenguaje.

((sum + 1000) / 2) ** doble(1) ** 1

Gramatica de expresiones aritmeticas (Laboratioro)
EA -> EA '+' TA | EA '-' TA | TA
TA -> TA '*' FA | TA '/' FA | FA
FA -> EbaseA '**' FA | EbaseA
EbaseA -> '(' EA ')' | NUMBER | ID | FN_CALL

EBA es una abreviacion de EbaseA



Un árbol de sintaxis abstracta es un árbol de derivación conciso, en donde se abstrae de los detalles innecesarios de las reglas de la gramática como paréntesis y otros no terminales. Dando un enfoque especial al significado intrínseco (semántico) que se espera de los nodos del árbol de sintaxis abstracta.

Sobrecarga de operadores:

La sobrecarga de operadores es un recurso sintáctico complejo que nos permite realizar múltiples usos de un operador dependiendo de los operandos con los que se acompañe. Este recurso se considera aceptable siempre y cuando no se vea afectada la legibilidad y fiabilidad del código en el que sea tratado. Comúnmente se usa en la informática con dirección científica en donde se permite manipular las representaciones computacionales de objetos matemáticos con la misma sintaxis que en papel.

Podemos hablar de dos sobrecargas:

- Sobrecarga predefinida por el lenguaje: Los propios diseñadores del lenguaje deciden que un operador tenga varios significados. Ejemplo: en Java, el + se usa tanto para números como para cadenas.
- Sobrecarga definida por el usuario: El programador puede extender el significado de un operador para aplicarlo a tipos de datos abstractos o definidos por el usuario. Ejemplo: Sobrecargar el * para permitir operar entre escalar y matriz.

Un ejemplo claro de una sobrecarga de operadores puede ser el & en el lenguaje C++, ya que si lo vemos como un operador binario este especifica una operación lógica AND a nivel de bits pero si lo vemos como un operador unario su significado es completamente distinto pues este acompañado de una variable le dará al usuario una dirección.

Al utilizar la sobrecarga debemos considerar que si bien puede mejorar la naturalidad y expresividad del código, su uso indebido puede provocar confusión, dificultar el mantenimiento y comprometer la consistencia semántica del programa. Por ello, debe aplicarse con criterio, limitándose a situaciones en las que aporte claridad y coherencia.

Sobrecarga del operador binario | |

En el caso de listas y conjunto el operador binario | | devuelve un número entero, que representa el número de elementos que contiene la estructura.

En el caso de número el operador | | devuelve el valor del número en valor absoluto .

Expresiones: Operaciones con Conjuntos y Listas

Las expresiones sobre conjuntos y listas son foco principal en el diseño de Tree-SL. Los conjuntos y listas son tipos de datos estructurados, al representar abstracciones de datos complejas es necesario especificar su construcción, manipulación, operaciones, precedencia, mutabilidad y su compatibilidad con otros tipos de datos.

La construcción de conjuntos y listas sintácticamente son casos iguales. Consiste entre llaves para un conjunto y entre corchetes para una lista, los elementos del objeto matemático. Para definir cada uno de sus miembros solo es necesario escribir las expresiones asociadas al valor que se le quiere dar al elemento en particular.

eg: [{“q0”, “q1”}, {“a”, “b”}];

Acerca de la precedencia y asociatividad de las expresiones con conjuntos y listas. Estas presentan mayor nivel de precedencia que las expresiones lógicas y relacionales, pero menor precedencia que las aritméticas. Aunque este último es necesario aclarar porqué problemas de precedencia con operadores aritméticos y de listas como. eg: 5 + 3 add []; Son resueltos sintácticamente al definir add 5 + 3 to [], para agregar elementos a una lista. El otro problema sería. eg: 5 + 3 unión {}; Este caso no es resuelto sintácticamente sino a través de la gramática de atributos y chequeo de tipos. Es decir, sintácticamente es una expresión entre conjuntos válidas aunque no pasa control estático de la semántica al no ser los dos operando de tipo conjunto.

Por último otra característica de los conjuntos y listas en Tree-SL es que son mutables y heterogéneos. Al ser mutables quiere decir que pueden ser foco y objetivo de transformacion atraves de algoritmos. Por otro lado, la heterogeneidad los vuelve excelentes candidatos para el almacenamiento de datos para el uso en algoritmos.

Expresiones: Operaciones con Conjuntos

Un conjunto es una colección bien definida de elementos distintos, considerados como un solo objeto a través de una agrupación sin orden.

Tree-SL permite sólo las operaciones básicas entre conjuntos son: unión, intersección y diferencia. En el lenguaje las únicas expresiones que evalúan a un conjunto son las tres operaciones básicas y la construcción con llaves de un conjunto.

La operación de **unión, intersección, diferencia** son operaciones binarias donde sus dos operandos deben ser de tipo conjunto, el operador **union** efectúa la unión correspondiente de dos conjuntos, el operador **inter** efectúa la intersección entre dos conjuntos y el operador **diff** efectúa la diferencia entre dos conjuntos. **unión** es el operador dentro de las expresiones con conjuntos con menor nivel de precedencia, es decir el último en ser evaluado. Luego en el siguiente nivel de precedencia sigue **inter** como el operador en primero ser evaluado **diff.** Los tres operandos son asociativos a la izquierda.

ej: partial_rtdo_set inter {0} union acum_set diff {1}

EXPRESION para listas

<exp>::= lit_struct
|<exp> union <exp>
|<exp> inter <exp>
|<exp> diff <exp>
|<exp> take <exp>
|<exp> take <exp>
|<exp> kick <exp>
| add <exp> to <exp>
<lit_struct>::= {} | [] | {<list_exp>} | [<list_exp>]
<lista_exp> ::= <exp> , <list_exp> | <exp>

Expresiones: Listas

```
<exp>::= lit_struct
        | take <exp> from <exp>
        | kick <exp> from <exp>
        | add <exp> to <exp>
<lit_struct>::= {}  | [] | {<list_exp>} | [<list_exp>]
<lista_exp> ::= <exp> , <list_exp> | <exp>
```

Expresiones: Operaciones relacionales

```
<exp>:=  <exp> CMP <exp>
```

Las expresiones relacionales, son aquellas que evalúan a un literal booleano, es decir, tienen un valor de verdad. Consisten en operaciones binarias infijas. Los operadores permitidos en Tree-SL son: <, >, <=, >=, ==, !=, in, contains. La condición para obtener una evaluación correcta de una expresión relacional en Tree-SL son dos: Los tipos de datos de los operandos deben ser iguales, en caso de operandos de tipo int y double, la comparación es permitida. No se permite encadenamiento eg: 10 < x < 20.

Acerca de la relación de la regla utilizada para definir las expresiones relacionales en Tree-SL, con respecto a las demás variables y reglas de la sintaxis del lenguaje. Es importante aclarar que las expresiones relacionales tienen un nivel de precedencia menor que las expresiones de Conjunto y Listas, pero mayor precedencia que las expresiones lógicas.

Todo tipo de expresión en Tree-SL tienen un valor de verdad ligado, lo que permite un enriquecimiento de la cantidad de combinaciones posibles con un significado importante en el lenguaje. Esto conlleva una mejora en la expresividad y ortogonalidad, a costa de empeorar la legibilidad del código al necesitar que los usuarios del lenguaje conozcan las convenciones de Tree-SL que son las estándares compartidas con muchos otros lenguajes. eg: {}; el conjunto vacío tiene un valor de verdad falso al igual que 0; el número cero

Algunas combinaciones válidas son: {} contains A; 3 < 3.5; 0; {};

Expresiones: Operaciones Lógicas

```
<exp> ::= bool_lit
```

De forma similar a las expresiones relaciones, las expresiones lógicas también evalúan a un literal booleano, obteniendo un valor de verdad de la expresión. Consisten en 2 operaciones binarias infijas, denotadas con los operandos **and** y **or** que efectúan la operación de las compuertas lógicas and or. El otro operando es **not** para la operación unaria efectuada por la compuerta lógica not.
Las expresiones lógicas son aquellas con el nivel de precedencia más bajo en Tree-SL, son las últimas en ser evaluadas.

Adicionalmente dentro de las distintas reglas de expresiones relacionales tenemos niveles de precedencia y asociatividad, debido a que las proposiciones lógicas son sumamente ricas en cuanto estructura. Los niveles de precedencia para los operandos son: **or**, con el menor nivel de precedencia **and**, con mayor nivel de precedencia que or pero menor que not **not**, el operando que se evalua primero. Por otro lado **and** y **or** ambos son asociativos a izquierda mientras que **not** asociativo a derecha. Un ejemplo valido en Tree-SL es.
eg: A contains B and i < n or A and b == 0;

Evaluación perezosa: Comúnmente utilizada en lenguajes de programación del paradigma funcional (tales como *LISP* o *Haskell*), la evaluación perezosa consiste en que un parámetro actual solo es evaluado cuando su valor resulta necesario para el funcionamiento de la función. De este modo, si una función tiene dos parámetros pero solo uno es utilizado, el otro no será evaluado.

La evaluación perezosa permite la creación de estructuras de datos potencialmente infinitas y una modularización más flexible, gracias a su capacidad de evaluar únicamente las expresiones requeridas durante la ejecución.

Evaluación ingenua: La evaluación ingenua es una estrategia en la cual todas las expresiones se evalúan tan pronto como son encontradas, antes de emplearse en cualquier operación posterior. Esta técnica garantiza un orden de ejecución determinista y predecible, lo que la hace especialmente adecuada para lenguajes estructurados e imperativos, como Tree-SL.

Estructuras de Control

El control del flujo de ejecución de instrucciones permite alterar el orden en el que cada sentencia es evaluada.

Ambigüedad de la estructura de control if:

La ambigüedad del **if** ocurre cuando en una estructura de selección anidada aparece un else, pero no queda claro a cuál **if** pertenece.

Esto sucede porque:

- Un else puede asociarse con más de un **if** posible.
- No siempre existen marcadores sintácticos (como llaves, “end”, indentación obligatoria) que indiquen claramente el alcance.
- La gramática del **if** es ambigua.

Sebesta explica que este es un problema clásico en el diseño de lenguajes.

Ejemplo:

```
if (sum == 0)
   if (count == 0)
      result = 0;
   else
      result = 1;
```

Aquí hay dos `if` y **un solo `else`**.

El compilador debe decidir si el `else` pertenece al `if (sum == 0)` o al `if (count == 0)`.

Interpretación 1 — El else pertenece al if interno: (Esta es la que adoptan Java, C, C++, C#, etc.)

```
if (sum == 0)          → if externo
        if (count == 0)    → if interno
                result = 0
        else               → else interno
                result = 1
```

Casos:

Si `sum == 0` y `count == 0` → result = 0

Si `sum == 0` y `count != 0` → result = 1

Si `sum != 0` → no hace nada

Esto quiere decir que el else pertenece al segundo if. Esta es la **regla del “else se asocia con el if no pareado más cercano”**.

Interpretación 2 — El else pertenece al if externo

Casos:

Si `sum == 0` y `count == 0` → result = 0

Si `sum == 0` y `count != 0` → no hace nada

Si `sum != 0` → result = 1

Esta interpretación es **diferente** y puede ser la intención del programador.

Esto suele representar un problema pues la lectura del código se vuelve tediosa y ademas el codigo es ambiguo (La misma secuencia de tokens puede entenderse de dos maneras)

¿Cómo lo solucionan los lenguajes?

1. Regla semántica: "else con el if más cercano" (Java, C, C++, C#)

Esto elimina la ambigüedad para el compilador, pero **no para el lector humano**.

2. Uso obligatorio de bloques (Lenguajes basados en C)

Para forzar la interpretación deseada se usan { } , las llaves son obligatorias cuando el bloque del if o else tienen más de una sentencia.

```
if (sum == 0) {
        if (count == 0)
                result = 0;
}
else
   result = 1;
```

3. Finalizadores explícitos de bloques (Ada: end if; Fortran: End If; Ruby: end; Lua: end)

Ejemplo en Ruby:

```
if sum == 0 then
        if count == 0 then
                result = 0
        end
else
   result = 1
end
```

Aquí **no hay ambigüedad posible**, porque cada `if` se cierra con un `end`.

En nuestro lenguaje adoptaremos la solución de finalizadores explícitos de bloques.

Sintaxis de la sentencia IF

```
<if> ::= if ( <condicion> ) <block> endif
       | if ( <condicion> ) <block> else <block> endif
```

Ejemplo de uso:

```
if (x not in A)
        A= A union {x}
endif
```

Sintaxis de las estructuras de control

Ejemplo de uso:

<while> ::= **while (** <condicion> **) do** <block> **end**

<forall> ::= **forall (id in** <expr> **) do** <block> **end**

<forany> ::= **forany (id in** <expr> | <condicion> **) do** <block> **end**

1) while (A != B) do

   …

   end

2) forall (T in Transition) do

   …

   end

3) forany(Vertex in Graph / Vertex != false) do

   …

   end

Subrutinas

Las subrutinas permiten la modularización de un programa, de manera que no deba repetir código cada vez que deseo obtener un resultado aplicando argumentos a un algoritmo bien determinado, al que podemos llamar función o procedimiento, dependiendo de si el mismo devuelve o no algún dato simple o estructurado.

Gramática completa de Aleph

```
tree-sl: header main defs
;
header:
      | header fn_prototype
;
main: MAIN ':' block ENDMAIN
;
defs:
      | defs fn_definition
;
block:
      | list_stm
;
list_stm:   stm
      | list_stm stm
;
stm:   assign_s ';'
      | assign_m ';'
      | while_stm
      | if_stm
```

```
assign_s: ID '=' exp
;
assign_m: LET ID FLECHA '(' list_id ')'
;
while_stm: WHILE '(' exp ')' DO block END
;
if_stm: IF '(' exp ')' block ENDIF
      | IF '(' exp ')' block ELSE block ENDIF
;
forall_stm: FORALL '(' ID IN exp ')' DO
block END
;
forany_stm: FORANY '(' ID IN exp '|' exp ')'
DO block END
;
exp: ID
      | exp OR exp
      | exp AND exp
      | NOT exp
      | exp CMP exp
      | exp UNION exp
```

```
      | exp CONCAT exp
      | ADD exp TO exp
      | KICK exp FROM exp
      | exp '+' exp
      | exp '-' exp
      | exp '*' exp
      | exp '/' exp
      | '|' exp '|'
      | '-' exp %prec NEGATIVO
      | NUMBER
      | BOOL_LIT
      | lit_struct
      | fn_call
      | '(' exp ')'
;
return_stm: RETURN exp ';'
;
fn_prototype: FN ID '(' list_id ')' ':' type ';'
;
fn_definition: FN ID '(' list_id ')' ':' block
ENDFN
```

```
list_id: ID
        | list_id ',' ID
;
lit_struct: '{' '}' | '[' ']'
        | '{' list_exp '}' | '[' list_exp ']'
;
list_exp: exp
        | list_exp ',' exp
;
fn_call: ID '(' list_id ')'
        | ID '(' ')'
;
type: INT | SET | LIST | STRING | BOOLEAN


;
;
```

3.4. Semántica: Métodos Formales

En esta sección se especificará la semántica, el significado, de las construcciones sintácticas descriptas en la sección anterior del presente informe. Las herramientas formales para esta tarea no son tan conocidas como las herramientas formales para la sintaxis. No obstante, estas herramientas pueden ser utilizadas para complementar las definiciones de algunas construcciones que exijan una mayor precisión en su descripción semántica que aquella que puede brindarse mediante una definición coloquial de su funcionamiento.

--->poner aquí Def Semántica

3.4.1. Semántica Estática

--->Poner aquí Def Semántica Estática

Gramática de Atributos

def: gram atributo:

Recordndo los tipos de datos de Tree-SL son: INT, STRING, DOUBLE, LIST, SET, BOOL, UNDEFINED

exp::= ID

exp.type = lookup_tipo(lexeme_de(ID))

exp.val = lookup_val(lexeme_de(ID))

predicado: //no hay

Operaciones Lógicas

Las convenciones que toma Tree-SL para valor de verdad en su distintos tipos de datos son las siguientes:

- Listas:
  - Una lista vacía se considera falsa.
  - Una lista con uno o más elementos se considera verdadera.
- Conjuntos:
  - Un conjunto vacío se considera falso.
  - Un conjunto con uno o más elementos se considera verdadero.
- Números:
  - Punto flotante extendido: El valor 0.0…0 es falso; cualquier otro número es verdadero.
  - Enteros: El valor 0 es falso; cualquier otro número es verdadero.
- Cadenas de texto:

Cualquier cadena, independientemente de su contenido (incluida la cadena vacía, aunque no se especifica explícitamente), se considera verdadera.

exp::= exp OR exp

exp.type = Bool

exp.val = fun_Or(exp[1],exp[2]);

predicado: no hay

exp:== exp AND exp

exp.type=Bool

exp.val = fun_And(exp[1],exp[2]);

predicado: no hay

exp:== NOT exp

exp.type = Bool

exp.val = fun_Not(exp[1])

predicado: no hay

Operacionales Relacionales

CMP es el token que representa cualquier operación relacional son <, >, <=, >=, ==, !=. Tree-SL verifica que los tipos de los operandos sean comparables, es decir, del mismo tipo.

exp:== exp CMP exp

exp.type= Bool

exp.val= comparaData(CMP,exp[1],exp[2])

predicado: [1].type == [2].type

Operaciones Binarias entre Conjuntos

exp:== exp UNION exp

exp.type= SET

exp.val= UniontData([1],[2])

predicado: [1].type == [2].type == SET

exp:== exp INTER exp

exp.type = SET

exp.valor = InterstData([1],[2])

predicado: [1].type == [2].type == SET

exp:== exp DIFF exp

exp.type= SET

exp.val= diferentData([1],[2])

predicado: [1].type == [2].type == SET

Operaciones Unaria de Listas

exp:== TAKE exp FROM exp

exp.type= LIST

exp.val = take(exp[1], exp[2])

predicado: exp[1].type == INT && exp[2].type==LIST //exp[1] >= 0

exp:== ADD exp TO exp

exp.type= LIST

exp.val = add(exp[1], exp[2])

predicado: exp[1].type != UNDEFINED && exp [2].type==LIST

exp:== KICK exp FROM exp //exp[1] es un nro

exp.type = LIST

exp.val = kick(exp[1], exp[2])

predicado: exp[1].type == INT and exp[2].type == LIST

Operaciones Binarias entre listas

exp:== exp CONCAT exp

exp.type= LIST

exp.val = concat(exp[1], exp[2])

predicado: [1].type == [2].type==LIST

Operaciones Binarias entre números

exp:== exp '+' exp

exp.type = INT if (exp[1].type == exp[2].type == INT) else: DOUBLE

exp.val = opSuma(exp[1],exp[2])

predicado: exp[1].type == INT or exp[1] == DOUBLE and exp[2].type == INT or exp[2].type == DOUBLE

exp:== exp '-' exp

exp.type = INT if (exp[1].type == exp[2].type == INT) else: DOUBLE

exp.val = opResta(exp[1],exp[2])

predicado: exp[1].type == INT or exp[1] == DOUBLE and exp[2].type == INT or exp[2].type == DOUBLE

exp:== exp '*' exp

exp.type = INT if (exp[1].type == exp[2].type == INT) else: DOUBLE

exp.val = opProd(exp[1],exp[2])

predicado: exp[1].type == INT or exp[1] == DOUBLE and exp[2].type == INT or exp[2].type == DOUBLE

exp:== exp '/' exp

exp.type = INT if (exp[1].type == exp[2].type == INT) else: DOUBLE

exp.val = opDiv(exp[1],exp[2])

predicado: exp[1].type == INT or exp[1] == DOUBLE and exp[2].type == INT or exp[2].type == DOUBLE

exp:== '-' exp

exp.type = exp[1].type

exp.val = menosUnario(exp[1])

predicado: exp[1].type == INT or exp[1] == DOUBLE

exp:== '|' exp '|'

exp.type = exp[1].type

exp.val = valAbs_cardinal(exp[1])

predicado: exp[1].type == INT or exp[1] == DOUBLE or exp[1] == LIST or exp[1] == SET

exp:== NUMBER_INT

exp.type = INT

exp.valor = atoi(lexeme_de(NUMBER_INT))

predicado: no hay

exp:== NUMBER_DOUBLE

exp.type = DOUBLE

exp.valor = atoi(lexeme_de(NUMBER_DOUBLE))

predicado: no hay

exp:== BOOL_LIT

exp.type = Bool

exp.val = lexemeToBool(lexeme_de(BOOL_LIT))

predicado: no hay

exp:== lit_struct

exp.type = lit_struct.type

exp.val = lit_struct.val

predicado: no hay

exp:== fn_call

exp.type = fn_call.type

exp.val = fn_call.val

predicado: no hay

exp:== '(' exp ')'

;

3.4.2. Semántica Dinámica

--->Poner aquí Def Semántica Dinámica

Semántica Operacional

El lenguaje intermedio para definir el comportamiento operacional de las estructuras de control de Tree-SL.

⟨Programa⟩ ::= ⟨Instrucción⟩*

⟨Instrucción⟩ ::=

x := e (asignación)

| if e goto L (salto condicional)

| goto L (salto incondicional)

| label L : (etiqueta)

⟨Expresión⟩ ::=

c (constante entera o real)

| x (variable)

| e1 + e2 | e1 - e2 | e1 * e2 | e1 / e2

| [e1, e2, …, en] (lista)

| {e1, e2, …, en} (conjunto)

| in(e1, e2) (pertenencia)

| len(e) (longitud lista o tamaño conjunto)

T ::= int | double | list(T) | set(T)

--->Poner aquí Def Semántica Operacional

Descripción las estructuras de control de Aleph

Definición de lenguaje intermedio:

Un Lenguaje Intermedio (IR) (del inglés, *Intermediate Representation*) es una estructura de datos o un lenguaje de programación abstracto utilizado internamente por un compilador o intérprete para representar el código fuente de un programa.

Se define el siguiente IR para la descripción de las estructuras de control del lenguaje Tree-SL:

Tipo	Instrucción	Parámetros	Semántica Operacional
Control	GOTO L	Linea L	Salta incondicionalmente a la etiqueta L.
	IF V, GOTO L	Value V, Linea L	**Si** el valor booleano en V es **TRUE**, salta a L.
	IFN V, GOTO L	Value V, Linea L	**Si** el valor booleano en V es **FALSE**, salta a L.
Evaluación	EVAL Exp	Expresión Exp	Evalúa la expresión Exp y le da un valor a la variable.
	ASSIGN ID, V	Identificador ID, Value V	Asigna el contenido del valor o variable V a la variable persistente ID en el entorno.
	DO	Bloque B	Ejecuta secuencialmente las instrucciones del bloque B
Colección	ITER_INIT I, D	Iterador I, Colección D	Inicializa el estado del iterador I sobre la colección D.
	ITER_HASNEXT V, I	Value V, Iterador I	Asigna TRUE a V si el iterador I tiene un elemento siguiente; FALSE si terminó.
	ITER_GETNEXT V, I	Value V, Iterador I	Asigna el siguiente elemento a V y avanza el estado interno de I.

Descripción de las estructuras de control en TREE-SL con semántica operacional:

Sentencia if: <if> ::= **if (** <condicion> **)** <block> **else** <block> **endif**

1. EVAL <condicion>
2. IFN <condicion> GOTO 5 //saltar a realizar el else si no entra al if
3. DO <block>[1]
4. GOTO 6
5. DO <block>[2]
6.

Sentencia while: <while> ::= **while (** <condicion> **) do** <block> **end**

1. EVAL <condicion>
2. IFN <condicion> GOTO 5 //si la condicion no es verdadera salir del bucle
3. DO <block>
4. GOTO 1 //volver a evaluar la condicion
5.

Sentencia forall: <forall> ::= **forall (ID in** <expr> [| <condicion>] **) do** <block> **end**

1. ITER_INIT Ite <expr> //inicia un iterador genérico de colecciones
2. IFN ITER_HASNEXT Ite GOTO 9 //se fija si tiene un siguiente elemento, sino sale
3. IT_GETNEXT V_elem Ite //toma el elemento siguiente
4. ASSIGN ID V_elem
5. EVAL <condicion>
6. IFN <condicion> GOTO 2
7. DO <block>
8. GOTO 2 //realiza la iteración por toda la colección
9.

Sentencia forany: <forany> ::= **forany (ID in** <expr> [| <condicion>] **) do** <block> **end**

1. ITER_INIT Ite <expr> //inicia un iterador genérico de colecciones
2. IFN ITER_HASNEXT Ite GOTO 9 //se fija si tiene un siguiente elemento, sino sale
3. IT_GETNEXT V_elem Ite //toma el elemento siguiente
4. ASSIGN ID V_elem
5. EVAL <condicion>

6. IFN <condicion> GOTO 2
7. DO <block>
8. GOTO 9 //tiene una terminación temprana
9.

Semántica Denotacional

Se ha explicado a detalle la sintaxis de Tree-SL. La forma o estructura que deben tener sus programas. Las cadenas válidas sintácticamente (expresiones, sentencias). En contraste con la sintaxis y las gramáticas de atributos, que son modelos utilizados en teoría de intérpretes para resolver " The syntax problem ". La **semántica denotacional** es un modelo formal utilizado para la descripción del significado de los programas y su comportamiento durante la ejecución de una construcción en un lenguaje de programación con precisión matemática.

"The semantic problem"

El problema de la semántica es mucho más complejo que el de la sintaxis, de hecho el lenguaje natural es sumamente ambiguo inconsistente e incompleto. Lo que lo vuelve difícil de modelar. Por ello no existe un lenguaje universal para la semántica ni el análisis semántico de lenguajes en general como si lo hay para modelar y estudiar las sintaxis de un lenguaje. Una ilustración que demuestra lo complejo que es el lenguaje Ingles para modelar su significado deterministicamente.

Ambigüedad: Palabras, frases y oraciones pueden llegar a tener distintos significados que son resueltos en el **contexto.** eg: "The back is by the river". Esta oración presenta ambigüedad léxica, donde "bank" tiene varios significados.

Incompletitud: Normalmente son omitidas palabras porque el diálogo y la comunicación está basado en el contexto. eg: A: "Where are you going?". B: "To the store".

Inconsistente: Debido a lo rico en cuanto significado tanto semántico y sintáctico (donde la forma de las oraciones moldea, cambia y dirige el significado de una cadena). Sumado a que el uso del lenguaje es amplio (chats, emails, llamadas, etc). Vuelve al lenguaje natural inconsistente. eg: palabras no concuerdan con su pronunciación como: "though", "though", "through", "thorough".

La semántica denotacional es una de los mas rigurosos modelos para aproximar y construir un modelo matemático del significado de un programa.

Definición: Define el significado de un programa asignando una función a cada construcción sintáctica. Dicha función modela el efecto computacional correspondiente de dicha construcción

La composición es una de las grandes ventajas de este modelo, al permitir que el significado se construya sistemáticamente de las partes. Además se abstrae del modelo de la máquina, lo que lo vuelve portable y no ambiguo. Por último es muy utilizado para la verificación del correcto funcionamiento de compiladores, aunque para esto es necesario matemáticas avanzadas.

Descripción las estructuras de control de Aleph

El dominio semántico de Tree-SL es el estado del programa.

> S = {<i1, v1>, <i2, v2>, … , <in, vn>}, es el estado del programa. Donde ik es el nombre de las variables y vk es el valor donde vk pertenece a Data, con k = 1…n. Donde Data es el siguiente conjunto: Data = {int, double, boolean, string, list, set}

Definamos la siguiente función, que nos ayudará a la formalización de la semántica del lenguaje. Además, el error es un valor semántico posible en la evaluacion de la semántica del lenguaje por lo tanto, en vez de negar su existencia lo tendremos en cuenta.

> Sea VARMAP la funcion que mapea la variable j-esima del estado del programa S a su correspondiente valor para el estado S
>
> VARMAP(ij, S) = vj

Primero definamos las funciones que denotan el significado de una expresión y asignación en un programa de Tree-SL. Una notación que usaremos es '**def:**' usada para definir las funciones de notacionales para una construcción sintáctica y distinguirla del '=' de las asignaciones en el pseudo-lenguaje utilizado para las definiciones. Como sabemos la variable no terminal <exp> en Tree-SL posee varias derivaciones, definamos una pseudo variable no terminal <exp>, de la siguiente forma. Adicionalmente supongamos que tenemos las subrutinas adecuadas: compute_op_data(), que efectúa la operación definida por el operador OP_DATA entre dos operandos de tipo data. Dicha función realiza el chequeo de tipos correspondiente, de forma similar agregar() agrega en una lista o conjunto un elemento del conjunto Data.

> <exp> ::= **LITERALES** | **ID** | <lit_struct>
>
> | <exp> **OP_DATA** <exp>
>
> <lit_struct> ::= [<list_exp>] | { <list_exp> }
>
> <list_exp> ::= <exp> | <exp> , <exp_list>

Sea Mstruct la funcion que mapea una construccion <lit_struct> y S el estado del programa a un elemento del conjunto Data union {error}

```
Mstruct(<lit_struct>, S) def:
        switch type of <lit_struct>:
        case [ <list_exp> ]:
                lista = []
                for <exp> in <list_exp>:
                        agregar( lista, Me(<exp>) )
                lista
        // igual en case conjunto
```

Sea Me la funcion que mapea para dada una expresion <exp> y el estado del programa S a un elemento del conjunto Data union {error}

```
Me(<exp>, S) def:
        switch type of <exp>:
        Case LITERALES: lit_to_data(<exp>)
        Case ID:
                if (VARMAP(ID, S) == undef) then error
                else  VARMAP(ID, S)
        Case <lit_struct>: Mstruct(<exp>, S)
        Case <exp> OP_DATA <exp>:
                compute_op_data(OP_DATA, <exp>[1], <exp>[2])
```

<asign> ::= **ID** = <exp>

Sea Masign la funcion que mapea la construccion <asign> y un estado del programa S a otro estado del programa S'.

Masign(<asign>, S) = Masign(<ID> = <exp>, S) def:

    if (Me(<exp>, S) == error) then error

    else:

        S' = {<i1, v1'>, <i1, v2'>, … , <i1, vn'>}

        for j = 1 … n

            if (ij == **ID**) then vj' = Me(<exp>, S)

Para la formalizacion de las estructuras de control de Tree-SL falta definir dos funciones, Mbool y Mblock. Mblock es una funcion que recibe un construccion de un bloque es decir un conjunto de 0 o mas sentencias las ejecuta secuencialmente modifican el estado del programa, esta funcion mapea a un nuevo estado del programa. Y supongamos que existe una subrutina bool_value() que computa el valor de verdad de un elemento del conjunto Data

Sea Mbool la funcion que mapea para dada una expresion <exp> y el estado del programa S a un elemento del conjunto {True, False, error}

Mbool(<exp>, S) def:

    switch type of Me(<exp>, S):

    Case **DATA**: bool_value(Me(<exp>, S))

    Case **ID**:

        if (VARMAP(**ID,** S) == undef) then error

        else bool_value(VARMAP(**ID,** S))

Sea Mblock la funcion que mapea una construccion de bloque y el estado del programa S a un nuevo estado del programa S'

Mblock(<block>, S) = Mblock(<stm>, <block>, S) def:

    if(Mstm(<stm>, S) == error) then error

    else Mblock(<block>, Mstm(<stm>, S))

<while> ::= **while (<exp>) do <block> end**

Sea Mwhile la funcion que mapea una construccion while y un estado del programa S a un estado del programa S' union {error}.

Mwhile(<while>, S) = Mwhile(<exp>, <block>, S) def:

```
if (Mbool( <exp>, S ) == error) then error
else:
    if (Mbool( <exp>, S ) == False) then S
    else Mwhile(<while>, Mblock(<block>, S ))
```

<if> ::= **if (<exp>) <block> else <block> endif**

Sea Mif la funcion que mapea una construccion if y un estado del programa S a un estado del programa S' union {error}.

Mif(<if>, S) = Mif(<exp>, <block>, <block>, S) def:

```
if (Mbool( <exp>, S ) == error) then error
else:
    if (Mbool( <exp>, S ) == False) then Mblock(<block>[0], S)
    else Mblock(<block>[1], S)
```

<forall> ::= **forall (ID in <exp> | <exp>) do <block> end**

Sea Mforall la funcion que mapea una construccion forall y un estado del programa S a un estado del programa S' union {error}.

Mforall(<forall>, S) = Mforall(<exp>, <exp>, <block>, S) def:

```
if (Me( <exp>[0], S ) == error) then error
elif (Mbool( <exp>[1], S ) == error) then error
else:
    for elem in Me( <exp>[0], S ):  // definir si <block> cambia S
        S' = Masign(ID, elem)      // definir si ID ya existe
        S = S'
        if (Mbool( <exp>[1], S ) == True) then
            S' = Mblock(<block>, S)
            S = S'
S // devolver S
```

<forany> ::= **forany (ID in** <expr> | <exp> **) do** <block> **end**

Sea Mforall la funcion que mapea una construccion forall y un estado del programa S a un estado del programa S' union {error}.

```
Mforany(<forany>, S) = Mforany(<exp>, <exp>, <block>, S) def:
        if (Me( <exp>[0], S ) == error) then error
        elif (Mbool( <exp>[1], S ) == error) then error
        else:
                for elem in Me( <exp>[0], S ):
                        S' = Masign(ID, elem)      // definir si ID ya existe
                        S = S'
                        if (Mbool( <exp>[1], S ) == True) then
                                S' = Mblock(<block>, S)
                                S = S'
                                break
                S // devolver S
```

Por ultimo se define el significado semantico de la asignacion multiple

<assign_m> ::= **let ID => (** <list_id> **)**

<list_id> ::= **ID** | **ID ,** <list_id>

Sea Massign_m la funcion que mapea una construccion de asignacion multiple y un estado del programa S a un estado del programa S' union {error}. Y supongamos que existe una subrutina es_struct que devuelve el valor de verdad si un elemento del conjunto Data es una lista o set. Otra sub rutina tamanio que determina el tamanio de listas o conjuntos. Y una sub rutina getElem() que retorna el valor del i-esimo elemento de una lista o conjunto

```
Massign_m(<forany>, S) = Masign_m(ID, <list_id>, S) def:
        if (VARMAP(ID, S)  == undef) then error
        elif ( not es_struct(VARMAP(ID, S))) then error
        elif (tamanio(VARMAP(ID, S)) < tamanio(<list_id>)) then error
        else
                for i = 1… tamanio(<list_id>) :
                        S' = Masign(<list_id>[i], getElem(i, ID))
        S' // devolver S
```

4. Fase 1 de implementación de Aleph: análisis léxico y sintáctico

4.1. Tipos de implementación

--->Def de los distintos enfoques de implementación de lenguajes

--->Enfoque de implementación de Aleph

4.2. Análisis Léxico

Un analizador léxico funciona como una interfaz de un analizador sintáctico. Para entender esta definición debemos entender como funciona este analizador:

Sabemos que un programa de entrada cuando se compila, es presentado como una sola cadena de caracteres sin significancia ni distinción entre esta cadena. Aquí es donde entra en juego el analizador léxico, este agrupa los caracteres correspondientes en grupos lógicos y según su estructura les asigna un código interno (hablando técnicamente se entiende como grupos lógicos a los lexemas y al código interno como tokens para el parser). En los inicios de los compiladores, los analizadores léxicos solían procesar un archivo de programa fuente completo y generar un archivo de tokens y lexemas. Sin embargo, ahora la mayoría de los analizadores léxicos son subprogramas que localizan el siguiente lexema en la entrada, determinan su código de token asociado y lo devuelven al analizador sintáctico, que es quien lo llama. Por lo tanto, cada llamada al analizador léxico devuelve un solo lexema y su token. La única vista del programa de entrada que ve el analizador sintáctico es la salida del analizador léxico, un token a la vez.

De lo enunciado anteriormente se pueden concluir las tareas de un analizador léxico:

- Eliminar detalles irrelevantes de código: ignora espacios en blanco, comentarios o tabulaciones que no tienen relevancia sintáctica.
- Identificar tokens: clasifica secuencias de caracteres como identificadores, palabras clave, literales, operadores, etc.
- Gestionar errores léxicos: detecta símbolos no reconocidos o secuencias mal formadas, informando de manera clara para ayudar al programador a encontrar errores.
- Insertar lexemas para identificadores definidos por el usuario en la tabla de símbolos, que se utiliza en fases posteriores del compilador.
- Mejorar la eficiencia: al separar el análisis léxico del sintáctico, se simplifica el diseño del parser y se mejora el rendimiento del compilador.

- Primer prototipo: scanner que imprime tokens reconocidos.

A continuación se expone el primer analizador léxico utilizando tokens correspondientes reconocidos en el bloque 2.2.1. Obsérvese que no se trabajó con el printf en el main porque resultó más fácil utilizar una línea de impresión en la segunda sección del código en flex que utilizar un switch o varios if en la última sección.

```
%{
#include <stdio.h>
%}
%%

"let" { printf("LET: %s\n",yytext); }
"return" { printf("RETORNO: %s\n",yytext); }

"if" { printf("IF: %s\n",yytext); }
"else" { printf("ELSE: %s\n",yytext); }
"endif" { printf("ENDIF: %s\n",yytext); }
"while" { printf("WHILE: %s\n",yytext); }
"forall" { printf("FORALL: %s\n",yytext); }
"forany" { printf("FORANY: %s\n",yytext); }
"do" { printf("DO: %s\n",yytext); }
"end" { printf("END: %s\n",yytext); }

"union" { printf("UNION: %s\n",yytext); }
"inter" { printf("INTERSECCION: %s\n",yytext); }
"diff" { printf("DIFERENCIA: %s\n",yytext); }
"in" { printf("IN: %s\n",yytext); }
"contains" { printf("CONTAINS: %s\n",yytext); }
"concat" { printf("CONCAT: %s\n",yytext); }
"take" { printf("TAKE: %s\n",yytext); }
"kick" { printf("SACAR: %s\n",yytext); }
"add" { printf("ANADIR: %s\n",yytext); }
```

```
"||" { printf("CARD: %s\n",yytext); }

"true" { printf("BOOLEAN: %s\n",yytext); }            /*booleanos*/
"false" { printf("BOOLEAN: %s\n",yytext); }

"+" { printf("SUMA: %s\n",yytext); }
"-" { printf("RESTA: %s\n",yytext); }
"*" { printf("PROD: %s\n",yytext); }
"/" { printf("DIV: %s\n",yytext); }
"%" { printf("MOD: %s\n",yytext); }
"==" { printf("IGUALDAD: %s\n",yytext); }
"!=" { printf("DESIGUALDAD: %s\n",yytext); }
">" { printf("MAYOR: %s\n",yytext); }
"<" { printf("MENOR: %s\n",yytext); }
"<=" { printf("MENOR_IG: %s\n",yytext); }
">=" { printf("MAYOR_IG: %s\n",yytext); }

"=" { printf("ASIGNACION_SIMP: %s\n",yytext); }
"=>" { printf("ASIGNACION_MULT: %s\n",yytext); }
"{" { printf("LLAVE_ABIERTA: %s\n",yytext); }
"}" { printf("LLAVE_CERRADA: %s\n",yytext); }         /*simbolos del lenguaje*/
"[" { printf("CORCHETE_ABIERTA: %s\n",yytext); }
"]" { printf("CORCHETE_CERRADA: %s\n",yytext); }
"(" { printf("PARENTESIS_ABIERTA: %s\n",yytext); }
")" { printf("PARENTESIS_CERRADA: %s\n",yytext); }
\" { printf("COMILLA: %s\n",yytext); }
";" { printf("PUNTO_COMA: %s\n",yytext); }
":" { printf("DOS_PUNTOS: %s\n",yytext); }
"," { printf("COMA: %s\n",yytext); }

[a-zA-Z][a-zA-Z0-9]* { printf("ID: %s\n",yytext); }     /*identificadores (el primer símbolo de un
identificador es una letra)*/
```

```
[ \t\n] ;   /* ignora espacios y saltos de línea */
. { printf("CARACTER DESCONOCIDO: %s\n", yytext); }

%%
/*tercera seccion*/
int main(void){
  yylex();
  return 0;
}
```

Otra forma de hacerlo utilizando enum para los tokens:

```
%{
/*primera seccion*/
#include <stdio.h>

enum yytokentype{
LET = 256,
RETURN = 257,

IF = 258,
ELSE = 259,
ENDIF = 260,
WHILE = 261,
FORALL = 262,
FORANY = 263,
DO = 264,
END = 265,

UNION = 266,
INTER = 267,
DIFF = 268,
IN = 269,
```

CONTAINS = 270,

CONCAT = 271,

TAKE = 272,

KICK = 273,

ADD = 274,

CARD = 275,

BOOLEAN = 276,

SUMA = 277,

RESTA = 278,

PROD = 279,

DIV = 280,

MOD = 281,

IGUALDAD = 282,

DESIGUALDAD = 283,

MAYOR = 284,

MENOR = 285,

MENOR_IG = 286,

MAYOR_IG = 287,

ASIGNACION_SIMP = 288,

ASIGNACION_MULT = 289,

LLAVE_ABIERTA = 290,

LLAVE_CERRADA = 291,

CORCHETE_ABIERTA = 292,

CORCHETE_CERRADA = 293,

PARENTESIS_ABIERTA = 294,

PARENTESIS_CERRADA = 295,

COMILLA = 296,

PUNTO_COMA = 297,

DOS_PUNTOS = 298,

COMA = 299,

```
ID = 300,
CARACTER_DESCONOCIDO = 301,
EOL = 302
};
int yylval;
%}
%%

"let" { return LET; }
"return" { return RETURN; }

"if" { return IF; }
"else" { return ELSE; }
"endif" { return ENDIF; }
"while" { return WHILE; }
"forall" { return FORALL; }
"forany" { return FORANY; }
"do" { return DO; }
"end" { return END; }

"union" { return UNION; }
"inter" { return INTER; }
"diff" { return DIFF; }
"in" { return IN; }
"contains" { return CONTAINS; }
"concat" { return CONCAT; }
"take" { return TAKE; }
"kick" { return KICK; }
"add" { return ADD; }
"||" { return CARD; }

"true" { return BOOLEAN; }
"false" { return BOOLEAN; }
```

```
"+" { return SUMA; }
"-" { return RESTA; }
"*" { return PROD; }
"/" { return DIV; }
"%" { return MOD; }
"==" { return IGUALDAD; }
"!=" { return DESIGUALDAD; }
">" { return MAYOR; }
"<" { return MENOR; }
"<=" { return MENOR_IG; }
">=" { return MAYOR_IG; }

"=" { return ASIGNACION_SIMP; }
"=>" { return ASIGNACION_MULT; }
"{" { return LLAVE_ABIERTA; }
"}" { return LLAVE_CERRADA; }
"[" { return CORCHETE_ABIERTA; }
"]" { return CORCHETE_CERRADA; }
"(" { return PARENTESIS_ABIERTA; }
")" { return PARENTESIS_CERRADA; }
\" { return COMILLA; }
";" { return PUNTO_COMA; }
":" { return DOS_PUNTOS; }
"," { return COMA; }

[a-zA-Z][a-zA-Z0-9]* { return ID; }      /*identificadores (el primer simbolo de un identificador es una letra)*/


\n { return EOL; }
[\t] ; /* ignora espacios y saltos de línea */
. { return CARACTER_DESCONOCIDO; }

%%
```

```
/*tercera seccion*/
int main(void){
   int tok;
   while(tok = yylex()) {
      printf("\nLexema: %s y Token: %d",yytext,tok);
   }
   return 0;
}
```

4.3. Análisis Sintáctico

Un analizador sintáctico es la parte del compilador que recibe la secuencia de tokens producida por el analizador léxico (scanner) y determina si esa secuencia forma una estructura sintácticamente correcta de acuerdo con las reglas gramaticales del lenguaje de programación.

Dicho de otro modo, el parser verifica la estructura del programa, asegurándose de que las sentencias, expresiones, bloques, etc., estén organizados de acuerdo con la gramática formal del lenguaje.

Por lo tanto se puede decir que el seguimiento que realiza una analizador sintáctico es:

1. Recibe los tokens del analizador léxico.
2. Consulta la gramática (en BNF o CFG).
3. Intenta derivar la secuencia de tokens desde el símbolo inicial de la gramática.
4. Si logra derivarla completamente la entrada es sintácticamente correcta. Si no se detecta un error sintáctico.

De las tareas anteriores del parser podemos apuntar a los objetivos que este tiene:

- Verificación sintáctica: Comprobar si el programa fuente cumple las reglas gramaticales. Cuando se encuentra un error, el analizador debe producir un mensaje diagnóstico y recuperarse. En este contexto, recuperarse significa volver a un estado normal y continuar el análisis del programa de entrada. Este paso es necesario para que el compilador encuentre la mayor cantidad posible de errores durante un solo análisis del programa. Si no se realiza adecuadamente, la recuperación de errores puede crear más errores o, al menos, más mensajes de error.
- Construcción de una representación estructural: Generar una representación jerárquica del programa, generalmente en forma de árbol sintáctico (parse tree) o árbol de sintaxis abstracta

(AST, Abstract Syntax Tree). Esta estructura es la base para las siguientes fases del compilador, como el análisis semántico o la generación de código.

Para profundizar más sobre el parser debemos saber sobre las dos clases de derivación que se utilizan:

Top-Down: Como su nombre lo dice, de arriba hacia abajo, el árbol se construye desde la raíz hacia las hojas. Esta construcción preorder (de orden previo) en la que cada nodo se visita antes de seguir sus ramas, y las ramas de un nodo se recorren de izquierda a derecha lo cual corresponde a una derivación por izquierda.

Dicho de otras palabras, dada una forma sentencial que es parte de una derivación por la izquierda, la tarea del analizador es encontrar la siguiente forma sentencial en esa derivación. Ahora vamos a exponer el funcionamiento que realiza esta clase de derivación:

Inicio desde el símbolo inicial: El analizador comienza con el símbolo inicial de la gramática (por ejemplo, <program> o <expr>) donde el símbolo representa todo el lenguaje, y el parser intentará derivar a partir de él la cadena de tokens que llega desde el analizador léxico.

Selección de reglas (producciones): En cada paso, el analizador busca el no terminal más a la izquierda en la forma sentencial actual y elige una regla de producción para expandirlo. Por ejemplo, si la forma actual es: <expr> y existen reglas como: <expr> → <term> + <expr> | <term> entonces el parser debe decidir cuál de las dos usar dependiendo del siguiente token de entrada.

Comparación con el siguiente token: El analizador compara el token actual de la entrada (por ejemplo, un id, (, +, etc.) con los posibles símbolos iniciales de las reglas que puede aplicar. Esto se hace con la ayuda del conjunto FIRST de cada regla (es decir, los símbolos con los que puede empezar una derivación de esa producción). Si el token coincide con alguno de los posibles primeros símbolos de una producción, se elige esa regla.

1. Expansión recursiva: Una vez elegida la regla, el analizador:
2. Sustituye el no terminal por el lado derecho de la producción (RHS).
3. Continúa con el siguiente no termina más a la izquierda.
4. Si el símbolo actual es un terminal, lo compara directamente con el token de entrada:
    - Si coinciden, consume el token (avanza al siguiente).
    - Si no, se produce un error sintáctico.

Este proceso se repite recursivamente hasta que: Se consumen todos los tokens, y se genera una derivación completa del símbolo inicial. Si esto ocurre sin errores, la cadena es sintácticamente correcta. El análisis termina cuando no quedan tokens de entrada, o el árbol generado a partir del símbolo inicial coincide completamente con la entrada.

En ese punto, el parser ha confirmado que la estructura del código fuente sigue las reglas gramaticales del lenguaje.

En los analizadores Top-Down existen dos tipos principales:

Analizadores LL donde LL viene de Left-to-right scan (el analizador lee la entrada de izquierda a derecha) y Leftmost derivation (genera una derivación por la izquierda). Un LL usa una tabla de análisis predictivo (*parsing table*) para decidir qué regla aplicar en cada paso.

1. Tiene una pila con símbolos de la gramática (empieza con el símbolo inicial).
2. Lee un token de entrada (desde el analizador léxico).
3. Mira la parte superior de la pila:
    - Si es un terminal y coincide con el token lo consume.
    - Si es un no terminal, consulta la tabla LL(1) para decidir qué producción aplicar.
4. Sustituye el no terminal por el lado derecho de la producción (RHS).
5. Repite hasta vaciar la pila y consumir todos los tokens.

Pero se deben cumplir ciertas condiciones para que una gramática sea LL, ya que no todas las gramáticas sirven. Debe estar libre de ambigüedades y sin recursión por la izquierda, además de cumplir que:

- Ningún par de reglas de un mismo no terminal tenga símbolos iniciales (FIRST) que se superpongan.
- Si una regla puede producir ε (cadena vacía), sus símbolos FOLLOW no deben superponerse con los FIRST de las demás reglas.

Analizadores Descendentes Recursivos es otro tipo de analizador Top-Down implementado directamente en el código, en la gramática BNF del lenguaje, escrita en forma de conjunto de procedimientos o funciones recursivas. Esta recursión refleja la naturaleza de los lenguajes de programación que incluyen varios tipos de estructuras anidadas entre sí. Por ejemplo, las sentencias suelen estar anidadas dentro de otras sentencias. Asimismo, los paréntesis en las expresiones deben

estar correctamente anidados. La sintaxis de estas estructuras se describe de forma natural mediante reglas gramaticales recursivas.

- Cada no terminal de la gramática se implementa como una función o subrutina.
- Cada función reconoce las construcciones que puede generar ese no terminal, según las reglas de producción de la gramática.
- Las llamadas recursivas reflejan las expansiones sintácticas.
- Los tokens terminales se comparan directamente con los símbolos de entrada.

De este modo, el proceso de análisis sigue la misma estructura que las reglas de la gramática.

Hay que resaltar que el descenso recursivo no puede usarse directamente con gramáticas que tienen recursión por la izquierda, porque esto causaría llamadas infinitas.

Funcionamiento del descenso recursivo:

1. El analizador comienza con el símbolo inicial del lenguaje.
2. Llama a la función correspondiente a ese no terminal.
3. Cada función puede:
    - Verificar tokens (cuando encuentra terminales).
    - Llamar a otras funciones (para expandir no terminales).
4. Si los tokens coinciden con la estructura esperada entonces el análisis prosigue.
5. Si no coinciden entonces se detecta un error sintáctico.

De estos dos tipos de analizadores podemos notar las claras diferencias:

Aspecto:	LL	Descendente recursivo
Implementación	Usa una tabla de análisis predictivo.	Usa funciones recursivas por no terminal.
Selección de reglas	Por tabla (algorítmica)	Por estructura del código
Requiere tabla FIRST/FOLLOW	Si	No necesariamente (se codifica manualmente)
Claridad	Más mecánico	Más intuitivo, cercano a la gramática
Facilidad de implementación	Más complejo, pero sistemático	Más fácil de programar a mano
Ejemplo típico	Generadores automáticos	Pequeños intérpretes o

	(como ANTLR)	compiladores educativos

--->Ejemplo de módulos descendentes recursivos de las expresiones de Aleph

Bottom-UP: Los analizadores sintácticos ascendentes, conocidos como parsers *bottom-up*, son una de las dos estrategias principales de análisis sintáctico utilizadas en los compiladores.

A diferencia de los analizadores descendentes, que construyen el árbol sintáctico desde la raíz hacia las hojas, los analizadores Bottom-UP inician el proceso desde los elementos más básicos de la cadena de entrada (las hojas) y avanzan progresivamente hasta llegar a la raíz del árbol, que representa el símbolo inicial de la gramática.

El análisis Bottom-UP puede interpretarse como el proceso inverso de una derivación por la derecha, esto significa que el analizador parte de la cadena completa de entrada y realiza una serie de reducciones que van reconstruyendo las reglas de la gramática en orden inverso al de la derivación original.

En otras palabras, en lugar de expandir los no terminales como en el enfoque descendente, el analizador Bottom-UP identifica subcadenas dentro de la entrada que coinciden con el lado derecho de una regla gramatical (RHS) y las reemplaza por el lado izquierdo (LHS) de dicha regla. Cada una de estas reducciones representa un paso en la reconstrucción del árbol sintáctico desde las hojas hacia la raíz.

Una de las tareas más importantes del análisis es detectar cuál es la subcadena correcta que debe reducirse en cada paso. Esa subcadena recibe el nombre de handle, y representa una ocurrencia válida del lado derecho de alguna regla de la gramática dentro de la cadena actual. Una vez identificado el handle, el analizador lo sustituye por su símbolo no terminal correspondiente, obteniendo así una nueva forma sentencial que se acerca un paso más al símbolo inicial del lenguaje. Ejemplo:

S->aAc

A->aA | b

S => aAc => aaAc => aabc

Encontrar el handle adecuado no siempre es sencillo. En muchos casos, una forma sentencial puede contener múltiples posibles coincidencias con los lados derechos de las reglas de la gramática. Por ello, los analizadores Bottom-UP deben incorporar mecanismos de decisión precisos para determinar cuál de esas coincidencias es la correcta en cada paso del análisis.

Algunas ventajas de utilizar este analizador son:

- Mayor capacidad para manejar gramáticas complejas: Los analizadores Bottom-UP pueden procesar un conjunto mucho más amplio de gramáticas libres de contexto que los analizadores Top-Down.
- Precisión y determinismo: Por cada estado del análisis y cada token de entrada, existe una única acción válida (desplazar o reducir).Esto evita la ambigüedad en las decisiones y mejora la fiabilidad del análisis.
- Mejor detección y recuperación de errores sintácticos: En este análisis se detectan errores sintácticos en el punto exacto donde se vuelven inevitables. Facilitando al compilador que produzca mensajes de error más precisos y en algunos casos pueda recuperarse y continuar el análisis, lo cual mejora la calidad del proceso de compilación.
- No requiere eliminar la recursión por la izquierda: A diferencia del enfoque Top-Down, este puede manejar directamente gramáticas con recursión por la izquierda, sin necesidad de transformarlas.
- Producción directa del árbol sintáctico: En este método se reconstruye el árbol sintáctico completo mientras se reduce las subcadenas de entrada permitiendo que el árbol resultante refleje de forma exacta y estructurada la jerarquía gramatical del programa, lo que es muy útil para las etapas posteriores del compilador (como la generación de código o la verificación semántica).

El **analizador LR** es uno de los tipos más potentes y ampliamente utilizados de analizadores **bottom-up**. Estos analizadores utilizan un programa relativamente pequeño y una tabla de análisis construida para un lenguaje de programación específico. El algoritmo LR original fue diseñado por Donald Knuth (Knuth, 1965). Este algoritmo, que a veces se denomina LR canónico, no se utilizó en los años inmediatamente posteriores a su publicación porque la producción de la tabla de análisis necesaria requería una gran cantidad de tiempo y memoria de computadora.

Este nombre proviene de su funcionamiento, pues:

- **L**: El analizador lee la **entrada de izquierda a derecha**.
- **R**: Se produce una **derivación por la derecha** pero en orden inverso.

El analizador LR funciona manteniendo una pila que almacena símbolos y estados, y utiliza una tabla de análisis que guía el proceso. Esta tabla indica, para cada combinación de estado actual y símbolo de entrada, qué acción debe realizar el analizador.

El proceso se basa en dos operaciones fundamentales:

1. Shift (desplazar): el analizador lee el siguiente token y lo coloca en la pila.
   Esto representa avanzar en el análisis de la cadena.
2. Reduce (reducir): el analizador identifica una secuencia de símbolos en la pila que coincide con el lado derecho (RHS) de una regla de producción. Esa secuencia se reemplaza por el no terminal correspondiente al lado izquierdo (LHS) de la producción.

5. Fase 2 de implementación de Aleph: primer análisis semántico

5.1. Construcción del AST y evaluación de expresiones

-->Definición de AST

-->Mostrar y explicar la estructura de datos y módulos que se usan para la construcción del AST

-->Mostrar y explicar la estructura de datos y módulos utilizados para evaluar los AST en Aleph

-->Evaluación de literales listas, conjuntos (sin elementos repetidos) y elemento (con sus operaciones no excluyente)

5.2. Variables y la Tabla de Símbolos

Variable

Una variable de programa es una abstracción de una celda o colección de celdas de memoria de una computadora. Los programadores suelen pensar en los nombres de las variables como nombres para ubicaciones de memoria, pero una variable es mucho más que solo un nombre.

Una variable se define a través de una secuencia de atributos clave: nombre, dirección, valor, tipo, tiempo de vida y alcance.

1- Nombre de la Variable : La mayoría de las variables poseen un nombre identificador.

Variables con nombre

En numerosos lenguajes de programación, especialmente aquellos derivados de C, la distinción entre letras mayúsculas y minúsculas en los identificadores es significativa, lo que se conoce como ser sensible a mayúsculas y minúsculas (*case sensitive*).

Ejemplos que representan entidades diferentes en lenguajes *case sensitive*:

- ROSE
- rose
- Rose

Para algunos programadores, esta sensibilidad puede comprometer la legibilidad, ya que nombres visualmente muy similares pueden referirse a entidades completamente distintas. En nuestro lenguaje, los nombres de las variables son sensibles a mayúsculas y minúsculas y no pueden ser palabras reservadas del lenguaje.

Reserved word (palabra reservada): Es una palabra especial de un lenguaje que no puede utilizarse como nombre de variable, función, etc.

En nuestro lenguaje tenemos las palabras reservadas :

Los tipos de datos: SET , ELEM, LIST; funciones: UNION, DIFERENCIA, | | (cardinal), etc .

Variables sin nombre:

Las variables que *no tienen nombre* son las *variables heap-dinámicas explícitas.* Estas son las variables creadas en el montón (heap) mediante instrucciones como new (en C++, Java, C#, etc.).

La clave es: La celda de memoria creada en el heap no tiene nombre.

El programador no le asigna un identificador.
Solo tiene una dirección, y esa dirección se guarda en un puntero o referencia.

Ejemplo: En C++:
int *p;
p = new int;

p sí tiene nombre → es un puntero,pero el int creado con **new int** no tiene nombre. Es solo una celda anónima en el heap.

Esa es la variable sin nombre que menciona Sebesta.

Más tarde puede ser liberada con **delete p**;, pero seguimos sin tener un identificador directo para la variable del heap — solo podemos acceder a ella a través del puntero.

Entonces una variable puede no tener nombre cuando se cumple esto:

✔ Se crea en el heap

✔ Es dinámica (se asigna en tiempo de ejecución)

✔ Se accede únicamente por un puntero o referencia

✔ No tiene un identificador propio

2- Dirección de una variable:

Es la ubicación en memoria donde se almacena. También se conoce como su **l-value**, porque es lo que se necesita cuando la variable aparece en el lado izquierdo de una asignación.

La asociación variable–dirección no siempre es fija:

En muchos lenguajes, una misma variable puede tener **diferentes direcciones en distintas ejecuciones**, por ejemplo cuando es una variable local asignada en el *stack* en cada llamada a un subprograma.

Alias:

Ocurre cuando **dos o más nombres de variables comparten la misma dirección de memoria**

- Esto dificulta la **legibilidad** porque un cambio en una variable afecta a la otra.
- También dificulta la **verificación del programa**.

Cómo se crean alias:

- Mediante **uniones (union)** en C/C++.
- Cuando **dos punteros o referencias apuntan al mismo lugar**.
- A través de **parámetros de subprogramas** en varios lenguajes.

3- Tipo de una variable:

El tipo de una variable determina el rango de valores que puede almacenar y el conjunto de operaciones definidas para los valores de ese tipo.

Por ejemplo:

El tipo `int` en Java especifica un rango de valores de -2147483648 a 2147483647 y operaciones aritméticas como la suma, la resta, la multiplicación, la división y el módulo.

4- Valor de una variable:

Es el contenido almacenado en la celda o celdas de memoria asociadas a esa variable.

El valor de una variable también se llama valor r, porque es lo que se necesita cuando la variable aparece en el lado derecho de una asignación.

Para obtener el valor r primero se debe conocer el valor l (la dirección).

Puede haber dificultad en determinar el valor, factores como las reglas de alcance (scope) pueden complicar determinar el valor correcto, según se explica más adelante.

5-Alcance:

El ámbito de una variable es el rango de instrucciones en las que la variable es visible. Una variable es visible en una instrucción si se puede hacer referencia a ella en dicha instrucción.

Las reglas de ámbito de un lenguaje determinan cómo se asocia una aparición particular de un nombre con una variable, o, en el caso de un lenguaje funcional, cómo se asocia un nombre con una expresión

6-Tiempo de Vida

El tiempo de vida de una variable es el período durante el cual la variable existe en la memoria del programa, es decir, desde que se le asigna espacio en memoria hasta que ese espacio se libera.

A veces, el ámbito y la duración de una variable parecen estar relacionados. Por ejemplo, consideremos una variable declarada en un método de Java que no contiene llamadas a otros métodos. El ámbito de dicha variable abarca desde su declaración hasta el final del método. La duración de esa variable es el periodo de tiempo que comienza cuando se entra al método y finaliza cuando termina su ejecución. Aunque el ámbito y la duración de la variable claramente no son lo mismo, ya que el ámbito estático es un concepto textual o espacial, mientras que la duración es un concepto temporal, al menos en este caso parecen estar relacionados.

Esta aparente relación entre ámbito y duración no se cumple en otras situaciones. En C y C++, por ejemplo, una variable declarada en una función con el especificador `static` se vincula estáticamente al ámbito de dicha función y también a la memoria. Por lo tanto, su ámbito es estático y local a la función, pero su duración se extiende durante toda la ejecución del programa del que forma parte.

Ejemplo:

```
#include <stdio.h>

void mostrarContador() {
   static int contador = 0;  // Se inicializa solo la primera vez
   printf("Contador = %d\n", contador);
   contador++;              // Luego siempre se incrementa
}

int main() {
   mostrarContador();  // Primera llamada → muestra 0
   mostrarContador();  // Segunda llamada → muestra 1
   mostrarContador();  // Tercera llamada → muestra 2
```

```
    return 0;
}
```

La variable contador solo es visible desde la función mostrarContador(), sin embargo su tiempo de vida se extiende a todo el programa, es decir se eliminará el valor almacenado junto con la referencia a su dirección una vez q el programa finalizó.

-->Tabla de símbolos, definición y explicar cómo se implementa la tabla de símbolos y mostrar código en aleph

5.3. Análisis Semántico

5.3.1. Ligaduras

Definición de Ligadura(*binding*):

Una ligadura (*binding)* es una asociación entre un atributo y una entidad, como entre una variable y su tipo o valor, o entre una operación y un símbolo. El momento en que ocurre una vinculación se llama *tiempo de vinculación* (*binding time*).

Las vinculaciones y los tiempos de vinculación son conceptos importantes en la semántica de los lenguajes de programación. Una vinculación puede ocurrir en el momento del diseño del lenguaje, en la implementación del lenguaje, en tiempo de compilación, de carga, de enlace (*link time*) o durante la ejecución (*run time*).

Ejemplo: Considera la siguiente instrucción de asignación en Java:

count = count + 5;

Algunas de las vinculaciones y sus tiempos de vinculación para las partes de esta instrucción son las siguientes:

- El tipo de *count* se vincula en tiempo de compilación.
- El conjunto de valores posibles de *count* se vincula en el diseño del compilador.
- El significado del operador ***+*** se vincula en tiempo de compilación, cuando se han determinado los tipos de sus operandos.
- La representación interna del literal 5 se vincula en el diseño del compilador.
- El valor de *count* se vincula en tiempo de ejecución con esta instrucción.

Tiempos de ligadura (Binding Times):

El tiempo de ligadura es el momento en el ciclo de vida del programa en que se establece una ligadura.

Sebesta identifica varios momentos típicos:

1. Tiempo de diseño del lenguaje (language design time)

El diseñador del lenguaje decide reglas y significados fundamentales.

Qué se liga aquí:

- operadores ⟶ operaciones
- palabras clave ⟶ significados
- estructuras básicas del lenguaje

Ejemplos:

- En Java, + para enteros significa suma.
- En Python, for significa un loop iterativo sobre iterables.
- El símbolo = significa asignación.

2. Tiempo de implementación del lenguaje (language implementation time)

El creador del compilador o intérprete concreta detalles técnicos.

Qué se liga aquí:

- tamaños de tipos primitivos
- representación interna de datos
- límites de la pila o heap
- forma exacta del ciclo de evaluación

Ejemplos:

- En una implementación de C, int ocupa 32 bits.
- El compilador decide cómo representar un float

3. Tiempo de compilación (compile time)

El compilador asigna propiedades a identificadores.

Qué se liga aquí:

- tipo de las variables (en lenguajes con tipado estático)
- direcciones relativas
- tamaño de estructuras
- resolución de sobrecarga (overloading)

Ejemplos:

- En C++: `int x;` liga el nombre `x` al tipo `int`.
- En Java: el compilador decide qué versión del método sobrecargado usar.
- En C: el compilador determina cuántos bytes ocupará una struct.

4. Tiempo de carga (load time)

Ocurre cuando el programa se carga en memoria antes de ejecutarse.

Qué se liga aquí:

- direcciones absolutas para variables estáticas
- ubicaciones del código en memoria

Ejemplos:

- Colocar una variable global en una posición fija de memoria del proceso.
- Asignar direcciones finales al segmento de código.
- Reservar memoria para constantes estáticas.

5. Tiempo de enlace (link time)

El linker resuelve referencias entre módulos.

Qué se liga aquí:

- llamadas a funciones externas → sus definiciones reales
- referencias a librerías → código de librería
- direcciones finales en el ejecutable

Ejemplos:

- Resolver una llamada a printf desde el ejecutable hacia la librería estándar.
- Asociar un símbolo global a su dirección de memoria definitiva.

6. Tiempo de ejecución (run time)

Todo lo que se decide dinámicamente mientras el programa corre.

Qué se liga aquí:

- valores actuales de las variables
- creación y destrucción de objetos
- asignación de memoria dinámica
- enlaces dinámicos (métodos virtuales, dinámicamente ligados)

Ejemplos:

- Al ejecutar `x = 5;` la variable `x` queda ligada al valor 5.
- En Python, al crear `a = []` se liga un nombre al objeto lista recién creado.
- Una llamada virtual en Java decide en *runtime* qué método usar (polimorfismo dinámico). `malloc()` o `new` reservan memoria del heap durante la ejecución.

-->Defina ligadura estática y ligadura dinámica

Definición de ligadura estática:

Una vinculación es *estática* si ocurre antes de que comience la ejecución y permanece sin cambios durante toda la ejecución del programa.

Definición de ligadura dinámica:

Si la vinculación ocurre por primera vez durante la ejecución o puede cambiar durante ella, se denomina *dinámica*.

-->Defina variables estáticas, dinámicas de pila, explícitas de montón dinámico, implícitas de montón dinámico. Enumere las ventajas y desventajas de cada una.(ya lo hago by martin)

-->Relacione estos conceptos con la implementación de Aleph **(ayuda)**

5.3.2. Ámbito

-->Defina: tiempo de vida, alcance, alcance estático y alcance dinámico. (ya lo hago by martin)

-->Defina: ancestro estático y dinámico de un subprograma (ya lo hago by martin)

-->Relacione estos conceptos con la implementación de Aleph

5.3.3. Tipos de Datos

-->Defina: Chequeo de tipos. ¿Qué es un tipo compatible?

-->Defina: error de tipo.

-->Defina: fuertemente tipado y enumere las ventajas de los lenguajes de este tipo.

-->Defina: conversión implícita y conversión explícita de tipos.

-->Relacione estos conceptos con la implementación de Aleph

5.3.4. Semántica de las asignaciones y expresiones

-->Explicar y mostrar la implementación de las asignaciones y expresiones de Aleph

--> Evaluación de asignaciones

(segundo parcial 25/11 presentación informe defensa lunes 28/11 15hs - recu: sábado 03/12)

6. Fase 3 de implementación de Aleph (examen final)

6.1. Semántica de las estructuras de control

-->Explicar y mostrar la implementación de todas las estructuras de control Aleph, esto incluye las condiciones lógicas y los operadores relacionales.

6.2. Semántica de las subrutinas

-->Explicar y mostrar la implementación de las subrutinas de Aleph