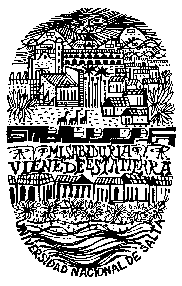
**Trabajo Práctico Transversal**

Teoría de la Computación II

2022



Licenciatura en Análisis de Sistemas

Universidad Nacional de Salta

Nombre: Mauricio Colque Medrano L.U: 225320

Nombre: Fernández Carlos Agustín DNI: 44017405

1. **Introducción**
   1. **Objetivo**

El objetivo de este trabajo practico transversal consiste en la realización de un lenguaje de programación orientado al uso de conjuntos, listas, elementos y números, con el propósito de crear programas que implementen dichos tipos de datos, entre ellos algunas operaciones con autómatas.

* 1. **Sobre el nombre del lenguaje**

El lenguaje a desarrollar se denominará **Aleph** puesto que este nombre tiene una fuerte vinculación con la Teoría de Conjuntos, estructura que será el pilar del desarrollo del nuevo lenguaje.

El termino alef es el signo gráfico, correspondiente a la letra hebrea א, empleado por Georg Cantor en la formulación de su teoría de conjuntos para representar la cardinalidad de los números infinitos, es decir, para ordenar los números transfinitos y así diferenciar los distintos tamaños de infinito.

* 1. **Sobre las herramientas a utilizar en el desarrollo**

Flex es una herramienta que se utiliza para generar analizadores léxicos o scanners. Su función principal es analizar un flujo de texto (como un programa fuente) y dividirlo en tokens, que son unidades léxicas significativas. Estos tokens pueden ser palabras clave, identificadores, números, operadores, símbolos, etc.

Bison es una herramienta que se utiliza para generar analizadores sintácticos o parsers. Su función principal es analizar la estructura gramatical de un lenguaje de programación o un formato de datos y construir un árbol de sintaxis abstracta (AST) que represente esa estructura.

1. **Análisis de requerimientos** 
   1. **Un caso de estudio: Los Autómatas Finitos y sus algoritmos**

Un autómata finito es una “máquina” la cual se encarga de aceptar o rechazar cadenas, tiene un conjunto de estados los cuales determinan su comportamiento. Se dividen en dos grupos:

Autómata finito determinista:

Un autómata finito determinista (afd) A es una 5-upla definida por los siguientes elementos:

* Q Es un conjunto finito no vacío de **estados**.
* Σ Es un conjunto finito no vacío de símbolos llamado **alfabeto de entrada**
* δ QxΣ→Q es una función completamente definida llamada **función de transición**
* q0 ∈ Q es el **estado inicial**
* F ⊆ Q es el conjunto de **estados de aceptación**

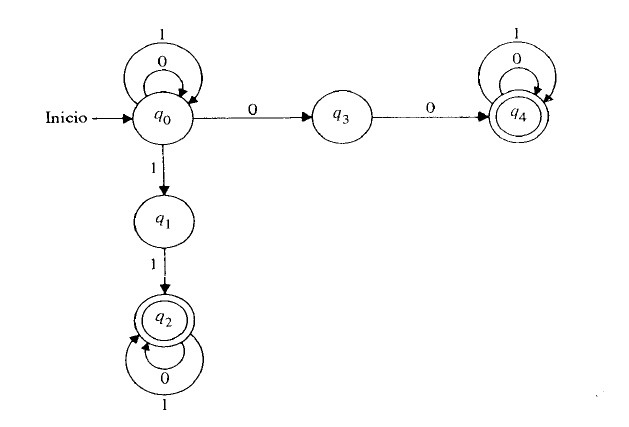
Autómata finito no determinista:

Un autómata finito no determinista (afnd) A es una 5-upla definida por los siguientes elementos:

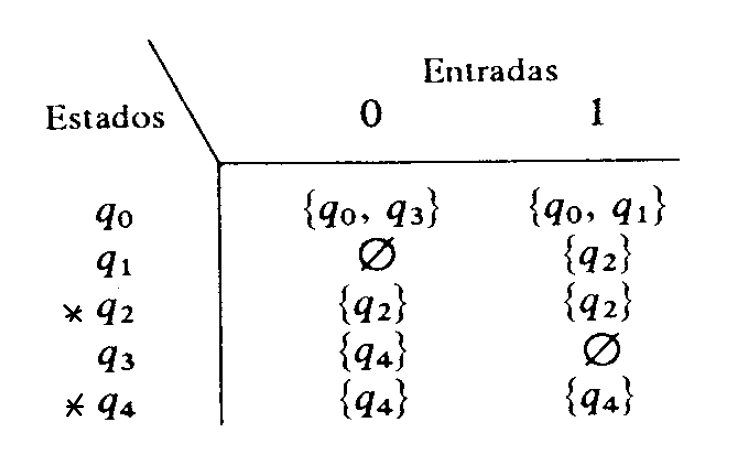
* Q Es un conjunto finito no vacío de **estados**.
* Σ Es un conjunto finito no vacío de símbolos llamado **alfabeto de entrada**
* δ QxΣxQ es una relación llamada **relación de transición**
* q0 ∈ Q es el **estado inicial**
* F ⊆ Q es el conjunto de **estados de aceptación**

Ejemplo de un autómata finito:

Representado en un diagrama de transiciones:



En una tabla:



Conjuntos y listas:

[**{**q0,q1,q2,q3,q4**}**,

**{**0,1**}**,

**{[**q0,0,**{q0,q3}]**,**[**q0,1,**{q0,q1}]**,**[**q1,0,**{}]**,**[**q1,1,**{q2}]**,**[**q2,0,**{q2}]**,**[**q2,1,**{q2}]**,**[**q3,0,**{q4}]**,**[**q3,1,**{}]**,**[**q4,0,**{q4}]**,**[**q4,1,**{q4}]}**,

**q0**,

**{**q2,q4**}**]

Los autómatas finitos deterministas y no deterministas aceptan lenguajes regulares, luego, dado un afnd podemos encontrar un afd equivalente, es decir, que acepta el mismo lenguaje, usando un algoritmo de conversión:

Pseudocódigo:

afnd2afd(A)=

sea A=( Q, Σ, δ, q0, F)

QB= {{q0}}

mientras ∃**R** ∈ QB tal que δB(**R,a**) este definido ∀ **a**∈ Σ

∀ **a**∈ Σ

δB(**R,a**) = **∪**δ(q,a) , q∈ **R**

QB= QB ∪ { δB(**R,a**) }

FB = {**S** ∈ QB | **S ∩** F ≠ Ø }

contestar (QB, Σ, δB, {q0},FB)

Luego, intentamos desarrollar este algoritmo en algún lenguaje de programación, en este caso C, el cual presentará algunas dificultades a la hora de intentar hacer uso de estructuras como Conjuntos y Listas, pues no fue desarrollado para ello:

Código en C:

tData ConversionAFD(tData AFND)

{

    tData sigmaND = returnElem(AFND, 2);

    tData transitionsND = returnElem(AFND, 3);

    tData initialStateND = returnElem(AFND, 4);

    tData acceptanceStatesND = returnElem(AFND, 5);

    tData AFD = newData("[]");

    tData initialStateD = newNestedData(initialStateND, SET);

    tData stateSetD = newNestedData(initialStateD, SET);

    tData sigmaD = copyData(sigmaND);

    tData transitionsD = newData("{}");

    tData acceptanceStatesD = newData("{}");

    tData aux1;

    tData aux2;

    tData partialState;

    tData newTransition;

    int positionInStateSetD = 1;

    while (positionInStateSetD <= CARDINAL(stateSetD))

    {

        for (int positionInSigmaND = 1; positionInSigmaND <= CARDINAL(sigmaND); positionInSigmaND++)

        {

            partialState = newData("{}");

            for (int positionInStateD = 1; positionInStateD <= CARDINAL(returnElem(stateSetD, positionInStateSetD)); positionInStateD++)

            {

                for (int positionInTransitionsND = 1; positionInTransitionsND <= CARDINAL(transitionsND); positionInTransitionsND++)

                {

                    if (isEqual(returnElem(returnElem(stateSetD, positionInStateSetD), positionInStateD), returnElem(returnElem(transitionsND, positionInTransitionsND), 1)))

                    {

                        if (isEqual(returnElem(sigmaND, positionInSigmaND), returnElem(returnElem(transitionsND, positionInTransitionsND), 2)))

                        {

                            aux1 = UNION(partialState, returnElem(returnElem(transitionsND, positionInTransitionsND), 3));

                            dataFree(&partialState);

                            partialState = aux1;

                            aux1 = NULL;

                            break;

                        }

                    }

                }

            }

            newTransition = newData("[]");

            PUSH(newTransition, copyData(returnElem(stateSetD, positionInStateSetD)));

            PUSH(newTransition, copyData(returnElem(sigmaND, positionInSigmaND)));

            PUSH(newTransition, copyData(partialState));

            aux1 = newNestedData(newTransition, SET);

            dataFree(&newTransition);

            newTransition = NULL;

            aux2 = UNION(transitionsD, aux1);

            dataFree(&transitionsD);

            dataFree(&aux1);

            transitionsD = aux2;

            aux2 = NULL;

            aux1 = NULL;

            if (!IN(stateSetD, partialState))

            {

                aux1 = newNestedData(partialState, SET);

                dataFree(&partialState);

                partialState = aux1;

                aux1 = UNION(stateSetD, partialState);

                dataFree(&stateSetD);

                stateSetD = aux1;

                aux1 = NULL;

            }

            partialState = NULL;

        }

        positionInStateSetD++;

    }

    for (int positionInStateSetD = 1; positionInStateSetD <= CARDINAL(stateSetD); positionInStateSetD++)

    {

        aux1 = INTER(returnElem(stateSetD, positionInStateSetD), acceptanceStatesND);

        if (!isEmpty(aux1))

        {

            dataFree(&aux1);

            aux1 = newNestedData(returnElem(stateSetD, positionInStateSetD), SET);

            aux2 = UNION(acceptanceStatesD, aux1);

            dataFree(&acceptanceStatesD);

            acceptanceStatesD = aux2;

            aux2 = NULL;

        }

        dataFree(&aux1);

    }

    PUSH(AFD, stateSetD);

    PUSH(AFD, sigmaD);

    PUSH(AFD, transitionsD);

    PUSH(AFD, initialStateD);

    PUSH(AFD, acceptanceStatesD);

    return AFD;

}

La principal dificultad que se encuentra a la hora de desarrollar el programa es debido a que el lenguaje no está pensado para este tipo de problemática.

Se complica mucho la escritura por la falta de estructuras de control que permitan trabajar fácilmente con conjuntos y listas, por ejemplo, una que permita hacer algo con cada elemento de un conjunto u otra que permita sobrescribir un conjunto con su union con otro sin tener que realizar manualmente la liberación de memoria.

Además, la lectura del código es muy complicada por el tema de los parámetros anidados cuando se quiere referenciar por ejemplo a un elemento dentro de un conjunto, dentro de una lista.

* 1. **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 es un sistema notacional para describir computaciones en una forma legible tanto para la maquina como para el ser humano.

Entendemos que una computación es aquel proceso que puede ser efectuado por una computadora, que abarca tanto la manipulación de datos como el procesamiento de texto, almacenamiento y recuperación de la información, entre otros.

Es importante que un lenguaje sea legible tanto para la máquina como para el ser humano, puesto que, para ser traducido a un lenguaje entendible por la computadora, éste debe tener una estructura que permita una traducción eficiente. Por nuestra parte, necesitamos que proporcione abstracciones de las acciones de las computadoras que sean fáciles de entender, por lo que en consecuencia el lenguaje debe parecerse a un lenguaje natural, para que la persona que lo use pueda describir de la mejor manera posible las instrucciones que le brinda a la computadora, basándose en su lenguaje natural.

* 1. **Abstracciones**

Las abstracciones de los lenguajes de programación se agrupan en dos clases generales: **abstracción de datos y abstracción de control,** y también, se agrupan en niveles, representando en cada uno la cantidad de información contenida de la abstracción.

Abstracciones de datos

Se dividen en tres grupos:

1. Abstracciones básicas: Representan valores de datos comunes en una computadora, conocidos como variables cuyo tipo de valor se conoce como tipo.
2. Abstracciones estructuradas: Se trata de colecciones de valores de datos relacionados entre sí, que pueden ser constituidos por diferentes tipos de datos.
3. Abstracciones unitarias: Representan componentes o contenedores, son estructuras de datos que a su vez contienen otros definidos por el usuario (bibliotecas).

Abstracciones de control

* 1. Abstracciones elementales:

Son aquellos enunciados o sentencias en un lenguaje que combinan unas cuantas instrucciones de máquina en una sentencia abstracta mas comprensible.

* 1. Abstracciones estructuradas: Dividen un programa en grupos de instrucciones que están anidadas dentro de pruebas que gobiernan su ejecución.
     1. **Abstracciones de Aleph**

Abstracciones de datos:

* Abstracciones básicas: Elementos, números
* Abstracciones estructuradas: Conjuntos, listas

Abstracciones de control:

* Abstracciones elementales: Asignación, unión, diferencia, intersección, cardinal, retorna elemento, pop, push, operaciones lógicas y aritméticas.
* Abstracciones estructuradas: Condicional (if), ciclo condicionado (while), para todo(foreach), asignación múltiple (let).
  1. **Dominio de programación de Aleph**

Aleph se ubica en los lenguajes con dominio científico, esto se debe a que permitirá usar estructuras matemáticas como conjuntos y listas con sus respectivas operaciones.

1. **Diseño** 
   1. **Sentencias**

Son cadenas de caracteres que representan las unidades ejecutable más pequeñas de un programa, especifican y controlan el flujo y orden de ejecución del programa.

* + 1. *Un primer diseño de sentencias de Aleph*

list afnd2afd(list A){

set Delta = A[3];

set F = A[5];

set Sigma = A[2]

set QB = {{A[4]}};

set DeltaB = {};

set FB = {};

foreach R in QB{

foreach a in Sigma{

list new\_trans = [R,a,{}];

set new\_q = {};

foreach q in R{

foreach T in Delta{

if ((q in T) && (a in T))

new\_q = new\_q | T[3];

}

}

new\_trans[3] = new\_q; //meto el nuevo estado en la transicion

DeltaB = DeltaB | {new\_q}; //meto la transicion en el DeltaB

QB = QB | {new\_q}; //añado el nuevo estado

}

}

foreach S in QB{

if((S & F) != {}) //Compara con el vacio

FB= FB | {S};

}

return [QB, Sigma, DeltaB, {A[4]}, FB];

}

* 1. **Reconocimiento de patrones. Tokens y Lexemas**

Un lexema es la unidad más pequeña y significativa en un lenguaje de programación. Es una secuencia de caracteres en el código fuente que representa un elemento individual y coherente del lenguaje, como palabras clave, identificadores, constantes, operadores, símbolos y comentarios. Los lexemas son identificados y reconocidos por el analizador léxico durante el proceso de análisis léxico y se utilizan como entrada para el análisis sintáctico y la generación de código en el proceso de compilación.

Un token es una unidad semántica en un lenguaje de programación, formada por una o más secuencias de caracteres (lexemas) que representan elementos significativos como palabras clave, identificadores o números. Los tokens son reconocidos durante el análisis léxico y se utilizan en el procesamiento de programas o texto.

* + 1. *Tokens y lexemas de Aleph*

|  |  |
| --- | --- |
| **Tokens** | **Lexemas** |
| { | **{** |
| } | **}** |
| [ | **[** |
| ] | **]** |
| + | **+** |
| - | **-** |
| / | **/** |
| \* | **\*** |
| = | **=** |
| , | **,** |
| ; | **;** |
| : | **:** |
| ( | **(** |
| ) | **)** |
| > | **>** |
| < | **<** |
| != | **!=** |
| == | **==** |
| >= | **>=** |
| <= | **<=** |
| | | **|** |
| & | **&** |
| &! | **&!** |
| && | **&&** |
| || | **||** |
| ! | **!** |
| ELEM | **'[a-zA-Z0-9]+** |
| IF | **if** |
| ELSE | **else** |
| WHILE | **while** |
| LET | **let** |
| FOREACH | **foreach** |
| POP | **pop** |
| PUSH | **push** |
| TO | **to** |
| IN | **in** |
| FUNC | **sqrt** |
| FUNC | **exp** |
| FUNC | **log** |
| FUNC | **print** |
| FUNC | **abs** |
| NAME | **[a-zA-Z][a-zA-Z0-9]\*** |
| NUMBER | **[0-9]+"."[0-9]\*{EXP}?** |
| NUMBER | **"."?[0-9]+{EXP}?** |

* 1. **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.

La sintaxis 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 gramáticas libres de contexto son un modelo matemático y una herramienta fundamental en la teoría de la computación. Estas gramáticas se utilizan para describir la estructura sintáctica de lenguajes formales, como lenguajes de programación o lenguajes de marcado. Se componen de reglas de producción que especifican cómo se pueden combinar símbolos no terminales (variables) y símbolos terminales (tokens) para formar cadenas válidas en el lenguaje. Las gramáticas libres de contexto son esenciales en la construcción de compiladores y analizadores sintácticos, ya que permiten analizar y entender la estructura jerárquica de un programa.

La sintaxis se refiere a las reglas y estructuras que gobiernan cómo se construyen expresiones, declaraciones y programas en un lenguaje de programación o en un lenguaje natural. Estas reglas definen cómo se deben colocar las palabras clave, operadores, identificadores y otros elementos léxicos para que un programa sea válido desde el punto de vista sintáctico. El análisis de sintaxis es un proceso crítico en la compilación y en la interpretación de código, ya que verifica si un programa cumple con las reglas sintácticas establecidas y, si es así, crea una representación estructurada, como un árbol de sintaxis, que facilita el procesamiento y la ejecución del programa. La sintaxis es un componente esencial en la comprensión y producción de código en cualquier lenguaje de programación.

La BNF (Forma Backus-Naur) es una notación formal utilizada para describir la sintaxis de lenguajes formales, como lenguajes de programación y lenguajes de marcado. Se compone de reglas de producción que especifican cómo se deben combinar los símbolos no terminales (variables) y los símbolos terminales (tokens) para formar estructuras válidas en el lenguaje. Las reglas se definen de manera recursiva, lo que permite describir la estructura jerárquica del lenguaje. BNF es ampliamente utilizado en la especificación de gramáticas formales y es una herramienta esencial en la construcción de compiladores y analizadores sintácticos.

La EBNF (Forma Extendida de Backus-Naur) es una extensión de la BNF que incluye una serie de convenios y notaciones adicionales para hacer las descripciones gramaticales más legibles y expresivas. EBNF permite el uso de símbolos como \* para indicar repeticiones, | para alternativas y () para agrupar elementos. Esto hace que las reglas gramaticales sean más concisas y más cercanas a cómo se expresan intuitivamente en el lenguaje natural.

* + 1. *Primera descripción formal de Aleph*

<if> -> “if” “(” <logicExpression> “)” “{” <statement> “}” [else “{” <statement> “}” ]

<foreach> -> “foreach” <identifier> “in” <set> “{” <statement> “}”

<while> -> “while” “(” <logicExpresion> “)” “{” <statement> “}”

<assignment> -> <identifier> = ( <set> | <list> | <elem> );

<declaration> -> ( set | list | elem ) <identifier>;

<assignDeclaration> -> ( set | list | elem ) <identifier> = ( <set> | <list> | <elem> );

<set> -> <identifier> | “{” <manyTypeElems> “}” | <union> | <inters> | <diff>

<list> -> <identifier> | “[” <manyTypeElems> “]” | <pop> |

<manyElems> -> ( <elem> | <set> | <list> ) [“, ” <manyTypeElems>]

<union> -> <set> “|” <set>

<inters> -> <set> “&” <set>

<diff> -> <set> “&!” <set>

* + 1. *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 de 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 Alpeh.

**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

<assign> -> ID **=** (<expr> | ID) **;**

<expr> -> <set> | <list> | ID | <elem>

Asignación Múltiple

<assign\_mult> -> ( <lista\_id> ) = ( <lista\_expr> )

<lista\_id> -> ID | ID , <lista\_id>

<lista\_expr> -> <expr> | <expr> , <lista\_expr>

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**

La precedencia de operadores determina el orden en el cual los operadores son evaluados uno respecto del otro. Por otro lado, la asociatividad es el orden de izquierda a derecha o derecha a izquierda para agrupar operandos en operadores que tienen la misma prioridad.

Estos conceptos se implementan en BNF mediante la estructura de reglas gramaticales y la posición relativa de los operadores en esas reglas

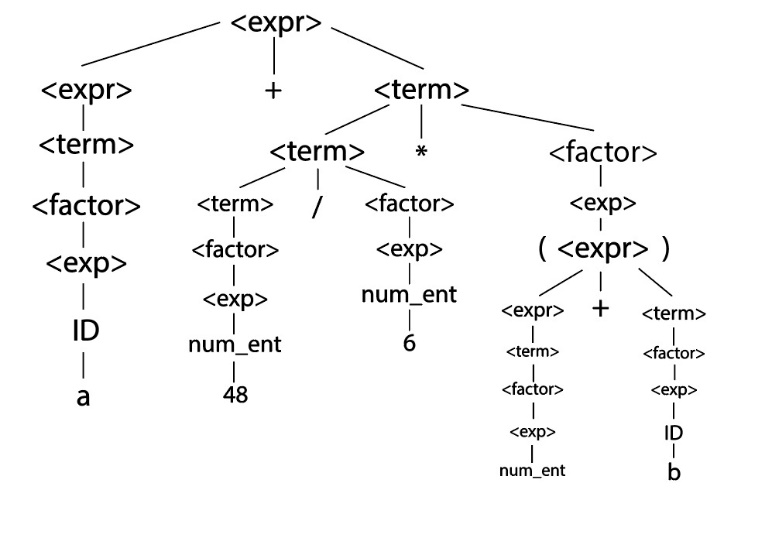
BNF de expresiones aritméticas por Sebesta:

<expr> -> <expr> + <term> | <expr> - <term> | <term>

<term> -> <term> \* <factor> | <term> / <factor> | <factor>

<factor> -> <exp> \*\* <factor> | <exp>

<exp> -> ( <expr> ) | id



Árbol de sintaxis abstracta:

Es una estructura de datos en forma de árbol, que permite “abstraer”, el árbol de sintaxis que el parser está utilizando, de tal manera que sea más simple y fácil de generar durante la etapa de análisis sintáctico. Este árbol se crea mientras se va leyendo la entrada, para una posterior ejecución luego de terminar la lectura de la entrada para la evaluación semántica de la entrada procesada.

Árbol de derivación:

Una de las características más atractivas de las gramáticas es que describen de manera natural la estructura sintáctica jerárquica de las oraciones de los idiomas que definen. Estas estructuras jerárquicas se llaman árboles de análisis sintáctico.

Operadores sobrecargados:

Los operadores aritméticos a menudo se utilizan para más de un propósito. Por ejemplo el símbolo “+” generalmente se utiliza para especificar la adición de enteros y la adición de números de punto flotante. Algunos lenguajes, como Java, también lo utilizan para la concatenación de cadenas. Este uso múltiple de un operador se llama sobrecarga de operadores y generalmente se considera aceptable, siempre y cuando ni la legibilidad ni la confiabilidad se vean afectadas.

**Expresiones: Operaciones con Conjuntos y Listas**

---> poner aquí la BNF de Operaciones con Conjuntos y Listas

<union> -> <set> **|** <set>

<inters> -> <set> **&** <set>

<diff> -> <set> **&!** <set>

<push> -> PUSH <expr> TO <list>

<pop> -> POP <list>

**Expresiones: Operaciones Relacionales**

<is\_equal> -> <expr> == <expr>

<not\_equal> -> <expr> != <expr>

**Expresiones: Operaciones Lógicas**

Evaluación perezosa: Significa que ninguna expresión se evalúa hasta que se requiere su valor. Esto conduce a algunas capacidades sorprendentes en el lenguaje.

Evaluación ingenua (estricta): Por el contrario, evalúa las expresiones tan pronto como se las encuentra.

<and> -> <logic\_expr> && <logic\_expr>

<or> -> <logic\_expr> || <logic\_expr>

<not> -> ! <logic\_expr>

<logic\_expr> -> <and> | <or> | <not> | <is\_equal> | <not\_equal> | (<logic\_expr>)

**Estructuras de Control**

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

BNF del if:

<if> -> if ( <logic\_expr> ) {<statement\_list> } | if (<expr>) { <statement\_list> } > else { <statement\_list> }

EBNF del if:

<if> -> if ( <expresion> ) <sentencias> [else <sentencias>]

<foreach> -> foreach ID in <list> { <statement\_list > } | foreach ID in <set> { <statement\_list> }

<while> -> WHILE ( <logic\_expr> ){ <statement\_list > }

**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.

--->poner la BNF de la subrutina

<subroutine> -> <data\_type> <name> ( <param\_list> ) {<sentences>}

<data\_type> -> <type> | void

<name> -> id

<param\_list> -> <param> | <param>, <param\_list>

<param> -> <type> id

**Gramática completa de Aleph**

<asign> -> id **=** (<expr> | id) **;**

<asign\_mult> -> ( <id\_list> ) = ( <expr\_list> )

< id\_list > -> id | id , < id\_list >

<expr\_list> -> <expr> | <expr> , <expr\_list>

<union> -> <set> **|** <set>

<inters> -> <set> & <set>

<diff> -> <set> **&!** <set>

<push> -> PUSH <expr> TO <list>

<pop> -> POP <list>

<set> -> {} | { <expr\_list> } | <union> | <inters> | <diff> |<pop>| ID

<list> -> [] | [<expr\_list>] | <pop> | ID

<elem> -> ELEM | <pop> | ID

<is\_equal> -> <expr> == <expr>

<not\_equal> -> <expr> != <expr>

<if> -> IF ( <expresion> ) <sentencias> | IF (<expresion>) <sentencias> ELSE <sentencias>

<foreach> -> FOREACH ID IN <list> { <statement\_list > } | FOREACH ID IN <set> { <statement\_list> }

<while> -> WHILE ( <logic\_expr> ){ <statement\_list > }

<subroutine> -> <data\_type> <name> ( <param\_list> ) {<sentences>}

<data\_type> -> <type> | VOID

<name> -> id

<param\_list> -> <param> | <param>, <param\_list>

<param> -> <type> id

<and> -> <logic\_expr> && <logic\_expr>

<or> -> <logic\_expr> || <logic\_expr>

<not> -> ! <logic\_expr>

<logic\_expr> -> <and> | <or> | <not> | <is\_equal> | <not\_equal> | (<logic\_expr>)

<statement\_list> -> <statement> ; <statement\_list > | <statement > ;

<statement> -> <if> | <while> | <foreach> | <def> | <assign> | <assign\_mult>

<subroutine> -> <data\_type> <name> ( <param\_list> ) {<sentences>}

<data\_type> -> <type> | void

<name> -> id

<param\_list> -> <param> | <param>, <param\_list>

<param> -> <type> id

* 1. **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.

**Semántica**: Se refiere al significado de las expresiones, sentencias y las unidades de un programa.

* + 1. *Semántica Estática*

La semántica estática tiene que ver con las formas legales de un programa. Muchas de estas reglas declaran las restricciones de tipos. La semántica estática es muy nombrada debido a que el análisis requerido para validar estas especificaciones puede ser realizado en tiempo de compilación.

**Gramática de Atributos**

Las gramáticas de atributos son gramáticas libres de contexto a las que se les han añadido atributos, funciones de cálculo de atributos y funciones de predicado. Los atributos, que están asociados con los símbolos de la gramática (los símbolos terminales y no terminales), son similares a variables en el sentido de que se les pueden asignar valores.

Las **funciones de cálculo de atributos**, a veces llamadas funciones semánticas, están asociadas con las reglas de la gramática. Se utilizan para especificar cómo se calculan los valores de los atributos. Las **funciones de predicado**, que establecen las reglas semánticas estáticas del lenguaje, están asociadas con las reglas de la gramática.

Asociado a cada símbolo de la gramática X hay un conjunto de atributos A(X). El conjunto A(X) se compone de dos conjuntos disjuntos, S(X) e I(X), llamados atributos sintetizados y atributos heredados, respectivamente. Los **atributos sintetizados** se utilizan para transmitir información semántica hacia arriba en un árbol de análisis, mientras que los **atributos heredados** transmiten información semántica hacia abajo y a través de un árbol.

Los atributos **intrínsecos** son atributos sintetizados de nodos hoja cuyos valores se determinan fuera del árbol de derivación.

Por otro lado, Asociado con cada regla de gramática hay un conjunto de funciones semánticas y posiblemente un conjunto vacío de funciones de predicado sobre los atributos de los símbolos en la regla de gramática. El valor de un atributo sintetizado en un nodo del árbol de derivación depende únicamente de los valores de los atributos en los nodos hijos de ese nodo, mientras que el valor de un atributo heredado en un nodo del árbol de derivación depende de los valores de los atributos en el nodo padre y los de sus nodos hermanos.

Un árbol de derivación de una gramática de atributos es el árbol de derivación basado en su gramática subyacente BNF, con un conjunto posiblemente vacío de valores de atributos adjuntos a cada nodo.

Una función predicado tiene la forma de una expresión booleana sobre la unión del conjunto de atributos {A(X0), **…** , A(X­­n)} y un conjunto de valores de atributos literales. Las únicas derivaciones permitidas con una gramática de atributos son aquellas en las que cada predicado asociado con cada no terminal es verdadero. Un valor de función predicado falso indica una violación de las reglas de sintaxis o semántica estática del lenguaje. Dado los valores de atributos intrínsecos en un árbol de derivación, las funciones semánticas pueden utilizarse para calcular los valores restantes de atributos.

Gramática de atributos para las expresiones de Aleph

Atributos sintetizados: actual\_type, node\_type

Union:

Syntax rule: <exp>[0] -> <exp>[1] | <exp>[2]

Semantic rule: <exp>[0].actual\_type <-

If (<exp>[1].node\_type = SET) and (<exp>[2]. node\_type = SET)

then SET

else INVALID\_OP

end if

Predicate: <exp>[0].actual\_type == SET

Intersection:

Syntax rule: <exp>[0] -> <exp>[1] & <exp>[2]

Semantic rule: <exp>[0].actual\_type <-

If (<exp>[1]. node\_type = SET) and (<exp>[2]. node\_type = SET)

then SET

else INVALID\_OP

end if

Predicate: <exp>[0].actual\_type == SET

Difference:

Syntax rule: <exp>[0] -> <exp>[1] &! <exp>[2]

Semantic rule: <exp>[0].actual\_type <-

If (<exp>[1]. node\_type = SET) and (<exp>[2]. node\_type = SET)

then SET

else INVALID\_OP

end if

Predicate: <exp>[0].actual\_type == SET

NAME:

Syntax rule: <exp> -> NAME

Semantic rule: <exp>.actual\_type <- lookup(NAME.string)

Literal Set:

Syntax rule: <exp> -> {} | {<explist>}

Semantic rule: <exp>.actual\_type <- SET

Literal List:

Syntax rule: <exp> -> [] | [<explist>]

Semantic rule: <exp>.actual\_type <- LIST

Elem:

Syntax rule: <exp> -> ELEM

Semantic rule: <exp>.actual\_type <- ELEM\_TYPE

Assign:

Syntax rule: <exp>[0] -> NAME = <exp>[1]

Semantic rule: <exp>[0].actual\_type <- <exp>[1].actual\_type

Positioned element:

Syntax rule: <exp>[0] -> <exp>[1] “[” <exp>[2] “]”

Predicate: (<exp>[1].actual\_type == SET || <exp>[1].actual\_type == LIST) && (<exp>[2].actual\_type == NUMBER)

Pop:

Syntax rule: <exp>[0] -> POP <exp>[1]

Predicate: <exp>[1].actual\_type == LIST

Is Contained:

Syntax rule: <exp>[0] -> <exp>[1] : <exp>[2]

Semantic rule: <exp>[0].actual\_type <- NUMBER

Predicate: (<exp>[1].actual\_type == SET && <exp>[2].actual\_type == SET)

* + 1. *Semántica Dinámica*

La semántica dinámica, se refiere a la interpretación del significado de un programa o sus componentes durante el tiempo de ejecución. Se centra en cómo se comporta un programa a medida que se ejecuta en una maquina o un entorno informático. Debido a la potencia y naturalidad de la notación disponible, describir la sintaxis es un asunto relativamente simple. Por otro lado, no se ha ideado una notación o enfoque universalmente aceptado para la semántica dinámica.

Existen 3 enfoques para los lenguajes imperativos:

* Semántica operacional
* Semántica denotacional
* Semántica axiomática

**Semántica Operacional**

La idea detrás de la semántica operacional es la de describir el significado de una sentencia o un programa mediante la descripción de los cambios que generan estos en los estados de una computadora al ejecutarlos.

Descripción las estructuras de control de Aleph:

El primer paso para crear una descripción de la semántica operativa de un lenguaje es diseñar un lenguaje intermedio adecuado, donde la característica principal del lenguaje sea la claridad. Cada construcción del lenguaje intermedio debe tener un significado obvio y sin ambigüedades. Este lenguaje se encuentra en un nivel intermedio, ya que el lenguaje de máquina es demasiado bajo nivel para ser comprendido fácilmente y otro lenguaje de alto nivel obviamente no es adecuado.

Como ejemplo, usaremos la siguiente lista de sentencias, para describir la semántica de las estructuras de control de aleph:

ident = var ;

ident = ident + 1 ;

ident = ident – 1 ;

goto label ;

if var relop var ? goto label ;

ident = set[number] ;

ident = list[number] ;

ident = size\_of(set) ;

ident = size\_of(list);

IF en Aleph:

If (exp) {statement\_TRUE} else {statement\_FALSE}

IF en semántica operacional:

If exp ? goto **true\_if**;

Statement\_false\_1;

Statement\_false\_2;

…

goto **END**;

**true\_if:**

Statement\_true\_1;

Statement\_true\_2;

…

goto **END**;

While en aleph:

while (exp) {stm\_list}

While en semántica operacional:

**while:**

if exp ? goto **true\_while**

goto **END**;

**true\_while**:

stm1;

stm2;

...

goto **while**;

Foreach en aleph:

foreach ident in exp {stm\_list}

Foreach en semántica operacional:

count = 0;

**foreach**:

if count == size\_of(exp) ? goto **END**;

ident = exp[count]

smt1;

smt2;

...

count = count + 1;

goto **foreach**;

**Semántica Denotacional**

La semántica denotacional es el método formal más riguroso y ampliamente conocido para describir el significado de los programas. Se basa en la teoría de dominios matemáticos y utiliza funciones matemáticas para asignar significado a los programas y a sus fragmentos, estas funciones se utilizan para mapear programas a elementos de un dominio matemático, que representan el significado de los programas.

Las funciones de asignación de una especificación de lenguaje de programación mediante semántica denotacional, al igual que todas las funciones en matemáticas, tienen un dominio y un codominio. El dominio es la colección de valores que son parámetros legítimos para la función; el codominio es la colección de objetos a los cuales se asignan los parámetros. En la semántica denotacional, el dominio se llama dominio sintáctico, porque son las estructuras sintácticas las que se asignan. El codominio se llama dominio semántico.

La semántica denotacional de un programa podría definirse en términos de cambios de estado en una computadora ideal. El estado "s" de un programa se puede representar como un conjunto de pares ordenados, de la siguiente manera:

s = {<i1, v1>, <i2, v2>, . . . , <in, vn>}

Cada "i" es el nombre de una variable, y los "v" asociados son los valores actuales de esas variables. Cualquiera de los "v" puede tener el valor especial "undef", lo que indica que su variable asociada actualmente no está definida.

Descripción las estructuras de control de Aleph

Para cada estructura, asumimos que ya existen las siguientes funciones:

Mb(B, s) donde B es una expresión booleana que evalúa la función y devuelve **true** o **false** según corresponda. También puede devolver **undef** si la expresión no está definida.

Msl(L, s) donde L es una lista de sentencias las cuales ejecuta la función, y devuelve un nuevo estado de la máquina.

Md(EXP, s) donde EXP es una expresión, y la función devuelve **true** si EXP es una lista o conjunto, y **false** si EXP no es lista o conjunto o no está definida.

Msizeof(EXP, s) donde EXP es una lista o conjunto, y la función devuelve el tamaño de EXP.

Mpos\_elem(EXP, i, s) donde EXP es una lista o conjunto, i es el índice y la función devuelve el elemento que está en la posición i de EXP.

Mfeach\_sl(L, EXP, s) donde L es una lista de sentencias las cuales ejecuta la función y EXP es una expresión que pueden usar las sentencias.

If en semántica denotacional:

Mif(if EXP then LT else LF ,s) Δ = if Mb(EXP, s)== **undef**

then **error**

else if Mb(EXP, s) == true

then Msl(LT, s)

else Msl(LF, s)

While en semántica denotacional:

Mw(while EXP do L, s) Δ= if Mb(EXP, s) == **undef**

then **error**

else if Mb(EXP, s) == false

then s

else if Msl(L, s) == **error**

then **error**

else Mw(while EXP do L, Msl(L, s))

Foreach en semántica denotacional:

Mfeach(foreach x in EXP do L, s) Δ = if Md(EXP, s) == **false**

then **error**

else Mfeach\_rec(foreach x in EXP do L, 0, s)

Mfeach\_rec(foreach x in EXP do L, i, s) Δ= if Msizeof(EXP, s) > i

then Mfeach\_rec(foreach x in EXP do L, i+1, Mfeach\_sl(L, Mpos\_elem(EXP, i, s), s))

else s

1. **Fase 1 de implementación de Aleph**
   1. **Tipos de implementación**

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

--->Enfoque de implementación de Aleph

* 1. **Análisis Léxico**

--->Def de Análisis Léxico

--->Descripción de las tareas de un analizador léxico

-->Avance de código: Primer analizador de Aleph (primer scanner que imprime por pantalla los tokens)

* 1. **Análisis Sintáctico**

--->Def de Análisis Sintáctico

--->Metas del Análisis Sintáctico

--->Analizadores Top-Down

--->Gramáticas LL

--->Analizadores Descendentes Recursivos

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

--->Analizadores Bottom-Up

--->Proceso

--->Def Frase, Frase Simple, Manejador

--->Ventajas

--->Funcionamiento de los analizadores LR

--->Funcionamiento del analizador generado por Bison

--> Primer Analizador Sintáctico de Aleph (un archivo .l y un archivo .y relacionados, lo entregado para el 1P)

1. **Fase 2 de implementación de Aleph**
   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)

* 1. **Variables y la Tabla de Símbolos**

-->Describa las componentes de la seis-upla que caracteriza a una variable.

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

* 1. **Análisis Semántico**
     1. *Ligaduras*

-->Defina ligadura y tiempos de ligadura.

-->Defina ligadura estática y ligadura dinámica

-->Defina variables estáticas, dinámicas de pila, explicitas de montón dinámico, implícitas de montón dinámico. Enumere las ventajas y desventajas de cada una.

-->Relacione estos conceptos con la implementación de Aleph

* + 1. *Ámbito*

-->Defina: tiempo de vida, alcance, alcance estático y alcance dinámico.

-->Defina: ancestro estático y dinámico de un subprograma

-->Relacione estos conceptos con la implementación de Aleph

* + 1. *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

* + 1. *Semántica de las asignaciones y expresiones*

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

--> Evaluación de asignaciones

1. **Fase 3 de implementación de Aleph (examen final)**
   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.

* 1. **Semántica de las subrutinas**

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