# Trabajo práctico especial N°2

# ‘TreeLang’

**Año:2022**


**Integrantes:**

Santiago Sandrini (61447)

Marcos Dedeu (60469)

Federico Rojas (60239)

Roberto Rodriguez (60089)

# Índice

# 1. Introducción

El objetivo de este trabajo fue diseñar un lenguaje de programación junto a su respectivo compilador, a partir de los conocimientos adquiridos tanto en la materia como en la lectura de la bibliografía sugerida.

Para lograr el objetivo se definió una gramática que posteriormente fue analizada y se utilizaron los analizadores Lex (léxico) y Bison (sintáctico). Por último se utilizó un compilador (main.c) para generar una representación intermedia como recomendó la cátedra, en este caso, Java bytecode, ya que éste es multi-plataforma. El nombre elegido para nuestro proyecto es TreeLang.

# 2. Idea General del Lenguaje

## 2.1. Resumen

*TreeLang* es un lenguaje diseñado para trabajar con árboles, la idea general es facilitarle al usuario el uso de los mismos sin tener que implementarlos. Se busca que el usuario pueda crear distintos tipos de árboles y realizar algunas funciones con ellos como pueden ser graficarlos, agregar nodos, quitar nodos, obtener la cantidad de nodos, filtrar nodos, multiplicar cada nodo por algún valor, etc.

## 2.2. Prestaciones

1. Se podrán crear árboles red-black trees y árboles AVL.
2. Se podrán insertar nodos en los árboles.
3. Se podrán eliminar nodos.
4. Las variables podrán ser del tipo Árbol, ints o strings.
5. Las variables podrán ser vectores de alguno de los tipos anteriores (excepto árboles).
6. Se proveerán operadores relacionales como $<$, $>$, $=$, $\neq$, $\leq$ y $\geq$.
7. Se proveerán operaciones aritméticas básicas como +, -, * y /.
8. Se proveerán operaciones lógicas básicas como AND, OR y NOT.
9. Se proveerán estructuras de control básicas de tipo IF-THEN-ELSE, FOR y WHILE.
10. Se podrán crear árboles a partir de otros árboles.
11. Se podrán graficar los árboles.

### 2.3. Consideraciones realizadas

Para el desarrollo del trabajo se decidió que los árboles manejan sólo dos tipos de datos esenciales: *int* y *string;* ya que era necesario que el tipo de dato sea comparable para poder implementar el balanceo de los árboles. También se decidió que los tipos de árboles serán árboles de búsqueda binaria, árboles rojo-negro y árboles AVL, quitando el árbol genérico que se planteó en un principio ya que complicaría mucho la sintaxis del lenguaje para lograr crear un árbol de cualquier tipo, y además, no es el objetivo del lenguaje manejar árboles genéricos. Hay que tener en cuenta que el scope de todas las variables es global y para realizar la tabla de símbolos utilizamos una librería de hashMap, kHash, que nos permitió un acceso rápido y sin complicaciones de implementación.

# 3. Descripción de la gramática/sintaxis diseñadas

## 3.1. Producciones de la gramática

- mainProgram→ MAIN OPEN_CURL_BRACKETS block CLOSE_CURL_BRACKETS
- block→ instruction block | instruction
- instruction→ statement semiColons | if | for | while
- statement→ declareAndAssign | assignation | function
- declareAndAssign→ declare ASSIGN expression | declare ASSIGN OPEN_CURL_BRACKETS parameterList CLOSE_CURL_BRACKETS | declare
- declare→ type symbol | treeType type symbol | type symbol OPEN_SQUARE_BRACKETS CLOSE_SQUARE_BRACKETS
- assignation→ symbol ASSIGN expression
- function→ symbol POINT noParamFunctions OPEN_PARENTHESIS CLOSE_PARENTHESIS | symbol POINT oneParamFunctions OPEN_PARENTHESIS expression CLOSE_PARENTHESIS | read | write
- read→ READ OPEN_PARENTHESIS symbol CLOSE_PARENTHESIS
- write→ WRITE OPEN_PARENTHESIS expression CLOSE_PARENTHESIS
- noParamFunctions→ PRINT | LENGTH | BALANCED | SIZE
- oneParamFunctions→ DELETE_NODE | NEW_NODE | TREE_MUL | FILTER
- if→ IF OPEN_PARENTHESIS expression CLOSE_PARENTHESIS OPEN_CURL_BRACKETS block if_close
- if_close→ CLOSE_CURL_BRACKETS | CLOSE_CURL_BRACKETS ELSE OPEN_CURL_BRACKETS block CLOSE_CURL_BRACKETS
- while→ WHILE OPEN_PARENTHESIS expression CLOSE_PARENTHESIS OPEN_CURL_BRACKETS block CLOSE_CURL_BRACKETS

- for→ FOR OPEN_PARENTHESIS declareAndAssign semiColons expression semiColons statement CLOSE_PARENTHESIS OPEN_CURL_BRACKETS block CLOSE_CURL_BRACKETS | FOR OPEN_PARENTHESIS assignation semiColons expression semiColons statement CLOSE_PARENTHESIS OPEN_CURL_BRACKETS block CLOSE_CURL_BRACKETS | FOR OPEN_PARENTHESIS semiColons expression semiColons statement CLOSE_PARENTHESIS OPEN_CURL_BRACKETS block CLOSE_CURL_BRACKETS
- expression→ expression ADD expression | expression SUB expression | expression MUL expression | expression DIV expression | expression GT expression | expression GE expression | expression LE expression | expression LT expression | expression NE expression | expression EQ expression | NOT expression | factor | function| vector
- factor→ OPEN_PARENTHESIS expression CLOSE_PARENTHESIS| constant | string | symbol
- constant→ INTEGER
- vector→ symbol OPEN_SQUARE_BRACKETS factor CLOSE_SQUARE_BRACKETS
- parameterList→ expression | parameterList COMMA expression
- type→ INT_TYPE | STRING_TYPE
- treeType→ BST_TREE_TYPE | AVL_TREE_TYPE | RED_BLACK_TREE_TYPE
- semiColons→ SEMI_COLON | semiColons SEMI_COLON
- symbol→ SYMBOL
- string→ STRING

## 3.2. Terminales

- **ADD → "-"**
- **SUB → "+"**
- **MUL → "*"**
- **DIV → "/"**
- **EQ → "=="**
- **LE → "<="**
- **GE → ">="**
- **NE → "!="**
- **LT → "<"**
- **GT → ">"**
- **OR →"||"**
- **AND →"&&"** .
- **NOT→ "!"**
- **ASSIGN → "="**
- **POINT → "."**
- **COMMA → ","**
- **INT_TYPE** → “int”
- **STRING_TYPE** →“string”

- **AVL_TREE_TYPE** → “avlTree”
- **AVL_TREE_TYPE** → “rbtree”
- **BST_TREE_TYPE** → “bstTree”
- **MAIN** → ”main”
- **PRINT** → “print”
- **READ** → “read”
- **WRITE** → “write”
- **NEW_NODE** → “createNode”
- **BALANCED** → **balanced**
- **SIZE** → **size**
- **LENGTH** → **length**
- **DELETE_NODE** → **deleteNode**
- **FILTER** → **filter**
- **TREE_MUL** → **mul**
- **FOR** → **"for"**
- **WHILE** → **"while"**.
- **IF**→ **"if"**
- **ELSE** → **"else"**
- **OPEN_PARENTHESIS** → **"("**
- **CLOSE_PARENTHESIS** → **")"**
- **OPEN_CURL_BRACKETS** → **"{"**
- **CLOSE_CURL_BRACKETS** → **"}"**
- **OPEN_SQUARE_BRACKETS** → **"["**
- **CLOSE_SQUARE_BRACKETS** → **"]"**
- **QUOTE** → **"'"**
- **SEMI_COLON** → **";"**

## 3.3. No terminales

- **mainProgram:** Programa principal
- **block**: Bloque de instrucciones.
- **instruction**: Instrucción propiamente dicha, inlcuyendo el fin de línea.
- **statement:** asignación, declaración, función o cualquier otra cosa sin terminación de línea.
- **declareAndAssign:** declaración y asignación en la misma línea.
- **declare:** Únicamente declaración.
- **assignation:** Únicamente asignación.
- **function:** Funciones con 2,1 o 0 parámetros.
- **read**: función que lee de entrada estándar.
- **write:** Función que escribe a salida estándar
- **noParamFunctions:** funciones como print, length, size o balanced.

- **oneParamFunctions**: funciones como deleteNode, createNode, mul o filter
- **if:** Instrucción if
- **if_close:** Instrucción else o cierre del if
- **while:** Instrucción while
- **for:** Instrucción for
- **expression:** Combinación con o sin operadores de factores, funciones y vectores
- **factor:** constantes, strings y nombres de variables
- **constant:** número entero
- **vector:** arreglo de constantes, string o variables.
- **parameterList:** Lista de parámetros
- **type:** tipo de dato. No incluye tipo de árbol pero si el tipo de sus nodos asociados
- **treeType:** tipo de árbol
- **semiColons:** fin de línea
- **symbol:** nombres de variables
- **string:** elementos de tipo string

# 4. Descripción del desarrollo

## 4.1. Flex y Bison para la construcción de un AST

La función de Flex y Bison además de encontrar errores en la sintaxis del programa, es la de transformar el archivo en texto plano a una estructura que sea más manejable para la compilación del código.

Para realizar el lenguaje atravesamos una serie de pasos comunes en la construcción de compiladores:

La primera de ellas es la parte **léxica**. Para ella definimos los tokens en flex.

Luego con esos tokens definidos , utilizamos bison para correr el análisis **sintáctico**.

Con esta misma herramienta , y con la ayuda externa de una librería para construir árboles AST en C generamos el árbol con el programa parseado y todos sus tipos de nodo, para realizar el análisis **semántico**. El AST tiene un nodo raíz que inicialmente tiene como hijos las instrucciones del programa en su capa más exterior. Cada instrucción es de un tipo particular y por ende se puede llegar a ramificar de forma diferente. Este genera una estructura jerárquica (y no text/plain como un archivo fuente) que facilita mucho el análisis de alto nivel del input.

Por último, generamos **código intermedio** en Java para resolver los desafíos que planteaba el backend del lenguaje. Esos archivos intermedios los compilamos y devolvemos el bytecode de java listo para ser ejecutado en cualquier plataforma.

### 4.2. Validación de las variables

Para la validación de las variables se optó por realizarla durante la generación del AST. Ya que la única manera de utilizar una variable es luego de haberla declarado, cuando ocurre la producción ***declare*** se agrega el nombre de la variable junto con el tipo de dato a la tabla de símbolos. Posteriormente en cada bison-action que se ejecuta después de las producciones se chequea que **la variable exista** en primer lugar (consultando la tabla de símbolos) y además se hace un chequeo del tipo de dato. Algunos de estos chequeos realizados son:

- Que no se asigne var1 = var2 si los tipos de datos son diferentes.
- En el caso de hacer arbol1 = arbol2 se valida que el tipo de dato de los nodos del arbol1 coincida con el tipo de dato de los nodos del arbol dos, y que además arbol1 y arbol2 sean el mismo tipo de arbol.
- Que no se intente hacer read(var) o write(var) cuando var es del tipo arbol.
- Que las expresiones que estén dentro de una estructura if o if-else sean expresiones evaluables, es decir, <,>,==, !=, etc.
- Que no se repita la declaración de una variable.
- Que no se asigne var1 = “a” o var1 = 1 cuando var1 fue inicializado como vector.

Varios de estos chequeos se pueden observar funcionando en los casos de prueba que el compilador rechaza. También se agregaron casos adicionales.

# 5. Especificaciones funcionales del lenguaje

### 5.1. Híbrido entre C y Java

Nuestro lenguaje tiene varias similitudes a C en lo que concierne a la estructura general del programa, comenzando con main, todo un programa con instrucciones ejecutadas linealmente con el tiempo, pero a su vez , al agregar el manejo de árboles, hacemos un híbirido con lenguajes orientados a objetos como lo es Java, utilizando por ejemplo,

"*arbol.print()*" o bien "*bstTree int arbol2;  bstTree int arbol = arbol2.filter(2);*", una sintaxis orientada a objetos.

### 5.2. Creación de árboles

La creación de un árbol es bien sencilla. Primero debemos declarar el tipo de árbol que queremos, seguido del tipo de dato del nodo que él contiene. Luego en otra línea, agregamos los nodos adecuados.

Los ejemplos pueden ser revisados en https://github.com/mdedeu/TP-TLA/tree/main/programs

### 5.3. Manipulación de árboles

Se pueden utilizar las funciones built-in de los árboles para realizar operaciones sobre los mismos. Podemos por ejemplo, obtener un nuevo árbol en base a un anterior, con todos sus elementos multiplicados por un factor, podemos obtener un nuevo árbol a partir de un anterior, filtrando por un valor específico.

Los ejemplos pueden ser revisados en https://github.com/mdedeu/TP-TLA/tree/main/programs

## 6. Dificultades encontradas

La primera dificultad que tuvimos fue adaptarnos a las herramientas sin entender por completo los conceptos de lenguaje, producciones. Este approach top-down igualmente estuvo bueno porque nos facilitó entender los conceptos de la teoría cuando fueron dados.

La segunda y siempre presente dificultad fue la del tiempo. Consideramos que nuestras primeras dos entregas fueron breves comparadas con la tercera, se podrían hasta juntar las primeras dos para concentrar esfuerzos. Nuestra percepción de dificultad en las diferentes etapas también se vio alterada por la complejidad del backend que teníamos que realizar, frente a frontend más simple y rápido.

Otra dificultad que tuvimos fue la de utilizar el yylval como estructura para el pasaje de datos, ya que estuvimos frenados dos o tres días por la falta de declaración o por las restricciones que el mismo boilerplate de la cátedra proponía. Por suerte esta dificultad fue sorteada con una clase de consulta que nos ayudó a destrabar el tema.

# 7. Posibles extensiones

Dentro de la inmensa posibilidad de funciones que se le pueden agregar al lenguaje mismo, podemos destacar algunas en particular.

- **Declaración de funciones:** Permite realizar código modularizado y mucho más legible. Esto es importante cuando estamos armando grandes árboles con muchas variaciones, diferentes funciones y objetivos sobre el tratamiento de los mismos.
- **Agregar más tipos de árbol:** Si bien en un momento lo planteamos, la dificultad y cantidad de combinaciones y situaciones diferentes nos hicieron descartar esta posibilidad.
- **Agregar "merging" de árboles:** Otra funcionalidad interesante podría haber sido combinar y "appendear" o mergear árboles con otros. Podría ser del mismo tipo o de tipos distintos, y hacer algún tipo de conversión. Esto haría el lenguaje muy potente pero tiene una complejidad acarreada importante.
- **Agregar búsqueda de nodos:** Una funcionalidad muy usada cuando estamos creando y manipulando árboles es la búsqueda de valores sobre un tipo de árbol específico. Por ejemplo, es muy común que el armado de un árbol binario de búsqueda sea para realizar búsqueda más eficientemente.

# 8. Conclusión

Para resumir, nos encontramos conformes con el lenguaje creado, si bien es cierto que no llegamos a cumplir con todo el scope original, creemos que pese a la dificultad del lenguaje elegido y a las adversidades que se presentaron pudimos crear un lenguaje bastante completo.

También sostenemos la postura de que este lenguaje en vista a futuro puede ser de gran utilidad y puede expandirse para volverse una herramienta realmente potente.

# 9. Referencias


9.1. Symbol table. URL https://en.wikipedia.org/wiki/Symbol_table.

9.2. Thomas Niemman. A guide to lex & yacc. URL https://arcb.csc.ncsu.edu/~mueller/codeopt/codeopt00/y_man.pdf.

9.3. La base del TP está tomada del repositorio boilerplate presentado por la cátedra: https://github.com/agustin-golmar/Flex-Bison-Compiler

9.4. Librería de hashmap https://github.com/attractivechaos/klib