Trabajo Práctico Especial

Alvarez Escalante, Tomás (60127)
tomalvarez@itba.edu.ar

Caeiro, Alejo (60692)
acaeiro@itba.edu.ar

Ferreiro, Lucas Agustín (61595)
lferreiro@itba.edu.ar

Gomez Kiss, Roman (61003)
romgomez@itba.edu.ar

Nimloth

The white tree of Numenor

72.39 Autómatas, Teoría de Lenguajes y Compiladores

Instituto Tecnológico de Buenos Aires

Tabla de contenidos

Introducción

El objetivo de este trabajo fue diseñar un lenguaje de programación junto a su respectivo compilador a partir de los conocimientos adquiridos a lo largo de la materia. La idea general de nuestro lenguaje es facilitarle al usuario la creación de árboles BST (Binary Search Tree), AVL (BST balanceados) y RBT (Red Black Tree), y su visualización. Para esto se hizo uso del lenguaje DOT proporcionado por Graphviz.

Desarrollo de Nimloth

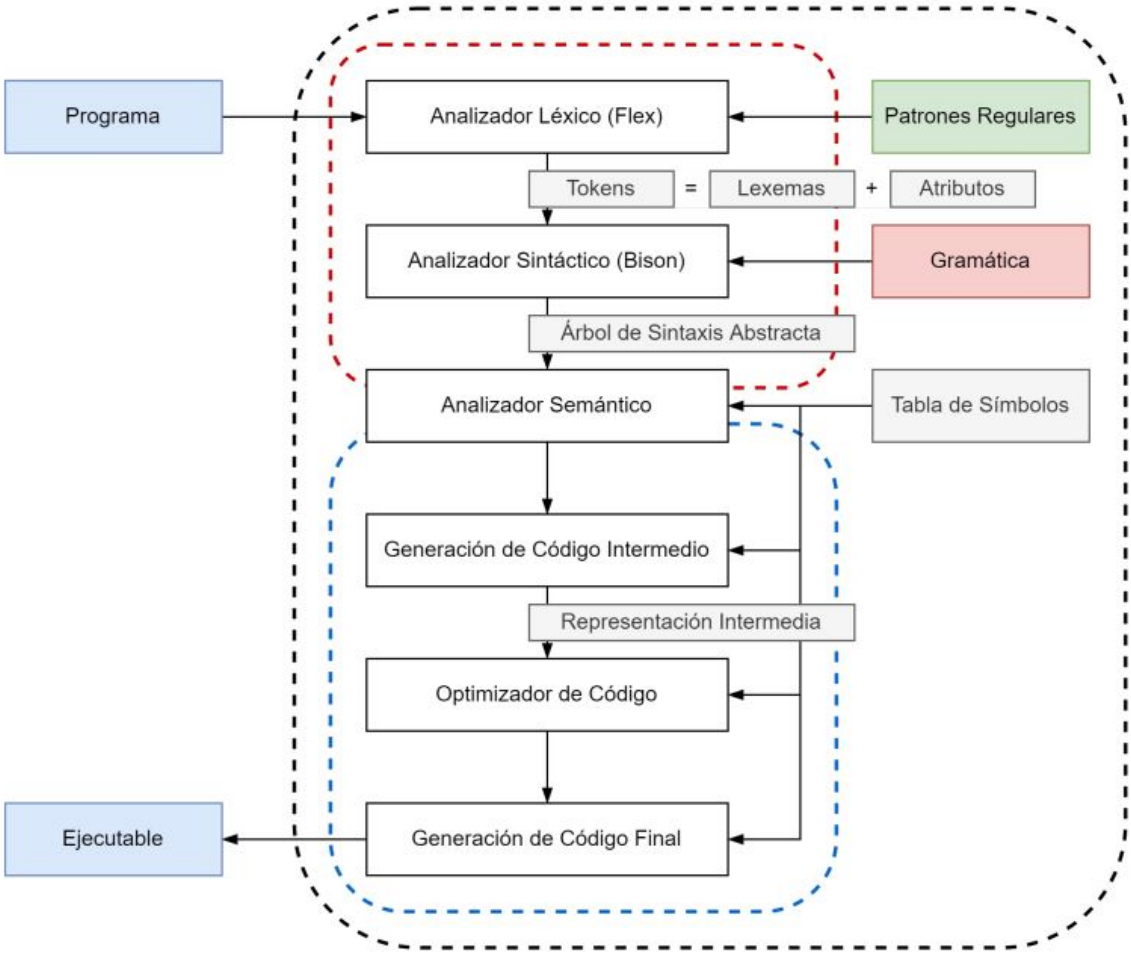

Figura 1: Diagrama de flujo tentativo de la arquitectura del compilador.

Frontend

Para comenzar con el desarrollo del trabajo, lo primero a realizar fue plantear, diseñar e implementar la gramática encargada de identificar nuestro lenguaje llamado 'Nimloth'.

Adicionalmente, se decidió que los árboles manejen unicamente el numeros positivos, y como se mencionó previamente, solamente manejamos árboles BST, AVL y RBT.

Prestaciones del lenguaje

Principalmente, Nimloth cuenta con tres 'acciones' detalladas a continuación.

En las ***declaraciones*** se busca que el usuario puede declarar las variables de tipo 'tree' e inicializarlas con al menos uno o más nodos. Por default, el tipo de árbol será un BST.

```
tree arbol(1);
tree arbol2(1,2,3,4);

```

En los ***bloques de configuración***, se busca que el usuario pueda configurar el árbol que va a desear imprimir. Al comenzar el bloque se podrá redefinir el tipo del árbol a configurar, y sino por default se mantendrá el tipo anterior que este tuviera. Luego se podrá agregar, buscar y eliminar nodos. En caso de que se inserte un nodo duplicado, o se desee eliminar un nodo inexistente, el programa lanzará un warning para notificarlo.

```
configure arbol {
    addNode(2);
    deleteNode(1)
    addNode(3);
    findNode(2);
}
configure avl arbol2 {
    deleteNode(1,4)
    addNode(8,9,10);
    findNode(2);
}
configure rbt arbolRbt {
    findNode(2,3);
}

```

Por último, en los ***bloques de creación***, el usuario podrá especificar el nombre de la imagen de salida del programa, los árboles presentes en la imagen, el path donde se encontrara el archivo y la configuración de títulos en la imagen. Estos títulos incluyen: el nodo de mayor o menor valor, la cantidad de nodos, la altura del árbol y si el mismo se encuentra o no balanceado. Si se agrega un título duplicado, el programa lanzará un warning para notificarlo. La salida generada por dicho bloque será un archivo .dot y un archivo .png con el nombre especificado.

```
create imagenDeSalida {
    addTree(arbol, arbol2);
    addFilePath("/miUsuario/")
    addLegends(max, min, count, height, balanced);
}

```

Consideraciones generales de sintaxis

- NO es posible declarar un árbol sin especificar al menos un nodo.
- Todas las líneas deben terminar con ';'.
- El contenido de los bloques de configuración y creación se debe encontrar entre llaves.
- Los bloques de configuración y de creación NO pueden estar vacíos, deben contar con al menos una instrucción. Esto se debe a que no tiene mucho sentido hacer un bloque que no realice modificación alguna sobre el árbol o archivo al que refiere.
- Si en los bloques de creación, la única instrucción presente es *addLegends()* o *addFilePath()*, el programa generará un árbol vacío, lo cual se notificará con un warning. Queda a responsabilidad del usuario agregar la instrucción *addTree()* para que se grafiquen los árboles requeridos.
- Es posible realizar comentarios de la siguiente forma */* comentario */*

Flex y Bison

Una vez definida la gramática del lenguaje, se hizo uso de Flex (para definir los tokens del lenguaje, como *tree*, *configure*, *create*, *(*, etc.) y Bison para encontrar errores en la sintaxis del programa mediante el uso de las reglas de producción definidas en nuestra gramática.

Backend

Creación del AST (Abstract Syntax Tree)

Para comenzar con el desarrollo del backend del compilador, se volvió a hacer uso del analizador sintáctico Bison para construir el árbol AST a medida que se va parseando el programa, construyendo así todos los tipos de nodos del árbol (cada nodo corresponde con cada símbolo definido en nuestra gramática).

Tabla de símbolos

Luego se prosiguió con la creación de una tabla de símbolos para almacenar las variables encontradas a medida que el parsing avanza. Nuestra tabla de símbolos se encargará unicamente de almacenar las variables de tipo *tree* y los nombres de los archivos que generará el compilador, es decir, *filenames*. Para realizar la tabla de símbolos se hizo uso una librería de hashMap llamada ***kHash*** que nos permitió un acceso rápido y sin complicaciones de implementación.

También se hizo uso de la tabla de símbolos para la validación de las variables, ya que la única manera de utilizar una variable del tipo *tree* es luego de haberla declarado. De esta forma, cada vez que se ejecuta una función de Bison donde se utiliza una variable *tree* se verifica que la variable exista en la tabla de símbolos y además se hace un chequeo del tipo de dato. De forma análoga, se verifica en los bloques de creación, que el nombre del archivo de salida no se encuentre repetido.

Generación de código

Para la generación de código utilizamos librerías externas levemente modificadas para el manejo de los árboles. Adicionalmente, manejamos estructuras creadas por nosotros para ir guardando cada árbol con sus nodos asociados y los archivos a generar. Como implementación se decidió que la imagen de salida se generara inmediatamente al salir del bloque de creación. Esto conlleva a que si el código a continuación de dicho bloque retorna algún tipo de error, la imagen de todas formas habrá sido generada y adicionalmente ahorramos el tener que guardarnos todos los archivos en memoria para después crearlos al final.

Posibles errores

- La compilación puede fallar en caso de utilizar un árbol que no haya sido declarado.
- La compilación puede fallar en caso de declarar un árbol sin nodos.
- La compilación puede fallar en caso de que los bloques de configuración o creación se encuentren vacíos.
- La compilación puede fallar por errores de sintaxis.
- Se contemplan fallos en el momento de generar el código de las funciones malloc y calloc, en cuyo caso se imprimirá el error correspondiente.
- Se utilizan las funciones fopen y system para generar los dos archivos de salida. En caso de que alguna de ellas falle se imprimirá el error correspondiente.

Guía de instalación y uso

Los requisitos para ejecutar el proyecto son los siguientes:

- Bison v3.8.2
- CMake v3.24.1
- Flex v2.6.4
- GCC v11.1.0
- Make v4.3
- DOT v2.40.1

Para construir el proyecto, basta posicionarse en la raíz del proyecto y ejecutar:

```
./script/build.sh

```

Tras esto, se debe crear un archivo llamado *program* en la raíz del proyecto con código correspondiente a nuestro lenguaje. Para compilar el programa se debe ejecutar:

```
./script/start.sh program

```

Ejemplo de uso

A continuación mostramos un ejemplo de la salida del siguiente programa:

```
/* Declaramos un arbol con nodos*/
tree miArbol(1,2,3,4);
/*Se balancea como AVL, agregamos y buscamos nodos*/
configure avl miArbol {
    addNode(5,6,7,8,9);
    findNode(5,7);
}
/*Creamos el archivo e imagen 'completoAvl'*/
create completoAvl {
    addTree(miArbol);
    addFilePath("/home/lferreiro/TLA/TPE-TLA/");
    addLegend(max, min, count, height, balanced);
}
/*Eliminamos dos nodos*/
configure miArbol {
    deleteNode(1,6);
    findNode(5,7);
```

```
18      }
19      /*Volvemos a imprimir el arbol con los nodos eliminados*/
20      create completoAvlSin1y6 {
21          addTree(miArbol);
22          addFilePath("/home/lferreiro/TLA/TPE-TLA/");
23          addLegend(max, min, count, height, balanced);
24      }
25      /*Cambiamos su balanceo de AVL a RBT y agregamos nodos*/
26      configure rbt miArbol {
27          addNode(1,6);
28          findNode(5,7);
29      }
30      /*Imprimimos el arbol balanceado como RBT*/
31      create completoRbt {
32          addTree(miArbol);
33          addFilePath("/home/lferreiro/TLA/TPE-TLA/");
34          addLegend(max, min, count, height, balanced);
35      }
36
```

Tras compilar esto, se generarán las siguientes imágenes:

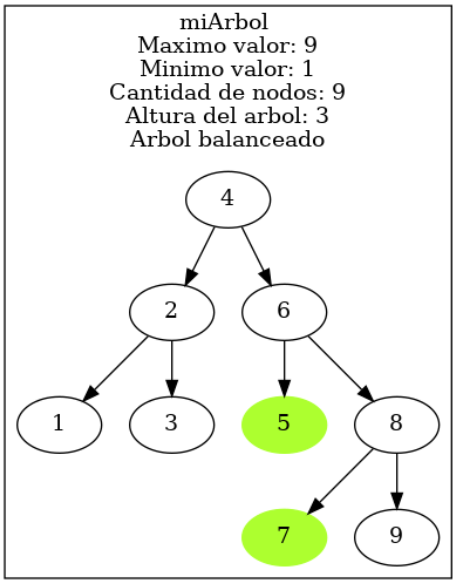

Figura 2: Árbol completoAvl.

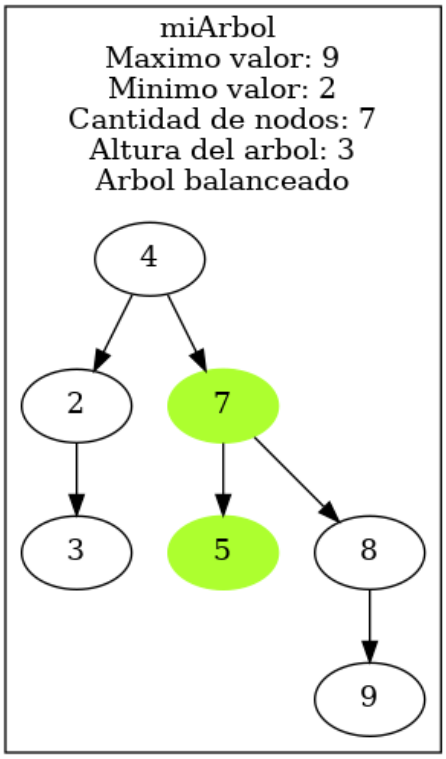

Figura 3: Árbol completoAvlSin1y6.

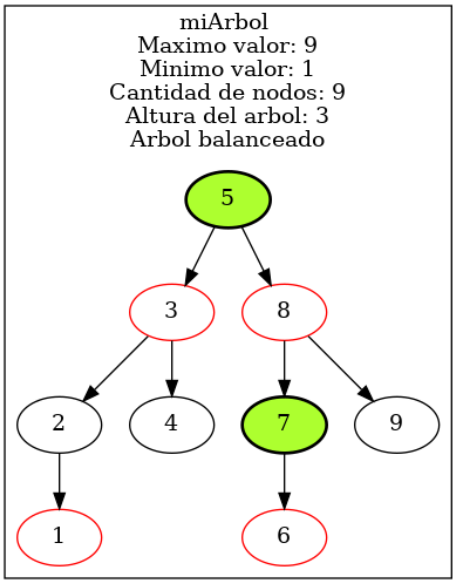

Figura 4: Árbol completoRbt.

Dificultades encontradas

Las dificultades encontradas a lo largo del desarrollo del lenguaje se encontraron principalmente en encontrar una implementación adecuada y correcta a nuestras necesidades, ya que definimos que se pueda hacer manejo de distintos tipos de árboles, los cuales necesitan hacer uso de métodos apropiados a la funcionalidad de cada uno de ellos.

Otra dificultad encontrada fue no lograr encontrar una correcta implementación de borrado de nodos para los árboles RBT. Por lo tanto, dicha funcionalidad no fue implementada y en caso de querer utilizarla se arrojará un warning. La solución a este problema se deja planteada como una futura extensión del proyecto.

Por otro lado, se tuvieron dificultades con errores del editor de texto/IDE elegido, ya que muchas veces se presentaban errores de include en el proyecto, los cuales se resolvían de manera muy peculiar, y sin mucho sentido lógico.

Futuras extensiones

Una posible manera de escalabilidad a Nimloth, seria que los nodos de los árboles puedan manejar distintos tipos de datos, como también hacer manejo de una mayor cantidad de tipos de árboles, lo que exigiría como consecuencia escalar en la cantidad de funciones o acciones que se puedan aplicar sobre estos.

Otro posible ampliación del proyecto sería extender nuestra gramática con el objetivo de potenciar todas las funcionalidades que nos ofrece el lenguaje DOT, ofreciendo más opciones de personalización al estilo de la salida generada.

Conclusión

Durante el desarrollo de este trabajo se obtuvo un gran conocimiento sobre como hacer uso de las tecnologías de Flex y Bison (y también el lenguaje DOT), como también aplicar los conocimientos teóricos vistos en clase, de manera que estos últimos no sean solamente plasmados en papel en un examen, sino también, vistos en acción y así profundizar el entendimiento en cuanto al funcionamiento de los lenguajes de programación y los compiladores.

Referencias y bibliografía

- https://graphviz.org/

- https://github.com/agustin-golmar/Flex-Bison-Compiler
- https://github.com/attractivechaos/klib/blob/master/khash.h
- https://www.programiz.com/dsa/binary-tree
- https://www.programiz.com/dsa/avl-tree
- https://www.codesdope.com/course/data-structures-red-black-trees-insertion/
- https://www.codesdope.com/course/data-structures-red-black-trees-deletion/
- https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/