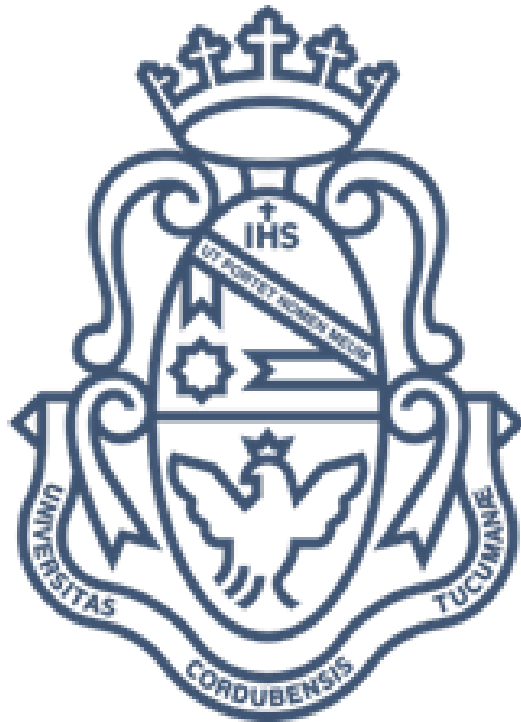


Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales



Práctica y Construcción de Compiladores

Proyecto Final

**Compilador básico de lenguaje C
con generación de TAC y optimización**

Daniele Francisco

Índice

| | |
|---|-----------|
| Índice | 2 |
| Introducción | 3 |
| Herramientas | 3 |
| Repositorio | 3 |
| Desarrollo | 4 |
| Sistema de archivos | 4 |
| App.java | 4 |
| declaraciones.g4 | 4 |
| Escucha.java | 5 |
| FileRW.java | 5 |
| Funcion.java | 5 |
| Generador.java | 5 |
| Id.java | 5 |
| Optimizador.java | 5 |
| TablaSimbolos.java | 5 |
| TipoDato.java | 5 |
| Variable.java | 6 |
| Visitor.java | 6 |
| Diagramas de clases | 6 |
| Manejo de archivos | 6 |
| Sistema de símbolos | 6 |
| Listener - Tabla de símbolos | 7 |
| Ecosistema Visitor | 7 |
| Optimizador | 8 |
| Diagrama completo | 9 |
| Diagrama de flujo | 10 |
| Gramática - Análisis léxico sintáctico | 10 |
| Listener - Análisis semántico | 11 |
| Visitor - Generación de Código de 3 Direcciones | 12 |
| Optimizaciones | 13 |
| Known Issues | 13 |
| Conclusión | 15 |
| Referencias | 16 |

Introducción

Este trabajo se elaboró según lo propuesto por el profesor Maximiliano Eschoyez de la materia Práctica y Construcción de Compiladores de la UNC - FCEFyN como cierre de la asignatura para asentar conocimientos y demostrar lo aprendido.

El objetivo de este proyecto es comprender cómo funciona un compilador y que va sucediendo en las distintas etapas de trabajo del mismo, desde que se tiene código fuente hasta obtener la salida final que es código máquina. El alcance del proyecto va desde el análisis léxico de código fuente hasta la obtención y optimizado de código intermedio y se fue desarrollando de forma incremental, iniciando al principio del semestre de cursado.

Se generó un compilador de lenguaje C básico, comenzando con su análisis léxico (y en poca medida sintáctico) con la creación de una gramática para ANTLR, luego se incorporó un Listener para realizar análisis sintáctico completo y crear el árbol sintáctico, y realizar el análisis semántico y obtener un árbol sintáctico con anotaciones y una tabla de símbolos. A continuación, un Visitor recorre el árbol obtenido y genera código intermedio, más precisamente código de tres direcciones para finalmente aplicar algunas optimizaciones al mismo.

Herramientas

Para desarrollar el proyecto se utilizaron las siguientes herramientas:

- Maven para generar el espacio de trabajo,
- Visual Studio Code como editor de texto,
- GitHub para control de versiones,
- Java como lenguaje de programación, y
- ANTLR como parser.

Repositorio

El repositorio del proyecto se encuentra en el siguiente link:

<https://github.com/frandaniele/Compiladores2022>

Allí se encuentra el código fuente generado a lo largo del desarrollo del proyecto, pudiéndose observar todo el progreso, a lo largo de los distintos cambios y versiones.

Desarrollo

El proyecto se divide en 4 partes, cada una con su propio objetivo tomando una entrada y generando una salida para que la siguiente parte tome como entrada y la procese.

Se parte de la generación de una gramática para un lenguaje específico y así determinar si nuestra entrada es propia de dicho lenguaje, y sigue con un analizador semántico para encontrar sentido al código de entrada. A continuación se genera código de tres direcciones a partir del código original y por último se realizan optimizaciones sobre ese código obtenido.

Sistema de archivos

App.java

Es el archivo con el *entry point* para ejecutar el sistema creado.

Contiene el método **main** el cuál lee el código fuente como input, crea un lexer que se alimenta de dicho input, genera un buffer de tokens a partir del lexer y un parser que se alimenta de los tokens. Luego se instancia un listener, lo agrega al parser y este último comienza su trabajo a partir del símbolo inicial. A continuación se instancia el visitor y este *visita* el árbol generado creando el código de 3 direcciones. Como último paso se crea el optimizador y realiza pasadas sobre el código hasta no poder optimizar más, generando como salida final el TAC con optimizaciones.

declaraciones.g4

Este es el archivo de la gramática para el idioma del proyecto. ANTLR lo toma como entrada para reconocer dicho idioma.

Contiene las reglas para lexer (operadores, keywords, números, etc) y parser (instrucciones, operaciones).

Esta gramática contiene todas las reglas para posibilitar el reconocimiento de un lenguaje símil C que consta de una regla inicial llamada *programa* que es una secuencia de instrucciones seguida de *EOF*. Las instrucciones reconocidas son las siguientes:

- declaraciones
- bloques (código entre llaves)
- while loops
- for loops
- if-elif-else
- funciones
- llamados a funciones
- asignaciones
- operaciones de post/pre incremento o decremento

Las operaciones aritmético-lógicas implementadas son, en su orden de precedencia descendiente:

- post/pre incremento o decremento (++ --)
- multiplicación, división y módulo (* / %)
- suma y resta (+ -)
- operadores de relación (< <= > >=)
- igualdad y desigualdad (== !=)
- AND bit a bit (&)

- OR bit a bit (|)
- AND lógica (&&)
- OR lógica (||)

Además se reconocen los comentarios de línea, precedidos por '//', y de bloque, de la forma '/* comentario */', para ignorarlos.

Escucha.java

Es el archivo del *listener* del sistema, el que se encarga de analizar el árbol generado por el parser (ANTLR) y realiza el análisis semántico. En caso de haber errores en el código fuente los marca, y caso contrario maneja la tabla de símbolos y genera como salida los símbolos del código de entrada.

FileRW.java

Esta clase contiene los métodos *readFile* y *writeFile* utilizados por otras clases para interactuar con archivos de texto, leyéndolos y escribiéndolos.

Funcion.java

Extiende a la clase *Id*, agregando una lista, *args*, para agregar sus parámetros mediante el método *addArg*, y un campo *estado* (Integer) para saber en qué estado se encuentra la función (0: no existe, 1: fue prototipada, 2: fue inicializada). Las instancias de esta clase se incorporan a la tabla de símbolos.

Generador.java

Esta clase implementa el patrón singleton y su función es generar y mantener la lista de labels y variables temporales utilizadas para la generación de código intermedio.

Id.java

Es una clase abstracta, base para la creación de funciones y variables para la tabla de símbolos. Tiene los campos *nombre* (String), *tipo* (TipoDato), *usado* (Boolean) e *init* (Boolean), con sus getters y setters.

Optimizador.java

Este archivo se encarga de analizar su entrada de texto (TAC) y reconocer patrones para aplicar distintas optimizaciones al mismo en varias pasadas. Genera como salida el TAC con las optimizaciones realizadas.

TablaSimbolos.java

Esta clase implementa el patrón singleton y su función es mantener los contextos con los símbolos generados por el código. Si no hay errores en el código, se va a encargar de generar un archivo con todos los símbolos.

TipoDato.java

Es un enum con los tipos de datos permitidos por el idioma creado: int, char, double y void.

Variable.java

Esta clase extiende a Id sin incorporar ninguna funcionalidad. Se usa para agregar variables a la tabla de símbolos.

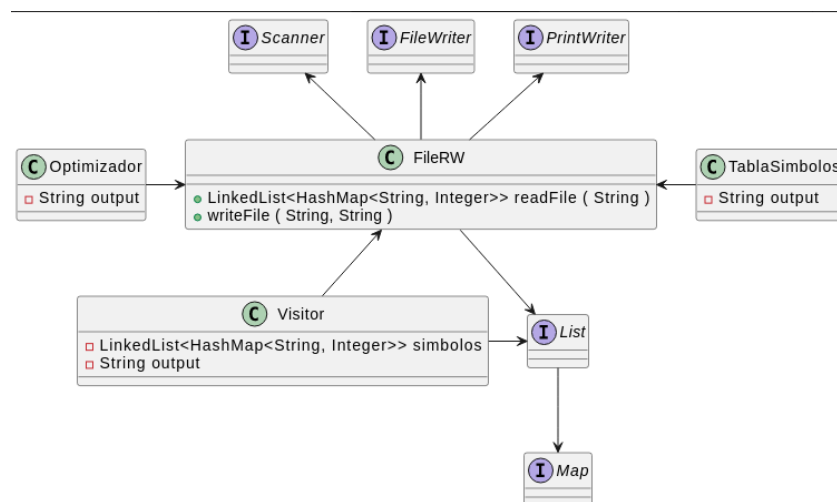
Visitor.java

El visitor recorre el árbol sintáctico ya generado, decide cuándo pasar de nodo a nodo y va creando el código intermedio (de tres direcciones) equivalente al código fuente original.

Diagramas de clases

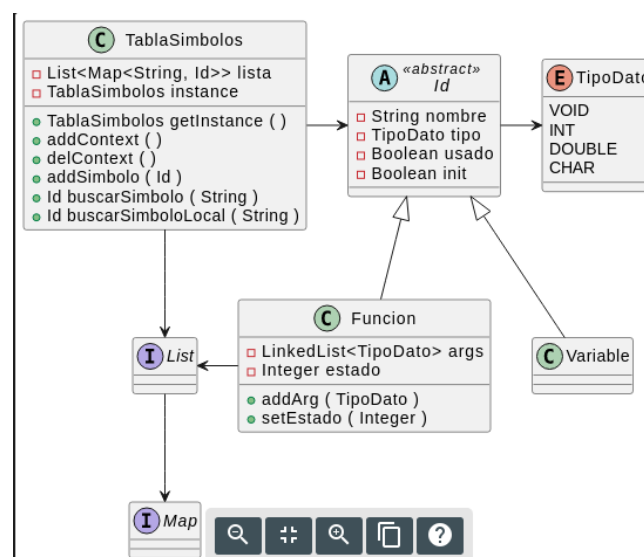
Manejo de archivos

Este diagrama muestra las clases que manipulan archivos a través de los métodos de la clase FileRW



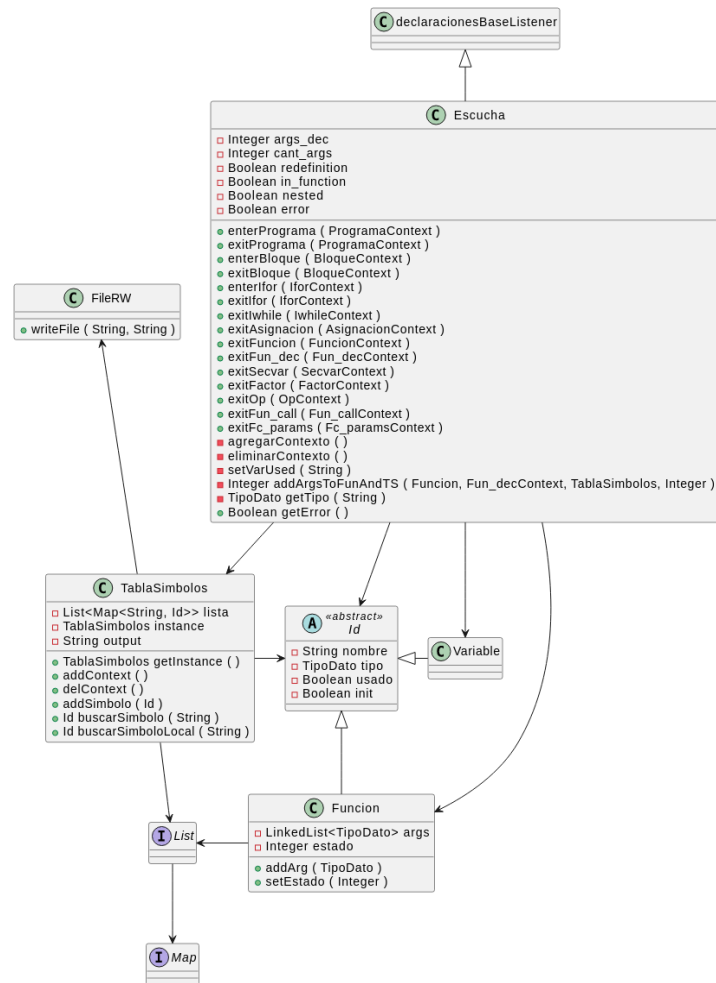
Sistema de símbolos

Aquí se muestra el diagrama de las clases que conforman la tabla de símbolos.



Listener - Tabla de símbolos

Este diagrama nos permite observar cómo se relaciona el listener con las demás clases. Usa la Tabla de Símbolos para agregar los símbolos del código de entrada y luego la Tabla se aprovecha del método de FileRW para escribir un archivo con su contenido.



Ecosistema Visitor

En este diagrama vemos a la clase Visitor y cómo se relaciona con las demás. Mediante FileRW lee para obtener los símbolos del archivo generado por la tabla y también mediante FileRW escribe su salida generada (código intermedio) en un archivo. Además usa la clase Generador para manejo de labels y variables temporales.

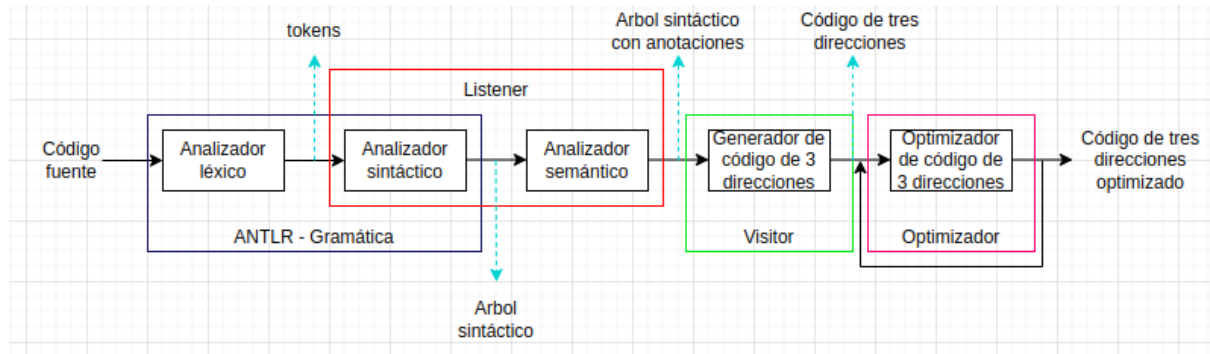


Optimizador

En este diagrama observamos la clase Optimizador con sus campos y métodos. Se ve cómo se vale de *Paths* y *Files* para leer las líneas de TAC para finalmente utilizar a FileRW para escribir la salida generada, el código de tres direcciones optimizado.

Diagrama de flujo

A continuación, se presenta un diagrama de flujo que muestra cómo interactúan cada una de las partes del proyecto, desde la primera entrada, pasando por las salidas de cada parte hasta la salida final.



Gramática - Análisis léxico sintáctico

El primer paso para la construcción del sistema propuesto fue interiorizarse con un parser como ANTLR, el elegido en este caso, configurarlo y lograr una gramática para que ANTLR parsee y reconozca el idioma creado en esa gramática, siendo aquí similar al lenguaje C.

Se comenzó con sentencias básicas y operaciones sencillas, a partir de ejemplos dados por el profesor y tareas propuestas, para ir avanzando hasta llegar a instrucciones más complejas (condicionales y loops), mayor cantidad de operaciones aritmético-lógicas con su orden de precedencia, y manejo de funciones.

La gramática en cuestión se empezó a construir con reglas simples, primero reglas léxicas para reconocer keywords, como *int*, *char*, *double*, *while*, *for*, etc, operandos (+, -, *, /...), identificadores, números y espacios en blanco (se ignoran con el comando *skip*). Progresivamente se fueron insertando *parser rules* empezando a moldear el lenguaje. Se empezó con reglas simples cómo reconocer sentencias entre paréntesis y llaves, como reconocer la estructura de un programa (secuencia de instrucciones y EOF como final del programa), operaciones simples (sumas, restas y multiplicaciones). Luego se fue incorporando otro tipo de instrucciones como *ifs*, *fors*, *whiles*, asignaciones y declaraciones de variables, para finalmente agregar lo más complejo como son muchas operaciones aritmético-lógicas con el orden de precedencia y las funciones y llamadas a las mismas.

A continuación se adjunta la imagen de un árbol construido a partir del parseo de un programa simple que consta de tres instrucciones para observar en detalle cómo se reconoce el idioma. El programa es el siguiente:

```
int x, y = 3;
x = 3*(y-2);
return x;
```

En el árbol se puede ver cómo la raíz es la regla inicial, programa, la cuál se forma por la regla instrucciones seguida de EOF. La regla instrucciones puede ser o una instrucción

(es una regla también) seguida recursivamente por instrucciones cómo vemos en el árbol, o la regla vacía. La regla instrucción puede ser una declaración, asignación, bloque, if, while, for, return, función, llamada a función... En este caso vemos cómo hay una declaración, una asignación y un return.



En resumen, el trabajo hasta aquí realizado permite el análisis léxico y sintáctico de una entrada. Es decir, encontrar los tokens que la conforman y determinar si pertenece al lenguaje que estamos analizando.

Listener - Análisis semántico

Una vez creada la gramática y logrado el reconocimiento del lenguaje buscado, se pasó al desarrollo de un listener que realice “al vuelo” el análisis semántico sobre el árbol mientras va siendo construido por el parser.

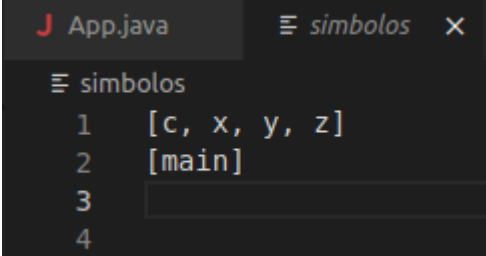
Esto consiste en analizar el significado del código fuente y verificar que tenga sentido según el idioma creado. Para eso se determina si existen errores que en ese caso se comunican y se detiene la compilación. Caso contrario se agregan a la Tabla de Símbolos todos los que contenga el programa, variables y funciones en sus contextos correspondientes, con sus características (usados, inicializados). Una vez finalizado, se

imprime el contenido de la tabla en un archivo para procesar en el siguiente paso del proceso de compilación.

En las siguientes imágenes podemos observar el caso de la salida de un programa con errores y el de uno sin, obteniéndose la tabla de símbolos.

```
+++++++PARSER+++++++  
  
error: expected expression before ')' token  
variable x not used  
error: void value not ignored as it ought to be  
variable x not used  
error: control reaches end of non-void function  
  
+++++++  
  
variable foo not used  
  
*****  
fin compilacion  
Termino el parseo.  
Sintaxis incorrecta. FIN
```

```
+++++++PARSER+++++++  
  
variable c2 not used  
  
+++++++  
  
*****  
fin compilacion  
Termino el parseo.
```



Para realizar el listener se extiende la clase `baseListener` que proporciona ANTLR y se utilizan los métodos para entrar y salir de cada regla, analizando los tokens que las conforman y determinando su significado según el contexto.

Una vez terminado el parseo y obtenido el árbol y la tabla de símbolos, podemos proceder con el siguiente paso que consiste en recorrer nodo a nodo el árbol para generar código intermedio.

Visitor - Generación de Código de 3 Direcciones

El visitor, al igual que el listener, se realiza a partir del `baseVisitor` que proporciona ANTLR y se sirve de los métodos para visitar las reglas (nodos). Lo primero que se hace es leer el archivo de la tabla de símbolos para así poder luego ignorar los nodos que involucren variables no usadas, y por consiguiente innecesarias, que no aparecen en la misma.

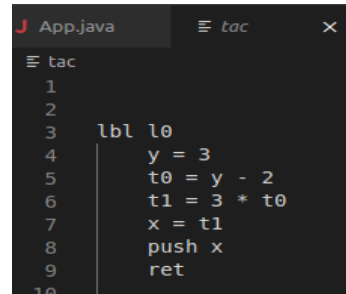
La función del visitor es generar código intermedio que es una representación simple independiente del hardware del código original. Este código es apropiado para realizar optimizaciones antes de generar código objeto o máquina.

Uno de los más populares de este tipo de código es el de tres direcciones, el cuál es el elegido para este proyecto. El código de tres direcciones, o TAC, representa el programa en un lenguaje en el que se utilizan hasta tres direcciones de memoria en el cuál el formato base es $x = y \text{ op } z$. Se sirve de un generador de etiquetas para el caso de los saltos y de variables temporales para ir dividiendo operaciones complicadas. A continuación podemos observar el código generado para el siguiente simple programa:

```
int main() {
    int x, y = 3;

    x = 3*(y-2);

    return x;
}
```



```
App.java  tac
tac
1
2
3    lbl l0
4        y = 3
5        t0 = y - 2
6        t1 = 3 * t0
7        x = t1
8        push x
9        ret
10
```

Cuándo se termina de recorrer el árbol y se genera el *three address code* se escribe la salida en un archivo.

Optimizaciones

Al finalizar la generación de TAC, se procede a aplicar algunas optimizaciones al mismo. Para ello se lee el archivo de texto del código generado y se usa reconocimiento de patrones de texto realizando posiblemente más de una pasada de optimización.

Los pasos que se siguen son:

1. Obtener línea a línea el TAC
2. Identificar los objetivos de saltos en el código
3. Dividir en bloques básicos
4. Aplicar optimizaciones
5. Escribir el código optimizado en un archivo
6. Repetir 1 a 5 hasta no poder optimizar más

Las optimizaciones aplicadas son locales:

- Propagación de constantes
- *Constant folding*
- Eliminación de variables no utilizadas
- Eliminación de código “muerto”
- Eliminación de operaciones innecesarias
- Eliminación de subexpresiones comunes
- Eliminación de código inalcanzable
- Resolución de identidades aritméticas

Para poder aplicarlas se usa un mapa que contiene las variables y sus valores a medida que se pueden ir resolviendo las operaciones y otro que contiene las variables y su última asignación. También se tiene una lista con las variables usadas en cada pasada para poder eliminar las innecesarias y una con las funciones a las que nunca se llaman para también eliminarlas.

Quedaron pendientes aplicar otras optimizaciones que van más allá de un bloque básico cómo son eliminar saltos no tomados, sacar operaciones repetitivas afuera de un loop, *loop unrolling*, aplicar las optimizaciones locales globalmente, etc.

Known Issues

En este apartado del informe se mencionan algunos problemas que se detectaron en el sistema, sobre los cuáles no se pudo trabajar más profundamente debido al tiempo

requerido y el disponible, y a que el proyecto se centra en el funcionamiento más general que específico y se da prioridad a trabajar con condiciones óptimas en las entradas al sistema.

A continuación se enumeran algunos *bugs* no resueltos

1. En las operaciones que involucren dos operandos con, uno con pre y otro con post incremento o decremento, por ej. $i++ * ++j$, la actualización de las variables no se hace en el momento correspondiente, sino que las dos se actualizan como si fueran la operación del segundo operando.
2. El optimizador transforma los *doubles* a enteros, por consiguiente no existen los *doubles* en el código optimizado.
3. Cuando se va a optimizar y hay un bloque de *if* sin su *else* y luego otro bloque *if*, el optimizador no detecta que no están relacionados y eso genera valores incorrectos de las variables en muchos casos por sobreescritura o por ignorar actualizaciones obligatorias.
4. No existe chequeo, y por consiguiente no se marca el error, cuando se prototipa una función de un tipo y se inicializa con otro tipo. Lo mismo ocurre con sus parámetros.
5. No existe chequeo exhaustivo de errores por lo que existen varias situaciones que pueden crashear el programa ya que no se salvan los errores. Por ejemplo, cuando se escriben instrucciones sin estar contenidas en una función o bloque.
6. Si el código fuente se escribe dentro de una función, por ej. *main* y al final no llamo a esta función el programa crashea.

Conclusión

Para dar cierre a este proyecto realizado a lo largo de todo el segundo semestre de cursado del año 2022 se pasan por escrito algunas reflexiones obtenidas.

Primeramente podemos decir que fue una propuesta muy interesante, donde se introdujeron temas (parseo, compilación...) que bajo el punto de vista desde el que los afrontamos son *nuevos* y nos ayudan a comprender un poco que va pasando cuándo escribimos código que luego ejecutamos, lo que anteriormente veíamos como “automático”, sin prestar mucha atención. Con esto también se pudo ver cómo funcionan los lenguajes, cómo se construyen, el porqué de su sintaxis y comprender algunas particularidades que antes parecían *trucos* de los mismos.

Otro aspecto del trabajo digno de mencionar es que al ser *grande y complejo* se debió trabajar con un gestor de proyectos como Maven, con GitHub para la gestión de versiones y hacer el mejor uso posible de ellos para evitar complicaciones y facilitar el desarrollo. Cómo autocrítica se menciona que se debería haber afrontado el proyecto con más diseño antes de desarrollar código y haber dado más importancia al testeado para tener mejor organización y evitar complicaciones.

En resumen, este trabajo terminó siendo uno de los más interesantes, complejos y grandes en términos de código generado y tiempo requerido. Su desarrollo fue muy provechoso ya que terminó siendo bastante abarcativo, introduciendo nuevos conceptos y ayudando a moldear una forma de trabajar para así facilitar las cosas en el futuro, cuándo todos los proyectos que se afronten serán complejos y de gran tamaño.

Referencias

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, & Tools", Second Edition, Addison-Wesley Iberoamericana, 2007
- <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/>
- <https://www.geeksforgeeks.org/compiler-design-detection-of-a-loop-in-three-address-code/>
- <https://www.geeksforgeeks.org/code-optimization-in-compiler-design/>
- <https://baptiste-wicht.com/posts/2012/02/local-optimization-on-three-address-code.html>