Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Ingeniería en Computación

Compilador para C Reducido

Trabajo Práctico Final, Práctica y Construcción de Compiladores

Integrantes: Cabrera, Augusto Gabriel
Villar, Federico Ignacio
Profesor: Eschoyez, Maximiliano Andrés
Fecha de entrega: 13 de diciembre de 2022
Córdoba, Argentina

Resumen

En el siguiente informe se desarrollarán las explicaciones correspondientes al desarrollo de un compilador para lenguaje C reducido. Este compilador al que se hace referencia fue iniciado en su desarrollo en las clases de la materia, luego, gracias a la lectura de la bibliografía, junto con la puesta en práctica de lo aprendido se logró crear lo mostrado en las siguientes páginas.

El software a desarrollar tiene como objetivo a partir de un archivo de código fuente en C el generar como salida una verificación léxica, gramatical y semántica reportando errores en caso de existir. Una vez completado el reporte anterior, se genera una salida de código intermedio de tres direcciones.

En este informe se incluirán conceptos teóricos, explicaciones de desarrollo de código, algoritmos y figuras que permitan la correcta interpretación del trabajo realizado.

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Consigna

El objetivo de este Trabajo Final es mejorar el filtro generado durante el cursado. Dado un archivo de entrada en C, se debe generar como salida el reporte de errores en caso de existir. Para lograr esto se debe construir un parser que tenga como mínimo la implementación de los siguientes puntos:

- Reconocimiento de bloques de código delimitados por llaves y controlar balance de apertura y cierre.
- Verificación de la estructura de las operaciones aritmético/lógicas y las variables o números afectadas.
- Verificación de la correcta utilización del punto y coma para la terminación de instrucciones.
- Balance de llaves, corchetes y paréntesis.
- Tabla de símbolos.
- Llamado a funciones de usuario.

Si las fases de verificación gramatical y semántica no han encontrado errores, se debe proceder a:

1. Detectar variables y funciones declaradas pero no utilizadas y viceversa.
2. Generar la versión en código intermedio utilizando código de tres direcciones, el cual fue abordado en clases y se encuentra explicado con mayor profundidad en la bibliografía de la materia.

En resumen, dado un código fuente de entrada el programa deberá generar dos archivos de salida:

1. Las tablas de símbolo para cada contexto.
2. La versión en código de tres direcciones del código fuente de entrada.

2. Marco Teórico

2.1. ANTLR

(ANother Tool for Language Recognition, por sus siglas) es una herramienta de lenguaje que proporciona un framework para la construcción de reconocedores, intérpretes, compiladores y traductores a partir de descripciones gramaticales que contienen acciones en varios lenguajes.

Se encarga de generar Scanners, Parsers y Tree Parsers automatizando la construcción de reconocedores de lenguaje. A partir de una descripción formal del lenguaje, ANTLR genera un conjunto de rutinas que determinan si una entrada forma parte de ese lenguaje tanto a nivel léxico como sintáctico.

Genera Parsers de tipo recursivo descendente para el lenguaje deseado. Este lenguaje puede ser tanto un lenguaje de programación como un formato de datos, pero ANTLR no sabe nada acerca del lenguaje especificado en la gramática. También se pueden incluir fragmentos de código que realicen determinadas tareas durante el proceso de reconocimiento. ANTLR permite anotar la gramática con operadores de forma que el usuario describe como quiere que sea el AST que espera que se construya.

La función principal de ANTLR es facilitar el trabajo al programador, automatizando aquellas tareas más complicadas que forman parte del proceso de reconocimiento a nivel léxico y sintáctico de un lenguaje.

ANTLR permite generar código en C, Java, Python, C#, Objective-C y otros que actualmente están en desarrollo como por ejemplo, C++.

2.1.1. Estructura de ANTLR

A partir de una gramática combinada G (Por ejemplo) donde se especifican tanto los tokens como la gramática propiamente dicha en un mismo fichero G.g4, ANTLR genera un analizador formado por los siguientes archivos:

- **GLexer.java**. Contiene el Lexer generado a partir del fichero G.g4
- **GParser.java**. Contiene el Parser generado por la gramática.
- **G.tokens**. Contiene una lista de los tokens de la gramática.

La extensión java indica el código en que se está generando el analizador.

2.2. Compilador

2.2.1. Concepto

Un compilador es un programa que traduce un código fuente en un lenguaje de alto nivel a otro lenguaje de programación de más bajo nivel, el código objeto, que puede ser directamente ejecutable. Para realizar la traducción, primero se debe comprobar la corrección del código fuente informando de posibles errores al usuario. El proceso de compilación se divide en diferentes etapas llevando a cabo en cada una de ellas una tarea diferente.

A modo de introducción, se dice que un compilador es un programa formado por las **Suites de Compilación**.

2.2.2. Suite de Compilación

Es el conjunto de herramientas y archivos que trabajan para producir la compilación, hoy en día se utiliza la siguiente representación de los compiladores.

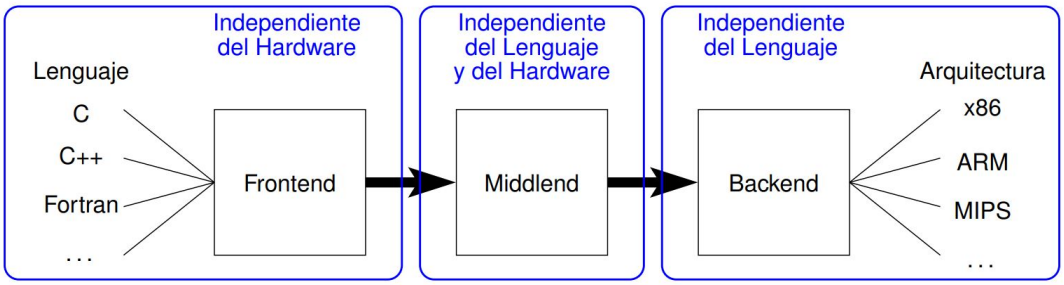

Figura 1: Esquema de una suite de compilación

2.2.3. Etapas del compilador

Funcionalmente, la compilación es un proceso lineal que se divide en diferentes etapas que se muestran en el siguiente gráfico en el orden que se realizan. Cada etapa se estructura en diferentes fases que realizan cada una de las tareas correspondientes. A continuación se muestra una figura donde aparece la estructura típica de un compilador con sus correspondientes fases.

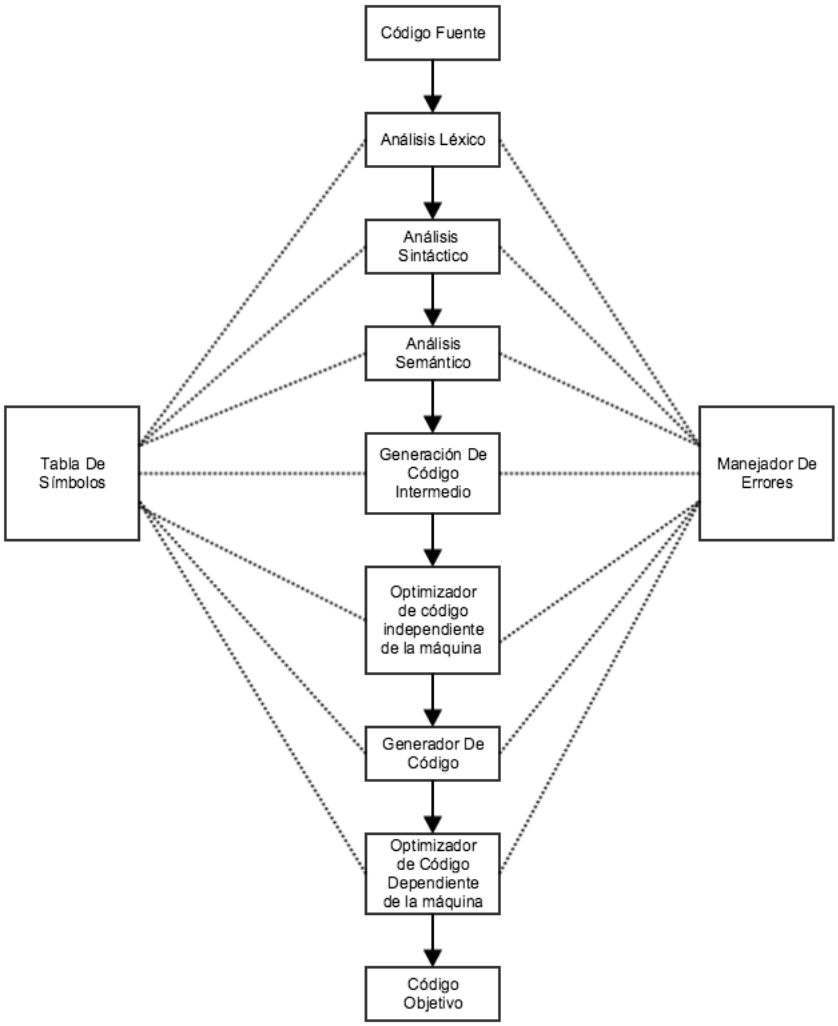

Figura 2: Etapas de la compilación

Se puede clasificar la estructura en diferentes etapas:

1. Front-End o análisis:
    - Análisis Léxico (scanning)
    - Análsis Sintáctico (parsing)
    - Análisis Semántico (TypeCheking)

2. Middle-End o etapa de síntesis de representación intermedia:
    - Generación de Código Intermedio
    - Optimización de Código Intermedio
3. Back-End o etapa de síntesis de código objeto:
    - Generación de Código Objeto
    - Optimización de Código Intermedio

Profundizando en las etapas anteriormente nombradas, se tiene lo siguiente:

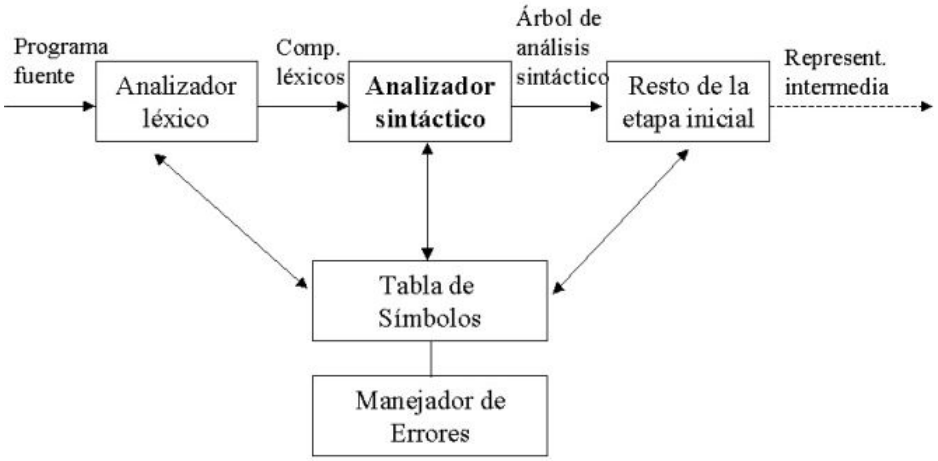

Figura 3: Analizador sintáctico

- **Análisis léxico o scanning**: durante esta fase se lee el código fuente y se fragmenta en tokens o componentes léxicos. Cada uno de estos tokens estará formado por una secuencia de caracteres, que es la unidad mínima de nuestro lenguaje. Los tokens se definen a partir el alfabeto del lenguaje fuente a tratar, y cada uno de ellos, tiene un significado propio. A medida que se fragmenta la entrada, también se comprueba la corrección de los caracteres de entrada, es decir, que si una sucesión de esos caracteres no coincide con alguno de los patrones definidos (tokens), se habrá encontrado un error léxico.

  Para el caso del siguiente lenguaje de alto nivel, se deberían obtener los tokens: PROGRAM, VARS, ID, INT, ENDVARS, ID, ASIG, INT_CONST, WRITE, ABREPAR, ID, PLUS, INT_CONST, CIERRAPAR, ENDPROGRAM.

Código 1: Ejemplo de lenguaje openCL

```
program
vars
a int
endvars
a := 3
write(a + 1)
endprogram
```

Un posible ejemplo de un error sería el caso en que en el anterior programa, en lugar de tener ":=" tenga "::" y por lo tanto, se produce un error léxico.

- **Análisis sintáctico o parsing**: en este punto se comprueba si la secuencia de tokens obtenida de la fase anterior es aceptada por la gramática del lenguaje que se está tratando, agrupando los tokens en frases gramaticales. Si es así, el programa fuente será sintáctica o estructuralmente correcto. La gramática del lenguaje se define mediante reglas recursivas. El resultado ahora será una jerarquía de tokens, representados en forma de árbol o AST (Árboles de Sintaxis Abstracta). La forma más eficiente de manejar la información proveniente de un lenguaje de programación es la forma arbórea; por éso la estructura de datos elegida es un árbol. Además, construyendo ASTs a partir de un texto se puede obviar mucha información irrelevante.

  Ahora, con el ejemplo del código 1, con la gramática definida como la que se muestra a continuación.

Código 2: Gramática de ejemplo en ANTLR

```
1    instr: ID ASIG expr | WRITE ABREPAR expr CIERRAPAR
2    expr: expr_simple (PLUS expr_simple) *
3    expr_simple : ID | INT_CONST
```

  Si se tuviera una entrada de la forma **write(a 1)** , no se podría reconocer por la gramática dada, ya que después de la **a** se espera o bien el token PLUS o el token CIERRAPAR. En este caso, el compilador emitirá un error sintáctico al encontrarse en ese punto, localizándolo y explicando a qué se debe.

  Conceptualmente, se entienden las fases de Análisis Léxico y Sintáctico como dos fases diferenciadas, pero en el momento de la ejecución, el compilador puede realizar estas dos etapas de forma conjunta. En este caso el analizador sintáctico solicita nuevos tokens al analizador léxico a medida que los va necesitando.

- **Análisis semántico**: llegados a este punto interesa comprobar que además que lo que que se tenía inicialmente en la entrada esté bien escrito y estructurado, tenga también un sentido.

  Con el ejemplo planteado del lenguaje natural, es posible comprobar que la oración tiene un sentido pero, si no se realizarán en esta etapa algunas comprobaciones, podríamos encontrar oraciones como la siguiente:

Código 3: Ejemplo de oración a analizar

```
1    El casa de mi abuela vuela normalmente.
```

  Según las reglas gramaticales del idioma castellano, la oración anterior es correcta, ya que consta de las partes necesarias: sujeto, predicado y punto. Sin embargo, ¿qué pasa con su significado? Lógicamente, una casa no vuela y por lo tanto, esta oración realmente no tiene un significado correcto para los parlantes de la lengua. Pero además, esta oración presenta otro problema que es la concordancia de género entre el determinante y el nombre del sujeto.

  En el caso del lenguaje de programación CL antes mencionado: Si en lugar de **write(a+1)** tuviéramos la instrucción **write(a+b)**, donde b hubiera sido una variable declarada de tipo booleano, el compilador emitiría un error semántico. Éste se debe a que a es de tipo entero y

dado que CL no permite la coerción de tipo booleano a entero, no se puede realizar la operación de suma.

- **Generación de código intermedio o representación intermedia**: antes de todo, es importante entender la importancia de que los compiladores utilizan una representación intermedia del código fuente antes de traducir al lenguaje objeto final. Un compilador acaba siendo,un traductor de un lenguaje de alto nivel a uno de más bajo nivel. Así, para diferentes lenguajes de alto nivel y diferentes arquitecturas habrá que realizar diferentes traducciones y por tanto, implementar diferentes compiladores. Para facilitar el desarrollo de una familia de compiladores es habitual definir una representación intermedia del código y una estructura de compilador como la que se muestra a continuación.

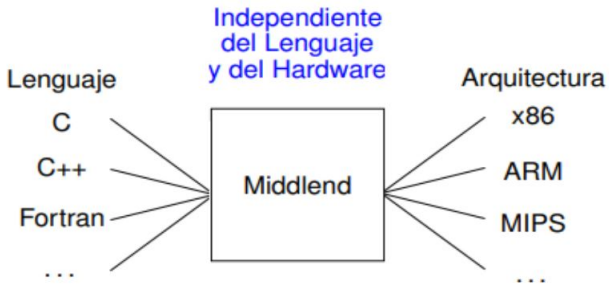

Figura 4: Generación de código intermedio

- **Optimización del código intermedio**: cualquier proceso de optimización lo que pretende es conseguir resultados más eficientes. En el caso de los compiladores, contamos con dos tipos de optimizaciones, las que son dependientes de la arquitectura y las que no. En esta fase, llevaremos a cabo las operaciones de optimización no dependientes de la arquitectura. El tiempo de respuesta de un compilador depende en gran parte del número y la complejidad de las optimizaciones que se realizan en esta fase. En general, se dedican a analizar la representación intermedia en búsqueda de instrucciones y/o cálculos redundantes, el uso de instrucciones que no implican ningún cambio en el resultado y que por tanto se pueden eliminar; y, en general, todo aquello que pueda hacer perder tiempo durante la ejecución del código sin afectar al significado del programa.

- **Generación código objeto**: en esta fase se realiza la traducción del código intermedio al código objeto concreto de la arquitectura en la que vamos a ejecutar el programa. A la hora de generar el código objeto es importante hacer un buen uso de los registros disponibles en la arquitectura, los modos de direccionamiento adecuados y traducir usando las instrucciones o conjuntos de instrucciones más apropiados.

2.3. Recorrido del árbol sintáctico

El recorrido del árbol AST es relativamente intuitivo:

- Se comienza con el nodo raíz como nodo principal a recorrer.
  - Si este nodo no tiene hijos, regresa a su padre.

  - De lo contrario, elige el primer nodo secundario que aún no ha visitado

- Ahora, este nodo es el principal y repite el ciclo. A esto se le llama búsqueda en profundidad DFS, porque baja antes de ir hacia los laterales. En enfoque alternativo sería una búsqueda en amplitud BFS, que es menos común.

El recorrido explicado anteriormente se muestra gráficamente en la figura 5. La flecha dibujada muestra el paso a paso de la explicación anterior. Cabe destacar que al final del recorrido, se vuelve al nodo padre, que en el caso de la figura es el que refiere al statement sequence. Es importante ver además, que un recorrido en profundidad permite una interpretación ligeramente más intuitiva que un recorrido en amplitud para la lectura que una persona realiza de forma apresurada, ya que los nodos padres preceden a los hijos.

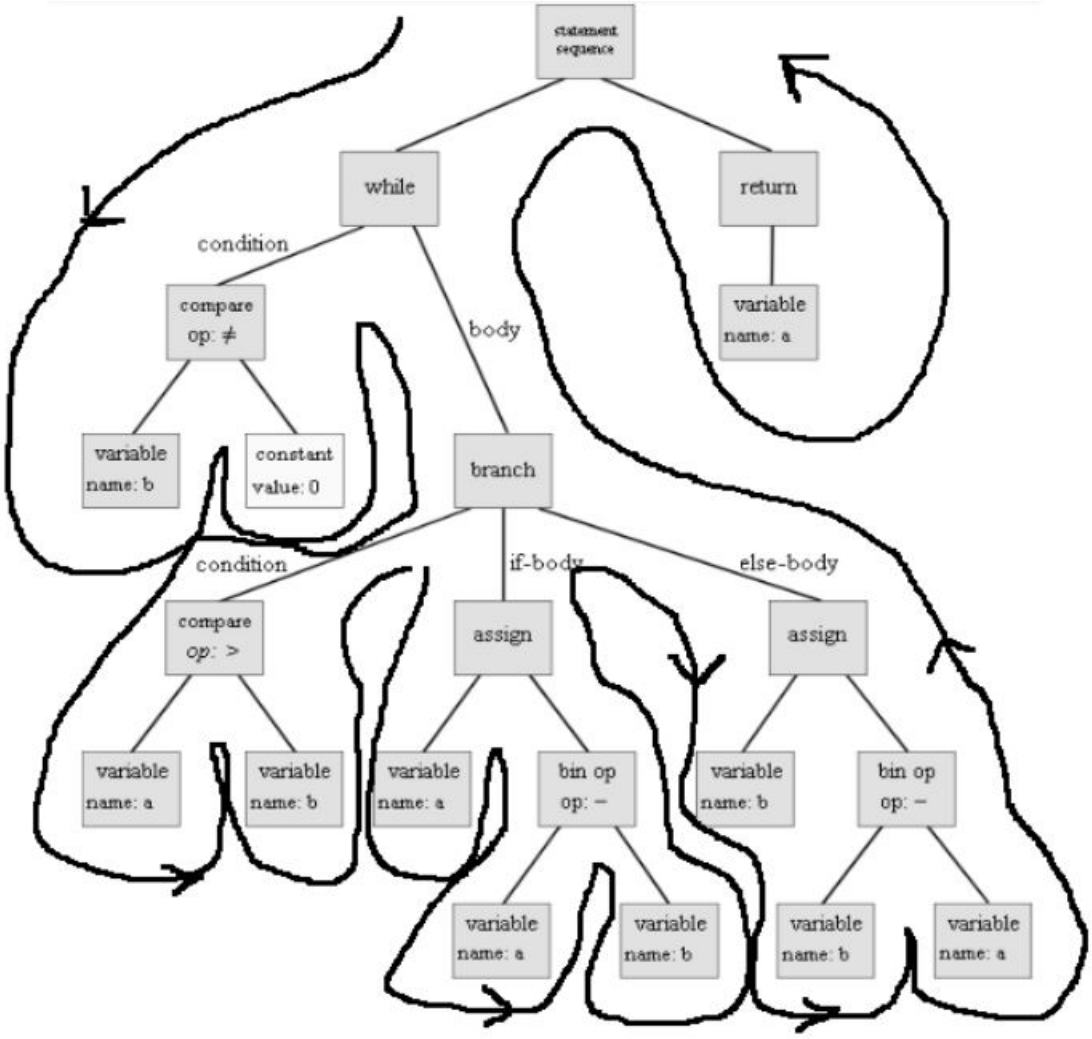

Figura 5: Recorrido del árbol

2.4. Listener y Visitors

El proceso de recorrer el árbol sintáctico anteriormente desarrollado, es fácil de entender y de aplicar a cualquier árbol, aunque no es aplicable por el usuario manualmente ya que hay mejores formas de hacerlo. Estas refieren a los "Listeners" (Oyentes) y los "Visitors (Visitantes)". ANTLR puede generar un visitante o un oyente.

Nota: Para este trabajo integrador en particular, se utilizará el Visitor para construir la parte semántica, generación y agregación de elementos en la tabla de signos. Y se utilizarán el Visitor y el Listener para generar el código intermedio.

2.4.1. Listeners

Son objetos de la clase Listeners, que se encargan de escuchar cuando suceda un evento determinado, cuando este ocurre se le avisa al Listener y este reacciona, lo que permite detectar evento del árbol.

2.4.2. Visitors

Son objetos de la clase Visitor, estos trabajan una vez que el árbol ya está construido y se encargan de iterar o visitar todos los nodos del árbol.

2.4.3. Comparación entre Listeners y Visitors

- Similitudes:
  - Ambos adoptan un algoritmo de profundidad primero.
  - Representan un aumento de la productividad que proporciona una forma fácil y estándar de procesar un árbol de análisis. Es importante porque al proporcionar una forma estándar de acceder al árbol, no tiene que idear su propia estrategia. Esto facilita la reutilización de la gramática con diferentes programas, incluso en diferentes idiomas.
  - Proporcionan formas estándar de ejecutar código cuando hay un determinado nodo en el árbol de análisis. Sin embargo, debe escribir el código que se ejecutará realmente.
  - Tanto un oyente como un visitante le permiten ejecutar fácilmente algún código cuando el árbol de análisis contiene un nodo, pero debe escribir el código que se ejecutará.
- Diferencias:
  - Con un oyente, no se puede controlar el proceso de atravesar el árbol. Sus funciones solo se llaman cuando el caminante del árbol se encuentra con un nodo del tipo correspondiente.
  - Con el visitante, se puede cambiar la forma en que se recorre el árbol; puede devolver valores de cada función de visitante e incluso detener al visitante juntos.
  - Desea utilizar el oyente cuando no necesita cambiar la estructura del árbol de análisis sintáctico para lograr sus objetivos. Esto significa que desea usarlo cuando su salida se puede generar sin alterar la estructura del árbol de análisis.
  - En su lugar, desea utilizar un visitante cuando necesita devolver resultados de cada nodo o necesita recopilar información de varios nodos.

2.5. Tabla de Símbolos

La Tabla de Símbolos almacena toda la información relativa a los símbolos y los ámbitos donde han sido declarados. Recibe entradas a medida que nos encontramos con declaraciones de símbolos y, posteriormente, se realizan consultas sobre ella para realizar las comprobaciones en el uso de esos símbolos.

Antes de realizar la inserción de un nuevo símbolo en la tabla, se deberá comprobar que no haya sido introducido previamente en el mismo ámbito y asegurar que su tipo sea correcto (para el caso de los nombres de tipo hay que verificar que existe como tal). Dado que una de la información más relevante para la parte que estamos tratando es el ámbito al que pertenece un símbolo (identificador), la tabla de símbolos organizará su información de manera que tendremos una tabla para cada ámbito del programa. Por tanto, la Tabla de Símbolos realmente consiste en una pila de ámbitos. Dentro de cada ámbito estarán declarados sus identificadores.

En resumen la tabla de símbolos debe contener:

- Registro de los diferentes identificadores encontrados.
- Poder decir:
  - Si una variable se usó o no.
  - Si una variable se inicializa o no.
  - De qué tipo de dato es una variable.

Hay muchas formas de modelar una tabla de símbolos, la cual varía la estructura de datos elegida, hay tablas de símbolos que van desde simples arreglos, y otras que ya son estructuras de datos dinámicas.

2.6. Recuperación de errores

La recuperación de errores no es una fase más del proceso de compilación. En lugar de eso, debería considerarse como una característica deseable del análisis léxico y semántico.

La recuperación de errores permite al compilador detectar un error, dar parte de él y seguir reconociendo la entrada, detectando de esta manera más errores. Si el programador puede ver más errores en cada compilación, podrá corregir más errores en cada compilación, por lo que será más eficiente.

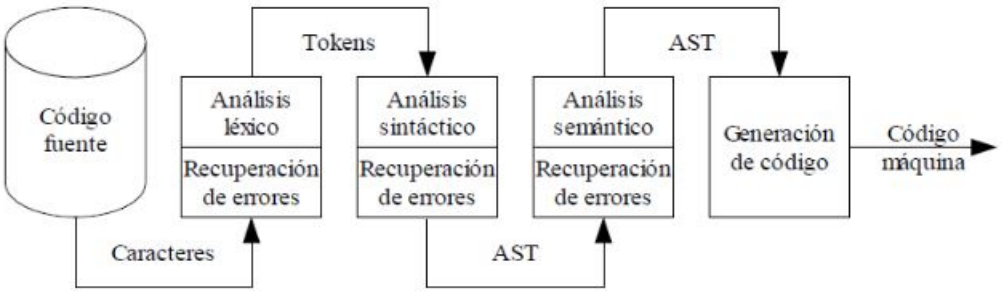

Figura 6: Recuperación de errores

2.6.1. Fases

Las tres fases básicas de la gestión de errores son las siguientes:

1. Detección. El error se localiza.
2. Informe. Se archiva o muestra un mensaje informativo explicando la naturaleza del error.
3. Recuperación. El analizador utiliza alguna téctica para "superar.[el] error y poder seguir efectuando el análisis.

2.7. Código Intermedio

El código intermedio es un paso intermedio entre el AST enriquecido y el código generado. Presenta varias ventajas:

- Es suficientemente abstracto para ser independiente de la máquina destino, pero suficientemente concreto para ser sometido a ciertas optimizaciones independientes de la máquina.
- Puede distribuirse en lugar del código objetivo, y convertirse en código máquina en la propia máquina objetivo, aplicándose las optmizaciones dependientes de la máquina.
- Facilita la reusabilidad.

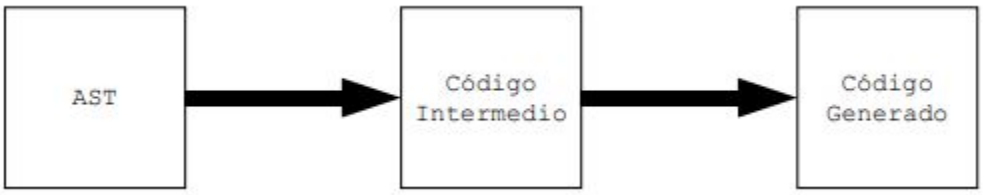

Figura 7: Lugar del código intermedio

2.7.1. Código de Tres Direcciones

3AC o TAC por sus siglas en inglés, el código de tres direcciones es un lenguaje intermedio que utilizan los compiladores optimizadores para ayudar en las transformaciones de mejora de código. Cada instrucción TAC tiene a lo sumo tres operandos, y normalmente consta de una combinación de asignación y operador binario. Por ejemplo **t1 := t2 + t3**. Las **t** hacen referencias a variables temporales.

Este código se usa como lenguaje intermedio en los compiladores, los operandos no contienen direcciones de memoria concretas, sino simbólicas que son luego convertidas en direcciones reales una vez asignados los registros.

Para el caso de una instrucción del tipo operación, al escribirse en C++ lo siguiente:

Código 4: Operación aritmética en C

```
1 y = x * 2 + z / 5;
```

Se genera la siguiente sucesión de instrucciones en código de tres direcciones:

Código 5: Código intermedio TAC para la entrada anterior

```
t0 = x * 2
t1 = z / 5
t2 = t0 + t1
y = t2
```

En el caso de un condicional If, se tiene lo siguiente:

Código 6: Condicional If en C

```
if (x >= 0)
    y = x;
else
    y = -x;
```

Se genera la siguiente sucesión de instrucciones en código de tres direcciones:

Código 7: Código intermedio TAC para la entrada anterior

```
t0 = x >= 0
if jmp t0, l0
y = x
jmp l1
label l0
y = -x
label l1
```

Para un bucle For:

Código 8: Bucle For en C

```
for (i = 0; i < 0; ++i)
    y += i;
```

Se genera la siguiente sucesión de instrucciones en código de tres direcciones:

Código 9: Código intermedio TAC para la entrada anterior

```
i = 0
label l0
t0 = i < 0
if jmp t0, l1
t1 = y + i
y = t1
i = i + 1
jmp l0
label l1
```

Finalmente, para un While se tiene:

Código 10: Bucle While en C

```
while (i < n)
   a = b + 1;
   i++;
```

Se genera la siguiente sucesión de instrucciones en código de tres direcciones:

Código 11: Código intermedio TAC para la entrada anterior

```
label l0
t0 = i < n
if jmp t0, l1
t1 = b + 1
a = t1
i = i + 1
jmp l0
label l1
```

3. Desarrollo

3.1. Implementación

Para el desarrollo del compilador, se siguió la hoja de ruta planteada durante la cursada. Es decir, de forma resumida, se planteó el siguiente proceso:

1. Declaración de las reglas léxicas y gramaticales.
2. Declaración de tokens a utilizar.
3. Configuración de listener y visitor.
4. Generador de la tabla de símbolos.
5. Generador de código de tres direcciones (TAC).

Una vez obtenido el resultado del anterior proceso, para un código fuente dado, el software desarrollado por el grupo, realiza lo siguiente:

1. Análisis sintáctico y léxico del código fuente.
2. Generación del arbol AST.
3. Generación de un reporte de errores.
4. Generación de la tabla de símbolos.
5. Generación de código de tres direcciones.
6. Generación de tabla de símbolos en formato .png.

A continuación, se explican los puntos anteriormente mencionados.

3.1.1. Reglas léxicas

Las reglas léxicas se declararon en el archivo **compilador.g4**, que es un archivo de ANTLR4. Para ello se utilizaron expresiones regulares. Las reglas léxicas son iterativas a diferencia de las recursivas (gramaticales). Se adjuntan en la figura a continuación los diagramas de vías de tren de las reglas.

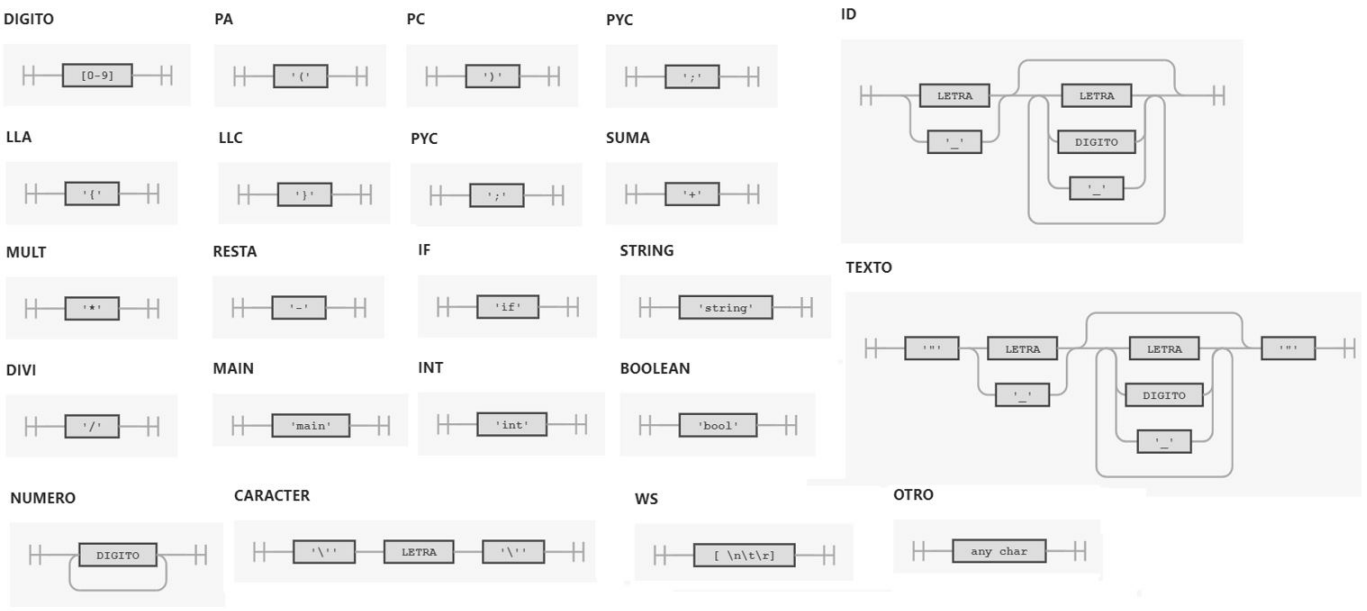

Figura 8: Reglas léxicas

3.1.2. Reglas gramaticales

Las reglas gramaticales, al igual que las léxicas son declaradas en el archivo **compilador.g4**. Estas reglas son recursivas, eso puede apreciarse en los diagramas de vías de tren que se adjuntan en las siguientes figuras.

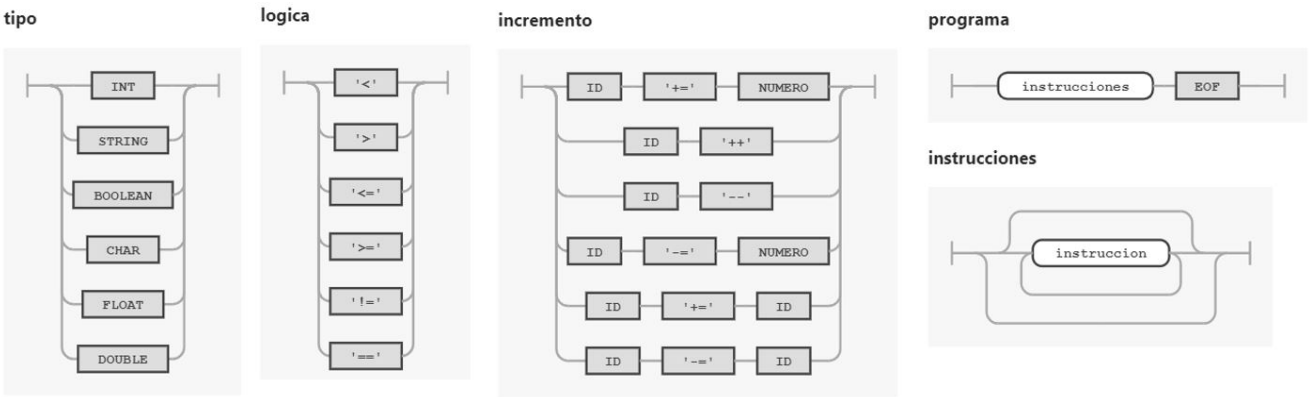

Figura 9: Reglas gramaticales (primera parte)

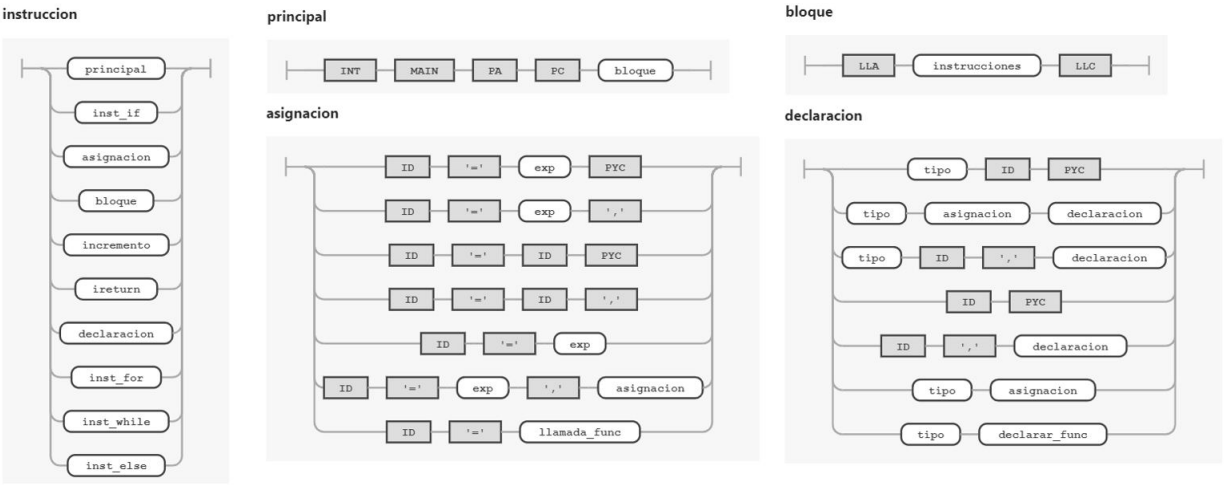

Figura 10: Reglas gramaticales (segunda parte)

Las reglas gramaticales declaradas son:

- **tipo**: refiere al tipo de variable a utilizar en el programa fuente.
- **lógica**: refiere a los diferentes operadores lógicos.
- **incremento**: son las diferentes maneras de incrementar una variable.
- **programa**: es la regla madre. (Básicamente, un programa es una secuencia automatizada de instrucciones previamente definidas.)
- **instrucciones**: concatenación (i.e. **Recursión**) de instrucciones individuales (**instrucción**).
- **instrucción**: sentencias reservadas del lenguaje C.
- **principal**: instrucción principal del programa, (i.e. **Main**).
- **bloque**: delimitación de un contexto.
- **asignación**: definir el valor de un dato en la dirección de memoria reservada para la variable.
- **declaración**: reserva de memoria para una variable.
- **inst_if**: sentencia reservada para el condicional (i.e. **If**).
- **inst_else**: parte de la instancia **inst_if**.
- **declarar_func**: declaración de un **ID** de una función de usuario.
- **inst_while**: sentencia reservada para un bucle (i.e. **While**).
- **ireturn**: sentencia reservada para la devolución del dato de una función de usuario.
- **llamada_func**: ejecución de una función de usuario.
- **exp**: expresión, sucesión de términos (i.e. **term**).

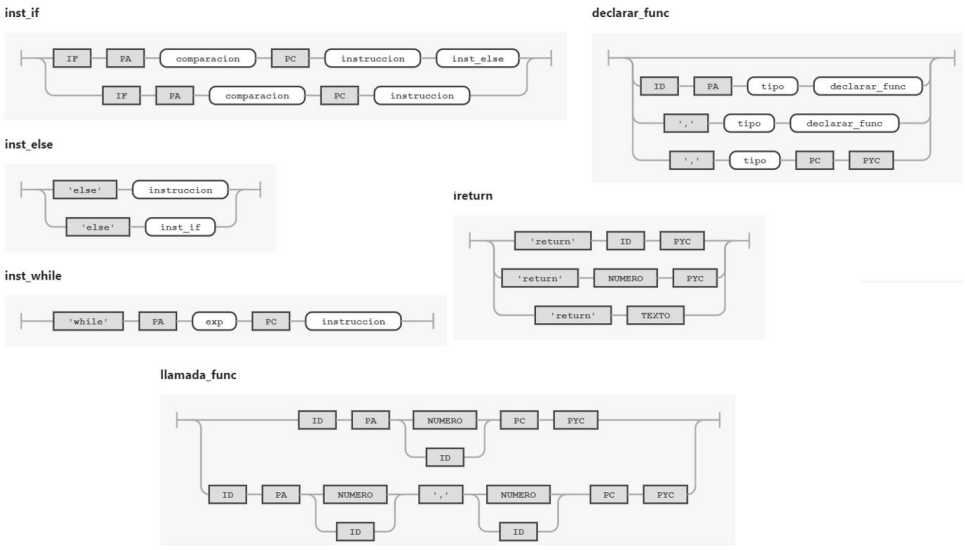

Figura 11: Reglas gramaticales (tercera parte)

- **opar**: sucesión de expresiones.
- **factor**: desde un número hasta una expresión compleja. Puede ser una expresión dentro de un paréntesis, por ejemplo.
- **comparación**: comparación entre los valores de dos variables.
- **term**: sucesión de factores (i.e. **factor**).
- **f**: expresión de un producto (multiplicación o división).
- **t**: expresión de una adición (suma o resta).

3.1.3. Generación del árbol AST

El árbol gramatical puede tener dos morfologías (de acuerdo a su construcción):

1. Ascendente: construcción desde los nodos hojas hacia la raíz del árbol, en diseño de compiladores, este tipo de morfología es la más óptima, pero más dificil de implementar.
2. Descendente: construcción desde la raíz hacia las hojas (así lo genera ANTLR).

C es un Lenguaje de una sola pasada, lo que obliga a tener todo definido a priori.
Véase sección de **Ejemplos**, para apreciar la construcción de un árbol.

3.1.4. Configuración de listener y visitor

Como el lenguaje para el cual se diseñó el compilador es de una sola pasada, con el listener sería suficiente para el diseño. Sin embargo, se utiliza un visitor de manera de abarcar más conceptos de la materia.

Tanto los listener como los visitor son interfaces, lo que implica que no son utilizables a priori, se les deben sobreescribir los métodos. El lenguaje ANTLR genera una versión de archivo básica para los listener y visitor llamada **compiladorBaseListener.java** para evitar escribir código redundante.

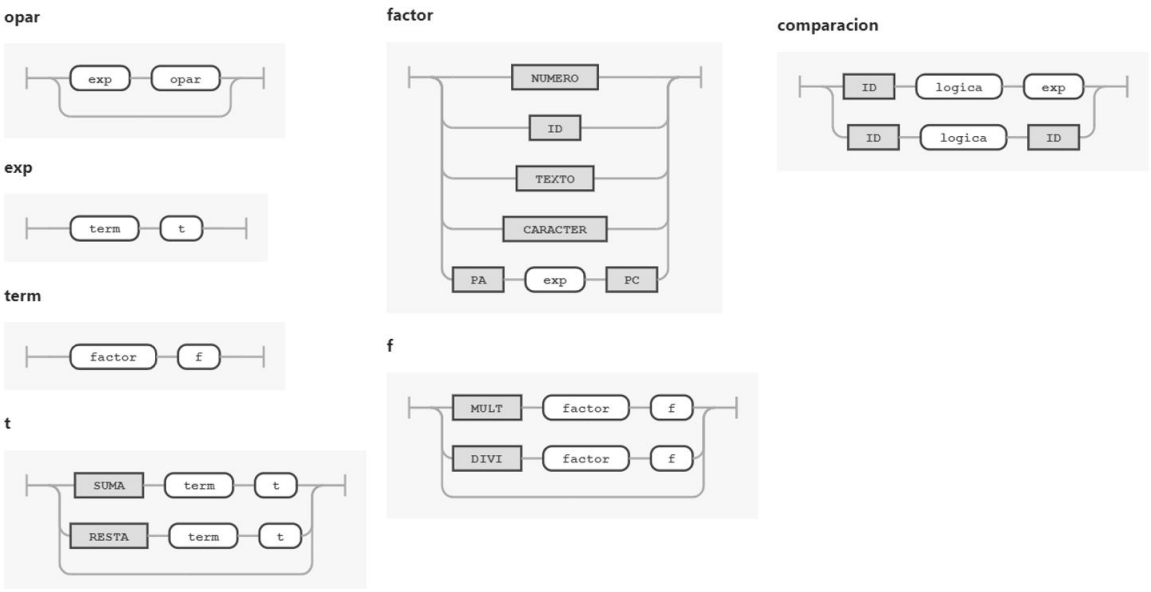

Figura 12: Reglas gramaticales (cuarta parte)

Con el visitor se puede recorrer el árbol todas las veces que sea necesario (post mortem). Lo que implica que el funcionamiento de este comienza una vez creado el árbol. El visitor visita cada uno de los nodos del AST, si se puede usar es porque ya está creado el árbol y todo funciona (siempre y cuando no se trabaje con un árbol roto, o sea, no deben faltarle ramas).

El listener en paralelo a la creación del AST entra y sale de cada una de las reglas gramaticales esperando sucesos para así reaccionar.

3.1.5. Generación de la tabla de símbolos

La complejidad de la tabla de símbolos es totalmente variable, es posible modificarla a gusto y necesidad del usuario. Ésta es una estructura de datos implementable de muchas maneras, por ejemplo, una lista o un árbol. Normalmente se trata de un diccionario del tipo Hash, al que es posible acceder con un token a un valor. La implementación realizada arroja una tabla como la siguiente.

	String	Nombre	TipoDato	Inicializado	Expresion	Valor	Usado
0	ID	Nombre	Tipo (int,string,bool)	Estado de la inicialización	Expresión asignada	Valor asignado	Utilización o no

Figura 13: Salida en .png de la tabla de símbolos

La implementación realizada se ve perfectamente descripta con el diagrama UML a continuación, en donde se aprecian cada uno de los métodos implementados.

Los métodos de la clase **TablaSimbolos** son los siguientes:

- **addContenxto()**: añade un contexto a la tabla.
- **delContexto()**: elimina un contexto.

- **addSimbolo(Id id)**: añade una variable a la tabla.
- **addContenxto(Id id)**: busca un símbolo en la tabla.
- **buscarSimboloLocal(Id id)**: busca un símbolo en el contexto local.
- **ReemplazarVariableMAPA(HashMap<String, String> copia)**: reemplaza las expresiones de las variables por su valor.
- **CopiarMapa(HashMap<String, String> copia)**: copia y ordena un mapa dejándolo listo para su posterior uso.
- **ImprimirTS()**: imprime la tabla de símbolos.

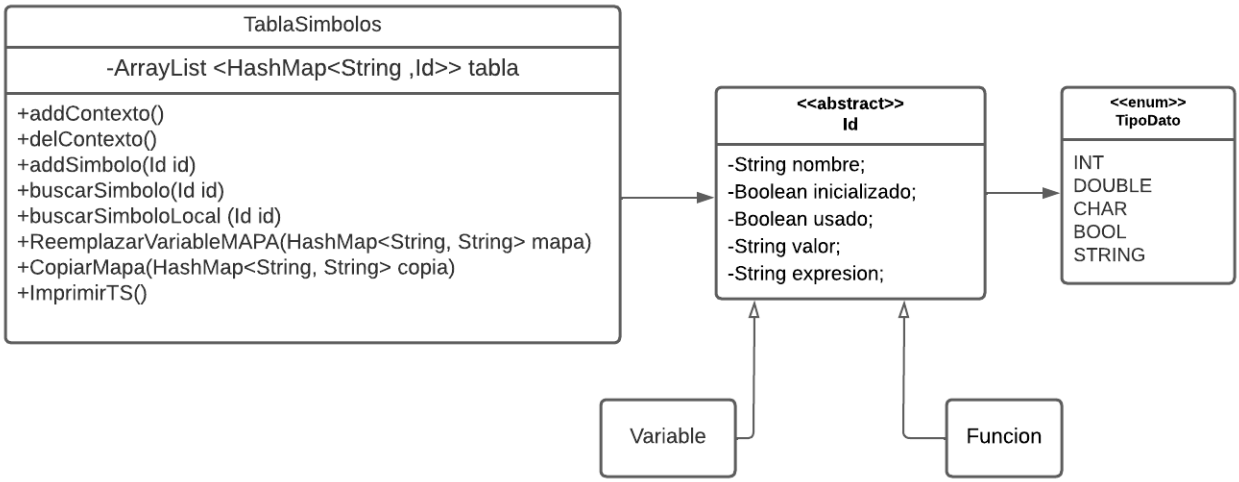

Figura 14: Diagrama UML de la tabla de símbolos

3.1.6. Generación de código intermedio

El código intermedio implementado es el de tres dimensiones (TAC), que es adimensional, es decir, que es inherente al lenguaje de programación compilado. Además, como se dijo anteriormente, es independiente de la arquitectura de computadora sobre la que correrá el programa.

Este código se genera gracias al árbol sintáctico. El TAC se generó parseando strings generados por el recorrido del visitor en donde se analizan cada una de las operaciones e instrucciones que se deban ejecutar, imprimiendo en consola el código necesario.

3.1.7. Generación de la tabla de símbolos en png

Para exportar la tabla de símbolos se utilizó un script de **Python** que genera un dataframe de Pandas, que luego es exportado en formato imagen mediante una librería que se encarga de convertir el dataFrame en un archivo con extensión **.png**.

3.2. Herramientas utilizadas

Para el desarrollo del proyecto fue necesario el uso de diferentes herramientas, pudiéndose enumerar de la siguiente forma:

- Herramientas de software:
  - Visual Studio Code: se añadieron las siguientes extensiones:
    - Better Comments.
    - ANTLR4 grammar syntax support.
    - Extension Pack for Java.
  - LucidChart
  - Overleaf
- Lenguajes de programación:
  - C.
  - Java.
  - Python: se utilizaron los paquetes:
    - Pandas
    - dataframe_image
  - ANTLR4.

3.3. Restricciones / limitaciones

Al momento del desarrollo del software surgieron ciertos inconvenientes, que, junto con el poco tiempo del que se estipuló para la realización el trabajo, derivaron en la aparición de ciertas restricciones al momento de escribir el código fuente en C. Ellas son:

1. No se pueden repetir instrucciones del tipo:
   - If / If else.
   - For.
   - While.
2. El incremento en un bucle While es la última operación dentro del contexto local.
3. Las asignaciones a variables enteras soportan únicamente números de una cifra.
4. Las instrucciones que no deban ir dentro del contexto local de alguno de los bloques nombrados en el item 1 deben ser escritas encima de éstos.
5. Al momento de escribir un incremento, se debe colocar **i = i + 1** en vez de **i++**.

3.4. Ejemplos de ejecución

3.4.1. Generación de código intermedio

Código 12: Ejemplo de código en C para generar código intermedio

```
int main() {
   int y;
   if (x < 4) {
      y = 1 - 5/7+62;
   }
   else {
      y = 25-9;
   }
}
```

Código 13: Código intermedio resultado

```
t1 = 5/7
t2 = 1-t1
t3 = t2+6
y = t3
t0 = x<4
if jmp t0, l0
t1 = 5/7
t2 = 1-t1
t3 = t2+6
y = t3
jmp l1
label l0
t1 = 5-9
y = t1
label l1
```

3.4.2. Generación del arbol AST

Código 14: Ejemplo de código en C para generar un árbol AST

```
int main() {
   int y;
   int x = 2;
   string dios = "Messi";
}
```

3.4.3. Generación de la tabla de símbolos

Código 15: Ejemplo de código en C para generar una tabla de símbolos

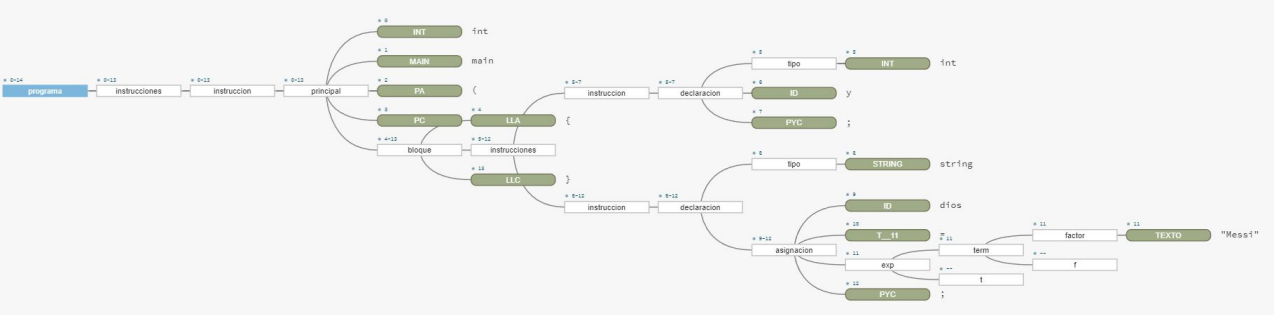

Figura 15: Arbol AST generado para el código 14

```
int main() {
    int y;
    int x = 10;
    int o, p, r, s;
    int a = 1, b = 2, c = 3;
    int w, t = 3, j = 0, i;
    w = a * b + c / 2;
}
```

	String	Nombre	TipoDato	Inicializado	Expresion	Valor	Usado
0	a	a	INT	True	Null	1	True
1	b	b	INT	True	Null	2	True
2	c	c	INT	True	Null	3	True
3	i	i	INT	False	Null	Null	False
4	j	j	INT	True	Null	0	False
5	o	o	INT	False	Null	Null	False
6	p	p	INT	False	Null	Null	False
7	r	r	INT	False	Null	Null	False
8	s	s	INT	False	Null	Null	False
9	t	t	INT	True	Null	3	False
10	w	w	INT	True	a*b+c/2	3	False
11	x	x	INT	True	Null	10	False
12	y	y	INT	False	Null	Null	False

Figura 16: Tabla de símbolos para el código 15

3.5. Conclusión

Luego de completar el desarrollo del proyecto final de la materia de práctica y construcción de compiladores, se puede realizar una conclusión a modo general del trabajo práctico.

En primer instancia, destacar que se pudieron cumplir los objetivos iniciales, que constaban de la posibilidad de poder ejecutar correctamente las sentencias escritas en un código fuente de C. Sin embargo, durante el correr de los días surgieron ciertos inconvenientes que pudieron ser resueltos, y algunos de ellos derivaron en ciertas limitaciones al momento de escribir código debido a que no se llegó a completar como se hubiese querido en el tiempo estipulado.

Luego, se puede decir que el desarrollar un compilador, en el lenguaje que sea, en este caso C, permitió comprender finalmente ciertos conceptos que hasta el momento no se habían terminado de afianzar, como lo era por ejemplo, la utilización de un árbol como estructura de datos, y su recorrido. Además, con la construcción del software fue posible entender ciertos errores que surgen al momento de uno mismo escribir código en un compilador o IDE de terceros.

Finalmente, y a modo de cierre, se plantean ciertas posibilidades para futuras versiones del software desarrollado, estas son:

- Eliminación de las limitaciones de la primera versión.
- Soporte para impresiones por consola.
- Soporte de comentarios.
- Soporte de implementación de funciones de usuario.
- Soporte de inclusión de librerías estándar de C.
- Implementación de código correspondiente a una cierta arquitectura de computadora.

Referencias

Aho, A. V. (2008). *COMPILADORES PRINCIPIOS TECNICAS Y HERRAMIENTAS.*

Cota, E. J. G. (2003). *Guía práctica de ANTLR 2.7.2.* http://www.lsi.us.es/~troyano/documentos/guia.pdf

Excel2Latex. (2017). *Plugin para hacer tablas con Excel.* https://www.ctan.org/tex-archive/support/excel2latex/

Rodríguez, L. (2010). *Generación de Código Intermedio Usando Semántica Funtorial.*

Anexo A. Representación de un árbol en ANTLR

ANTLR representa un árbol utilizando una notación basada en LISP. A modo de ejemplo, partiendo del siguiente código en openCL, se obtiene el árbol de sintaxis abstracta para el siguiente código del algoritmo de Euclides.

Código A.1: Ejemplo de lenguaje openCL

```
1  while b != 0
2  if a > b
3  a := a - b
4  else
5  b := b - a
6  return a
```

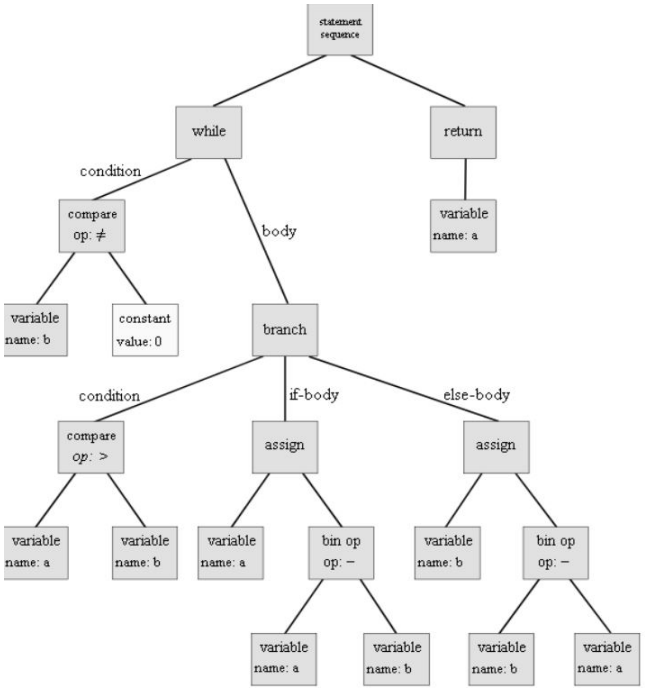

Figura A.1: Árbol generado para el ejemplo de código anterior