

UNIVERSIDAD AUTÓNOMA DE CIUDAD JUÁREZ

Instituto de Ingeniería y Tecnología

Departamento de Ingeniería Eléctrica y Computación



# ANÁLISIS AUTOMATIZADO DE LA COMPLEJIDAD TEMPORAL DE ALGORITMOS IMPLEMENTADOS EN C++

Reporte Técnico de Investigación presentado por:

Mauricio Alexis Tapia Alanis 176787

Requisito para la obtención del título de:

INGENIERO EN SISTEMAS COMPUTACIONALES

ASESOR:

M. en C. José Saúl González Campos.

Ciudad Juárez, Chihuahua, a 13 de Mayo de 2024


Asunto: Liberación de Asesoría

**Mtro. Ismael Canales Valdiviezo**  
**Jefe del Departamento de Ingeniería**  
**Eléctrica y Computación**  
**Presente.-**

Por medio de la presente me permito comunicarle que, después de haber realizado las asesorías correspondientes al reporte técnico Análisis automatizado de la complejidad temporal de algoritmos implementados en C++, del alumno Mauricio Alexis Tapia Alanis. de la Licenciatura en Ingeniería en Sistemas Computacionales, considero que lo ha concluido satisfactoriamente, por lo que puede continuar con los trámites de titulación intracurricular.

Sin más por el momento, reciba un cordial saludo.

Atentamente:

  
M. en C. José Saúl González Campos.  
Profesor Investigador

Ccp: Mtro. David Absalón Uruchurtu Moreno  
Coordinador del Programa de Sistemas Computacionales  
Mauricio Alexis Tapia Alanis.  
Archivo

Ciudad Juárez, Chihuahua, a 22 de Mayo de 2024

Asunto: Autorización de publicación

**C. Mauricio Alexis Tapia Alanis**

**Presente.-**

En virtud de que cumple satisfactoriamente los requisitos solicitados, informo a usted que se autoriza la publicación del documento de ANÁLISIS AUTOMATIZADO DE LA COMPLEJIDAD TEMPORAL DE ALGORITMOS IMPLEMENTADOS EN C++, para presentar los resultados del proyecto de titulación con el propósito de obtener el título de Licenciado en Ingeniería en Sistemas Computacionales.

Sin otro particular, reciba un cordial saludo.

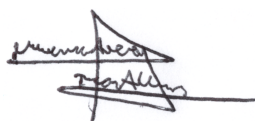


Dr. Everardo Santiago Ramírez

Profesor Titular de Seminario de Titulación II

## Declaración de Originalidad

Yo, Mauricio Alexis Tapia Alanis declaro que el material contenido en esta publicación fue elaborado con la revisión de los documentos que se mencionan en el capítulo de Bibliografía, y que la solución obtenida es original y no ha sido copiada de ninguna otra fuente, ni ha sido usada para obtener otro título o reconocimiento en otra institución de educación superior.

A handwritten signature in dark ink, appearing to read 'Mauricio Alexis Tapia Alanis', with a stylized, overlapping structure.

Mauricio Alexis Tapia Alanis

# Agradecimientos

Agradezco a mi asesor, el M. en C. José Saúl González Campos, por el diligente asesoramiento otorgado a lo largo del desarrollo del presente reporte técnico de investigación y del producto propuesto en él. A los doctores Jesús Ochoa Domínguez y Everardo Santiago Ramírez por la orientación otorgada en las clases de seminario de titulación, y al resto de los profesores del departamento de sistemas computacionales que aportaron a mi formación como profesionista.

# Dedicatoria

Dedico este logro a mis padres, Alicia y Alejandro, sin cuyo apoyo incondicional esto no hubiese sido posible. Al Sr. Jesús Soriano y a la Sra. Leticia Andrade, que no solo otorgaron a mis padres y por extensión al resto de mi familia una oportunidad invaluable, sino que también nos han tratado como familia desde el principio. A mis amigos que conocí durante mi proceso de formación profesional, especialmente a los miembros y coordinadores de los clubes de Algoritmia fundados en la ciudad, gracias por compartir mis intereses y ser una fuente continua de inspiración.

# Resumen

En el presente reporte técnico se cubre el desarrollo de un prototipo para el análisis automatizado de la complejidad temporal de algoritmos implementados en C++. Utilizando una metodología de desarrollo de software apropiada para este proyecto, fueron desarrollados módulos de análisis léxico y sintáctico con el fin de obtener una representación del código analizado que sea más fácil de manipular algorítmicamente. También fue desarrollado un módulo para la detección de dependencias circulares entre funciones a partir de algoritmos para el conteo de Componentes Fuertemente Conectados. Posteriormente, fue desarrollado un módulo para el análisis de complejidad, el cual en función de la estructura del código y a partir de una serie de reglas que tratan de generalizar este proceso, obtiene una complejidad temporal estimada.

Así mismo son exploradas las limitantes del prototipo desarrollado a través de pruebas funcionales, donde fueron comparados los resultados de un análisis de complejidad realizado por un humano con los obtenidos por el prototipo. Demostrando de manera empírica la existencia de limitantes en cuanto al análisis de complejidad automatizado.

**Palabras claves:** Análisis automatizado de la complejidad temporal, análisis léxico, análisis sintáctico

# Índice general

<b>1. Planteamiento del Problema</b>	<b>1</b>
1.1. Antecedentes . . . . .	1
1.2. Definición del problema . . . . .	4
1.3. Objetivo general . . . . .	4
1.3.1. Objetivos específicos . . . . .	5
1.4. Justificación . . . . .	5
1.5. Alcances y limitaciones . . . . .	6
<b>2. Marco Teórico</b>	<b>7</b>
2.1. Análisis de algoritmos. . . . .	7
2.1.1. Sobre el modelo RAM . . . . .	8
2.1.2. Sobre el análisis de complejidad temporal y la notación <i>big Oh</i> . . . . .	8
2.2. Lenguajes formales . . . . .	10
2.2.1. Expresiones regulares . . . . .	10
2.2.2. Gramáticas libres de contexto . . . . .	11
2.3. Compiladores . . . . .	12



2.3.1. Sobre el análisis léxico . . . . .	12
2.3.2. Sobre el análisis sintáctico . . . . .	13
<b>3. Desarrollo del Proyecto</b>	<b>15</b>
3.1. Producto propuesto . . . . .	15
3.2. Descripción de la metodología . . . . .	17
3.3. Análisis y Definición de Requisitos . . . . .	18
3.3.1. Análisis . . . . .	18
3.3.2. Requisitos del sistema . . . . .	22
3.4. Diseño . . . . .	22
3.4.1. Diseño del módulo de análisis léxico . . . . .	23
3.4.2. Diseño del módulo de Análisis sintáctico . . . . .	26
3.4.3. Diseño del módulo de análisis de jerarquía y dependencia . . . . .	42
3.4.4. Diseño del módulo de análisis de complejidad . . . . .	53
3.5. Implementación . . . . .	61
3.5.1. Implementación del módulo de análisis léxico . . . . .	61
3.5.2. Implementación del módulo de análisis sintáctico . . . . .	68
3.5.3. Implementación del módulo de análisis de jerarquía y dependencia . . . . .	98
3.5.4. Implementación del módulo de análisis de complejidad . . . . .	106
3.5.5. Implementación del módulo coordinador y proceso de compilado . . . . .	141
3.6. Validación . . . . .	143
<b>4. Resultados y discusiones</b>	<b>148</b>

<i>ÍNDICE GENERAL</i>	x
4.1. Resultados . . . . .	148
4.1.1. Recolección y etiquetado de soluciones . . . . .	148
4.2. Discusiones . . . . .	152
<b>5. Conclusiones</b>	<b>158</b>
5.1. Con respecto al objetivo general . . . . .	158
5.2. Recomendaciones para trabajo a futuro . . . . .	159
<b>Bibliografía</b>	<b>160</b>

# Índice de figuras

2.1. Gráfica de las notaciones $\Theta, O$ y $\Omega$ . . . . .	9
3.1. Arquitectura del prototipo. . . . .	16
3.2. Metodología Cascada usada para el desarrollo del prototipo. . . . .	18
3.3. Representación simplificada del AST correspondiente al código mostrado en el Listado 3.18. . . . .	44
3.4. Representación simplificada del AST correspondiente al código mostrado en el Listado 3.21. . . . .	48
3.5. Componentes fuertemente conectados en el AST simplificado correspondiente al código mostrado en el Listado 3.21. . . . .	49
3.6. Grafo de llamadas correspondiente al código plasmado en el Listado 3.68 . .	100
3.7. Grafo de llamadas descomprimido correspondiente al código plasmado en el Listado 3.68. . . . .	101

# Índice de tablas

3.1. Métodos, contenedores y tipos del estándar de C++ aceptados por el prototipo.	21
3.2. Complejidades asociadas a los métodos definidos por el lenguaje, donde $n$ es la cardinalidad del elemento pasado como argumento o en su defecto la del contenedor. . . . .	56
4.1. Comparación de la complejidad real con la complejidad obtenida por el prototipo sobre los códigos que conforman las pruebas principales. . . . .	150
4.2. Comparación de la complejidad real con la complejidad obtenida por el prototipo sobre los códigos que conforman las pruebas complementarias. . . . .	151

# Índice de códigos fuente

3.1. Definición recursiva de la sucesión de Fibonacci. . . . .	19
3.2. Declaración de una función recursiva de cola. . . . .	19
3.3. Código usado para el diseño de la gramática . . . . .	27
3.4. Gramática con reglas de producción para cada operador . . . . .	29
3.5. Gramática con reglas de producción que agrupan terminales representando operadores . . . . .	29
3.6. Reglas agrupadoras de operadores aritmeticos, operadores de asignación y de constantes. . . . .	30
3.7. Reglas de producción correspondientes a la forma mas básica que puede tomar una expresión aritmética. . . . .	31
3.8. Reglas de producción relacionadas con expresiones aritméticas formadas por llamadas a funciones. . . . .	31
3.9. Regla de producción relacionada con expresiones aritméticas formadas por accesos a posiciones en arreglos. . . . .	32
3.10. Reglas de producción correspondientes a expresiones aritméticas formadas por otras expresiones aritméticas. . . . .	32
3.11. Reglas agrupadoras de operadores booleanos y de definición de expresiones booleanas compuestas por dos expresiones. . . . .	34
3.12. Reglas de producción correspondientes a los métodos de entrada y de salida.	36
3.13. Reglas de producción correspondientes a las declaraciones con y sin asignación.	36

3.14. Reglas de producción correspondientes a las estructuras iterativas y de control de flujo. . . . .	38
3.15. Reglas de producción correspondientes a los bloques de código. . . . .	40
3.16. Reglas de producción relacionadas a la declaración de funciones. . . . .	41
3.17. Reglas de producción correspondientes a la definición del programa. . . . .	41
3.18. Ejemplo de un código con dependencias entre funciones. . . . .	43
3.19. Algoritmo DFS para encontrar un ordenamiento topológico. . . . .	44
3.20. Ejemplo de un código con dependencias circulares entre dos funciones. . . . .	46
3.21. Ejemplo de un código con dependencias circulares entre tres funciones. . . . .	46
3.22. Algoritmo de Kosaraju-Sharir para encontrar componentes fuertemente conectados. . . . .	49
3.23. Algoritmo de Tarjan para encontrar componentes fuertemente conectados. . . . .	51
3.24. Estructura de un programa de Lex. . . . .	61
3.25. Función <code>main</code> implementada por defecto por Lex. . . . .	62
3.26. Declaración de la clase <code>cl_atributos_tokens</code> . . . . .	62
3.27. Código de C++ incluido en la sección de definiciones. . . . .	63
3.28. Algunas definiciones de patrones monomorfos. . . . .	64
3.29. Sintaxis de las reglas en Lex. . . . .	64
3.30. Ejemplo de la implementación de una regla correspondiente a un <i>token</i> monomorfo. . . . .	65
3.31. Patrones para la identificación y eliminación de comentarios, espacios y saltos de línea. . . . .	66
3.32. Patrones para la identificación de identificadores y constantes. . . . .	66
3.33. Estructura de un programa de Bison. . . . .	68
3.34. Prologo usado en el analizador escrito con Bison. . . . .	69
3.35. Declaración de los tokens y sus atributos. . . . .	70
3.36. Sintaxis de las reglas de producción en Bison. . . . .	71

3.37. Declaración de los terminales asociados con el símbolo <code>expresionAritmetica</code> .	72
3.38. Implementación de las reglas de producción correspondientes a los símbolos agrupadores <code>operadorAritmetico</code> y <code>operadorAsignacion</code> .	73
3.39. Declaración de la clase <code>cl_argumentos_llamada</code> .	73
3.40. Implementación de la regla de producción correspondiente al símbolo argu- mentosLLamada.	74
3.41. Declaración de las clases <code>cl_llamada_funcion</code> y <code>cl_expresion_aritmetica</code> .	74
3.42. Implementación de las reglas de producción correspondientes a las expresiones aritméticas formadas por una constante.	75
3.43. Implementación de las reglas de producción correspondientes a las expresiones aritméticas formadas por accesos a un arreglo o llamadas a funciones.	77
3.44. Implementación de las reglas de producción correspondientes a las expresiones aritméticas formadas por otras expresiones aritméticas.	78
3.45. Declaración de los terminales asociados con el símbolo <code>expresionBooleana</code> .	79
3.46. Declaración de la clase <code>cl_expresion_booleana</code> .	80
3.47. Implementación de las reglas de producción correspondientes al símbolo ex- presionBooleana.	80
3.48. Declaración de los terminales asociados con el símbolo <code>entradaSalida</code> .	82
3.49. Implementación de las reglas de producción correspondientes a a las rutinas de entrada y salida <code>cin</code> y <code>cout</code> .	82
3.50. Declaración de los terminales asociados con el símbolo <code>declaracionVariable</code> .	83
3.51. Declaración de las clases <code>cl_tipoDato</code> y <code>cl_declaracion</code> .	84
3.52. Implementación de las reglas de producción correspondientes a los símbolos agrupadores <code>tipoPrimitivo</code> y <code>tipoContenedor</code> .	85
3.53. Implementación de las reglas de producción correspondientes al símbolo de- claracionVariable.	86

3.54. Declaración de los terminales asociados con estructuras iterativas y de control de flujo. . . . .	87
3.55. Declaración de las clases <code>cl_if</code> y <code>cl_ciclo</code> . . . . .	88
3.56. Implementación de las reglas de producción correspondientes a estructuras iterativas y de control de flujo. . . . .	89
3.57. Declaración de los terminales asociados con el símbolo <code>bloqueCodigo</code> . . . . .	90
3.58. Declaración de la clase <code>cl_bloque_codigo</code> . . . . .	91
3.59. Implementación de las reglas de producción correspondientes al símbolo <code>instruccion</code> . . . . .	91
3.60. Implementación de las reglas de producción correspondientes a los símbolos <code>instrucciones</code> y <code>bloqueCodigo</code> . . . . .	92
3.61. Declaración de los terminales asociados con el símbolo <code>declaracionFuncion</code> . . . . .	94
3.62. Declaración de las clases <code>cl_lista_argumentos</code> y <code>cl_declaracion_funcion</code> . . . . .	94
3.63. Implementación de las reglas de producción correspondientes a los símbolos <code>listaArgumentos</code> y <code>declaracionFuncion</code> . . . . .	95
3.64. Declaración de los símbolos <code>cabecera</code> , <code>cabeceras</code> y <code>declaracionesFuncion</code> . . . . .	96
3.65. Implementación de las reglas de producción correspondientes a los símbolos <code>cabecera</code> , <code>cabeceras</code> y <code>declaracionesFuncion</code> . . . . .	96
3.66. Precedencia de los operadores en la gramática. . . . .	97
3.67. Definición inicial de la clase <code>analizador_precedencia</code> . . . . .	98
3.68. Ejemplo de un código con dependencias circulares entre dos funciones. . . . .	99
3.69. Alternativa a una pila con acceso aleatorio. . . . .	103
3.70. Implementación genérica del método <code>DFS</code> . . . . .	103
3.71. Implementación del método <code>esRecursoivo</code> . . . . .	105
3.72. Definición inicial de la clase <code>cl_factor</code> . . . . .	106
3.73. Implementación del método <code>multiplicables</code> en <code>cl_factor</code> . . . . .	107
3.74. Implementación del método de multiplicación en la clase <code>cl_factor</code> . . . . .	108



3.75. Implementación del método <i>menor que</i> en la clase <code>cl_factor</code> . . . . .	109
3.76. Definición inicial de la clase <code>cl_sumando</code> . . . . .	110
3.77. Implementación del método de multiplicación en la clase <code>cl_sumando</code> . . . . .	111
3.78. Implementación del método de igualdad en la clase <code>cl_sumando</code> . . . . .	112
3.79. Definición inicial de la clase <code>cl_complejidad</code> y del método <code>eliminar_terminos_semejantes</code> . . . . .	114
3.80. Implementación del método <code>imprimir_expresion</code> en la clase <code>cl_complejidad</code> . . . . .	115
3.81. Implementación del método de multiplicación en la clase <code>cl_complejidad</code> . . . . .	117
3.82. Implementación del método de suma en la clase <code>cl_complejidad</code> . . . . .	118
3.83. Definición inicial de la clase <code>cl_analizador</code> . . . . .	119
3.84. Fragmento de la implementación del método <code>inicializador_complejidades_predefinidas</code> . . . . .	120
3.85. Implementación inicial del constructor de la clase <code>cl_analizador</code> . . . . .	121
3.86. Implementación del método <code>obtener_complejidad</code> para funciones no conte- nidas. . . . .	122
3.87. Implementación del método <code>obtener_complejidad</code> para los métodos contenidos. . . . .	125
3.88. Implementación del método <code>analizar</code> para la clase <code>cl_bloque_codigo</code> . . . . .	126
3.89. Implementación del método <code>analizar</code> para la clase <code>cl_declaracion</code> . . . . .	127
3.90. Implementación del método <code>analizar</code> para la clase <code>cl_expresion_aritmetica</code> . . . . .	128
3.91. Implementación del método <code>analizar</code> para la clase <code>cl_expresion_booleana</code> . . . . .	131
3.92. Implementación del método <code>analizar</code> para la clase <code>cl_if</code> . . . . .	133
3.93. Implementación del método <code>analizar</code> para la clase <code>cl_ciclo</code> . . . . .	134
3.94. Implementación del método <code>analizar</code> para la clase <code>cl_declaracion_funcion</code> . . . . .	135
3.95. Modificaciones a la implementación del método <code>analizar</code> de la clase <code>cl_if</code> para soportar llamadas recursivas de cola. . . . .	136
3.96. Modificaciones a la implementación del método <code>analizar</code> de la clase <code>cl_expresion_aritmetica</code> para soportar llamadas recursivas de cola. . . . .	137
3.97. Definición del método <code>analizar_recursiva</code> . . . . .	139
3.98. Definición de la función <code>analizador_complejidad</code> . . . . .	141

3.99. Implementación del módulo coordinador. . . . .	142
3.100Expresiones regulares correspondientes a los operadores ++ y -. . . . .	143
3.101Reglas de producción correspondientes a los operadores ++ y -. . . . .	144
3.102Reglas de producción correspondientes a los símbolos <b>dimension</b> y <b>dimensiones</b> .145	
3.103Regla de producción relacionada con expresiones aritméticas formadas por accesos a posiciones en arreglos. . . . .	146
3.104Verificación de la existencia de un <b>else</b> dentro de la regla del símbolo <b>ifStatement</b> .147	
4.1. Fragmento de código correspondiente a CF-103960A. . . . .	152
4.2. Fragmento de código correspondiente a CF-1702B. . . . .	153
4.3. Fragmento de código correspondiente a CF-1612C. . . . .	154
4.4. Fragmento de código correspondiente a CF-102951B. . . . .	154
4.5. Fragmento de código correspondiente a CF-1760C. . . . .	155
4.6. Fragmento de código correspondiente a CF-1490C. . . . .	156

# Introducción

No todos los algoritmos son creados iguales, a pesar de que resuelvan un problema computacional en común, es posible hacer distinciones entre distintos algoritmos en función de los recursos que consumen. Los recursos consumidos por un algoritmo en particular pocas veces son medidos de manera rigurosa, esto debido a que una estimación lo suficientemente cercana al valor real basta para proporcionar una perspectiva adecuada sobre los recursos consumidos. El análisis de algoritmos es un proceso en el cual mediante un conjunto de herramientas matemáticas es estimado el uso de un recurso, generalmente tiempo (complejidad temporal) y memoria (complejidad espacial).[1]

Ha sido demostrado por otras investigaciones el hecho de que es imposible crear un sistema experto capaz de analizar la complejidad temporal de cualquier algoritmo[2]. Sin embargo, algunos autores han realizado esfuerzos en automatizar este proceso [3][4][5] con el fin de explorar la clase de algoritmos que pueden ser analizados por medios automatizados.

Entre las investigaciones que exploran la creación de herramientas automatizadas, es posible identificar propuestas que dependen de los conocimientos del usuario o acotan el análisis a programas iterativos. Son estas limitantes en las que se basa la presente propuesta, la cual tiene como objetivo explorar la creación de un prototipo para la automatización del análisis de la complejidad temporal de programas en C++ a través del análisis de la estructura sintáctica de dichos programas y la implementación de un algoritmo heurístico de análisis de complejidad.

# Capítulo 1

## Planteamiento del Problema

A lo largo del presente capítulo será descrito el problema abordado por este proyecto, así como los objetivos, alcances y limitaciones que lo caracterizan. De igual manera serán discutidas propuestas anteriores que guardan relación con el presente proyecto en una variedad de aspectos incluyendo, pero no limitado a la similitud del problema a resolver o utilización de técnicas similares.

### 1.1. Antecedentes

Durante el proceso de creación de *software*, asumiendo que el producto es de un tamaño considerable, es casi una certeza que habrá defectos que comprometan el cumplimiento de requerimientos funcionales y no funcionales, seguridad, consumo de memoria y/o tiempo de ejecución (*bugs* de rendimiento). Como se propone en [6], esto se debe, no al hecho de que los programadores sean descuidados o incompetentes, sino al hecho de que la habilidad de un programador para entender el código fuente de un proyecto disminuye conforme la complejidad del código incrementa.

A lo largo de las décadas se han hecho esfuerzos para ayudar a detectar, eliminar y prevenir los defectos mencionados anteriormente en proyectos de *software*, esto a través de la introducción de patrones de diseño, que proveen soluciones comprobadas a problemas

recurrentes en el diseño de *software* [7] y técnicas de *testing* que, a través de la detección de *bugs*, ayudan a incrementar la calidad del *software* [8].

El *testing* es una etapa en la cual el *software* se somete a diferentes pruebas para detectar errores, esta etapa ha demostrado ser imprescindible en el desarrollo de *software*, al grado de que en promedio entre el treinta y el sesenta por ciento de los recursos del proyecto son asignados a esta fase, por lo que se ha vuelto una necesidad automatizar este proceso en la medida de lo posible. De esta necesidad nacieron diversos sistemas de administración de pruebas, cada uno especializado en una tecnología y técnica de *testing* [9].

Sin embargo, la mayoría de los sistemas de *testing* son relativamente limitados en su capacidad de encontrar *bugs* de rendimiento, ya que para lograr esto es necesario tener el contexto suficiente acerca de las tecnologías utilizadas, carga de trabajo esperada y límites del tamaño de la información sobre la que trabaja un programa. Es por esto que tecnologías que no requieran dicho contexto son valiosas para el desarrollador de *software*, dentro de éstas destacan aquellas capaces de determinar la complejidad de tiempo de códigos.

El análisis de la complejidad temporal de algoritmos es una técnica popularizada por Knuth [10] (pero que ya había atraído interés tan temprano como en 1967 [11]), que permite, dado un programa, determinar la tasa en la que la cantidad de operaciones que ejecuta crece en función de la entrada. Desde la introducción del análisis de algoritmos, se han hecho esfuerzos para automatizar este proceso, todos conscientes de que un sistema que pueda lograr esto para cualquier algoritmo es imposible de crear, pues como lo menciona [2], para determinar la complejidad de un programa es necesario a su vez determinar si termina de ejecutarse en una cantidad finita de pasos, esto es imposible para ciertos programas.

Algunos autores han logrado la automatización del análisis de complejidad de algoritmos mediante el análisis del código (métodos analíticos), de los tiempos de ejecución (métodos experimentales) o alguna combinación de los dos (métodos híbridos). Dentro de los métodos que solamente analizan el código destaca [3], quien propone un sistema que, dado un

algoritmo codificado en LISP [12] y las probabilidades de que cada variable booleana sea verdadera, retorna la complejidad temporal en el peor, mejor y promedio de los casos; esto lo logra imitando la forma en la que un compilador analiza un código e identifica componentes sintácticos y sus relaciones entre sí, para después, a través de su conversión a ecuaciones de tiempo, determinar la complejidad del programa en su totalidad. Este método tiene una fuerte dependencia en el usuario, lo cual lo hace particularmente débil si su usuario no tiene el entrenamiento suficiente en matemáticas. Esta limitante fue uno de los puntos de partida para [4] quienes propusieron un método para analizar la complejidad de tiempo en el peor de los casos de algoritmos implementados en Java [13], con una intervención mínima por parte del usuario. Esto lo lograron acotando los programas que el sistema puede analizar a aquellos que no usan funciones recursivas y cuya complejidad de tiempo es cualquiera de las siguientes  $O(1)$ ,  $O(\log n)$ ,  $O(n \log n)$  y  $O(n^c)$ , y de una forma similar al método de [3], identifican funciones, operaciones primitivas y ciclos a través de un analizador de código. Una vez identificadas la dependencia entre funciones, el anidamiento y variables de control de los ciclos, calcularon la complejidad de cada ciclo partiendo del más interno a través de las variables de control, para después multiplicar su complejidad con la de los ciclos más externos, y es a partir de la suma de la complejidad de cada ciclo y función llamada, que determinaron la complejidad final.

Los métodos analíticos hasta ahora propuestos han demostrado ser limitados en cuanto a la precisión de sus resultados (en función de los antecedentes matemáticos del usuario final o en la amplitud de algoritmos e implementaciones que puede analizar), por lo que en [5] propusieron un sistema capaz de analizar la complejidad de tiempo de métodos implementados en Java ejecutándolos con entradas de diferentes tamaños, una vez completado este paso, el sistema buscaba la clase de complejidad que mejor se adaptaba al método analizado, comparando los tiempos de ejecución con funciones de complejidad comunes, concluyendo que la complejidad del método es aquella cuya raíz Cuadrada del Error Promedio Cuadrático (RMSE, por sus siglas en inglés) sea menor. El sistema está limitado a un pequeño sub-

conjunto de clases de complejidad, además su ejecución requiere más tiempo de ejecución e intervención por parte del usuario comparado con otros métodos enlistados anteriormente.

Otras propuestas han intentado resolver el problema del rendimiento del *software*, éstas van desde optimizaciones que compiladores, como GCC, implementan en el código máquina generado [14], hasta la utilización de tecnologías de aprendizaje automático para detectar problemas de rendimiento, tal como [15], donde a partir del tiempo de ejecución de métodos de *Javascript* [16] en miles de equipos de clientes, fue entrenado un modelo capaz de detectar cuellos de botella en el rendimiento de la aplicación en su totalidad o [17], donde fue propuesto un modelo de inteligencia artificial capaz de detectar *bugs* de rendimiento en aplicaciones de *Python* [18] a través de los tiempos de ejecución de diferentes componentes del programa y estructura del código, auxiliado de *code profilers* para poder detectar secciones problemáticas del código.

## 1.2. Definición del problema

Actualmente, de acuerdo a la investigación documental expuesta en la Sección 1.1, no existen sistemas de análisis de complejidad temporal para códigos escritos en C++ cuyos prototipos sean de uso público y con una curva de aprendizaje mínima.

## 1.3. Objetivo general

Desarrollar un prototipo para la automatización del análisis de la complejidad temporal de programas en C++ basado en las técnicas de análisis de código comúnmente utilizadas en compiladores.

### 1.3.1. Objetivos específicos

- Delimitar la estructura e instrucciones que hacen que un código sea apto para ser analizado por el sistema.
- Diseñar etapas de análisis de código inspiradas en el análisis léxico y sintáctico cuyos resultados se adapten al caso de aplicación del sistema.
- Implementar un método que determine las dependencias entre las funciones y estructuras iterativas del código.
- Implementar un método que determine la cantidad de iteraciones de un ciclo en función de su estructura y declaración.
- Desarrollar un método capaz de calcular la complejidad temporal de una función recursiva de forma análoga a la de un ciclo.
- Crear un prototipo funcional del sistema.
- Probar la fiabilidad del sistema utilizando códigos escritos por personas ajenas al presente proyecto.

## 1.4. Justificación

En el presente documento se propone la creación de un sistema analizador de complejidad temporal de códigos escritos en C++, el cual es un lenguaje ampliamente utilizado en la implementación de algoritmos de bajo nivel, este sistema utilizará análisis de código para inferir la complejidad total del código, representando una alternativa a los sistemas propuestos en [3], [4], [5], [19]; cuyo uso requiere al usuario pasar por una inclinada curva de aprendizaje.



Los *bugs* de rendimiento son relativamente costosos comparado con otros defectos en programas, afectan la experiencia del usuario y en promedio toman treinta por ciento más de tiempo en ser resueltos [20]. De concretarse, un sistema lo suficientemente robusto podría reducir el impacto que los *bugs* de rendimiento tienen sobre la industria, debido a que los ingenieros de software podrán proponer o descartar con mayor rapidez que un algoritmo ineficiente es el causante de un mal rendimiento en alguna instancia de su proyecto.

La presente propuesta no solo es de valor para la industria, pues de concretarse, tendría grandes implicaciones en la forma en la que es enseñado el análisis y diseño de algoritmos, ya que la automatización del análisis de complejidad puede ser utilizada para crear sistemas de evaluación más eficientes o como una herramienta de consulta para la verificación de resultados.

## 1.5. Alcances y limitaciones

El desarrollo del prototipo fue delimitado en función de los siguientes puntos:

- Se limitó la interacción del prototipo con el usuario a una interfaz de consola.
- La complejidad temporal estimada por el prototipo fue limitada a ser una combinación de las clases  $O(1)$ ,  $O(\log n)$  y  $O(n^c)$ .
- Se acotó el análisis realizado por el prototipo a programas iterativos y con funciones recursivas de cola que usen un conjunto de instrucciones previamente aprobados.

Los resultados y conclusiones de este proyecto pueden verse limitados por el siguiente aspecto:

- No se cuenta con acceso a un banco de programas de C++ etiquetados con su complejidad temporal.

# Capítulo 2

## Marco Teórico

En este capítulo son definidos los conceptos y técnicas necesarios para la comprensión del problema a resolver así como del producto propuesto. El presente capítulo está dividido en tres secciones independientes, donde serán abordados los conceptos esenciales de análisis de complejidad algorítmica, lenguajes formales y compiladores.

### 2.1. Análisis de algoritmos.

Un algoritmo se define como una secuencia finita de pasos a seguir para resolver un problema algorítmico. Un problema algorítmico define de manera generalizada una salida, una entrada y la relación existente entre ambas, donde una entrada en específico es considerada una *instancia* del problema. Un algoritmo es considerado correcto si obtiene la salida correcta para cada instancia del problema que resuelve.

El cómputo es un área con recursos finitos (dígase memoria o velocidad de procesamiento), para poder aproximar la cantidad de memoria y tiempo de ejecución que un algoritmo usaría se desarrolló el análisis de algoritmos, en particular el análisis de complejidad temporal es una técnica que expresa la cantidad de operaciones básicas (también llamadas operaciones primitivas) que ejecuta un algoritmo en función de su entrada [1].

### 2.1.1. Sobre el modelo RAM

El análisis de algoritmos es realizado en el contexto de un modelo de cómputo llamado Random Access Machine (RAM), donde se trata de generalizar el comportamiento que siguen las computadoras que eventualmente los ejecutarían. Dentro del modelo RAM se presume lo siguiente:

- Las instrucciones son ejecutadas de manera secuencial.
- Las operaciones de acceso y modificación de memoria, así como operaciones lógicas, aritméticas y de control de flujo son consideradas operaciones primitivas, es decir son ejecutadas en una cantidad constante de tiempo.
- Las estructuras iterativas y las llamadas a subrutinas no son consideradas operaciones primitivas.
- La memoria de la computadora es infinita.

[1], [21]

Si bien este modelo resulta no ser completamente representativo del comportamiento real de una computadora, es una abstracción necesaria para el análisis de complejidad, pues de no ser utilizada, el análisis de complejidad sería dependiente de la arquitectura y características de un equipo de cómputo en específico.

### 2.1.2. Sobre el análisis de complejidad temporal y la notación *big Oh*

Usando el modelo *RAM* son contadas las operaciones primitivas, prestando especial atención a las estructuras iterativas y a las subrutinas llamadas, al resultado de este conteo se

le conoce como complejidad temporal. La complejidad temporal puede representar los casos mejor, peor y promedio, en cuanto a la cantidad de operaciones a realizar.

El polinomio resultante del conteo de operaciones de un algoritmo tiende a contar con varios sumandos, cada uno con factores constantes y variables, esto dificulta comparar la complejidad de distintos algoritmos, por lo que es necesario expresarlos en su notación *big Oh*, la cual reduce el tamaño del polinomio y sigue siendo representativa de la tasa en que las operaciones crecen.

La notación *big Oh* define las funciones  $\Theta$ ,  $O$  y  $\Omega$ , las cuales representan cotas para la función de complejidad temporal  $f(n)$ , el comportamiento de estas funciones es ilustrado en la figura 2.1 y descrito a continuación:

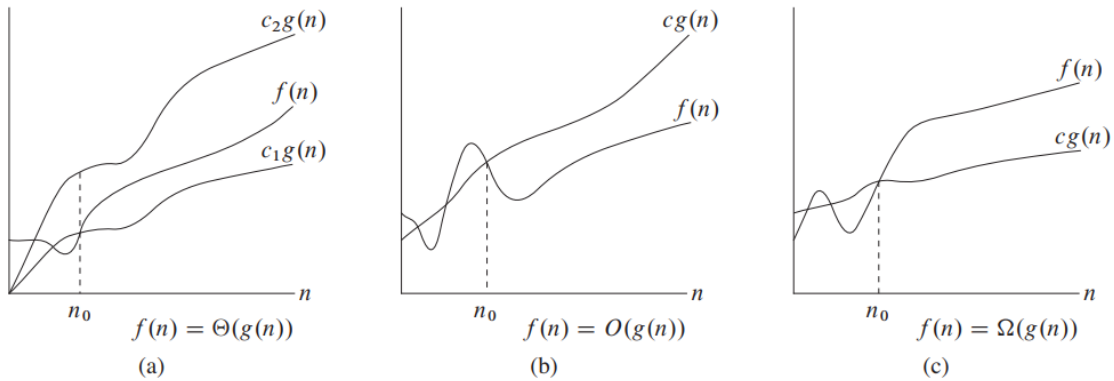


Figura 2.1: Gráfica de las notaciones  $\Theta$ ,  $O$  y  $\Omega$ , tomada de [1].

- La función  $\Omega(g(n))$  representa una cota inferior, es decir que existe una función  $g(n)$  y una constante  $c$ , tal que  $f(n) \geq c \cdot g(n)$  para todo  $n \geq n_0$  [21].
- La función  $O(g(n))$  representa una cota superior, es decir que existe una función  $g(n)$  y una constante  $c$ , tal que  $f(n) \leq c \cdot g(n)$  para todo  $n \geq n_0$  [21].
- La función  $\Theta(g(n))$  representa una cota inferior y superior, es decir que existe una función  $g(n)$  y dos constantes  $c_1$  y  $c_2$ , tal que  $f(n) \geq c_1 \cdot g(n)$  y  $f(n) \leq c_2 \cdot g(n)$  para

todo  $n \geq n_0$  [21].

## 2.2. Lenguajes formales

Para definir de manera precisa el concepto de lenguaje, es necesario definir primero otros conceptos que lo forman. Se define un alfabeto, denotado por el símbolo  $\Sigma$ , como una colección finita y no vacía de símbolos. Los símbolos dentro de  $\Sigma$  pueden ser concatenados entre ellos creando cadenas, una cadena puede estar vacía o estar formada por una concatenación arbitraria de símbolos, esto se denota con la estrella de Kleene ( $\Sigma^*$ ) [22].

Se dice que un lenguaje es un subconjunto de  $\Sigma^*$  y puede ser clasificado en función de los modelos computacionales a través de los cuales puede ser representado, la completa extensión de estos modelos y la relación existente entre ellos va mas allá del enfoque del presente proyecto, por lo que solo serán cubiertos los fundamentos necesarios para la comprensión de las expresiones regulares y las gramáticas libres de contexto.

### 2.2.1. Expresiones regulares

Propuestas por primera vez por el matemático norteamericano Stephen Kleene [23], un lenguaje regular es definido de manera recursiva como una expresión consistente de expresiones regulares primitivas y no primitivas [22], una expresión regular primitiva puede tomar cualesquiera de las tres siguientes formas:

- Un símbolo  $a$  perteneciente al alfabeto del lenguaje denotado por  $\Sigma$
- Una cadena vacía, denotada por el símbolo  $\epsilon$
- Un lenguaje vacío, denotado por  $\emptyset$

Una expresión regular también puede ser no primitiva, este tipo de expresiones son obtenidas al aplicar a una o mas expresiones regulares uno de los operadores descritos a continuación:

- El operador de unión ( $+$ ), considere los lenguajes  $R_1$  y  $R_2$ , el resultado de  $R_1 + R_2$  es el conjunto de cadenas presentes en  $R_1$ , en  $R_2$  o en ambas.
- El operador de concatenación ( $\cdot$ ), considere los lenguajes  $R_1$  y  $R_2$ , el resultado de  $R_1 \cdot R_2$  es el conjunto de cadenas presentes tanto en  $R_1$  y en  $R_2$ . Note que la concatenación de cualquier lenguaje con el lenguaje vacío resultara en el lenguaje vacío.
- La estrella de Kleene ( $*$ ), considere el lenguaje  $R_1$ , el resultado de  $R_1^*$  es el conjunto de cadenas obtenido al concatenar entre cero e infinitas veces elementos arbitrarios de  $R_1$ .

### 2.2.2. Gramáticas libres de contexto

Una manera alternativa de definir lenguajes formales es a través del uso de Gramáticas Libres de Contexto (CFG, por sus siglas en inglés), una CFG se define como la tupla  $CFG = (V, T, S, P)$  donde  $V$  representa un conjunto finito de símbolos no terminales,  $T$  es un conjunto finito de símbolos terminales,  $S$  es un símbolo de  $V$  designado como la variable de inicio y  $P$  es una cantidad finita de reglas de producción [22].

Los elementos contenidos dentro del conjunto  $T$  son análogos al alfabeto definido dentro de una expresión regular, pues de manera similar a ésta, las palabras del lenguaje definido por una CFG son formadas por una combinación de los símbolos encontrados en éste.

A diferencia de aquellos símbolos enlistados en  $T$ , los símbolos definidos dentro del conjunto  $V$  no forman parte de ninguna palabra contenida en el lenguaje descrito por la CFG, pues el propósito de estos es eventualmente ser reemplazados en su totalidad por símbolos no terminales, este proceso no es realizado de manera arbitraria y se rige por una serie de

equivalencias (reglas de producción) definidas dentro del conjunto  $P$ . Las reglas de producción dentro de una  $CFG$  tienen la forma  $x \rightarrow y$  donde  $x$  es un miembro del conjunto de los símbolos no terminales y  $y$  es un miembro del conjunto definido por  $(V \cup T)^*$  [22].

## 2.3. Compiladores

Un lenguaje de programación es una noción utilizada para describir computación tanto para humanos como para máquinas, los lenguajes de programación comúnmente contienen abstracciones que facilitan su comprensión y manipulación por parte de los programadores humanos, sin embargo para poder ejecutar los códigos escritos en estos lenguajes es necesario traducirlos a una forma que una computadora pueda comprender. Esta traducción es realizada por sistemas de software conocidos como compiladores [24].

En términos generales, un compilador es un programa que traduce un código de origen a un código de destino a través de una secuencia de representaciones intermedias resultado del análisis léxico, el análisis sintáctico, el análisis semántico, generación de código intermedio, optimización de código independiente de la máquina objetivo, generación de código y una fase de optimización de código dependiente de la máquina objetivo. De las fases previamente enlistadas, serán cubiertos los aspectos fundamentales del análisis léxico y del análisis sintáctico, debido a que estas fases fueron las implementadas durante el desarrollo del presente proyecto.

### 2.3.1. Sobre el análisis léxico

El análisis léxico es la primera etapa del proceso de compilación de un programa y la única que interactúa de manera directa con el código de origen, el propósito de esta etapa es identificar *palabras* propias del lenguaje de origen a partir del código fuente, estas palabras

también son conocidas como *lexemas*, *items* léxicos o *tokens* léxicos, son consideradas como la unidad mínima dentro de la definición del lenguaje y son enviados a la fase de análisis sintáctico junto con atributos que le son asignados. Además de la identificación de palabras, el analizador léxico también cuenta con la tarea de compactar espacios y saltos de línea, además de descartar comentarios hechos dentro del código fuente.

La forma que pueden tomar los *lexemas* dentro de un lenguaje son definidos a través de *patrones*, los cuales pueden ser autómatas finitos deterministas o de manera alternativa a una colección de expresiones regulares que eventualmente son traducidas a un autómata por un generador de analizadores léxicos como *Flex*. Debido a que varias cadenas pueden ser identificadas por un mismo patrón, los *tokens* identificados durante esta fase pueden tener uno o mas atributos asociados con el fin de que preserven su significado original.

### 2.3.2. Sobre el análisis sintáctico

El análisis sintáctico (también conocido como *parsing*) es la segunda etapa dentro del proceso de compilación, la entrada de esta etapa consiste en una serie de *tokens* identificados por la etapa de análisis sintáctico, los cuales son analizados con el fin de verificar que las instrucciones que describen sean sintácticamente correctas, si este proceso es exitoso esta fase retorna una representación intermedia conocida como Árbol Abstracto de Sintaxis (AST por sus siglas en inglés) [25].

La estructura sintáctica de un lenguaje de programación es frecuentemente descrita a través del uso de gramáticas libres de contexto, donde el conjunto de símbolos terminales está formado por los *tokens* definidos por el analizador léxico [24].

Existen diversas técnicas para interpretar una CFG que define un lenguaje, entre las cuales destacan los métodos: *universal*, *bottom-up* y *top-down*. Los métodos universales como los descritos por los algoritmos de *Earley* y de *Cocke-Younger-Kasami*, tienden a ser muy



ineficientes para ser usados con fines prácticos. Los métodos *top-down* y *bottom-up* son comúnmente usados en el desarrollo de compiladores y como es posible inferir por sus nombres, el primero construye el AST a partir de la raíz derivándola hasta llegar a los nodos hojas o símbolos terminales, el segundo inicia la construcción del AST en las hojas de éste y al aplicar inversamente las reglas de producción definidas por la CFG llegan al símbolo de inicio  $S$  [25].

Los algoritmos de *parsing bottom-up* son capaces de interpretar un mayor numero de CFG's comparado con sus contra partes *top-down* y generalmente son creados a través de herramientas automatizadas [25], [24]. Estos algoritmos constan de dos operaciones principales: mover (*shift*) un símbolo a una pila y convertir un sufijo de esta pila a un símbolo no terminal (*reduce*). El *parsing bottom-up* siempre puede ser implementado usando una derivación de tipo LR, donde se deriva siempre el no terminal más a la derecha en la regla de producción actual [25].

Existen dos tipos de errores comunes dentro del *parsing bottom-up*, los conflictos de tipo *shift/reduce* ocurren cuando dos reglas de producción pueden ser cumplidas al mismo tiempo, donde una de ellas puede ser obtenida reduciendo un sufijo de la pila mientras que la otra puede ser obtenida añadiendo el símbolo actual en la pila; también pueden ocurrir conflictos de tipo *reduce/reduce* donde dos sufijos no necesariamente iguales de la pila pueden ser reducidos. Estos errores tienen su origen en la definición de la gramática y por lo tanto solo pueden ser resueltos modificándola.

# Capítulo 3

## Desarrollo del Proyecto

En el presente capítulo, se describen diversos aspectos del producto, entre ellos su diseño, implementación, y tecnologías a partir de las cuales fue creado.

### 3.1. Producto propuesto

El producto propuesto es un prototipo para estimar la complejidad temporal de códigos escritos en C++. El prototipo fue dividido en cuatro módulos de análisis (léxico, sintáctico, de jerarquía y dependencia, de complejidad), coordinados por un módulo denominado coordinador. El módulo de análisis léxico fue construido usando Lex v 2.6.4, el módulo de análisis sintáctico fue desarrollado usando Bison v 3.5.1, el resto de los módulos fueron construidos usando C++.

El prototipo fue construido con el objetivo de que sea ejecutado de forma local como una utilidad de consola en el equipo del usuario final. La arquitectura del prototipo es ilustrada por medio de la Figura 3.1 y descrita a continuación:

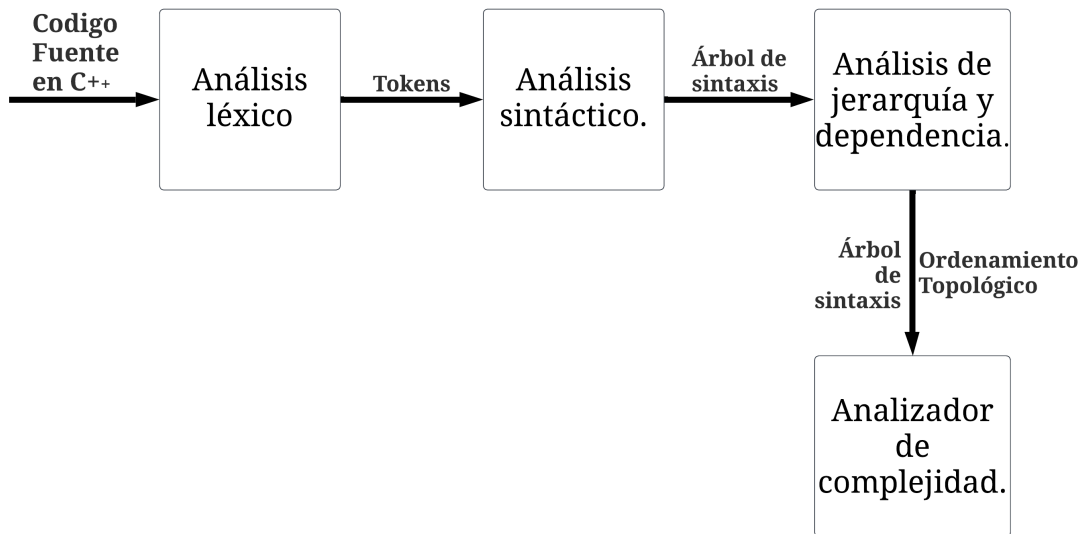


Figura 3.1: Arquitectura del prototipo.

- **Análisis léxico:** Recibirá como entrada el código fuente escrito en C++, del cual extraerá los *tokens* que lo conforman utilizando los métodos que tradicionalmente utiliza la etapa de la compilación con el mismo nombre. Una vez obtenidos los *tokens*, serán analizados con el fin de verificar si solamente fueron utilizados los métodos especificados por el sistema, de ser así, serán enviados a la próxima etapa.
- **Análisis sintáctico:** Esta etapa recibirá como entrada los *tokens* obtenidos por el análisis léxico, los cuales utilizará para crear un árbol de sintaxis por medio de las técnicas comúnmente utilizadas por compiladores en la fase con el mismo nombre.
- **Análisis de jerarquía y dependencia:** Este módulo recibirá como entrada el árbol de sintaxis del programa y determinará si no existen dependencias circulares entre funciones, exceptuando aquellas creadas por las llamadas recursivas. En caso de ser acíclico el grafo de llamadas, se creará un ordenamiento topológico de las llamadas y funciones, donde los primeros elementos serán los de menor jerarquía en el código, este ordenamiento topológico y el árbol de sintaxis, serán enviados a la siguiente etapa.

- **Análisis de complejidad:** Este módulo representa la última etapa del sistema, recibirá como entrada el árbol de sintaxis del programa y el ordenamiento topológico del mismo. Usando el ordenamiento topológico para definir el orden en el que las instrucciones del programa serán analizadas, se analizará de forma gradual la complejidad temporal de cada componente del código. Si el componente siendo analizado es una función ya definida en la especificación del lenguaje, su complejidad será retornada en función de sus argumentos; si es una estructura iterativa, su complejidad será calculada en función de las instrucciones incluidas en el cuerpo del bucle y la complejidad inferida a través del valor inicial y el *step* de su variable de iteración; en caso de ser una función, su complejidad será la suma de las complejidades de las instrucciones en el cuerpo de la función; si la función es recursiva, su *step* recursivo y casos bases serán utilizados para convertirla en un ciclo y analizar su complejidad usando esta nueva forma.

## 3.2. Descripción de la metodología

La metodología de cascada genérica descrita en [26] fue elegida para el desarrollo del presente. La versión de metodología usada en este documento unió las fases de definición de requisitos y análisis en una sola debido a que el producto resultante de la segunda es un resultado del producto de la primera, además fueron omitidas las fases de mantenimiento y retiro del producto, debido a que el periodo de desarrollo asignado no permite desarrollarlas de manera adecuada.

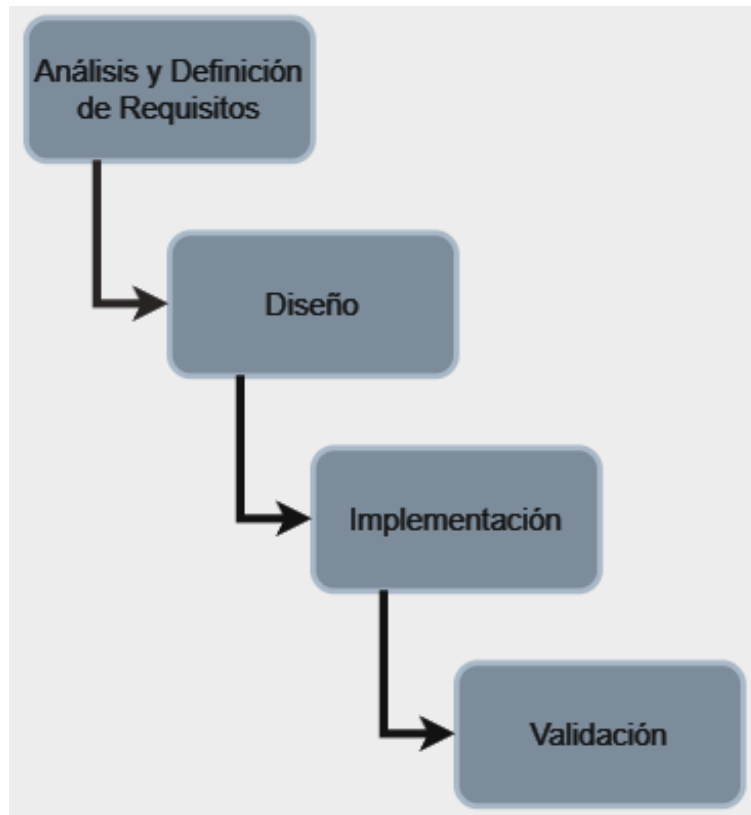


Figura 3.2: Metodología Cascada usada para el desarrollo del prototipo.

### 3.3. Análisis y Definición de Requisitos

Durante esta fase fueron definidas de manera sucinta las cualidades y estructura del producto final. Además se delimitó qué clase de programas serían admitidos por el sistema.

#### 3.3.1. Análisis

Durante este apartado se delimitan las características con las que deben cumplir los códigos para ser analizables por el prototipo.

**Acerca de las funciones recursivas.**

Considere el fragmento de código plasmado en el Listado 3.1. En el se especifica una función que retorna el n-ésimo número de Fibonacci, nótese que la función se llama a sí misma para obtener el resultado, a esto se le conoce como una función recursiva.

```
1 int fib(int n){  
2     if(n==0) return 0;  
3     if(n==1) return 1;  
4     return fib(n-1)+fib(n-2);  
5 }
```

Listado 3.1: Definición recursiva de la sucesión de Fibonacci.

A las funciones recursivas que solo utilizan una sola llamada, como la descrita en el código mostrado en el Listado 3.2, se les conoce como funciones recursivas de cola. Su estructura puede ser comparada con aquella de un ciclo, donde el caso base de la recursión puede ser usado como la condición de terminación, el paso recursivo como el incremento o decremento de la variable de control y el cuerpo de la función como el código que es ejecutado en cada iteración. Por ello se decidió que el prototipo acepte solo funciones recursivas de este tipo.

```
1 int suma_hasta(int n){  
2     if(n==0) return 0;  
3     return suma_hasta(n-1)+n;  
4 }
```

Listado 3.2: Declaración de una función recursiva de cola.

**Métodos, contenedores y tipos válidos.**

Se decidió acotar los métodos validos a un subconjunto del estándar del lenguaje, que el autor del presente documento considera más utilizado en cursos de estructura de datos y análisis de algoritmos. La Tabla 3.1 , mostrada a continuación, describe dicho subconjunto

del estándar, haciendo distinción entre las características propias de la *Standard Template Library (STL)* y aquellas que no lo son.

Tabla 3.1: Métodos, contenedores y tipos del estándar de C++ aceptados por el prototipo.

C++ vanilla	El uso de <i>namespaces</i> no está permitido, por lo que es imperativo para que un código sea analizable por el sistema que cuente con la directiva <i>using namespace std</i> ;
	Todos los tipos de datos primitivos tales como <i>bool</i> , <i>int</i> , <i>long long int</i> , <i>float</i> , <i>long double</i> y <i>double</i> son permitidos.
	Funciones definidas por el usuario, siempre y cuando no hagan uso de <i>templates</i> o polimorfismo.
	Funciones recursivas de cola, cuyo <i>step</i> recursivo dependa de un solo argumento
	Estructuras iterativas definidas a través de: <i>while</i> , <i>for</i>
	Estructuras y procedimientos de control de flujo: <i>if</i> , <i>else</i> , <i>break</i> , <i>continue</i>
	Solo será válida la salida y entrada por consola de C++ a través de los métodos <i>cin</i> , <i>cout</i>
	Variables y arreglos definidos de la forma tipo_de_dato identificador[tamaño]
	Operadores aritméticos.
	Operadores lógicos, exceptuando las palabras clave <i>and</i> , <i>or</i> , <i>not</i>
De la STL	Las funciones <i>swap</i> , <i>min</i> , <i>max</i> , <i>lower_bound()</i> , <i>upper_bound()</i> y <i>sort</i>
	Contenedor <i>stack</i> y los siguientes metodos asociados: <i>push()</i> , <i>pop()</i> , <i>top()</i> , <i>size()</i>
	Contenedor <i>vector</i> y los siguientes metodos y operadores asociados: <i>push_back()</i> , <i>pop_back()</i> , <i>size()</i> , <i>[]</i> , <i>front()</i> , <i>back()</i> , <i>swap()</i> , <i>clear()</i> , <i>shrink_to_fit()</i> , <i>at()</i>
	Contenedores <i>map/unordered_map</i> y los siguientes métodos y operadores asociados: <i>insert()</i> , <i>[]</i> , <i>clear()</i>
	Contenedores <i>set/multiset</i> y los siguientes métodos y operadores asociados: <i>insert()</i> , <i>erase()</i> , <i>size()</i> , <i>clear()</i> , <i>lower_bound()</i> , <i>upper_bound()</i> , <i>find()</i> , <i>count()</i>
	Contenedor <i>queue</i> y los siguientes metodos y operadores asociados: <i>size()</i> , <i>push()</i> , <i>pop()</i> , <i>front()</i> , <i>back()</i>
	Contenedores <i>string</i> y los siguientes metodos y operadores asociados: <i>push_back()</i> , <i>pop_back()</i> , <i>size()</i> , <i>[]</i> , <i>front()</i> , <i>back()</i> , <i>swap()</i> , <i>clear()</i> , <i>shrink_to_fit()</i> , <i>at()</i> , <i>=</i>



### 3.3.2. Requisitos del sistema

A continuación se presenta una lista de características que de manera general trazan cómo el prototipo resuelve el problema, así como la manera con la que interactúa con el usuario final.

- Es posible invocar al prototipo desde una interfaz de comandos.
- El prototipo divide el análisis en cuatro módulos independientes: análisis léxico, análisis sintáctico, análisis de jerarquía y dependencia, analizador de complejidad.
- El prototipo debe terminar el análisis en aproximadamente en el tiempo requerido para compilar el código.
- El prototipo describe la complejidad en términos de los identificadores usados en el código.

## 3.4. Diseño

Se decidió que el prototipo divida el análisis en cuatro fases secuenciales. Cada fase es manejada por un módulo independiente, los cuales reciben una representación intermedia del módulo anterior. Siendo los módulos de análisis léxico y sintáctico los primeros, estos procesan el código fuente y lo transforman a un formato, al cual le sea mas fácil manipular y extraer información a los módulos subsecuentes. El módulo de análisis de jerarquía y dependencia tiene como objetivo determinar el orden en el que deben ser calculadas las complejidades de cada instrucción y función del programa. Finalmente el módulo de análisis de complejidad, utilizando la información extraída por los módulos anteriores, calcula la complejidad total del programa.

### 3.4.1. Diseño del módulo de análisis léxico

Este módulo es el primero en ser llamado, recibe como entrada el código fuente a analizar y debe de devolver una lista de *tokens*, cada uno de ellos con atributos que les caracterizan. Los *tokens* que el módulo es capaz de identificar deben ser acotados de acuerdo a la especificación descrita en el Capítulo 3.3.1 y deben de contar con los siguientes atributos:

- Un entero que funja como identificador único del *token*.
- Un entero que denote la línea en la que fue encontrado el *token*.
- Un entero denotando el valor entero del *token*, sí cuenta con uno.
- Una cadena de caracteres denotando el nombre del identificador en el caso del *token* variable y la cadena asociada, en el caso de cadenas definidas por el usuario.
- Un carácter denotando el valor carácter del *token*, sí cuenta con uno.

De los *tokens* identificados, para efectos de este documento es posible hacer una distinción entre ellos en dos categorías distintas en función de la forma de las expresiones regulares que las representan: Aquellos donde existe una relación uno a uno entre la cadena que representa al *token* y su expresión (*tokens monomorfos*) y aquellos cuya cadena que los representa puede tomar más de una forma (*tokens polimorfos*).

#### **Tokens monomorfos**

Se dice que un *token* es *monomorfo* cuando existe una relación uno a uno entre la cadena que representa al *token* y su expresión, dentro de esta categoría podemos encontrar palabras reservadas y caracteres especiales como: tipos de datos (`int`, `char`, `float...`), nombres de contenedores (`vector`, `map`, `set...`), instrucciones de control de flujo y de iteración (`for`,

`while`, `continue`, `break`), operadores aritméticos, operadores booleanos y caracteres especiales usados como signos de puntuación (, ; [] () {}).

Es esta relación uno a uno lo que hace posible que la expresión regular que exprese estos elementos del lenguaje sea solamente la cadena que representa al *token*, por ejemplo, basta con la expresión *vector* para representar al nombre del contenedor **vector**.

Es importante notar que, a pesar de ya tener un identificador definido, los métodos miembro de los contenedores y las funciones ya implementadas dentro del lenguaje, serán tomadas como *tokens polimorfos*, con el fin de mantener la generalidad de las gramáticas que representan al lenguaje, la distinción se hará hasta la etapa de análisis de complejidad.

### **Tokens polimorfos**

Se definen los *tokens polimorfos* como aquellos que pueden ser representados por más de una palabra, un ejemplo concreto de este comportamiento podría ser ilustrado con el token *identificador*, las palabras **arreglo**, **i**, **\_\_esPrimo** son todos identificadores válidos, así como cualquier combinación de caracteres del alfabeto inglés, por lo que representar al token identificador mediante una colección de cadenas resulta no viable.

Entre los *tokens polimorfos* que se identificaron en el lenguaje se encuentran comentarios de una sola línea, comentarios de múltiples líneas, constantes enteras, constantes con punto flotante, constantes de tipo carácter, constantes de tipo cadena de caracteres e identificadores. Las cadenas representativas de cada uno de estos *tokens* serán definidas a través de una expresión regular.

### **Comentarios**

En C++, los comentarios existen solo dentro del código fuente con el fin de ser leídos por un humano, durante la fase de análisis léxico deben de ser descartados. Existen dos tipos

de comentarios dentro de la especificación del lenguaje, aquellos de una sola línea y los que pueden ocupar mas de una línea.

Los comentarios de una sola línea están definidos por un inicio que consiste de dos barras inclinadas (`//`), el cuerpo del comentario puede ser cualquier carácter a excepción de un salto de línea (`'\n'`) y el final del comentario está marcado por un solo salto de línea (`'\n'`). La expresión regular que representa a este tipo de comentarios es `[/]{2}[^\n]*`, la cual puede ser interpretada como una palabra con exactamente dos barras (`//`) al inicio y cero o más ocurrencias de cualquier carácter a excepción del carácter salto de línea (`'\n'`).

En cambio, los comentarios de múltiples líneas delimitan su inicio con una barra inclinada seguida de un asterisco (`/*`), su final esta marcado por un asterisco seguido de una barra inclinada (`*/`) y dentro del cuerpo del comentario puede ir cualquier combinación de caracteres a excepción de la combinación que marca el fin. La expresión regular que define este tipo de comentarios es `"/*" [^\0]* "*/"`, la cual indica que una palabra es un comentario multilínea siempre y cuando inicie con la cadena `/*`, contenga cualquier combinación de cero o más caracteres y termine con la cadena `*/`.

### Constantes numéricas

En el contexto del analizador léxico, se define como una constante numérica a cualquier valor numérico dentro del código fuente, una constante numérica puede ser de tipo entero (11,9898,4562, etc) o de tipo flotante (22.65 , 3336.0, etc.). Las constantes enteras pueden ser representadas mediante la expresión `[0-9]+`, la cual representa a todas las palabras resultantes de la combinación de uno más dígitos. Es importante recalcar que las constantes numéricas consideradas bajo esta expresión regular son expresadas en el sistema numérico decimal. Por su parte, las constantes de tipo flotante, pueden ser representadas por una combinación de un dígito o más, seguido por un punto y concluido con una combinación de un dígito o más, tal y como se lee en la expresión `[0-9]+.[0-9]+`

### Constantes de tipo carácter

Las constantes de tipo carácter, son aquellas que inician y terminan con una comilla simple y entre ellas contienen cualquier carácter a excepción de una comilla simple, como se lee en la expresión `'[^']'`

### Constantes de tipo cadena

Se definen a las cadenas especificadas por el usuario como una constante de tipo cadena, ejemplos de constantes de tipo cadena pueden ser `"cadena"`, `"a"` y `"22Mingus79"`. Este tipo de constante está marcada por un inicio y final que consisten de una aparición de comillas dobles y un cuerpo que puede contener cualquier combinación de cero o más caracteres exceptuando las comillas dobles, la expresión que la define es `["][^"]["]`.

### Identificadores

Se definen como identificadores a los nombres que les son asignados a las variables o funciones dentro de un programa. En C++ un identificador debe de tener como primer carácter un guión bajo o alguna letra mayúscula o minúscula, el resto de los caracteres pueden ser estos o algún dígito. Esto es representado mediante la expresión `[A-Za-z][A-Za-z0-9_-]+`

#### 3.4.2. Diseño del módulo de Análisis sintáctico

Una vez obtenidos los *tokens*, es necesario identificar las estructuras que forman y extraer significado de ellas. Para lograr esto fue diseñado un módulo de análisis sintáctico, el cual a través de los *tokens* identificados por el analizador léxico, un conjunto de reglas en forma de gramáticas y clases para el almacenamiento de sus características, define su estructura en un árbol abstracto de sintaxis, el cual puede ser manipulado de forma algorítmica.

## Definición de la gramática

Para la creación de la gramática que representa el lenguaje, se decidió seguir un enfoque de divide y vencerás. Primero, se propuso el código mostrado en el Listado 3.3, el cual contiene una variedad de los elementos que se desean identificar.

Este código fue dividido en sus componentes principales y estos en subcomponentes, esta división fue realizada hasta llegar a los *tokens*, que para efectos de este módulo, son considerados como elementos indivisibles, lo cual los convierte en símbolos terminales dentro del contexto de la gramática que define a C++. Es importante resaltar que el único comentario dentro del código fuente fue ignorado, debido a que en la práctica el analizador sintáctico no interactuaría con éste, pues es descartado por el analizador léxico.

A pesar de que la gramática fue diseñada a través de la disección del código mostrado en el Listado 3.3, el autor del presente documento considera más intuitivo introducir al lector a las producciones partiendo de aquellas formadas por una combinación de símbolos terminales y no terminales propios y de manera gradual introducir a aquellas reglas que las contengan, continuando de esta manera hasta llegar a las reglas que definen al programa en su totalidad. También es importante hacer mención del hecho que junto con cada gramática, serán definidos los atributos que le acompañan.

```
1 #include "bits/stdc++.h"
2 using namespace std;
3
4 int suma(int l, int r){
5     int res = 0;
6     while(l<=r && l<=r){
7         res+=r;
8         r--;
9     }
10    return res;
```

```
11 }
12 ///definir el arbol de sintaxis que mejor funcionaria para este programa
13 int main(){
14     int n;
15     cin>>n;
16     vector<int> arr;
17     int res = 0;
18     for(int i = 0; i<n; i+=1){
19         for(int j = i; j<n; j*=2){
20             cout<<suma(i,j)<<endl;
21             arr.push_back(suma(i,j));
22             res+=suma(i,j);
23         }
24     }
25     cout<<"res = "<<res;
26 }
```

Listado 3.3: Código usado para el diseño de la gramática

### Sobre los símbolos terminales

Extendiendo sobre lo mencionado en la Sección 3.4.2, todos los *tokens* identificados por el módulo de análisis léxico serán tratados como símbolos terminales. Cada símbolo terminal conserva los atributos que le fueron asignados durante el módulo anterior y sigue una convención de nomenclatura donde al nombre del *token* se le añade el prefijo *t\_*, esto para poder diferenciarlos de manera inmediata de los símbolos no terminales.

### Acerca de los símbolos no terminales agrupadores

Considere la gramática mostrada en el Listado 3.4, la cual define expresiones aritméticas (denotadas por el símbolo *EA*) formadas por identificadores y los cuatro operadores *+-\*/*

representados por los símbolos terminales `t_add`, `t_sub`, `t_mult`, `t_div` respectivamente.

```

1 EA → t_identificador
2 EA → EA t_add EA
3 EA → EA t_sub EA
4 EA → EA t_mult EA
5 EA → EA t_div EA

```

Listado 3.4: Gramática con reglas de producción para cada operador

De la gramática se interpreta que una expresión aritmética en su forma más básica está formada por un identificador y también puede ser expresada como la suma, resta, multiplicación y división de dos expresiones. Si bien, esta gramática es correcta, una manera alternativa de expresarla es a través de la introducción de símbolos no terminales denominados como *agrupadores* para efectos de este documento.

Al abstraer los operadores con el símbolo no terminal agrupador *operador* en la gramática mostrada en el Listado 3.5, el autor del presente documento considera que se mejoró la legibilidad de la definición de *EA*, debido a que fue compactada respecto al número de reglas de producción, eliminando el sentido de redundancia inherente al mantenimiento de reglas similares para diferentes operadores. Por esta razón, se decidió favorecer la creación de no terminales agrupadores que funjan como símbolos auxiliares en la definición de los no terminales de la gramática.

```

1 operador → t_add
2 operador → t_sub
3 operador → t_mult
4 operador → t_div
5
6 EA → t_identificador
7 EA → EA operador EA

```



---

Listado 3.5: Gramática con reglas de producción que agrupan terminales representando operadores

### Expresiones aritméticas

Una expresión aritmética puede ser monolítica o el resultado de la aplicación de diferentes operadores sobre una o más expresiones aritméticas. Para reducir el sentido de redundancia dentro de la definición del no terminal `expresionAritmetica`, se decidió crear los símbolos agrupadores `operadorAritmético`, `operadorAsignación` y `constanteGenerica`.

Como se muestra en la gramática mostrada en el Listado 3.6, el símbolo `operadorAritmético` puede ser derivado en cualquiera de los tokens representativos de los operadores de suma, resta, multiplicación y división; mientras que un operador de asignación resulta de un solo operador de igualdad(=) o de este operador precedido de un operador aritmético. Finalmente, una `constanteGenerica` puede ser derivada en cualquiera de los valores constantes definidos en el módulo anterior.

```
1 operadorAritmetico → t_sum
2 operadorAritmetico → t_sub
3 operadorAritmetico → t_div
4 operadorAritmetico → t_astk
5
6 operadorAsignación → operadorAritmetico t_eq
7 operadorAsignación → t_eq
8
9 constanteGenerica → t_intconst
10 constanteGenerica → t_true
11 constanteGenerica → t_false
```

```

12 constanteGenerica → t_doubleconst
13 constanteGenerica → t_hardstr
14 constanteGenerica → t_charconst

```

Listado 3.6: Reglas agrupadoras de operadores aritmeticos, operadores de asignación y de constantes.

Una vez definidos los no terminales agrupadores, se puede definir un no terminal **expresionAritmetica** en función de estos. En su forma más básica, se tiene que una expresión aritmética puede estar formada por un solo identificador o valor constante, esto es ilustrado por la gramática mostrada en el Listado 3.7.

```

1 expresionAritmetica → t_identificador
2 expresionAritmetica → constanteGenerica

```

Listado 3.7: Reglas de producción correspondientes a la forma mas básica que puede tomar una expresión aritmética.

A su vez, se tiene que una expresión aritmética monolítica también puede ser formada por una llamada a una función. Una función puede tener cero o más argumentos, estos pueden ser expresiones aritméticas no monolíticas o expresiones aritméticas monolíticas como identificadores, constantes o llamadas a otras funciones. Por lo tanto, es necesaria la creación de un no terminal agrupador **argumentosLLlamada** dentro de la gramática mostrada en el Listado 3.8 para definir de manera recursiva los argumentos de la función, además también se requiere crear una regla que defina a las llamadas a funciones miembros de los contenedores de STL.

```

1 argumentosLLlamada → ε
2 argumentosLLlamada → expresionAritmetica
3 argumentosLLlamada → argumentosLLlamada t_comma
                        expresionAritmetica

```

```

4
5 expresionAritmetica → t_identificador t_opregb argumentosLLamada
    t_cloregb
6 expresionAritmetica → t_identificador t_dot t_identificador
    t_opregb argumentosLLamada t_cloregb

```

Listado 3.8: Reglas de producción relacionadas con expresiones aritméticas formadas por llamadas a funciones.

Una expresión aritmética, como se muestra en la gramática plasmada en el Listado 3.9, también puede representar al acceso de una posición en un arreglo o estructura de datos asociativa, donde la posición puede ser, a su vez, una expresión aritmética.

```

1 expresionAritmetica → t_identificador t_opsqrb
    expresionAritmetica t_closqrb

```

Listado 3.9: Regla de producción relacionada con expresiones aritméticas formadas por accesos a posiciones en arreglos.

Finalmente, en la gramática mostrada en el Listado 3.10, se muestra cómo una expresión aritmética puede ser obtenida a partir de la aplicación de un operador aritmético o de asignación a dos expresiones aritméticas o a través de la contención de una expresión aritmética dentro de paréntesis.

```

1 expresionAritmetica → expresionAritmetica operadorAritmetico
    expresionAritmetica
2 expresionAritmetica → expresionAritmetica operadorAsignación
    expresionAritmetica
3 expresionAritmetica → t_opregb expresionAritmetica t_cloregb

```

Listado 3.10: Reglas de producción correspondientes a expresiones aritméticas formadas por otras expresiones aritméticas.

Una vez definidas todas las formas que puede tomar una expresión aritmética, resulta prudente definir los atributos que caracterizan a una instancia de ésta, a continuación se describe de forma general dichos atributos.

- **Un booleano que defina si la expresión es monolítica:** Extendiendo sobre lo mencionado anteriormente en la Sección 3.4.2, para efectos de este documento decimos que una expresión aritmética es monolítica si solamente está formada por una llamada a una función, un identificador o una constante. Es importante hacer esta distinción, pues las expresiones monolíticas tienen atributos distintos a sus contrapartes.
- **Apuntadores a las expresiones que la forman:** En caso de que la expresión resulte ser no monolítica, es seguro asumir que la expresión actual está formada por dos distintas expresiones unidas por un operador, conservar la ubicación de las expresiones que forman a una expresión no monolítica resulta esencial para la creación del árbol abstracto de sintaxis.
- **Un booleano denotando si es una constante o identificador y un valor relacionado:** Dentro de las expresiones monolíticas se tienen aquellas que consisten de un identificador o alguna constante, ambas necesitan de la existencia de una bandera booleana que servirá para indicar si la expresión monolítica está formada por estos, además se requiere de la creación de atributos que almacenen el nombre en el caso del identificador, o el valor que representa en caso de las constantes.
- **Un booleano que denote si es una llamada a una función y un objeto que guarde los atributos de la llamada:** El último tipo de expresión monolítica consiste de aquellas formadas por una llamada a una función, además de la bandera que la define como tal, se debe contar con el identificador de la función, el identificador del contenedor (si aplica) y una lista de argumentos en forma de expresiones aritméticas, debido a la cantidad de características de una llamada, se tomó la decisión de encapsular estos

atributos dentro de una clase propia, cuyas instancias serán referenciadas desde las expresiones aritméticas que las contienen.

## Expresiones booleanas

Una expresión booleana, en su forma más básica, está caracterizada por la aplicación de un operador lógico sobre dos expresiones aritméticas, también es posible expandir una expresión booleana existente a través de la aplicación de un operador lógico sobre una expresión booleana y una expresión aritmética. Este comportamiento es ilustrado por la gramática mostrada en el Listado 3.11, donde de manera similar a las expresiones aritméticas, fue creado un no terminal agrupador para los operadores lógicos para mejorar la legibilidad de la regla que define al no terminal `expresionBooleana`.

Si bien la gramática mostrada en el Listado 3.11 no lo expresa de manera explícita, es posible la existencia de una expresión booleana formada por una sola expresión aritmética, pero solamente existirá contenida dentro de una expresión booleana más grande. Esta encapsulación nos permitirá mantener una definición de clase más concisa.

```
1 operadorBooleano → t_andand
2 operadorBooleano → t_oror
3 operadorBooleano → t_eqeq
4 operadorBooleano → t_npeq
5 operadorBooleano → t_lteq
6 operadorBooleano → t_gteq
7 operadorBooleano → t_lt
8 operadorBooleano → t_gt
9
10 expresionBooleana → expresionAritmetica operadorBooleano
    expresionAritmetica
11 expresionBooleana → expresionBooleana operadorBooleano
```

```
expresionAritmetica
```

Listado 3.11: Reglas agrupadoras de operadores booleanos y de definición de expresiones booleanas compuestas por dos expresiones.

Una vez identificada una expresión booleana, será necesario representarla dentro del árbol abstracto de sintaxis por medio de una clase, la cual contiene los siguientes atributos.

- **Un booleano que defina si la expresión es monolítica:** Como ya fue mencionado, una expresión booleana puede ser monolítica si forma parte de una expresión booleana más grande. El valor de ésta variable define qué atributos existen para la expresión booleana actual.
- **Apuntadores a las expresiones que la forman:** En caso de que la expresión resulte ser no monolítica, es seguro asumir que la expresión actual está formada por dos distintas expresiones unidas por un operador, conservar la ubicación de las expresiones que forman a una expresión no monolítica resulta esencial para la creación del árbol abstracto de sintaxis.
- **Un apuntador a la expresión aritmética que la forma:** En caso de que la expresión booleana actual sea monolítica, es necesario mantener la ubicación de la expresión aritmética que encapsula.

### Rutinas de entrada y salida

De acuerdo a lo discutido en la Sección 3.3.1, para el analizador sintáctico solo serán tomadas en cuenta las rutinas de entrada y salida denotadas por los terminales `t_cin` y `t_cout`. Para efectos de este módulo, se definió que la entrada/salida de datos en C++ puede ser efectuada siempre y cuando el dato pueda ser encapsulado dentro de una expresión aritmética. Esta definición dio forma a la gramática mostrada en el Listado 3.12, donde fueron

creados los símbolos auxiliares `argEntrada` y `argSalida` con el fin de agrupar las expresiones que serán impresas/leídas.

```

1 argEntrada → ε
2 argEntrada → argEntrada t_gt t_gt expresionAritmetica
3 argEntrada → argEntrada t_lt t_lt t_endl
4
5 argSalida  → ε
6 argSalida  → argSalida t_lt t_lt expresionAritmetica
7 argSalida  → argSalida t_lt t_lt t_endl
8
9 entradaSalida → t_cin argEntrada
10 entradaSalida → t_cout argSalida

```

Listado 3.12: Reglas de producción correspondientes a los métodos de entrada y de salida.

Si bien, es posible crear una clase separada para almacenar los atributos relacionados con las rutinas de entrada y salida, es posible tratar estas rutinas con llamadas a una función, donde los argumentos de entrada y salida serían almacenados dentro de los argumentos de la clase función y el identificador correspondería a la rutina llamada (`cin/cout`).

### Declaración de variables

Como se muestra en la gramática plasmada en el Listado 3.13, las declaraciones de variable que son cubiertas por el prototipo pueden ser formadas solamente por un identificador de tipo seguidas de un identificador o pueden agregar un símbolo `t_eq` seguido de una expresión aritmética para formar lo que para efectos de este documento es conocido como una *declasnación*. Los identificadores de tipo fueron divididos en dos categorías representadas por los símbolos no terminales `tipoPrimitivo` y `tipoContenedor`.

```

1 tipoPrimitivo → t_void

```

```

2  tipoPrimitivo → t_bool
3  tipoPrimitivo → t_char
4  tipoPrimitivo → t_float
5  tipoPrimitivo → t_short t_int
6  tipoPrimitivo → t_long t_long t_int
7  tipoPrimitivo → t_long t_int
8  tipoPrimitivo → t_int
9  tipoPrimitivo → t_long t_double
10 tipoPrimitivo → t_double
11
12 tipoContenedor → t_vector t_gt tipoPrimitivo t_lt
13 tipoContenedor → t_set t_gt tipoPrimitivo t_lt
14 tipoContenedor → t_multiset t_gt tipoPrimitivo t_lt
15 tipoContenedor → t_queue t_gt tipoPrimitivo t_lt
16 tipoContenedor → t_pqueue t_gt tipoPrimitivo t_lt
17 tipoContenedor → t_stack t_gt tipoPrimitivo t_lt
18 tipoContenedor → t_bitset t_gt t_intconst t_lt
19 tipoContenedor → t_string
20 tipoContenedor → t_map t_gt tipoPrimitivo t_comma tipoPrimitivo
    t_lt
21 tipoContenedor → t_umap t_gt tipoPrimitivo t_comma tipoPrimitivo
    t_lt
22
23 declaracionVariable → tipoPrimitivo t_identificador
24 declaracionVariable → tipoPrimitivo t_identificador t_eq
    expresionAritmetica
25 declaracionVariable → tipoContenedor t_identificador

```

Listado 3.13: Reglas de producción correspondientes a las declaraciones con y sin asignación.



De la definición del símbolo `declaracionVariable` en la gramática mostrada en el Listado 3.13, se pueden inferir los siguientes atributos:

- **Cadenas que almacenen el tipo de dato:** En caso de que la declaración use un tipo primitivo solo será necesario almacenar en una cadena esto, sin embargo en el caso de los contenedores, además del identificador del contenedor, también será necesario almacenar los identificadores de los tipos contenidos.
- **Una cadena representando el identificador.**

### Estructuras iterativas y de control de flujo

Dentro de esta categoría, se incluyen las estructuras definidas por los símbolos terminales `if`, `for` y `while`. Es importante hacer mención al hecho de que estas estructuras comparten un símbolo no terminal que aún está por definir en las subsecuentes secciones, pues los símbolos que lo conforman aún no son definidos, se trata del símbolo no terminal `bloqueCodigo` el cual almacena sus atributos dentro de una clase, también por definir.

La gramática mostrada en el Listado 3.14 ilustra los símbolos que conforman las estructuras `while`, `for` e `if`, entre las cuales destaca la definición de esta última, pues debido a la forma que toma la regla que la define, puede tener cero, uno o más símbolos `else` asociados; este comportamiento será corregido en la fase de implementación, donde se verificará que el símbolo `if` actual no tenga un `else` asociado antes de asignarle uno nuevo.

```

1 ifStatement → t_if t_opregb expresionBooleana t_cloregb
    bloqueCodigo
2 ifStatement → ifStatement t_else bloqueCodigo
3
4 forStatement → t_for t_opregb declaracionVariable t_semicolon
    expresionBooleana t_semicolon expresionAritmetica t_cloregb
    bloqueCodigo

```

```
5
6 whileStatement → t_while t_opregb expresionBooleana t_cloregb
    bloqueCodigo
```

Listado 3.14: Reglas de producción correspondientes a las estructuras iterativas y de control de flujo.

Es posible hacer una distinción entre los símbolos definidos en la gramática mostrada en el Listado 3.14 en función de si representan estructuras iterativas o de control de flujo. Los atributos que definen a las estructuras iterativas son:

- Una condición de continuación en forma de una expresión booleana, de la cual se tendrá la referencia en memoria.
- Una expresión aritmética representando la operación o *step* a realizar por cada iteración.
- Un apuntador al objeto que contiene los atributos del bloque de código asociado.

Mientras que los atributos que definen a la única estructura de control de flujo (*if*) son:

- Una condición en forma de una expresión booleana, de la cual se tendrá la referencia en memoria.
- Un apuntador al objeto que contiene los atributos de los bloques de código asociados.

### Bloques de código

Se define un bloque de código como una colección de instrucciones delimitadas por llaves. Son utilizadas en la definición de funciones y en el uso de estructuras iterativas y de control de flujo para agrupar un conjunto de instrucciones asociadas. La gramática plasmada en el Listado 3.15 muestra como las instrucciones que pueden estar contenidas pueden ser estructuras iterativas y de control (*while*, *for*, *if*, *else*) o pueden ser declaraciones, métodos

de entrada o salida, expresiones aritméticas o booleanas; todas delimitando su final por un punto y coma (;).

```
1 instruccion → expresionAritmetica t_semicolon
2 instruccion → expresionBooleana t_semicolon
3 instruccion → entradaSalida t_semicolon
4 instruccion → declaracionVariable t_semicolon
5 instruccion → t_return expresionAritmetica t_semicolon
6 instruccion → t_return expresionBooleana t_semicolon
7 instruccion → forStatement
8 instruccion → whileStatement
9 instruccion → ifStatement
10
11 instrucciones → instruccion
12 instrucciones → instrucciones instruccion
13
14 bloqueCodigo → t_opcrlyb instrucciones t_clocrlyb
```

Listado 3.15: Reglas de producción correspondientes a los bloques de código.

En cuanto a los atributos que definen a un bloque de código, es posible inferir que aquello que lo caracteriza son las instrucciones que contiene, por lo que la clase que lo representa contendrá referencias a todas aquellas instrucciones contenidas en él.

### Definición de funciones

La estructura de la definición de funciones contempladas por el prototipo es ilustrada por la gramática plasmada en el Listado 3.16. Donde se recurrió a la creación de un símbolo no terminal para definir de manera recursiva los argumentos, los cuales pueden ser cero o más instancias del símbolo `declaracionVariable` separados por comas (,).

Haciendo uso de las reglas de producción descritas en la gramática mostrada en el Listado 3.16, se definen los atributos que caracterizan una definición de una función como el tipo de dato de retorno denotado por el símbolo `tipoPrimitivo`, una cadena indicando el identificador y un apuntador a los atributos del bloque de código asociado.

```

1 listaArgumentos → ε
2 listaArgumentos → declaracionVariable
3 listaArgumentos → listaArgumentos t_comma declaracionVariable
4
5 declaracionFuncion → tipoPrimitivo t_identificador t_opregb
   listaArgumentos t_cloregb bloqueCodigo

```

Listado 3.16: Reglas de producción relacionadas a la declaración de funciones.

### Sobre el símbolo `programa`

Una vez creados todos los símbolos que pueden estar contenidos en el código fuente, es posible definir un símbolo que encapsule en su totalidad al programa. Como se muestra en la gramática plasmada en el Listado 3.17, se consideró a un programa como una colección de declaraciones de funciones y cabeceras, estas últimas siendo la directiva `using namespace std` y directivas de tipo `#include`.

Debido a que aún no se considera que el prototipo soporte el uso de directivas, todo aquello contenido en el símbolo `cabeceras` será ignorado. Por lo tanto, el único atributo que define a un programa es una lista de apuntadores a las clases que contienen a los atributos de las definiciones de funciones contenidas en éste.

```

1 cabecera → t_include t_hardstr
2 cabecera → t_nspacestd
3
4 cabeceras → ε

```

```
5 cabeceras → cabeceras cabecera
6
7 declaracionesFuncion → ε
8 declaracionesFuncion → declaracionesFuncion declaracionFuncion
9
10 programa → cabeceras declaracionesFuncion
```

Listado 3.17: Reglas de producción correspondientes a la definición del programa.

### 3.4.3. Diseño del módulo de análisis de jerarquía y dependencia

El propósito de este módulo es determinar si no existen dependencias circulares entre las funciones declaradas dentro del código y determinar un orden de análisis que seguirá el siguiente módulo. Esto lo logra a partir de la aplicación de algoritmos de ordenamiento topológico y de conteo de componentes fuertemente conectados sobre el árbol abstracto de sintaxis.

#### Sobre el análisis de jerarquía

La complejidad temporal asociada a las funciones sin importar si son definidas por el usuario o por el estándar del lenguaje, influyen en la complejidad temporal de las funciones donde éstas son llamadas.

Considere el código mostrado en el Listado 3.18, donde las funciones `a()`, `b()`, `c()` y `d()` son definidas. Es posible observar que para terminar de ejecutarse la función `c()`, ésta requiere llamar a las funciones `a()` y `b()`, las cuales tienen una complejidad temporal asociada e impactarían aquella de la función que las llama. Por lo que para calcular la complejidad de un programa en su totalidad, es necesario priorizar las funciones que no realicen llamadas a funciones definidas por el usuario y seguir con aquellas que dependan de éstas hasta llegar a

la función `main()`.

```
1
2
3 void a(){
4     //hace algo
5 }
6 void b(){
7     //hace algo
8 }
9 void c(){
10    //hace algo
11    //...
12    a();
13    b();
14    //...
15 }
16 void d(){
17    //hace algo
18 }
```

Listado 3.18: Ejemplo de un código con dependencias entre funciones.

Este procedimiento debe ser realizado utilizando el árbol abstracto de sintaxis obtenido durante la etapa de análisis sintáctico, a continuación se muestra una representación simplificada de dicha estructura para el código plasmado en el Listado 3.18.

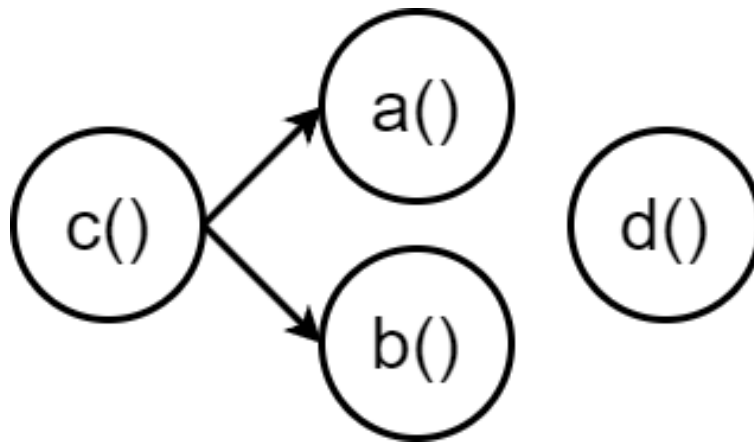


Figura 3.3: Representación simplificada del AST correspondiente al código mostrado en el Listado 3.18.

Una vez descrito el problema a resolver en términos generales y la representación a utilizar, es posible establecer que el problema es una instancia del problema de planificación de trabajos, cuya solución modificada para este caso de uso, consiste en la creación de un ordenamiento topológico de las funciones llamadas dentro del AST.

Este ordenamiento puede ser creado a través de la implementación de una variedad de algoritmos, sin embargo debido a la estructura del AST, se ha decidido hacer uso del hecho que el reverso de la lista resultante de un recorrido post-orden sobre un grafo acíclico dirigido (DAG, por sus siglas en inglés) es un ordenamiento topológico válido, por lo que la implementación del algoritmo mostrado en el Listado 3.19 bastaría para resolver este problema.

```
1 visitados = {} //Se declara una lista global vacía de nodos
   visitados.
2 función DFS_save(Grafo, nodoActual, listaPostOrden){
3     visitados+ = nodoActual
4     for(vecino_de_nodoActual en Grafo){
5         if(vecino_de_nodoActual NO esta en visitados){
6             DFS_save(Grafo, inicial, listaPostOrden)
```

```
7         }
8     }
9     listaPostOrden+ = nodoActual
10    return orden_recorrido
11 }
12 función Ordenamiento_Topológico(Grafo, nodoActual, visitados,
    nodos){
13     orden_recorrido = {}//El orden se declara como una lista
    vacía.
14     visitados{}//Se declara una lista vacía de nodos visitados.
15     for(nodo en nodos){
16         if(nodo NO esta en visitados){
17             orden_recorrido+ = DF S_save(Grafo, inicial, {})
18         }
19     }
20     orden_recorrido = invertir(orden_recorrido)
21     return orden_recorrido
22 }
```

Listado 3.19: Algoritmo DFS para encontrar un ordenamiento topológico.

Es importante hacer mención que el algoritmo contenido en el Listado 3.19 asume que el grafo es acíclico y dirigido, esta última característica está garantizada debido al hecho de que los grafos representan llamadas de funciones en forma de AST, sin embargo existe la posibilidad de que el grafo contenga por lo menos un ciclo, por lo tanto fue necesario introducir una rutina que garantice que el grafo de llamadas a funciones sea acíclico.



### Sobre el análisis de dependencias

En el apartado anterior, se mencionó que el grafo formado por las llamadas a funciones debe ser acíclico para ser considerado *analizable* por el prototipo. Considere el código mostrado en el Listado 3.20, en él son definidas las funciones `a()` y `b()`, las cuales dependen mutuamente entre ellas para terminar su ejecución, si bien existe la posibilidad de que la complejidad aportada por estas funciones sea analizable por métodos convencionales, el método algorítmico propuesto por el prototipo no tiene contemplada una forma consistente de lograrlo, por lo tanto es necesario descartar los códigos que contengan dependencias circulares.

```
1 {cpp}
2
3 void a(){
4     //...
5     b();
6     //...
7 }
8 void b(){
9     //...
10    a();
11    //..
12 }
```

Listado 3.20: Ejemplo de un código con dependencias circulares entre dos funciones.

De una manera similar al análisis de jerarquía descrito en la Sección 3.4.3, se tomó en cuenta la representación ya existente del programa en forma del AST, el cual fue simplificado durante la etapa de diseño, considere el código mostrado en el Listado 3.21 y su grafo de llamadas ilustrado por la Figura 3.4.

```
1
2 void a(){
```

```
3      //...
4      b();
5      //...
6  }
7  void b(){
8      //...
9      c();
10     //...
11 }
12 void c(){
13     //...
14     a();
15     //...
16 }
17 void d(){
18     //...
19     //hace algo
20     //...
21 }
22 int main(){
23     //...
24     a();
25     d();
26     //...
27 }
```

Listado 3.21: Ejemplo de un código con dependencias circulares entre tres funciones.

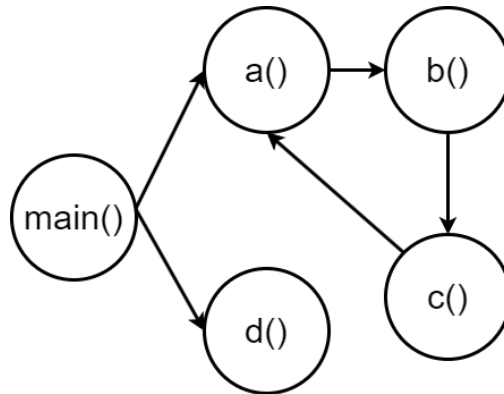


Figura 3.4: Representación simplificada del AST correspondiente al código mostrado en el Listado 3.21.

El código mostrado en el Listado 3.21, no sería considerado analizable por el prototipo, debido a que para calcular la complejidad que aporta la función `a()` a `main()`, es necesario conocer la complejidad de `b()`, la cual no puede ser obtenida sin antes conocer la complejidad de `c()`, la cual depende de la complejidad aportada por `a()`. Por lo tanto, es necesario implementar un algoritmo capaz de identificar ciclos en el grafo formado por las llamadas a funciones dentro del código a analizar, una manera de lograr esto es a través del conteo de los Componentes Fuertemente Conectados (SCC, por sus siglas en inglés).

Un componente fuertemente conectado se define como un subconjunto de nodos en el grafo para el cual existe un camino entre cualesquiera dos nodos dentro de este subconjunto, es decir un subgrafo cuyos nodos forman un ciclo. Usando la definición y el hecho de que un grafo analizable no debe contener ciclos, es posible inferir que el grafo de llamadas de un código analizable debe de tener tantos componentes fuertemente conectados como nodos.

Para ilustrar este comportamiento, considere la Figura 3.5, la cual corresponde al grafo de llamadas del código mostrado en el Listado 3.21, donde fueron coloreados los tres componentes fuertemente conectados del grafo, debido a que existen 3 SCCs correspondientes a 5 nodos, se concluye que el código representado por este grafo no es analizable por el prototipo.

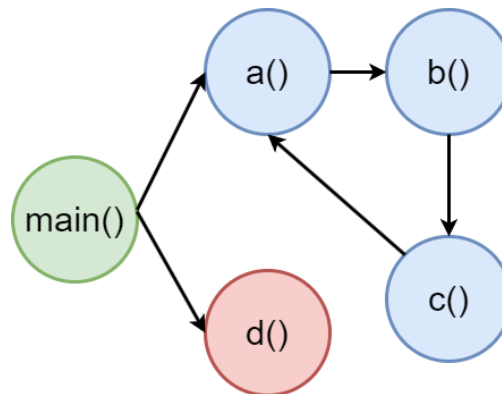


Figura 3.5: Componentes fuertemente conectados en el AST simplificado correspondiente al código mostrado en el Listado 3.21.

Para la identificación de SCCs fueron considerados los algoritmos de Kosaraju-Sharir y aquel propuesto por Tarjan, ambos resultantes de la modificación del algoritmo de Búsqueda en Profundidad (DFS, por sus siglas en inglés).

El Algoritmo contenido en el Listado 3.22, muestra como el algoritmo de Kosaraju-Sharir explota el hecho de que un grafo y su traspuesta comparten componentes fuertemente conectados, por lo que parte obteniendo una lista invertida de los nodos visitados en post-orden. Una vez obtenida esta lista, invierte todas las aristas del grafo e inicializa el valor de visitado de cada nodo en falso, para después verificar que cada nodo en la lista haya sido visitado, aquellos que no lo han sido son considerados el inicio de un SCC por lo que se agrega uno al contador de estos y se hace una llamada a `DFS()` para marcar como visitados a todos los nodos de este componente fuertemente conectado.

```
1 función kosaraju_sharir(Grafo, nodos){ //Donde grafo es una lista
    de
2 adyacencia y nodos es el conjunto que contiene todos los vértices
    en Grafo.
3     orden_recorrido = {}// El orden se declara como una lista
    vacía.
```

```
4     for(vértice inicial en nodos){
5         if(visitado[inicial] == false){
6             orden_recorrido+ = DF S_save(Grafo, inicial)//La
7             lista de orden es
8             poblada a través de los llamados a DFS_save
9         }
10
11     orden_recorrido = invertir(orden_recorrido)
12     Grafo_transpuesto = invertir_grafo(Grafo)
13     componentes_fuertemente_conectados = 0
14     visitados.reset() //El status de visitados se marca en falso
15     para todos los nodos
16     for(vértice inicial en orden_recorrido){
17         if(visitado[inicial] == false){
18             DFS(Grafo, inicial)
19             componentes_fuertemente_conectados+= 1
20         }
21     }
```

Listado 3.22: Algoritmo de Kosaraju-Sharir para encontrar componentes fuertemente conectados.

El algoritmo de Kosaraju-Sharir fue considerado como el algoritmo a implementar para el análisis de dependencia en etapas tempranas del prototipo, sin embargo una vez definidas las clases que formarían el AST, se consideró que la creación del grafo transpuesto sería no trivial, pues las clases que definen a cada elemento deberían cambiar para poder apuntar a elementos sintácticos a los cuales no apuntarían normalmente.

Por lo tanto, se recurrió a la utilización del Algoritmo de Tarjan definido en el Listado 3.23 como una alternativa que requiere modificar en menor medida las clases del AST. El algoritmo de Tarjan parte de una modificación al algoritmo de DFS, donde ahora se requiere que cada función cuente con los atributos `scc_id` y `low_link_value`, el primero es un identificador entero único, asignado en función del orden en que el nodo fue accedido por el DFS, mientras que `low_link_value` es el mínimo entre los valores `low_link_value` de las funciones llamadas por la función actual y el `scc_id` propio.

```
1 id_disponible = 0
2 Pila = {}
3 SCC = 0
4 función DFS(Grafo, actual){
5     inicial.scc_id = id_disponible
6     inicial.low_link_value = id_disponible
7     id_disponible+ = 1
8     Pila.push(actual)
9     for(vecino en Grafo[actual]){
10         if(vecino.scc_id == NULO){
11             DFS(Grafo, vecino)
12             inicial.low_link_value = min(inicial.low_link_value,
vecino.low_link_value)
13         }else{
14             if(vecino esta en Pila){
15                 inicial.low_link_value = min(inicial.
low_link_value, vecino.low_link_value)
16             }
17         }
18     }
19     while(Pila.top() != actual){
```

```
20     Pila.pop()
21 }
22 if(inicial.scc_id == inicial.low_link_value){
23     SCC+ = 1
24 }
25 }
26
27 función Tarjan(Grafo, nodos){
28     for(inicial en nodos){
29         inicial.scc_id = NULO
30         inicial.low_link_value = NULO
31     }
32     for(inicial en nodos){
33         if(vecino.scc_id == NULO){
34             DFS(Grafo, vecino)
35             inicial.low_link_value = min(inicial.low_link_value,
vecino.low_link_value)
36         }
37     }
38 }
```

Listado 3.23: Algoritmo de Tarjan para encontrar componentes fuertemente conectados.

Una vez implementado el algoritmo de Tarjan, bastará con verificar por cada definición de función dentro del AST que los nuevos atributos `scc_id` y `low_link_value` coincidan.

### Unión de los análisis de jerarquía y dependencia

Finalmente, es posible observar que ambos algoritmos son modificaciones del algoritmo de recorrido en profundidad DFS, por lo tanto es posible unir ambas implementaciones del método de tal forma que resulte en uno solo, esto se puede lograr creando un arreglo global de orden de visita dentro del algoritmo de Tarjan al cual se vayan agregando los nodos en post-orden de acuerdo a la función DFS, una vez obtenidos solo basta con invertir la lista al finalizar el método Tarjan, reduciendo de esta manera la redundancia inherente a la creación de dos métodos DFS().

#### 3.4.4. Diseño del módulo de análisis de complejidad

El propósito de este módulo es estimar la complejidad temporal en el peor caso del programa que ha sido analizado hasta esta etapa, esto a partir del árbol abstracto de sintaxis obtenido por el módulo de análisis sintáctico y del ordenamiento topológico resultante del análisis de jerarquía y precedencia.

#### Sobre la representación de la complejidad temporal

Se decidió que la complejidad temporal asociada a las instrucciones sea expresada en notación *big Oh*, que para fines del presente proyecto se decidió generalizar su obtención en la forma de polinomios que ignoran factores constantes y que reducen en la medida de lo posible sumandos con factores semejantes. Se profundizará sobre este comportamiento a lo largo de este apartado.

La complejidad algorítmica dentro del presente proyecto fue representada como un polinomio, el cual contiene sumandos, cada sumando esta caracterizado por una lista de factores que lo conforman siendo este elemento el más elemental en esta representación.



Entre los factores que podrían formar parte de la complejidad temporal estimada por el prototipo se encuentran aquellos con una de las siguientes formas  $k$ ,  $(\log id)^c$ ,  $id^c$  donde  $k$  es una constante arbitraria,  $id$  es el identificador de alguna variable y  $c$  es un valor numérico arbitrario que denota un exponente. Por lo tanto, se infiere que una clase que representa un factor dentro del polinomio de complejidad debe de contener los siguientes atributos y métodos:

- Un entero denotando si se trata de un factor constante, logarítmico o lineal/exponencial, debido a que solo fueron definidos estos tres tipos de factores se decidió asignarles los identificadores numéricos cero, uno y dos respectivamente.
- Una entero denotando el exponente para los factores de tipo logarítmico y lineal/exponencial.
- Una cadena de caracteres denotando el identificador sobre el cual se realizan las operaciones para los factores de tipo logarítmico y lineal/exponencial.
- Un operador que denote si dos factores son iguales, dos factores son considerados iguales si comparten tipo de factor, exponente e identificador.
- Un método que denote si dos factores son multiplicables, un factor constante puede ser multiplicado por cualquier factor, el resto de los factores solo pueden ser multiplicados por factores constantes o aquellos que compartan tipo e identificador.
- Un método que multiplique dos factores multiplicables, el resultado debe de seguir la ley de los exponentes.

Como ya ha sido mencionado anteriormente, un sumando está caracterizado por una lista de factores que lo conforma, a continuación se definen los métodos que deben ser capaz de ser efectuados con un sumando:

- Un método que le permita a un sumando ser multiplicado por otro.
- Un método que permita verificar si dos sumandos son iguales. Un sumando es considerado igual a otro si cuenta con los mismos factores, omitiendo aquellos que sean de tipo constante.

De una manera similar, se describen los métodos y atributos que caracterizan a la clase complejidad:

- Una lista con los sumandos que conforman el polinomio de complejidad.
- Un método que sume dos complejidades.
- Un método que multiplique dos complejidades.
- Un método para reducir términos semejantes.
- Un método para imprimir la complejidad caracterizada por la lista de sumandos.

### **Sobre la estructura y métodos del analizador de complejidad**

Se decidió que el analizador de complejidad realice dos tareas principales: mantener registro de la complejidad temporal de cada función y analizar la complejidad de cada elemento sintáctico dentro de la definición del programa. La primera tarea es lograda a través de la creación de una estructura de datos asociativa que mantenga una relación identificador-complejidad, pues al no permitirse el polimorfismo en los códigos a analizar de acuerdo a lo especificado en la Sección 3.3.1 es eliminada la posibilidad de registros duplicados. La segunda tarea se logra definiendo métodos para el análisis de definiciones de funciones iterativas/recursivas, estructuras de control de flujo (`if`), estructuras iterativas, declaración de variables, expresiones booleanas y de expresiones aritméticas; cada definición cuenta con un comportamiento único, el cual será definido en los apartados restantes de esta sección.

### Sobre el registro de la complejidad de funciones predefinidas

Posterior a la creación de la estructura de datos asociativa que mantiene la relación identificador-complejidad, se inicializará con la complejidad asociada con las funciones definidas dentro del lenguaje, esta inicialización será realizada siguiendo lo especificado en la Tabla 3.2.

Tabla 3.2: Complejidades asociadas a los métodos definidos por el lenguaje, donde  $n$  es la cardinalidad del elemento pasado como argumento o en su defecto la del contenedor.

Contenedor	Función	Complejidad
Ninguno	cin, cout, swap, min, max	$O(1)$
Ninguno	reverse	$O(n)$
Ninguno	sort	$O(n \log(n))$
vector/string	push_back, pop_back, size, [], front, back, at	$O(1)$
vector/string	clear, shrink_to_fit	$O(n)$
map/unordered_map	insert, []	$O(\log(n))$
map/unordered_map	clear	$O(n \log(n))$
set/multiset	size	$O(1)$
set/multiset	insert, erase, lower_bound, find	$O(\log(n))$
set/multiset	clear	$O(n \log(n))$
set	count	$O(\log(n))$
multiset	count	$O(n)$
queue	size, push, pop, front, back	$O(1)$
stack	size, push, pop, top	$O(1)$
arreglos tipos primitivos	[]	$O(1)$

### Sobre el análisis de complejidad de expresiones aritméticas

Una expresión aritmética puede tener dos efectos dentro del análisis propuesto: aportar complejidad temporal y modificar la tasa en la que crece una variable. De acuerdo a lo discutido durante la definición de los factores dentro de la complejidad temporal, se decidió asumir que una variable solo puede crecer en esos ordenes de magnitud, por lo que es necesario crear una estructura de datos asociativa que mantenga una relación identificador-nivel de anidamiento-tasa de crecimiento, donde este ultimo es representado por identificadores numéricos similares a los definidos para los factores (0 para crecimiento constante, 1 para crecimiento lineal, 2 para crecimiento exponencial).

Por defecto, se consideró que una expresión aritmética aporta una complejidad constante, complejidad que solo puede cambiar si la expresión aritmética en cuestión contiene una llamada a una función o un acceso a una posición en un arreglo/estructura de datos asociativa, en caso de contener alguno de estos elementos la complejidad resultante es la suma de estos.

En cuanto a la tasa de crecimiento, se consideró que un cambio en la tasa de crecimiento de una variable arbitraria solo puede suceder dentro de una asignación ( $=$ ,  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ), las condiciones bajo las que son definidos los ordenes de magnitud en cuanto al crecimiento de estas variables son descritas a continuación:

- Si el operador de asignación es de la forma  $*=$ ,  $/=$  entonces invariablemente la tasa de crecimiento/decrecimiento de la variable en cuestión es exponencial.
- Si el operador de asignación es de la forma  $+=$ ,  $-=$  la tasa de crecimiento/decrecimiento es lineal a menos de que del lado derecho de la asignación se encuentre por lo menos una ocurrencia del identificador de la variable que está siendo modificada.
- Si el operador de asignación es de la forma  $=$  la tasa de crecimiento/decrecimiento se asume lineal. Si del lado derecho de la asignación existen dos ocurrencias del identificador de la variable modificada o si el lado derecho de la asignación corresponde a

una operación de multiplicación o división, se asume que la tasa de crecimiento/decrecimiento es exponencial. Finalmente, si a la variable le es asignado el valor de otra variable, entonces la tasa de crecimiento de la variable modificada es igual a la de la variable a la que fue igualada.

### Sobre el análisis de complejidad de expresiones booleanas

De una manera similar a las expresiones aritméticas, las expresiones booleanas pueden aportar complejidad temporal y a su vez afectar el cómo aportan complejidad temporal las estructuras iterativas y funciones recursivas.

La complejidad temporal aportada por una expresión booleana es igual a la suma de la complejidad temporal de los elementos que la conforman, sean otras expresiones booleanas o expresiones aritméticas. La forma en la que una expresión booleana afecta la complejidad aportada por otras estructuras, es a través del mantenimiento de un estado, por ejemplo una estructura iterativa sigue ejecutándose siempre y cuando la expresión booleana que contiene como argumento de continuación evalúe verdadero. Debido a este comportamiento, se decidió añadir un atributo de tipo complejidad a las expresiones booleanas, la cual denota la tasa en la que se espera que la expresión cambie su estado.

Como se discutió en la Sección 3.4.2, una expresión booleana puede estar conformada por dos expresiones aritméticas unidas por un operador de comparación ( $<$ ,  $<=$ ,  $==$ ,  $>$ ,  $>=$ ) o por dos expresiones booleanas unidas por los operadores  $\&\&$ ,  $||$ . Una expresión booleana resultante de la unión de dos expresiones aritméticas tiene una complejidad de crecimiento descrita en función de un identificador del lado izquierdo de la expresión para los operadores  $<$ ,  $<=$  y del lado derecho para los operadores  $==$ ,  $>$ ,  $>=$ . El orden en que crece este tipo de expresión se asume lineal a menos de que algún identificador en cualesquiera de los lados de la expresión presente una tasa de crecimiento exponencial. Mientras tanto, una expresión booleana resultante de la unión de dos expresiones booleanas a través de los operadores  $\&\&$ ,  $||$

tiene una complejidad de crecimiento igual a la complejidad menor de las dos expresiones booleanas que la conforman para el primer operador y una complejidad igual a la máxima para el caso del operador `||`.

### **Sobre el análisis de complejidad de declaraciones**

Una declaración se asume como un proceso con una complejidad constante, esto de acuerdo al modelo RAM generalmente usado para el análisis manual de complejidad. En el contexto del lenguaje especificado por la gramática descrita en la Sección 3.4.2, una declaración puede contener una inicialización que asigna a la variable en cuestión un valor obtenido por una expresión aritmética, la evaluación de dicha expresión aporta complejidad al programa analizado, por lo tanto la complejidad de dicha expresión es sumada a la complejidad total de la declaración que la contiene.

### **Sobre el análisis de complejidad de las estructuras de control de flujo**

Una estructura de control de flujo (`if`) fue definida en la Sección 3.4.2 como una estructura que contiene una condición y uno o dos bloques de código, por lo tanto la complejidad que este aporta es igual a la suma de la complejidad aportada por la condición más la complejidad de los elementos sintácticos contenidos dentro de los bloques que contiene.

### **Sobre el análisis de complejidad de las estructuras iterativas**

De una manera similar a la estructura `if`, la complejidad aportada por una estructura iterativa es igual a la suma de las complejidades aportadas por sus argumentos en caso de estar definidos (declaración de variable de control, argumento de continuación y la operación de incremento entre iteraciones) más la complejidad aportada por los elementos sintácticos contenidos en el bloque de código asociado a la estructura. Pero a diferencia de la estructura

`if`, esta complejidad es multiplicada por la complejidad de la condición de continuación para obtener la complejidad total del ciclo.

### **Sobre el análisis de complejidad de una declaración de una función no recursiva**

Se asumió que la complejidad aportada por una función en cuya declaración no se realizaron llamadas a otra instancia de sí misma, es igual a la suma de las complejidades de los elementos sintácticos contenidos en el bloque que le fue asociado.

### **Sobre el análisis de complejidad de una declaración de una función recursiva**

Para efectos del análisis realizado por el presente prototipo, se consideró que el calculo de la complejidad aportada por una función recursiva es análogo al realizado para las estructuras iterativas. Donde primero son sumadas las complejidades de los elementos sintácticos contenidos en el bloque asociado, asignando una complejidad constante a la llamada recursiva en cuanto sea encontrada.

La tasa de crecimiento de los argumentos de la función es obtenida a través de la llamada recursiva, donde cada expresión aritmética en la lista de argumentos es convertida en una expresión donde se iguala el argumento representado a la expresión en cuestión, esta expresión es entonces analizada para obtener la tasa de crecimiento a través del análisis definido para las expresiones aritméticas.

Una vez obtenida la tasa de crecimiento de las variables en la lista de argumentos, es ubicada la estructura `if` que contiene el retorno correspondiente al caso base y es calculada la complejidad de crecimiento de su condición, la cual es análoga a la complejidad de crecimiento de la condición de continuación de una estructura iterativa y, de una manera similar a ésta, es multiplicada por la complejidad total de los elementos contenidos en el bloque de código asociado para obtener la complejidad de la función analizada.

## 3.5. Implementación

A lo largo de la presente sección se presenta lo referente la implementación de los módulos que conforman al prototipo.

### 3.5.1. Implementación del módulo de análisis léxico

#### Acerca de Lex

Para la implementación del módulo de análisis léxico se decidió utilizar Lex v 2.6.4, una herramienta para la generación de *scanners*. Un programa de Lex consiste de tres partes principales:

```
1 DEFINICIONES
2 %%
3 REGLAS
4 %%
5 SUBROUTINAS DEL USUARIO
```

Listado 3.24: Estructura de un programa de Lex.

En la sección de definiciones, es posible realizar dos acciones distintas:

- Incrustar código de C++, esto es posible colocando el código entre `%{ ... %}`. Esto resulta particularmente útil si el código ejecutado por los *tokens* requiere de librerías externas, llevar variables globales o definir funciones.
- Asignar a ciertos patrones un identificador, de manera similar a la directiva `#define` de C++, es posible definir expresiones regulares y asignarles un identificador para su uso futuro.

En la sección de reglas, se definen los *tokens* a detectar. Cada definición es de la forma



Expresión regular {código a ejecutar si la expresión se cumple}. Dentro del código a ejecutar se tiene acceso a las variables `char* yytext` e `int yyleng`, representativas de la palabra detectada y su longitud, respectivamente.

En la sección de subrutinas del usuario, se define una función `int main()` que permite al código generado por Lex ser ejecutado de manera independiente al analizador sintáctico, en caso de no ser definida, la función `int main()` por defecto será implementada, esto es ilustrado en el código plasmado en el Listado 3.25.

```
1 int main()
2 {
3     yylex();
4     return 0;
5 }
```

Listado 3.25: Función `main` implementada por defecto por Lex.

### Creación de nodos.h

De acuerdo a lo discutido en la Sección 3.4.1, fue creada la clase `cl_atributos_tokens` conteniendo los atributos representativos de cada token en el lenguaje y descrita en el código mostrado en el Listado 3.26.

```
1 class cl_atributos_tokens{
2 public:
3     int token_id;
4     int line_number;
5     long long int value_int;
6     long double value_double;
7     std::string value_string;
8     char value_char;
9 };
```

Listado 3.26: Declaración de la clase `cl_atributos_tokens`.

Si bien, para el funcionamiento del analizador léxico, hubiese bastado con definir la clase dentro de la sección de definiciones, será necesario para el correcto funcionamiento del analizador sintáctico la definición de la clase, por lo que se decidió crear un *header file* llamado `nodos.h` que contiene la definición de las clases necesarias para estos dos módulos.

### Sección de definiciones

Como se muestra en el código plasmado en el Listado 3.27, en este apartado del programa fueron incluidas las librerías `nodos.h`, que contiene la definición de la clase con los atributos de cada *token* y `analizadorSintactico.hpp`, la cual es generada por el analizador sintáctico y contiene una unión `yyval` con atributos a los cuales el analizador léxico y el analizador sintáctico podrán acceder, además de contener definiciones que el analizador léxico debe seguir para ser compatible con el analizador sintáctico. También fueron declaradas las variables globales `line_number` y `token_id`, las cuales son usadas para llevar un registro de la línea que esta siendo analizada y la cantidad de *tokens* que fueron aceptados hasta el momento. Además, se especificó que la función `yywrap()`, declarada por Lex para definir el comportamiento del analizador al finalizar de leer la entrada, tiene una interfaz de C.

```
1 %{  
2     #include "nodos.h"  
3     #include "analizadorSintactico.hpp" // Salida de bison  
4     extern "C" int yywrap() { }  
5     int line_number = 0;  
6     int token_id = 0;  
7 }%
```

Listado 3.27: Código de C++ incluido en la sección de definiciones.

Dentro de esta sección, también se definieron los patrones monomorfos y ciertos caracteres especiales dentro de nuestro alfabeto tal como letras o dígitos, asignándoles identificadores en mayúsculas, esto para mejorar la legibilidad de las reglas en el siguiente apartado. En

el código plasmado en el Listado 3.28 se muestran algunas de estas definiciones con fines demostrativos.

```
1 DEF "#define"
2 SET "set"
3 MULTiset "multiset"
4 SUM \+
5 SUB -
6 GT ">"
7 NTEQ "!="
8 CLOSQRB \]
9 OPCRLYB \{
10 CLOCRLYB \}
11 WHTSPCE [\t\r]
12 LETRA [A-Za-z]
13 DIGITO [0-9]
```

Listado 3.28: Algunas definiciones de patrones monomorfos.

## Sección de reglas

En esta sección deben de ser definidas las reglas que eventualmente dividen a la entrada en *tokens* digeribles por un algoritmo. Como se muestra en el código contenido en el Listado 3.29, una regla está dividida en dos partes separadas por un espacio, una expresión regular y el código que será ejecutado en caso de que una palabra dada cumpla con lo establecido en la expresión. Es importante resaltar que debe de existir un salto de línea para establecer el fin de una regla, de otra manera el comportamiento de Lex puede llegar a ser impredecible al intentar separar una definición de la otra.

```
1 {REGEX1} {
2     ...
3     //código
```

```
4     ...
5 }
6 {REGEX2} {
7     ...
8     //código
9     ...
10 }
```

Listado 3.29: Sintaxis de las reglas en Lex.

Debido a que los *tokens* monomorfos solo corresponden a una sola palabra, como fue discutido en la Sección 3.4.1, solo es necesario mantener los atributos de `line_number` y `token_id`, pues el tipo de *token* es lo suficientemente descriptivo como para inferir la palabra identificada. Por lo que todas las reglas de los tokens monomorfos fueron definidas usando las definiciones creadas en la Sección 3.5.1 y ejecutando un código similar al mostrado en el código plasmado en el Listado 3.30, donde los atributos de cada *token* son almacenados en un objeto de tipo `cl_atributos_tokens` contenido en la variable global `yylval`.

```
1 {MULTISET} {
2     yylval.atributos = new cl_atributos_tokens;
3     token_id++;
4     yylval.atributos->token_id = token_id;
5     yylval.atributos->line_number = line_number;
6
7     return t_multiset;
8 }
```

Listado 3.30: Ejemplo de la implementación de una regla correspondiente a un *token* monomorfo.

Cabe destacar que los tokens polimorfos representan la mayor variedad en cuanto a los patrones que los representan, así como de sus atributos.

Como se muestra en el código plasmado en el Listado 3.31, los comentarios pueden ser

aceptados sin la necesidad de retornar un *token*, por lo que solo basta iterar sobre la cadena identificada para contar la cantidad de saltos de línea que fueron descartados junto con el comentario. También cabe resaltar que se crearon patrones similares para desechar espacios en blanco y saltos de línea, manteniendo el número de línea actualizado en el caso de este último.

```

1 [ \t\r] {}/* eat up whitespace */
2 [\n] { line_number++;}
3 {DIV}{2}[^\n]* {line_number++;}/*COMENTARIOS */
4 ["\*"] [^\0']*["*\"] {
5     for(int i = 0;i<yyleng; i++){
6         if(yytext[i]=='\n') line_number++;
7     }
8 }

```

Listado 3.31: Patrones para la identificación y eliminación de comentarios, espacios y saltos de línea.

De manera similar, las reglas de identificación de identificadores, así como de constantes de tipo entero, flotante, carácter y cadena de caracteres, consisten de su expresión regular y código auxiliar que se encarga de traducir el texto aceptado al respectivo tipo que almacenará su valor dentro de `yylval`, como se ilustra en el código mostrado en el Listado 3.32.

```

1 [A-Za-z\_][0-9A-Za-z\_\-]* {
2     yylval.atributos = new cl_atributos_tokens; token_id++;
3     yylval.atributos->token_id = token_id;
4     yylval.atributos->line_number = line_number;
5
6     yylval.atributos->value_string = "";
7     for(int i = 0;i<yyleng; i++){
8         yylval.atributos->value_string+=yytext[i];
9     }
10    return t_identificador;

```

```
11 }
12 {DIGITO}+ {
13     yylval.atributos = new cl_atributos_tokens; token_id++;
14     yylval.atributos->token_id = token_id;
15     yylval.atributos->line_number = line_number;
16
17     yylval.atributos->value_string = "";
18     for(int i = 0; i < yyleng; i++){
19         yylval.atributos->value_string += yytext[i];
20     }
21     yylval.atributos->value_int = stoll(yylval.atributos->value_string);
22     return t_intconst;
23 }
24 {DIGITO}+.{DIGITO}+ {
25     yylval.atributos = new cl_atributos_tokens; token_id++;
26     yylval.atributos->token_id = token_id;
27     yylval.atributos->line_number = line_number;
28
29     yylval.atributos->value_string = "";
30     for(int i = 0; i < yyleng; i++){
31         yylval.atributos->value_string += yytext[i];
32     }
33     yylval.atributos->value_double = stold(yylval.atributos->value_string)
34     ;
35     return t_doubleconst;
36 }
37 {SINGLQUOT}[0-9A-Za-z]{SINGLQUOT} {
38     yylval.atributos = new cl_atributos_tokens; token_id++;
39     yylval.atributos->token_id = token_id;
40     yylval.atributos->line_number = line_number;
41     yylval.atributos->value_char = yytext[1];
42     return t_charconst;
```

```

42 }
43 {DBLQUOTE}[^\"' ]*{DBLQUOTE} {
44     yylval.atributos = new cl_atributos_tokens; token_id++;
45     yylval.atributos->token_id = token_id;
46     yylval.atributos->line_number = line_number;
47
48     yylval.atributos->value_string = "";
49     for(int i = 0; i < yyleng; i++){
50         yylval.atributos->value_string += yytext[i];
51     }
52     return t_hardstr;
53 }

```

Listado 3.32: Patrones para la identificación de identificadores y constantes.

Finalmente, se decidió que la sección de subrutinas del usuario se dejara en blanco, pues es más apropiado que un programa externo a cualquiera de los módulos se encargue de realizar llamadas a estos.

### 3.5.2. Implementación del módulo de análisis sintáctico

#### Acerca de Bison

El analizador sintáctico fue escrito a través de Bison v 3.5.1, el cual es un generador de analizadores sintácticos de tipo *bottom-up*. Para lograr esto, es necesario contar con un analizador léxico escrito en Lex o en cualquier programa cuyo formato sea reconocido por Bison, una vez creado es necesario crear un archivo de gramática Bison, el cual especifica la forma que tomará el analizador sintáctico y toma el formato descrito en el Listado 3.33.

```

1 %{
2 Prologo
3 %}

```

```
4 Declaraciones de Bison
5 %%
6 Reglas de la gramatica
7 %%
8 Epilogo
```

Listado 3.33: Estructura de un programa de Bison.

### Prologo del archivo de gramática

En esta sección se define código de C++ que es usado a lo largo del archivo de gramática y es copiado al inicio del código fuente del analizador sintáctico. Delimitado por los símbolos `%{` y `%}`, en el código plasmado en el Listado 3.34 se muestra como fueron incluidas librerías estándares del lenguaje y la librería `nodos.h`, donde fueron definidas las clases que describen cada uno de los elementos sintácticos del lenguaje.

```
1
2 %{
3 #include "nodos.h"
4 #include <iostream>
5 #include <cstdlib>
6 #include <string>
7 #include <map>
8
9
10 cl_raiz* raiz;
11 void yyerror(const char *s) { printf("Error: %s \n", s); }
12 extern int yylex();
13
14 %}
```

Listado 3.34: Prologo usado en el analizador escrito con Bison.



En esta sección fue creado un apuntador a un objeto que representa el programa en su totalidad, a través de una instancia de la clase `cl_raiz`, la cual será definida durante la sección de reglas de la gramática del analizador. Además, es definida la función `yterror()`, la cual será llamada en caso de errores de sintaxis. Finalmente, a través de la palabra reservada `extern` fue declarada la existencia de la función `yylex()` al compilador, la cual es definida por el analizador léxico y encontrará durante el proceso de compilación.

### Declaraciones de Bison

Durante esta sección se declara el tipo de dato que toma la variable global `yylval` a la cual accede el analizador sintáctico, además son declarados los símbolos terminales y no terminales en conjunto con el tipo de dato que mejor represente sus atributos.

Por defecto, Bison declara la global `yylval` con un tipo `int`, sin embargo es posible modificar este comportamiento al declarar una estructura de tipo `%union` que contenga variables y objetos representado cada uno de los atributos que pueden tomar los símbolos de la gramática.

Durante la creación del analizador léxico, `yylval` contenía inicialmente un apuntador a un objeto `atributos` de la clase `cl_atributos_tokens` y una variable entera simbólica `nada` que sería utilizada para los símbolos sin atributos. Además, como se muestra en el código plasmado en el Listado 3.35, por medio de la rutina de declaración `%token<tipo>simboloTerminal`, fueron declarados los tipos de datos de los símbolos terminales identificados por el analizador léxico. Los únicos *tokens* de tipo `atributos` son aquellos considerados polimorfos, mientras que al resto se les asigna el tipo simbólico `nada`, por simplicidad fue omitida la lista completa de estos últimos *tokens*.

```
1 %union {  
2     cl_atributos_tokens *atributos;  
3     int nada;
```

```
4 }
5 %token<nada> t_float t_cout t_define t_opsqrb t_astk ...
6
7 %token<atributos> t_include t_hardstr t_charconst
8 %token<atributos> t_identificador t_intconst t_doubleconst
```

Listado 3.35: Declaración de los tokens y sus atributos.

En esta sección también fueron definidos los símbolos no terminales y sus características, sin embargo, el autor del presente documento ha decidido enlistar los cambios hechos en esta sección del código fuente de forma progresiva a la definición de las reglas de la gramática, esto en un esfuerzo de mejorar la legibilidad y entendimiento de la implementación del módulo en su totalidad.

## Reglas de gramática

Durante esta sección del código fuente, fueron implementadas en Bison las reglas definidas a lo largo de la Sección 3.4.2 en el orden en que fueron introducidas.

El código mostrado en el Listado 3.36 ilustra la estructura básica de una regla de producción dentro de Bison, donde el símbolo objetivo puede ser logrado a través de distintas combinaciones de símbolos las cuales son separadas por *pipes* (`|`), el fin de dicho listado de derivaciones está denotado por un punto y coma (`;`).

```
1 simboloObjetivo:
2     simbolo1 simbolo2 ... {
3         //Codigo a ejecutar en caso de match
4         $$ = ...
5     }
6     | simboloA simboloB simboloC ...{
7
8     }
```

```
9 ;
```

Listado 3.36: Sintaxis de las reglas de producción en Bison.

Es importante mencionar que dentro del código a ejecutar es posible modificar los atributos del símbolo objetivo a través de la manipulación de la variable `$$` y de cada uno de los símbolos que lo conforman con las variables `$i` donde *i* es un entero entre uno y la cantidad de símbolos.

### Expresiones aritméticas

Para la creación de las reglas que definen a las expresiones aritméticas, se añadieron a `yylval` los atributos enlistados en la estructura `union` dentro del código mostrado en el Listado 3.37, además fueron definidos los símbolos no terminales y su tipo.

```
1 %union{
2     ...
3     cl_expresion_aritmetica* expresion_aritmetica;
4     bool es_exponencial;
5
6     cl_argumentos_llamada *argumentos_llamada;
7 }
8 ...
9 %type<es_exponencial> operadorAritmetico
10 %type<es_exponencial> operadorAsignacion
11 %type<expresion_aritmetica> expresionAritmetica
12 %type<argumentos_llamada> argumentosLLamada
```

Listado 3.37: Declaración de los terminales asociados con el símbolo `expresionAritmetica`.

En la sección de reglas, fueron añadidas las definiciones de los símbolos no terminales `operadorAritmetico` y `operadorAsignacion`, ambos fungiendo como símbolos agrupadores, como se muestra en el código plasmado en el Listado 3.38. Se decidió clasificar estos símbolos en

función de su capacidad de formar expresiones cuyo crecimiento sea exponencial, donde se asume que las expresiones que involucran a los operadores de multiplicación o división tienden a crecer o decrecer de forma exponencial.

```

1 operadorAritmetico:
2     t_sum {$$=false;}
3     | t_sub {$$=false;}
4     | t_div {$$=true;}
5     | t_astk {$$=true;}
6 ;
7 operadorAsignacion:
8     operadorAritmetico t_eq {$$ = $1;}
9     | t_eq{$$ = false; }
10 ;

```

Listado 3.38: Implementación de las reglas de producción correspondientes a los símbolos agrupadores `operadorAritmetico` y `operadorAsignacion`.

De acuerdo a lo discutido en la Sección 3.4.2, se consideró que los argumentos correspondientes a la llamada a una función son una lista de cero o mas expresiones aritméticas, tomando esto en cuenta, se agregó la definición de la clase `cl_argumentos_llamada` a `nodos.h`, donde su único atributo es una lista de apuntadores a las expresiones aritméticas que representan sus argumentos.

```

1 ...
2 class cl_argumentos_llamada{
3 public:
4     vector<cl_expresion_aritmetica*> argumentos;
5 };

```

Listado 3.39: Declaración de la clase `cl_argumentos_llamada`.

De una manera similar, como se muestra en el código plasmado en el Listado 3.40, se definió la lista de argumentos en función de sí misma, donde los casos base denotan el inicio

de la lista y por lo tanto la creación del objeto, mientras que la definición del paso recursivo solamente añade al final de una lista de argumentos ya existente una expresión aritmética.

```

1 ...
2 argumentosLLamada:
3     /*empty*/ {$$ = new cl_argumentos_llamada;}
4     |expresionAritmetica{
5         $$ = new cl_argumentos_llamada;
6         $$->argumentos.push_back($1);
7     }
8     |argumentosLLamada t_comma expresionAritmetica{
9         $1->argumentos.push_back($3);
10    }
11 ;

```

Listado 3.40: Implementación de la regla de producción correspondiente al símbolo `argumentosLLamada`.

Expandiendo sobre los atributos de una expresión aritmética discutidos en la Sección 3.4.2, fue declarada la clase `cl_llamada_funcion`, además fue creada una clase `cl_expresion_aritmetica`, la cual representa los atributos de expresiones aritméticas monolíticas y compuestas. La estructura básica de estas clases, incluyendo sus atributos y métodos, son ilustrados por el código plasmado en el Listado 3.41.

```

1 ...
2
3 class cl_llamada_funcion{
4 public:
5     string identificador;
6     //si es un metodo miembro de una clase
7     bool esta_contenida;
8     string id_contenedor;
9     vector<cl_expresion_aritmetica*> argumentos;
10 };

```

```
11 class cl_expresion_aritmetica{
12 public:
13     cl_expresion_aritmetica* izquierda;
14     cl_expresion_aritmetica* derecha;
15     bool es_exponencial;
16
17     bool es_terminal;
18     bool es_llamada_funcion;
19     cl_llamada_funcion* llamada_val;
20     bool es_constante;
21     long double valor_numerico;
22     bool es_cadena;
23     string cadena;
24     bool es_identificador;
25     string identificador;
26     bool es_posicion_arreglo;
27     bool es_asignacion;
28 };
```

Listado 3.41: Declaración de las clases `cl_llamada_funcion` y `cl_expresion_aritmetica`.

Una vez definida la clase que contiene los atributos del símbolo `expresionAritmetica`, es posible crear las reglas que definen su estructura. En el código plasmado por el Listado 3.42, primero se define que una expresión aritmética monolítica puede ser una constante de cualquier tipo, cuyo valor será almacenado en su atributo y el booleano `es_terminal` es marcado como `true`.

```
1 ...
2 expresionAritmetica:
3     t_intconst {
4         $$= new cl_expresion_aritmetica;
5         $$->es_terminal = true;
6         $$->es_constante = true;
```

```
7     $$->valor_numerico = $1->value_int;
8     }
9     |t_true {
10         $$= new cl_expression_aritmetica;
11         $$->es_terminal = true;
12         $$->es_constante = true;
13         $$->valor_numerico = 1;
14     }
15     |t_false {
16         $$= new cl_expression_aritmetica;
17         $$->es_terminal = true;
18         $$->es_constante = true;
19         $$->valor_numerico = 0;
20     }
21     |t_doubleconst {
22         $$= new cl_expression_aritmetica;
23         $$->es_terminal = true;
24         $$->es_constante = true;
25         $$->valor_numerico = $1->value_double;
26     }
27     |t_hardstr{
28         $$= new cl_expression_aritmetica;
29         $$->es_terminal = true;
30         $$->es_cadena;
31         $$->cadena = $1->value_string;
32     }
33     |t_charconst{
34         $$= new cl_expression_aritmetica;
35         $$->es_terminal = true;
36         $$->es_cadena;
37         $$->cadena = "";
38         $$->cadena+=$1->value_char;
```

```
39 }
```

Listado 3.42: Implementación de las reglas de producción correspondientes a las expresiones aritméticas formadas por una constante.

Si bien, los analizadores generados por Bison resuelven los conflictos de tipo *shift-reduce* al favorecer las operaciones de *shift*, con el fin de obtener un analizador más estable, fue favorecida la definición de las reglas de manera que redujeran la dependencia a este comportamiento.

Considerando que un *token* `t_identificador` puede ser derivado a una expresión aritmética monolítica y formar parte a su vez de una derivación más grande, como es el caso de llamadas a funciones, se decidió definir estas últimas en función del símbolo no terminal `expresionAritmetica` bajo la suposición que se trata de una expresión aritmética monolítica de tipo identificador. Esta decisión de diseño es ilustrada por el código mostrado en el Listado 3.43, donde fueron definidas reglas para las expresiones aritméticas consistentes de accesos a posiciones de un arreglo, llamadas a funciones y llamadas a métodos miembros de manera respectiva.

```
1 ...
2 expresionAritmetica:
3     ...
4     | expresionAritmetica t_opsqrb expresionAritmetica t_closqrb {
5         $$ = new cl_expresion_aritmetica;
6         $$->es_terminal = true;
7         $$->es_identificador = true;
8         $$->identificador = $1->identificador;
9         $$->es_posicion_arreglo = true;
10        delete $1; delete $3;
11    }
12    | expresionAritmetica t_opregb argumentosLLamada t_cloregb{
13        $$ = new cl_expresion_aritmetica;
```



```

14     $$->es_terminal = true;
15     $$->es_llamada_funcion = true;
16     $$->llamada_val = new cl_llamada_funcion;
17     $$->llamada_val->argumentos = $3->argumentos;
18     $$->llamada_val->identificador = $1->identificador;
19     delete $1; delete $3;
20 }
21 |expresionAritmetica t_dot expresionAritmetica{
22     //asume que la segunda expresion es una llamada
23     $$= new cl_expresion_aritmetica;
24     $$->es_terminal = true;
25     $$->es_llamada_funcion = true;
26     $$->llamada_val = $3->llamada_val;
27     $$->llamada_val->esta_contenida= true;
28     $$->llamada_val->id_contenedor= $1->identificador;
29     delete $1;
30 }

```

Listado 3.43: Implementación de las reglas de producción correspondientes a las expresiones aritméticas formadas por accesos a un arreglo o llamadas a funciones.

Finalmente, como se muestra en el código contenido en el Listado 3.44, fueron definidas las reglas que componen a las expresiones encapsuladas por paréntesis así como las expresiones aritméticas no monolíticas, donde dos símbolos de tipo `expresionAritmetica` fueron unidos a través de un operador cuyo valor de `es_exponencial` es conservado como atributo de la nueva expresión aritmética.

```

1 ...
2 expresionAritmetica:
3     ...
4     |expresionAritmetica operadorAritmetico expresionAritmetica {
5         $$= new cl_expresion_aritmetica;
6         $$->es_terminal = false;

```

```

7      $$->izquierda = $1;
8      $$->derecha = $3;
9      $$->es_exponencial = $2;
10     }
11     |expresionAritmetica operadorAsignacion expresionAritmetica {
12         $$= new cl_expresion_aritmetica;
13         $$->es_terminal = false;
14         $$->izquierda = $1;
15         $$->derecha = $3;
16         $$->es_exponencial = $2;
17     }
18     |t_opregb expresionAritmetica t_cloregb {
19         $$= $2;
20     }
21 ;

```

Listado 3.44: Implementación de las reglas de producción correspondientes a las expresiones aritméticas formadas por otras expresiones aritméticas.

## Expresiones booleanas

De una manera similar a las expresiones aritméticas, fueron definidos dos símbolos terminales, uno de ellos agrupador de los distintos operadores lógicos considerados por el prototipo. Además fueron creadas dos variables denotando los atributos correspondientes a estos.

```

1 %union{
2     ...
3     cl_expresion_booleana *expresion_booleana;
4     int es_conector;
5 }
6 ...
7 %type<expresion_booleana> expresionBooleana;

```

```
8 %type<es_conector> operadorBooleano;
```

Listado 3.45: Declaración de los terminales asociados con el símbolo `expresionBooleana`.

Considerando lo discutido en la Sección 3.4.2, la clase que define los atributos de una expresión aritmética contiene un apuntador a la expresión aritmética que es contenida en ella en caso de ser monolítica, además de apuntadores a las dos expresiones booleanas que las contienen en caso de no serlo. La estructura básica de esta clase, incluyendo sus atributos y métodos, es ilustrada por el código plasmado en el Listado 3.46.

```
1 class cl_expresion_booleana{
2 public:
3     cl_expresion_booleana* izquierda;
4     cl_expresion_booleana* derecha;
5     //0->puede ser terminal o puede ser cualquier operador, 1-> ||, 2-> &&
6     int es_conector;
7
8     bool es_terminal;
9     cl_expresion_aritmetica* valor;
10};
```

Listado 3.46: Declaración de la clase `cl_expresion_booleana`.

El símbolo no terminal `expresionBooleana`, como se muestra en el código plasmado en el Listado 3.47, fue definido en su forma más básica como la unión de dos expresiones aritméticas a través de operadores booleanos. A su vez, una expresión booleana puede ser formada por dos o más expresiones de este tipo unidas a través de un operador lógico conector (`&&`), (`||`), los cuales serán identificados por el valor de 2 y 1 para su futuro uso en el módulo de análisis de complejidad.

```
1 ...
2 expresionBooleana:
3     expresionAritmetica operadorBooleano expresionAritmetica{
4         $$ = new cl_expresion_booleana;
```

```

5      $$->es_terminal = false;
6      $$->es_conector = $2;
7      $$->izquierda = new cl_expresion_booleana;
8      $$->izquierda->es_terminal = true;
9      $$->izquierda->valor = $1;
10     $$->derecha = new cl_expresion_booleana;
11     $$->derecha->es_terminal = true;
12     $$->derecha->valor = $3;
13 }
14 |expresionBooleana t_andand expresionBooleana {
15     $$ = new cl_expresion_booleana;
16     $$->es_terminal = false;
17     $$->es_conector = $2;
18     $$->izquierda = $1;
19     $$->derecha $3;
20 }
21 |expresionBooleana t_oror expresionBooleana {
22     $$ = new cl_expresion_booleana;
23     $$->es_terminal = false;
24     $$->es_conector = $2;
25     $$->izquierda = $1;
26     $$->derecha $3;
27 }
28 ;

```

Listado 3.47: Implementación de las reglas de producción correspondientes al símbolo `expresionBooleana`.

### Rutinas de entrada y salida

De acuerdo a lo discutido en la Sección 3.4.2, las rutinas de entrada y salida (denotadas por `cin/cout`) serán tratadas como llamadas a funciones contenidas dentro de expresiones

aritméticas monolíticas. Por lo tanto, como se muestra en el código contenido en el Listado 3.48, fueron definidos los símbolos no terminales que definen la llamada a estos métodos así como sus argumentos de una manera similar a aquellos usados en las llamadas a funciones dentro del símbolo `expresionAritmetica`.

```

1 ...
2 %type<expresion_aritmetica> entradaSalida
3 %type<argumentos_llamada> argEntrada argSalida

```

Listado 3.48: Declaración de los terminales asociados con el símbolo `entradaSalida`.

De una manera similar a la definición de los argumentos de una llamada a una función convencional, los argumentos de los métodos de entrada y salida fueron definidos de manera recursiva, donde los elementos fueron separados por el respectivo operador del método (`«`) o (`»`).

```

1 ...
2 argSalida:
3     /*empty*/ {$$ = new cl_argumentos_llamada;}
4     |argSalida t_lt t_lt expresionAritmetica
5     {
6         $1->argumentos.push_back($4);
7     }
8     |argSalida t_lt t_lt t_endl
9     {
10    }
11 ;
12 argEntrada:
13     /*empty*/ {$$ = new cl_argumentos_llamada;}
14     |argEntrada t_gt t_gt expresionAritmetica
15     {
16         $1->argumentos.push_back($4);
17     }
18 ;

```

```

19 entradaSalida:
20     t_cin argEntrada{
21         $$= new cl_expresion_aritmetica;
22         $$->es_terminal = true;
23         $$->es_llamada_funcion = true;
24         $$->llamada_val = new cl_llamada_funcion;
25         $$->llamada_val->argumentos = $2->argumentos;
26         $$->llamada_val->identificador = "cin";
27         delete $2;
28     }
29     | t_cout argSalida{
30         $$= new cl_expresion_aritmetica;
31         $$->es_terminal = true;
32         $$->es_llamada_funcion = true;
33         $$->llamada_val = new cl_llamada_funcion;
34         $$->llamada_val->argumentos = $2->argumentos;
35         $$->llamada_val->identificador = "cout";
36         delete $2;
37     }
38 ;

```

Listado 3.49: Implementación de las reglas de producción correspondientes a a las rutinas de entrada y salida cin y cout.

### Declaración de variables

En esta sección fueron definidas las reglas que definen las declaraciones de variables y de variables inicializadas de acuerdo a lo especificado en 3.4.2. Como se muestra en el código plasmado en el Listado 3.50, fueron creados los símbolos no terminales `declaracionVariable`, `tipoPrimitivo` y `tipoContenedor` así como los objetos conteniendo sus atributos.

```

1 %union{

```

```

2      ...
3      cl_declaracion *declaracion;
4      cl_tipoDato *tipoDato;
5  }
6  ...
7  %type<tipoDato> tipoPrimitivo tipoContenedor
8  %type<declaracion> declaracionVariable

```

Listado 3.50: Declaración de los terminales asociados con el símbolo `declaracionVariable`.

Para la definición de la clase que contiene a los atributos de los símbolos `tipoPrimitivo` y `tipoContenedor` fue creada una cadena de caracteres que contiene el identificador de tipo o de contenedor y dos que contienen los tipos de datos asociados en caso de ser un contenedor. La clase que caracteriza a una declaración contiene los atributos del tipo de dato además de una cadena representando el identificador y un apuntador a la expresión aritmética que le fue asignada en caso de haber sido inicializada. La estructura básica de estas clases, incluyendo sus atributos y métodos, son ilustrados por el código plasmado en el Listado 3.51.

```

1  class cl_tipoDato{
2  public:
3      string tipo;
4      //aplica solo para el tipo de contenedores
5      string subtipo;
6      string subtipo2;
7  };
8
9  class cl_declaracion{
10 public:
11     string identificador;
12     //either primitive or the name of the container
13     string tipo;
14     string subtipo;
15     string subtipo2;

```

```
16
17     bool inicializada;
18     cl_expresion_aritmetica* valor_predeterminado;
19 };
```

Listado 3.51: Declaración de las clases `cl_tipoDato` y `cl_declaracion`.

En la sección de reglas, fueron declaradas las derivaciones que resultan en los símbolos no terminales agrupadores `tipoPrimitivo` y `tipoContenedor`, entre ellas destacan las ilustradas en el código mostrado en el Listado 3.52, donde se muestra un subconjunto de las reglas que definen a los tipos de datos y que se consideran representativas, en el caso de los contenedores se destacan algunas reglas donde son usados los atributos `subtipo` y `subtipo2`.

```
1 ...
2 tipoPrimitivo:
3     t_void {$$ = new cl_tipoDato;$$->tipo = "void";}
4     |t_bool {$$ = new cl_tipoDato;$$->tipo = "bool";}
5     ...
6 ;
7 tipoContenedor:
8     t_vector t_gt tipoPrimitivo t_lt {
9         $$ = new cl_tipoDato;
10        $$->tipo = "vector";
11        $$->subtipo = $3->tipo;
12        delete $3;
13    }
14    |t_string {
15        $$ = new cl_tipoDato;
16        $$->tipo = "string";
17    }
18    |t_map t_gt tipoPrimitivo t_comma tipoPrimitivo t_lt {
19        $$ = new cl_tipoDato;
20        $$->tipo = "map";
```



```

21     $$->subtipo = $3->tipo;
22     $$->subtipo2 = $5->tipo;
23     delete $3; delete $5;
24 }
25 ...
26 ;

```

Listado 3.52: Implementación de las reglas de producción correspondientes a los símbolos agrupadores `tipoPrimitivo` y `tipoContenedor`.

Una vez declaradas las reglas que agrupan los diferentes tipos de datos, es posible definir la estructura de una declaración. Continuando con lo discutido en la Sección 3.4.2 e implementado en el código plasmado en el Listado 3.53, una declaración está formada por un tipo de dato y un identificador o por un tipo de dato, un identificador y una asignación. Con el fin de reducir la cantidad de errores de *shift-reduce*, se identificó que el símbolo terminal `t_identificador` puede ser reducido a una expresión aritmética, también puede serlo una asignación, por lo tanto la regla que define al no terminal `declaracionVariable` fue declarada en función del símbolo `expresionAritmetica` asumiendo que éste es un monolítico formado por un identificador o una expresión compuesta denotando una asignación.

```

1 ...
2 declaracionVariable:
3     tipoPrimitivo expresionAritmetica{
4         $$ = new cl_declaracion;
5         $$->tipo = $1->tipo;
6         if($2->es_identificador){
7             //asume que expresionAritmetica es un identificador
8             $$->identificador = $2->identificador;
9         }else{
10            //asume que expresionAritmetica es una asignacion
11            $$->identificador = $2->izquierda->identificador;
12            $$->valor_predeterminado = $2->derecha;

```

```
13     }
14 }
15 |tipoContenedor expresionAritmetica{
16     //asume que expresionAritmetica es un identificador
17     $$ = new cl_declaracion;
18     $$->tipo = $1->tipo;
19     $$->subtipo = $1->subtipo;
20     $$->subtipo2 = $1->subtipo2;
21     $$->identificador = $2->identificador;
22     delete $1;
23 }
24 ;
```

Listado 3.53: Implementación de las reglas de producción correspondientes al símbolo `declaracionVariable`.

### Estructuras iterativas y de control de flujo

De una manera similar a la Sección 3.4.2 se hará referencia a un símbolo no terminal `bloqueCodigo`, el cual será definido en el siguiente apartado. Para la definición de los símbolos no terminales correspondientes a las estructuras iterativas y de control de flujo fueron añadidos a `yyval` objetos que contienen sus atributos así como aquellos del símbolo por definir bloque de código, también fueron declarados los tipos de estos símbolos como se muestra en el código contenido en Listado 3.54.

```
1 %union{
2     ...
3     cl_ciclo *ciclo;
4     cl_if* iif;
5     cl_bloque_codigo* bloque_codigo;
6 }
7 ...
```

```
8 %type<iif> ifStatement
9 %type<ciclo> forStatement
10 %type<ciclo> whileStatement
11
12 %type<bloque_codigo> bloqueCodigo
```

Listado 3.54: Declaración de los terminales asociados con estructuras iterativas y de control de flujo.

Crear una definición de clase para los atributos correspondientes a un bloque de código está fuera del enfoque de la presente sección y será relevado a aquella correspondiente al símbolo bloque de código. Los atributos correspondientes a la estructura de control de flujo *if* son una expresión booleana a ser evaluada, un bloque de código a ejecutar si la expresión resulta verdadera y en caso de existir, un bloque de código a ejecutar en caso de evaluar como falso. A pesar de aparentar tener una estructura sintáctica diferente, es posible identificar tres atributos en común entre las estructuras iterativas *for* y *while*, un argumento de terminación; un *step* que representa el cambio a realizar sobre las variables dentro del ciclo y un bloque de código a ejecutar repetidamente, dichos atributos fueron encapsulados dentro de la clase `cl_ciclo` contenida en `nodos.h`. La estructura básica de estas clases, incluyendo sus atributos y métodos, son ilustrados por el código plasmado en el Listado 3.55.

```
1 class cl_if{
2 public:
3     cl_expresion_booleana* argumento;
4     cl_bloque_codigo* bloque_if;
5     bool tiene_else;
6     cl_bloque_codigo* bloque_else;
7 };
8 class cl_ciclo{
9 public:
10     cl_expresion_booleana* argumento_terminacion;
11     cl_expresion_aritmetica* step;
```

```
12     cl_bloque_codigo* bloque_codigo;  
13 };
```

Listado 3.55: Declaración de las clases `cl_if` y `cl_ciclo`.

Con respecto a las reglas que definen a las estructuras discutidas en esta sección primero definidas en la Sección 3.4.2 e implementadas en el código plasmado en el Listado 3.56, no fueron realizadas modificaciones para reducir los conflictos de *shift-reduce*, sin embargo es importante resaltar el hecho que las expresiones booleanas y aritméticas contenidas en los argumentos de estas estructuras fueron agregadas al bloque de código asociado, pues existe la posibilidad que agreguen de manera significativa a la complejidad temporal. Además se inicializó el *step* de la estructura iterativa `while` a `NULL`, pues no existe una gramática capaz de identificarla por lo que es necesario relevar su identificación al módulo de análisis de complejidad.

```
1 ifStatement:  
2     t_if t_opregb expresionBooleana t_cloregb bloqueCodigo{  
3         $$ = new cl_if;  
4         $$->argumento = $3;  
5         $$->bloque_if = $5;  
6         $$->bloque_if->booleanas.push_back($3);  
7     }  
8     |ifStatement t_else bloqueCodigo{  
9         $1->tiene_else = true;  
10        $1->bloque_else = $3;  
11    }  
12    ;  
13    forStatement:  
14        t_for t_opregb declaracionVariable t_semicolon expresionBooleana  
15        t_semicolon expresionAritmetica t_cloregb bloqueCodigo  
16        {  
            $$ = new cl_ciclo;
```

```

17         $$->bloque_codigo = $9;
18         $$->argumento_terminacion = $5;
19         $$->step = $7;
20         $$->bloque_codigo->booleanas.push_back($5);
21         $$->bloque_codigo->aritmeticas.push_back($7);
22         delete $3;
23     }
24 ;
25 whileStatement:
26     t_while t_opregb expresionBooleana t_cloregb bloqueCodigo
27     {
28         $$ = new cl_ciclo;
29         $$->bloque_codigo = $5;
30         $$->argumento_terminacion = $3;
31         $$->step = NULL;
32         $$->bloque_codigo->booleanas.push_back($3);
33     }
34 ;

```

Listado 3.56: Implementación de las reglas de producción correspondientes a estructuras iterativas y de control de flujo.

### Bloques de código

Siguiendo la definición descrita en la Sección 3.4.2, se declaro la existencia del símbolo `bloqueCodigo` además de los símbolos agrupadores `instruccion` e `instrucciones`, con los cuales comparte tipo de dato, pues al ser heterogéneos en cuanto el tipo de instrucciones que pueden representar se decidió tratarlos como pequeños bloques de una sola instrucción que eventualmente serían añadidos al bloque que los contiene. Estas declaraciones son mostradas en el código contenido en el Listado 3.57.

```

1 %type<bloque_codigo> bloqueCodigo

```

```

2 %type<bloque_codigo> instrucciones
3 %type<bloque_codigo> instruccion

```

Listado 3.57: Declaración de los terminales asociados con el símbolo `bloqueCodigo`.

Dentro de `nodos.h`, se declaró la estructura de la clase que contiene los atributos correspondientes a un bloque de código, el cual al ser meramente un elemento sintáctico agrupador contiene algunos vectores con apuntadores a los elementos que contiene. La estructura básica de esta clase, incluyendo sus atributos y métodos, es ilustrada por el código plasmado en el Listado 3.58.

```

1 class cl_bloque_codigo{
2 public:
3     vector<cl_expresion_aritmetica*> aritmeticas;
4     vector<cl_expresion_booleana*> booleanas;
5     vector<cl_declaracion*> declaraciones;
6     vector<cl_ciclo*> ciclos;
7     vector<cl_if*> ifs;
8 };

```

Listado 3.58: Declaración de la clase `cl_bloque_codigo`.

Como se ilustra en el código mostrado en el Listado 3.59, el símbolo no terminal `instruccion` es utilizado como un símbolo agrupador de todos los no terminales definidos durante la implementación del analizador sintáctico, agregando derivaciones que cubren el retorno de expresiones aritméticas y booleanas.

```

1 instruccion:
2     expresionAritmetica t_semicolon {
3         $$ = new cl_bloque_codigo;
4         $$->aritmeticas.push_back($1);
5     }
6     |expresionBooleana t_semicolon {
7         $$ = new cl_bloque_codigo;

```

```

8      $$->booleanas.push_back($1);
9  }
10 |entradaSalida t_semicolon {
11     $$ = new cl_bloque_codigo;
12     $$->aritmeticas.push_back($1);
13 }
14 |declaracionVariable t_semicolon{
15     $$ = new cl_bloque_codigo;
16     $$->declaraciones.push_back($1);
17 }
18 |t_return expresionAritmetica t_semicolon {
19     $$ = new cl_bloque_codigo;
20     $$->aritmeticas.push_back($2);
21 }
22 |t_return expresionBooleana t_semicolon {
23     $$ = new cl_bloque_codigo;
24     $$->booleanas.push_back($2);
25 }
26 |forStatement { $$ = new cl_bloque_codigo; $$->ciclos.push_back($1);}
27 |whileStatement{ $$ = new cl_bloque_codigo; $$->ciclos.push_back($1);}
28 |ifStatement{ $$ = new cl_bloque_codigo; $$->ifc.push_back($1);}
29 ;

```

Listado 3.59: Implementación de las reglas de producción correspondientes al símbolo *instruccion*.

Finalmente, el símbolo de instrucciones fue declarado de manera recursiva como una colección de cero o más símbolos *instruccion*, donde se trató la heterogeneidad del símbolo que pudiese ser contenido agregando todos los elementos de los vectores de cada tipo de instrucción, esto con el fin de evitar ofuscar el código mediante la implementación de condicionales. Esto es ilustrado por el código contenido en el Listado 3.60.

```

1  instrucciones:

```

```

2      /*empty*/ {$$ = new cl_bloque_codigo;}
3      |instruccion {$$ = new cl_bloque_codigo;}
4      |instrucciones instruccion {
5          for(int i = 0; i<($2)->aritmeticas.size(); i++){
6              $1->aritmeticas.push_back($2->aritmeticas[i]);
7          }
8          for(int i = 0; i<($2)->booleanas.size(); i++){
9              $1->booleanas.push_back($2->booleanas[i]);
10         }
11         for(int i = 0; i<($2)->ciclos.size(); i++){
12             $1->ciclos.push_back($2->ciclos[i]);
13         }
14         for(int i = 0; i<($2)->ifsize.size(); i++){
15             $1->ifsize.push_back($2->ifsize[i]);
16         }
17         delete $2;
18     }
19 ;
20 bloqueCodigo:
21     t_opcrlyb instrucciones t_clocrlyb
22     {
23         $$ = $2;
24     }
25 ;

```

Listado 3.60: Implementación de las reglas de producción correspondientes a los símbolos `instrucciones` y `bloqueCodigo`.

### Definición de funciones

Una vez definido el símbolo correspondiente a los bloques de código y como es ilustrado por el Listado 3.61, es posible crear una regla para la declaración de funciones. Dentro de la



sección de declaraciones de Bison, se estableció la existencia de los símbolos no terminales `declaracionFuncion` y `listaArgumentos`, además fueron agregados a `yylval` objetos que contienen sus atributos.

```
1 %union{
2     ...
3     cl_declaracion_funcion *declaracion_funcion;
4     cl_lista_argumentos *lista_argumentos;
5 }
6 ...
7 %type<declaracion_funcion> declaracionFuncion
8 %type<lista_argumentos> listaArgumentos
```

Listado 3.61: Declaración de los terminales asociados con el símbolo `declaracionFuncion`.

Dentro de `nodos.h` fueron declaradas las clases que encapsulan los atributos de cada símbolo, los cuales fueron definidos en la Sección 3.4.2. La estructura básica de estas clases, incluyendo sus atributos y métodos, son ilustrados por el código plasmado en el Listado 3.62.

```
1 class cl_lista_argumentos{
2 public:
3     std::vector<cl_declaracion> argumentos;
4 };
5 class cl_declaracion_funcion{
6 public:
7     string identificador;
8     std::vector<cl_declaracion> argumentos;
9     cl_bloque_codigo* bloque_codigo;
10    string tipo;
11 };
```

Listado 3.62: Declaración de las clases `cl_lista_argumentos` y `cl_declaracion_funcion`.

Finalmente, fueron implementadas las reglas siguiendo la gramática sugerida durante la fase de diseño. Esto se muestra en el código plasmado en el Listado 3.63.

```

1 listaArgumentos:
2     /*empty*/ {$$ = new cl_lista_argumentos;}
3     | declaracionVariable{
4         $$ = new cl_lista_argumentos;
5         $$->argumentos.push_back((*$1));
6         delete $1;
7     }
8     | listaArgumentos t_comma declaracionVariable {
9         $1->argumentos.push_back((*$3));
10        delete $3;
11    }
12 ;
13 declaracionFuncion:
14     tipoPrimitivo t_identificador t_opregb listaArgumentos t_cloregb
15     bloqueCodigo{
16         $$ = new cl_declaracion_funcion;
17         $$->argumentos = $4->argumentos;
18         $$->bloque_codigo = $6;
19         $$->tipo = $1->tipo;
20         $$->identificador = $2->value_string;
21         delete $1; delete $4;
22     }
23 ;

```

Listado 3.63: Implementación de las reglas de producción correspondientes a los símbolos `listaArgumentos` y `declaracionFuncion`.

### Sobre el símbolo `programa` y precedencia

Reiterando lo discutido en la Sección 3.4.2, un programa es considerado por el prototipo como una colección de cabeceras y funciones. Debido a que el prototipo no contempla en su arquitectura a los módulos de generación de código, se ha decidido ignorar la existencia

de las cabeceras dentro del AST, resultando en el tipado dentro de la declaración de los no terminales relacionados directamente con el símbolo **programa** y expresada en el código contenido en Listado 3.64.

```

1 %union{
2     ...
3     vector<cl_declaracion_funcion*>* funciones_declaradas;
4 }
5 ...
6 %type<nada> cabecera cabeceras
7 %type<funciones_declaradas> declaracionesFuncion

```

Listado 3.64: Declaración de los símbolos `cabecera`, `cabeceras` y `declaracionesFuncion`.

De una manera similar al apartado anterior, las declaraciones de las reglas correspondientes a los símbolos `cabecera`, `cabeceras` y `declaracionesFuncion` siguieron lo recomendado durante la fase de diseño y su implementación es descrita por el código contenido en el Listado 3.65.

```

1 programa:
2     cabeceras declaracionesFuncion{
3         raiz = new cl_raiz;
4         raiz->funciones_declaradas = (*$2);
5         delete $2;
6     }
7 ;
8
9 cabecera:
10     t_include t_hardstr {$$ = 0;}
11     | t_nspacestd {$$ = 0;;}
12 ;
13 cabeceras:
14     /*empty*/ {$$ = 0;}
15     | cabeceras cabecera { $1 = 0;}

```

```

16 ;
17 declaracionesFuncion:
18     /*empty*/ {
19         $$ = new vector<cl_declaracion_funcion*>;
20     }
21     | declaracionesFuncion declaracionFuncion{
22         $1->push_back($2);
23     }
24 ;

```

Listado 3.65: Implementación de las reglas de producción correspondientes a los símbolos `cabecera`, `cabeceras` y `declaracionesFuncion`.

Una vez definido el símbolo programa, fue necesario establecerlo como el símbolo inicial de la gramática, esto mediante la instrucción `%start programa` dentro del apartado de declaraciones de Bison.

Finalmente, Bison por defecto establece el orden de precedencia de los *tokens* en función del orden que aparecen dentro de su definición, además de establecer que no tienen asociatividad por defecto. Para establecer un orden y una asociatividad a la izquierda para cada uno de los operadores matemáticos y booleanos se utilizó la instrucción `%left operador` donde a los últimos operadores a los que esta instrucción es aplicada son aquellos con mayor precedencia.

```

1 %left t_epeq t_npeq t_lteq t_gteq t_lt t_gt
2 %left t_andand t_oror
3 %left t_sum t_sub
4 %left t_astk t_div

```

Listado 3.66: Precedencia de los operadores en la gramática.

### Sobre el proceso de compilación

La implementación del analizador sintáctico descrito a lo largo del presente apartado fue almacenada en el archivo *AnalizadorSintactico.y*, el cual fue compilado utilizando el comando `bison d o analizadorSintactico.cpp analizadorSintactico.y`.

El proceso de compilación resulta en la creación del archivo `analizadorSintactico.hpp` en donde se definen los *tokens* usados por el módulo de análisis léxico, también es obtenido el archivo `analizadorSintactico.cpp` donde es definida, entre otras cosas, la rutina `yyparse` que puede ser utilizada para invocar el analizador léxico y en consecuencia el analizador sintáctico.

La implementación del programa Lex descrita a lo largo la Sección 3.5.1, fue almacenada en el archivo `analizadorLexico.L` y por medio del comando de compilación `lex -o analizadorLexico.cpp analizadorLexico.l` fue creado el archivo `analizadorLexico.cpp`, el cual contiene la función `yylex` utilizada por el analizador sintáctico para invocar el analizador léxico.

#### 3.5.3. Implementación del módulo de análisis de jerarquía y dependencia

Este módulo es el encargado de definir si no existen dependencias circulares entre las funciones definidas en el código del usuario además de definir el orden que el módulo de análisis seguirá. Se decidió que el módulo fuera contenido dentro `analizadorPrecedencia.hpp`, los contenidos de este archivo serán introducidos de manera gradual en este apartado. Inicialmente el código contenido en el Listado 3.67 muestra la definición inicial de la clase que contiene al módulo, el cual recibe la raíz del AST generado por el analizador sintáctico.

```
1 class analizador_precedencia{  
2 private:
```

```
3     cl_raiz* raiz;  
4 public:  
5     analizador_precedencia(cl_raiz* raiz);  
6 };  
7 analizador_precedencia::analizador_precedencia(cl_raiz* raiz){  
8     this->raiz = raiz;  
9 }
```

Listado 3.67: Definición inicial de la clase `analizador_precedencia`.

### Sobre los grafos de llamadas

En la Sección 3.4.3 fueron introducidos grafos de llamadas a funciones como una representación simplificada del AST. Considere el código descrito en el Listado 3.68, donde son definidas las funciones `a()` y `b()` cuyo contenido puede tener cualquier elemento sintáctico válido.

```
1 void a(){  
2     ...  
3     b();  
4     ...  
5 }  
6 void b(){  
7     ...  
8 }
```

Listado 3.68: Ejemplo de un código con dependencias circulares entre dos funciones.

El grafo de llamadas simplificado está ilustrado por la Figura 3.6, donde una arista dirigida es creada del nodo `a()` al nodo `b()` debido a la llamada que la primera hace de la segunda en el cuerpo de su declaración.

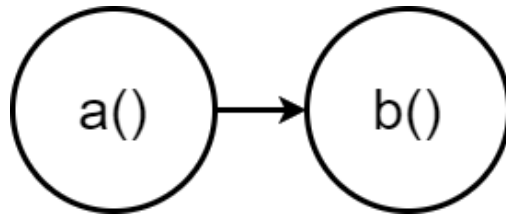


Figura 3.6: Grafo de llamadas correspondiente al código plasmado en el Listado 3.68

Debido a que las aristas son solo creadas a partir de una llamada a una función, fue posible descomponer los nodos del grafo en los elementos sintácticos relevantes, en particular aquellos que pueden contener llamadas a funciones o a los símbolos que las contienen, resultando en el grafo ilustrado en la Figura 3.7.

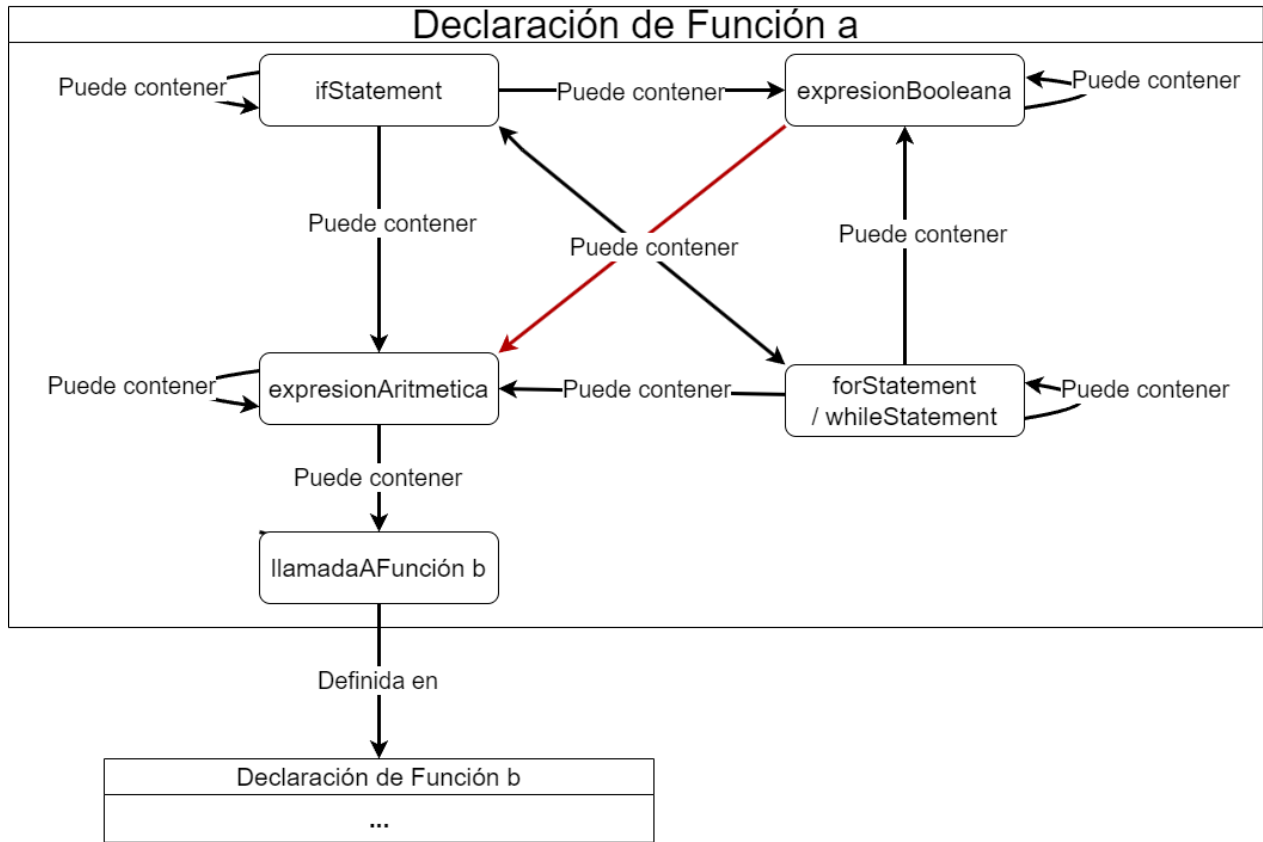


Figura 3.7: Grafo de llamadas descomprimido correspondiente al código plasmado en el Listado 3.68.

Los subgrafos contenidos en la declaración de las funciones dentro de la Figura 3.7 pueden ser obtenidos a través del AST resultante del módulo de análisis sintáctico. Sin embargo, la arista que une la llamada a una función con su definición no existe en esta representación, esto fue resuelto mediante la adición de una estructura de datos asociativa la cual asigna el identificador de cada llamada definida dentro del código a su respectiva declaración, basta con establecer el identificador de la función como llave debido a que en la Sección 3.3.1 se estableció que el polimorfismo de funciones no sería considerado. En el caso de las funciones definidas por el estándar del lenguaje y soportadas por el prototipo, se decidió usar otra estructura de datos asociativa con el fin de marcar estos métodos como predefinidos y evitar



que sean explorados de manera innecesaria.

Tomando en cuenta la estructura definida en la Figura 3.7 y el algoritmo descrito durante la fase de diseño de este módulo, fueron agregados a los símbolos `expresionAritmetica`, `expresionBooleana`, `ifStatement`, `forStatement`, `whileStatement` y `declaracionFuncion` los atributos `int low_link_value` e `int scc_id`, representando el identificador asignado a el símbolo actual por la función DFS y el identificador más pequeño al que los vecinos del símbolo actual tienen acceso respectivamente. A consecuencia de este cambio, también fueron agregados a la clase que contiene al módulo una variable entera `SINVISITAR` inicializada en cero y representando el `scc_id` asignado a los símbolos no visitados por el método DFS, además también fue agregada la variable `id_disponible` inicializada en uno y representa el primer entero sin asignar al `scc_id` de un símbolo.

### Implementación del algoritmo de análisis de jerarquía y dependencia

El algoritmo especificado en la Sección 3.4.3, contempla la utilización de una sola pila para el almacenamiento de los símbolos visitados, lo cual requiere de una implementación de este contenedor capaz de soportar miembros con diferente tipo de dato, se decidió evitar esta implementación al solo almacenar los identificadores numéricos correspondientes a cada objeto, pues al tener una relación uno a uno con los símbolos dentro del programa es posible hacer la conversión al objeto correspondiente fuera del contenedor.

También se requiere de la utilización de una pila (o cualquier estructura de datos que soporte operaciones de tipo *first-in first-out*) que soporte la verificación de pertenencia a ésta. C++ no cuenta con ninguna estructura ya implementada con estas características, sin embargo es posible simular este comportamiento mediante la adición de una estructura de datos asociativa que asocie el identificador de cada elemento sintáctico con un valor booleano indicando si se encuentra en la pila o no. Estos cambios a la clase se muestran en el código ilustrado en el Listado 3.69, donde además fue creado un vector con el fin de almacenar el

ordenamiento topológico correspondiente al orden de análisis de complejidad.

```

1 class analizador_precedencia{
2 private:
3     ...
4     vector<int> call_stack;
5     map<int,bool> on_stack;
6 public:
7     vector<cl_declaracion_funcion*> orden_topologico;
8 };

```

Listado 3.69: Alternativa a una pila con acceso aleatorio.

Una vez abordados los cambios necesarios para la implementación del algoritmo en C++, fueron implementados métodos DFS para cada uno de los símbolos contemplados en la figura 3.7. La implementación de DFS demostró ser la porción mas extensa del módulo, donde la definición de este método siguió una estructura similar a la demostrada en el código 3.70 para una clase imaginaria `cl_simbolo`, debido a su extensión la implementación de este método puede ser encontrada dentro del repositorio <https://github.com/MAlexxiT/automated-complexity-analysis/tree/main/src>.

```

1 void analizador_precedencia::dfs(cl_simbolo* actual){
2     actual->scc_id = id_disponible;
3     actual->low_link_value = id_disponible;
4     id_disponible++;
5     call_stack.push_back(actual->scc_id);
6     on_stack[actual->scc_id] = true;
7
8     //llamada a cada simbolo contenido
9     for(cl_simbolo* simbolo: actual){
10         if(simbolo->scc_id == SINVISITAR){
11             dfs(simbolo);
12         }
13         if(on_stack[simbolo->scc_id]){

```

```
14         low_link_value = min(low_link_value,simbolo->low_link_value);
15     }
16 }
17
18
19 //si es el primer nodo del ciclo
20 if(actual->scc_id==actual->low_link_value){
21     while(call_stack.back()!=actual->scc_id){
22         on_stack[call_stack.back()] = false;
23         call_stack.pop_back();
24     }
25     call_stack.pop_back();;
26     on_stack[actual->scc_id] = false;
27 }
28
29 //Si el simbolo es una declaración de una funcion
30 //agregarlo a la lista de ordenamiento topologico
31 //orden_topologico.push_back(actual);
32 }
```

Listado 3.70: Implementación genérica del método DFS.

Dentro de la implementación de DFS para cada símbolo, la lógica que rige las llamadas al método para los símbolos contenidos fue definida dentro de cada implementación, sin embargo debido a que los símbolos representados por las clases `cl_declaracion_funcion`, `cl_if` y `cl_ciclo` tienen bloques de código encapsulando los símbolos contenidos en estos, se decidió crear un método `explorar_bloque` con el fin de encapsular el fragmento de código que tienen estos símbolos en común.

### Sobre el método `esRecursoivo`

Una vez implementadas las instancias del método DFS, solo resta definir el método `esRecursoivo`, el cual inicia las llamadas recursivas. Como se muestra en el código contenido en el Listado 3.71, esta función llama DFS para cada función sin explorar y de acuerdo a lo discutido en la Sección 3.4.3 determina que la función forma parte de una dependencia circular si su `scc_id` difiere de su `low_link_value`.

```
1 //Esta funcion es llamada una sola vez
2 bool analizador_precedencia::esRecursoivo(){
3     bool respuesta = false;
4     for(int i = 0; i<raiz->funciones_declaradas.size(); i++){
5         cl_declaracion_funcion* actual = raiz->funciones_declaradas[i];
6         if(actual->scc_id == SINVISITAR){
7             dfs(raiz->funciones_declaradas[i]);
8         }
9         if(actual->scc_id != actual->low_link_value){
10             respuesta = false;
11         }
12     }
13     //al arreglo de declaraciones en postorden
14     //basta con invertirlo para convertirlo en el orden topologico
15     //el primer elemento del orden topologico
16     //es el primer elemento sin dependencias
17     //asumiendo que el grafo sea una DAG
18     reverse(orden_topologico.begin(),orden_topologico.end());
19     return respuesta;
20 }
```

Listado 3.71: Implementación del método `esRecursoivo`.

El método `esRecursoivo()` será llamado sola una vez por el módulo de análisis de complejidad, el valor retornado de esta función además del ordenamiento topológico obtenido de

su ejecución fue utilizado durante la implementación de este módulo.

### 3.5.4. Implementación del módulo de análisis de complejidad

Este módulo es el encargado de estimar la complejidad temporal del programa analizado y está contenido en el archivo `analizadorComplejidad.cpp`. Con el fin de modularizar en la medida de lo posible, se decidió dividir el desarrollo del presente módulo en la creación del tipo de dato que contiene al polinomio de complejidad, la creación de una clase que contenga los métodos y atributos necesarios para la implementación del analizador de complejidad y la definición de una función encargada de coordinar el análisis de complejidad.

#### Sobre el tipo de dato `cl_complejidad`

La complejidad temporal toma forma de un polinomio y es expresada en función de los identificadores presentes dentro del código analizado, estos polinomios son almacenados y manipulados a través de una clase llamada `cl_polinomio`, contenida en el archivo `polinomios.hpp`.

Como se discutió en la Sección 3.4.4, la clase polinomio fue definida a partir de las clases que contiene, partiendo de la clase `cl_factor`. La clase factor fue definida inicialmente con los atributos mostrados en el código plasmado en el Listado 3.72, donde además fueron creados métodos de acceso para cada uno de los atributos y funciones booleanas que evalúan si el factor en cuestión es de tipo constante, lineal/exponencial o logarítmico. Además se definió una constante global, la cual determina un identificador único a las variables constantes.

```
1 #define CONSTANTE "__CONSTANTE__"
2 class cl_factor{
3 private:
4     //0 = constante, 1 = log(id)^exp, 2 = id^exp
5     int tipo;
```

```
6     double exponente;
7 public:
8     std::string identificador;
9     //TIP0: 0 = constante, 1 = log(id)^exp, 2 = id^exp
10    cl_factor(int tipo, int exponente, std::string identificador){
11        this->tipo = tipo;
12        this->exponente = exponente;
13        this->identificador = identificador;
14        this->usado = false;
15    }
16    //METODOS DE ACCESO
17    double obtenerExponente(){
18        return this->exponente;
19    }
20    string obtenerIdentificador(){
21        return this->identificador;
22    }
23    int obtenerTipo(){
24        return this->tipo;
25    }
26 }
```

Listado 3.72: Definición inicial de la clase `cl_factor`.

Una vez creada la estructura básica de la clase `cl_factor`, fue definida la función auxiliar `multiplicables` y fueron sobrecargados los operadores de igualdad(==), desigualdad(!=), multiplicación (\*) y menor que (<).

La función auxiliar `multiplicables` plasmada en el código mostrado en el Listado 3.73, tiene como propósito definir si es posible multiplicar dos factores, se asumió que es posible siempre y cuando alguno de los factores sea constante o ambos coincidan en cuanto a su tipo e identificador.

```
1 class cl_factor{
2 public:
3     ...
4     bool multiplicables(cl_factor otro){
5         //Un factor constante cuenta como termino semejante de cualquier
        otro factor
6         if(this->esConstante() || otro.esConstante()){
7             return true;
8         }
9         if(this->obtenerIdentificador() != otro.obtenerIdentificador()){
10            return false;
11        }
12        if(this->obtenerTipo() != otro.obtenerTipo()){
13            return false;
14        }
15        return true;
16    }
17 }
```

Listado 3.73: Implementación del método multiplicables en cl\_factor.

El operador de multiplicación (\*) sobrecargado en el fragmento de código plasmado en el Listado 3.74, fue definido debido a que es un prerequisite para llamar a la función `sort` sobre una lista de tipo `cl_factor`, el comportamiento de este operador será explicado en cuanto sea relevante.

```
1 class cl_factor{
2 public:
3     ...
4     cl_factor operator*(cl_factor otro){
5         assert((this->multiplicables(otro)));
6         int tipo_resultante = max(this->obtenerTipo(), otro.obtenerTipo())
        ;
    }
}
```

```

7      int exponente_resultante =this->obtenerExponente()+otro.
obtenerExponente();
8      string nuevo_identificador = this->obtenerIdentificador();
9      if(nuevo_identificador == CONSTANTE){
10         nuevo_identificador = otro.obtenerIdentificador();
11     }
12     cl_factor resultado(tipo_resultante,exponente_resultante,
nuevo_identificador);
13
14     return resultado;
15 }
16 }

```

Listado 3.74: Implementación del método de multiplicación en la clase `cl_factor`.

El operador *menor que*(<) sobrecargado en el fragmento de código plasmado en el Listado 3.75, fue sobrecargado de tal manera que termine la ejecución del programa si se intentan multiplicar dos factores para los cuales `multiplicables` evalúa falso. En caso de no haber terminado la ejecución, el operador retorna un nuevo factor el cual conserva el identificador y con un exponente acorde al obtenido de la suma de los dos exponentes de los multiplicandos.

```

1 class cl_factor{
2 public:
3     ...
4     bool operator<(cl_factor otro){
5         if(this->obtenerTipo() != otro.obtenerTipo()){
6             return this->obtenerTipo() > otro.obtenerTipo();
7         }
8         if(this->obtenerIdentificador()!=otro.obtenerIdentificador()){
9             return this->obtenerIdentificador() > otro.
obtenerIdentificador();
10        }
11        return this->obtenerExponente()<otro.obtenerExponente();

```



```

12     }
13 }

```

Listado 3.75: Implementación del método *menor que* en la clase `cl_factor`.

De acuerdo a lo especificado en la Sección 3.4.4, fue definida una clase `cl_sumando`, la cual en su forma más elemental está caracterizada por una lista de factores que la conforman. Además de este único atributo, también fue creado el método `tieneFactor` donde fueron sobrecargados los operadores de multiplicación (`*`) e igualdad (`==`).

El método `tieneFactor` fue definido con el fin de determinar si dentro de un sumando existe un factor determinado, en el código plasmado en el Listado 3.76 se muestra cómo se considera que un factor de tipo constante siempre está presente en cualquier sumando, pues al ser el polinomio representativo de una complejidad temporal en su notación *big Oh* existe la posibilidad de que haya sido descartado de manera temprana, el resto de la implementación solo verifica cada factor dentro del sumando y determina que el factor está presente si y solo si existe un factor que coincide en tipo e identificador con el factor buscado.

```

1  class cl_sumando{
2  public:
3      vector<cl_factor> factores;
4      bool tieneFactor(cl_factor factor_objetivo){
5          if(factor_objetivo.esConstante()){
6              return true;
7          }
8          bool lo_contiene;
9          for(cl_factor factor: factores){
10              lo_contiene = true;
11              lo_contiene = lo_contiene && (factor_objetivo.obtenerTipo() ==
factor.obtenerTipo());
12              lo_contiene = lo_contiene && (factor_objetivo.identificador ==
factor.identificador);
13              lo_contiene = lo_contiene && (factor.obtenerExponente() >=

```

```
factor_objetivo.obtenerExponente());  
14         if(lo_contiene){  
15             return true;  
16         }  
17     }  
18     return false;  
19 }  
20 }
```

Listado 3.76: Definición inicial de la clase `cl_sumando`.

El método operador de multiplicación (`*`), cuya implementación es ilustrada por el código contenido en el Listado 3.77, fue implementado definiendo un sumando vacío como resultado, para después poblar los factores contenidos en éste a través de la multiplicación de todos los factores compatibles entre el sumando que invocó al operador y el sumando `otro`, donde al final son anexados aquellos factores que no fueron multiplicados en el proceso. Es importante resaltar la añadidura del atributo `bool usado` de manera retroactiva a la clase `cl_factor`.

```
1 class cl_sumando{  
2 public:  
3     ...  
4     cl_sumando operator*(cl_sumando otro){  
5         cl_sumando resultado;  
6         bool encontrado;  
7         for(cl_factor fac_propio: factores){  
8             encontrado = false;  
9             for(cl_factor& fac_otro: otro.factoros){  
10                 if(fac_propio.multiplicables(fac_otro)){  
11                     resultado.factoros.push_back(fac_propio*fac_otro);  
12                     encontrado = true;  
13                     fac_otro.usado = true;  
14                     break;  
15                 }  
16             }  
17         }  
18         return resultado;  
19     }  
20 }
```

```
16         }
17         if(!encontrado){
18             resultado.factoros.push_back(fac_propio);
19         }
20     }
21     for(cl_factor& factor: otro.factoros){
22         if(!factor.usado){
23             resultado.factoros.push_back(factor);
24         }
25     }
26     return resultado;
27 }
28 }
```

Listado 3.77: Implementación del método de multiplicación en la clase `cl_sumando`.

En el método operador de igualdad (`==`) ilustrado en el código contenido en el Listado 3.78, fueron ordenadas las listas de factores presentes en ambos sumandos, donde debido a la definición del operador `<` en la clase `cl_factor` son obtenidas dos listas donde los factores constantes son posicionados al final de éstas. Tomando ventaja de esta propiedad, el método verifica que no existan discrepancias entre las posiciones en común de ambas listas y que todos los elementos en la lista más larga sean factores constantes.

```
1 class cl_sumando{
2 public:
3     ...
4     bool operator==(cl_sumando otro){
5         sort(factoros.begin(),factoros.end());
6         sort(otro.factoros.begin(),otro.factoros.end());
7         //ignora factores constantes que podrian estar al final del vector
8         for(int i = 0; i<min(factoros.size(),otro.factoros.size()); i++){
9             if(!(factoros[i] == otro.factoros[i])){
10                 return false;
11             }
12         }
13         return true;
14     }
15 }
```

```
11     }
12 }
13
14     for(int i = factores.size(); i<otro.factoros.size(); i++){
15         if(otro.factoros[i].esConstante() == false){
16             return false;
17         }
18     }
19     for(int i = otro.factoros.size(); i<factores.size(); i++){
20         if(factores[i].esConstante() == false){
21             return false;
22         }
23     }
24     return true;
25 }
26 }
```

Listado 3.78: Implementación del método de igualdad en la clase `cl_sumando`.

Finalmente, se tiene que la clase `cl_complejidad`, la cual contiene un polinomio de complejidad temporal expresado en su notación *big Oh* de acuerdo a lo descrito en la Sección 3.4.4, tiene como atributo una lista de sumandos y métodos para eliminar sumandos semejantes y para imprimir la complejidad temporal a consola además de sobrecargas a los operadores de suma(+) y multiplicación(\*). Por defecto un objeto de tipo `cl_complejidad` cuenta con un único sumando con un solo factor constante.

El método para eliminar sumandos con los mismos factores fue nombrado `eliminar_terminos_semejantes` y su implementación es ilustrada en el código plasmado en el Listado 3.79. Primero fueron eliminados aquellos sumandos que fuesen considerados iguales. Una vez obtenida una lista de sumandos únicos, fueron marcados todos los sumandos en donde todos sus factores estuviesen contenidos en otro sumando, una vez eliminados se iguala la lista de sumandos del objeto que llamó a la función a la lista obtenida. Es importante resaltar

la añadidura del atributo `bool` duplicado de manera retroactiva a la clase `cl_sumando`.

```
1
2 class cl_complejidad{
3 private:
4     void eliminar_terminos_semejantes(){
5         vector<cl_sumando> copia_sumandos = sumandos;
6
7         for(int i = 1; i<copia_sumandos.size(); i++){
8             for(int j = 0; j<i; j++){
9                 if(copia_sumandos[i] == copia_sumandos[j]){
10                     copia_sumandos[j].duplicado = true;
11                 }
12             }
13         }
14
15         sumandos.clear();
16         for(cl_sumando s: copia_sumandos){
17             if(s.duplicado == false){
18                 sumandos.push_back(s);
19             }
20         }
21
22         copia_sumandos.clear();
23         copia_sumandos = sumandos;
24         bool lo_contiene;
25         for(int i = 0; i<copia_sumandos.size(); i++){
26             for(int j = 0; j<copia_sumandos.size(); j++){
27                 if(i==j){
28                     continue;
29                 }
30                 lo_contiene = true;
31                 for(cl_factor factor: copia_sumandos[j].factores){
```

```

32         lo_contiene = (lo_contiene&&copiar_sumandos[i].
tieneFactor(factor));
33     }
34     if(lo_contiene){
35         copia_sumandos[j].duplicado = true;
36     }
37 }
38 }
39
40 sumandos.clear();
41 for(cl_sumando s: copia_sumandos){
42     if(s.duplicado == false){
43         sumandos.push_back(s);
44     }
45 }
46 }
47 }
48 public:
49     vector<cl_sumando> sumandos;
50 }

```

Listado 3.79: Definición inicial de la clase `cl_complejidad` y del método `eliminar_terminos_semejantes`.

El método para imprimir la complejidad temporal a consola fue nombrado `imprimir_expresion` y su implementación es ilustrada en el código mostrado en el Listado 3.80. Nótese como el valor `CONSTANTE` definido al inicio del archivo es reemplazado por un 1 correspondiente a la manera estándar de representar complejidades constantes.

```

1 class cl_complejidad{
2     ...
3 public:
4     void imprimir_expresion(){

```

```
5      bool primero = true;
6      bool primer_factor;
7
8      cout<<"0( ";
9
10     for(cl_sumando sumando: sumandos){
11         if(!primero){
12             cout<<" + ";
13         }
14         primero = false;
15         primer_factor = true;
16         for(cl_factor factor: sumando.factor){
17             if(!primer_factor){
18                 cout<<"*";
19             }
20             primer_factor = false;
21             if(factor.identificador == CONSTANTE){
22                 cout<<"1";
23             }else{
24                 if(factor.esLogaritmico()){
25                     cout<<"log(";
26                 }
27                 cout<<factor.identificador;
28                 if(factor.esLogaritmico()){
29                     cout<<")";
30                 }
31                 if(factor.obtenerExponente()>1){
32                     cout<<"^"<<factor.obtenerExponente()<<" ";
33                 }
34             }
35         }
36     }
```

```
37     }
38     cout<<" )" <<endl;
39 }
40 }
```

Listado 3.80: Implementación del método `imprimir_expresion` en la clase `cl_complejidad`.

En el método operador de multiplicación (`*`) ilustrado en el código contenido en el Listado 3.81, fue creada una instancia de `cl_complejidad`, cuya lista de sumandos fue poblada con el resultado de multiplicar todos los pares posibles de sumandos presentes en el objeto que llamó al operador y otro, para después reducir los sumandos semejantes presentes en el resultado.

```
1
2 class cl_complejidad{
3 ...
4 public:
5     cl_complejidad operator*(cl_complejidad otro){
6         cl_complejidad respuesta;
7         for(cl_sumando propio_sumando: sumandos){
8             for(cl_sumando otro_sumando: otro.sumandos){
9                 respuesta.sumandos.push_back(propio_sumando*otro_sumando);
10            }
11        }
12
13        respuesta.eliminar_terminos_semejantes();
14
15        return respuesta;
16    }
17 }
```

Listado 3.81: Implementación del método de multiplicación en la clase `cl_complejidad`.

En el método operador de suma (`+`) ilustrado en el código contenido en el Listado 3.82, fue creada una instancia de `cl_complejidad` en cuya lista de sumandos fueron almacenados todos



los sumandos presentes, tanto en el objeto que llamó al operador como aquellos presentes en **otro**, para después reducir los sumandos semejantes presentes en el resultado.

```
1
2 class cl_complejidad{
3 ...
4 public:
5     cl_complejidad operator+(cl_complejidad otro){
6         cl_complejidad respuesta;
7         for(cl_sumando propio_sumando: sumandos){
8             respuesta.sumandos.push_back(propio_sumando);
9
10        }
11        for(cl_sumando otro_sumando: otro.sumandos){
12            respuesta.sumandos.push_back(otro_sumando);
13        }
14
15        respuesta.eliminar_terminos_semejantes();
16
17        return respuesta;
18    }
19 }
```

Listado 3.82: Implementación del método de suma en la clase `cl_complejidad`.

### Sobre la clase `cl_analizador`

De acuerdo a lo especificado en la Sección 3.4.4, las funciones y variables necesarios para llevar a cabo el proceso de análisis de complejidad fueron encapsulados dentro de la clase `cl_analizador`. En el código plasmado en el Listado 3.83 se ilustra la estructura fundamental de la clase `cl_analizador` donde fueron enlistados los métodos y atributos que contiene, cuyas definiciones serán descritas a lo largo del presente apartado.

```
1
2 class cl_analizador{
3 private:
4     bool estaPredefinida(string identificador);
5     cl_expresion_booleana* retorno_pendientes;
6
7     vector<cl_declaracion_funcion*> orden_topologico;
8     map<string,cl_declaracion_funcion*> definiciones;
9     map<string, cl_complejidad> complejidad_funcion;
10    map<string, bool> modifica_tam_contenedor;
11    //tipo de contenedor, identificador de la funcion, complejidad asociada
12    map<string,map<string,cl_complejidad>> complejidad_funcion_contenida;
13    //identificador, tipo de dato
14    map<string,cl_tipoDato> tipado_de_variable;
15
16    map<string,int> apariciones_en_aritmetica;
17    //0->constante, 1->líneal, 2->exponencial
18    //lvl de anidamiento, id, tasa de crec
19    map<int,map<string,int>> tasa_crecimiento_variable;
20    int anidamiento_actual;
21
22    void inicializador_complejidades_predefinidas();
23    cl_complejidad obtener_complejidad(cl_llamada_funcion* llamada);
24    cl_complejidad obtener_complejidad(string identificador_funcion,string
    identificador,cl_tipoDato tipoContenedor);
25    cl_complejidad analizar(cl_if* actual);
26    cl_complejidad analizar(cl_ciclo* actual);
27    cl_complejidad analizar(cl_declaracion* actual);
28    cl_complejidad analizar(cl_expresion_booleana* actual);
29    cl_complejidad analizar(cl_expresion_aritmetica* actual);
30    cl_complejidad analizar(cl_bloque_codigo* actual);
31 public:
```

```
32     string identificador_declaracion_funcion;  
33     cl_analizador(vector<cl_declaracion_funcion*> orden_topologico);  
34     cl_complejidad analizar(cl_declaracion_funcion* actual);  
35     cl_complejidad analizar_rekursiva(cl_declaracion_funcion* actual);  
36 };
```

Listado 3.83: Definición inicial de la clase `cl_analizador`.

Inicialmente fue declarado el cuerpo de la función `inicializador_complejidades_predefinidas`, el propósito de este método es poblar las estructuras de datos asociativas que almacenan la complejidad temporal de las funciones. Las complejidades almacenadas son expresadas en función de los argumentos de llamada de la función o del contenedor que los llamó, a este tipo de definiciones se le denominarán *genericas* dentro del presente documento, pues fungirán como plantilla para determinar la complejidad temporal de estas funciones cuando sean llamadas.

En el código plasmado en el Listado 3.84 se muestra un fragmento de la implementación de este método, donde las complejidades no constantes fueron declaradas en función de la variable genérica `n` representando la cardinalidad del argumento o del contenedor desde donde la función es llamada.

```
1  
2 void cl_analizador::inicializador_complejidades_predefinidas(){  
3     cl_factor ologn(1,1,"n");  
4     cl_factor on(2,1,"n");  
5  
6     //O(1)  
7     cl_complejidad constante;  
8     //O(logn)  
9     cl_complejidad logaritmica;  
10    logaritmica.sumandos[0].factores.pop_back();  
11    logaritmica.sumandos[0].factores.push_back(ologn);  
12    //O(n)
```

```

13     cl_complejidad líneal;
14     líneal.sumandos[0].factores.pop_back();
15     líneal.sumandos[0].factores.push_back(on);
16     //O(nlogn)
17     cl_complejidad loglíneal;
18     loglíneal.sumandos[0].factores.pop_back();
19     loglíneal.sumandos[0].factores.push_back(on);
20     loglíneal.sumandos[0].factores.push_back(ologn);
21
22
23     //no containerizadas
24     complejidad_funcion["cin"] = constante;
25     ...
26     complejidad_funcion["sort"] = loglíneal;
27     ...
28     //multiset
29     complejidad_funcion_contenida["multiset"]["upper_bound"] = logaritmica
30     ;
31     ...
32     complejidad_funcion_contenida["double"]["[]"] = constante;
33 }

```

Listado 3.84: Fragmento de la implementación del método inicializador\_complejidades\_predefinidas.

Después de definido el método que puebla la tabla de complejidades genéricas de los métodos definidos por el estándar del lenguaje, fue implementado el constructor de la clase `cl_analizador`, como muestra el código contenido en el Listado 3.85, en este método le fue asignado a cada identificador de una función un apuntador a su declaración, además fueron pobladas las tabla de funciones predefinidas por el lenguaje y fueron identificados los métodos que modifican el tamaño de su contenedor.

```

2 cl_analizador::cl_analizador(vector<cl_declaracion_funcion*>
    orden_topologico){
3     this->orden_topologico = orden_topologico;
4     for(cl_declaracion_funcion* declaracion_actual: orden_topologico){
5         definiciones[declaracion_actual->identificador] =
        declaracion_actual;
6     }
7     inicializador_complejidades_predefinidas();
8     anidamiento_actual = 0;
9     modifica_tam_contenedor["push_back"] = true;
10    modifica_tam_contenedor["pop_back"] = true;
11    ...
12 }

```

Listado 3.85: Implementación inicial del constructor de la clase `cl_analizador`.

Retomando el hilo temático de las complejidades genéricas descritas anteriormente, fueron declaradas dos versiones del método `obtener_complejidad`, una para métodos contenidos y otra para funciones, el propósito de estos métodos es retornar la complejidad temporal que aporta una llamada a una función a partir de la complejidad genérica que tiene asignada.

En el código plasmado en el Listado 3.86 se muestra la implementación del método `obtener_complejidad` para funciones no contenidas, dentro de éste se almacena la complejidad genérica en el objeto `complejidad` y por medio de una estructura de datos asociativa se establece una relación identificador genérico-identificador de llamada, donde se mantiene una relación identificador genérico-identificador genérico para aquellos identificadores a los que no se les haya encontrado una equivalencia. Una vez establecida la relación fueron reemplazados todos los identificadores dentro de `complejidad` por sus identificadores equivalentes y esta nueva forma del objeto fue retornada.

```

1 bool cl_analizador::estaPredefinida(string identificador){
2     if(identificador == "cin") return true;

```

```
3     if(identificador == "cout") return true;
4     if(identificador == "swap") return true;
5     ...
6     return false;
7
8 }
9
10 //asume que no esta contenida
11 cl_complejidad cl_analizador::obtener_complejidad(cl_llamada_funcion*
    llamada){
12     string identificador_funcion = llamada->identificador;
13     cl_complejidad complejidad;
14
15     complejidad = complejidad_funcion[identificador_funcion];
16
17     string identificador_generico;
18     string identificador_final;
19     map<string,string> idGenerico_a_idFinal;
20
21     if(estaPredefinida(llamada->identificador)){
22         identificador_generico = "n";
23         identificador_final = llamada->argumentos[0]->identificador;
24
25         idGenerico_a_idFinal[identificador_generico] = identificador_final
26     ;
27     }else{
28         for(cl_sumando& sumando: complejidad.sumandos){
29             for(cl_factor& factor: sumando.factor){
30                 identificador_generico = factor.identificador;
31                 identificador_final = factor.identificador;
32                 idGenerico_a_idFinal[identificador_generico] =
33                 identificador_final;
```

```

32         }
33     }
34     for(cl_declaracion argumento: definiciones[identificador_funcion
]->argumentos){
35         identificador_generico = argumento.identificador;
36         //asume que el argumento es un identificador
37         identificador_final = argumento.identificador;
38         idGenerico_a_idFinal[identificador_generico] =
identificador_final;
39     }
40 }
41
42
43     for(cl_sumando& sumando: complejidad.sumandos){
44         for(cl_factor& factor: sumando.factor){
45             if(factor.esConstante()) continue;
46             factor.identificador = idGenerico_a_idFinal[factor.
identificador];
47         }
48     }
49     return complejidad;
50 }

```

Listado 3.86: Implementación del método `obtener_complejidad` para funciones no contenidas.

La implementación del método `obtener_complejidad` para los métodos contenidos demostró ser más directa, puesto a que la complejidad genérica está definida en función de la cardinalidad del contador desde donde se llama. Por lo que se estableció una equivalencia entre la variable `n` usada en la complejidad genérica y el identificador del contenedor, para después modificar la complejidad reemplazando todas las instancias del identificador `n` por su identificador equivalente. Esto es ilustrado por el código contenido en el Listado 3.87.

```
1 cl_complejidad cl_analizador::obtener_complejidad(string
    identificador_funcion, string identificador, cl_tipoDato tipoContenedor){
2     string contenedor = tipoContenedor.tipo;
3     cl_complejidad complejidad;
4
5     complejidad = complejidad_funcion_contenida[contenedor][
    identificador_funcion];
6
7     map<string, string> idGenerico_a_idFinal;
8     idGenerico_a_idFinal["n"] = identificador;
9
10    for(cl_sumando& sumando: complejidad.sumandos){
11        for(cl_factor& factor: sumando.factoros){
12            if(factor.esConstante()) continue;
13            factor.identificador = idGenerico_a_idFinal[factor.
    identificador];
14        }
15    }
16
17    return complejidad;
18 }
```

Listado 3.87: Implementación del método `obtener_complejidad` para los métodos contenidos.

A modo de preámbulo al resto de los métodos de análisis y con el fin de modularizar fragmentos de código frecuentemente utilizados a lo largo del resto del código fuente, fue definida la función `analizar` para la clase `cl_bloque_codigo`, donde la complejidad aportada por este elemento fue la suma de todos los elementos sintácticos que éste contiene. En el código plasmado en el Listado 3.88 se muestra la implementación de la función, de la cual cabe resaltar que el orden de análisis no es arbitrario, pues es necesario analizar primero las declaraciones con el fin de poblar la estructura de datos asociativa que mantiene registro del



tipo de dato de cada variable definida.

```
1 cl_complejidad cl_analizador::analizar(cl_bloque_codigo* actual){
2     cl_complejidad complejidad;
3
4     //declaraciones
5     for(cl_declaracion* declaracion_actual: actual->declaraciones){
6         complejidad = complejidad + analizar(declaracion_actual);
7     }
8     //aritmeticas
9     for(cl_expresion_aritmetica* aritmetica_actual: actual->aritmeticas){
10         complejidad = complejidad + analizar(aritmetica_actual);
11     }
12     //booleanas
13     for(cl_expresion_booleana* booleana_actual: actual->booleanas){
14         complejidad = complejidad + analizar(booleana_actual);
15     }
16     //ciclos
17     for(cl_ciclo* ciclo_actual: actual->ciclos){
18         complejidad = complejidad + analizar(ciclo_actual);
19     }
20     //ifs
21     for(cl_if* if_actual: actual->if){
22         complejidad = complejidad + analizar(if_actual);
23     }
24     return complejidad;
25 }
```

Listado 3.88: Implementación del método `analizar` para la clase `cl_bloque_codigo`.

El método de análisis de complejidad correspondiente a la clase `cl_declaracion` es definido de manera concisa, pues como se muestra en el código contenido en el Listado 3.89, éste es el encargado de poblar la estructura de datos que asocia cada identificador con su tipo de

dato, asimismo la complejidad aportada por una declaración es considerada constante por el prototipo salvo que sea inicializada con una expresión aritmética, en cuyo caso adopta la complejidad que ésta aporta.

```
1 {cpp}
2 cl_complejidad cl_analizador::analizar(cl_declaracion* actual){
3     tipado_de_variable[actual->identificador] = *(actual->tipo);
4
5     cl_complejidad complejidad;
6
7     if(actual->inicializada){
8         complejidad = complejidad + analizar(actual->valor_predeterminado)
9     };
10    }
11    return complejidad;
12 }
```

Listado 3.89: Implementación del método `analizar` para la clase `cl_declaracion`.

Enfatizando sobre lo discutido en la Sección 3.4.4, en el caso de las expresiones aritméticas, el analizador debe de ser capaz de no solo identificar la complejidad que estas aportan, sino también identificar cómo aceptan la tasa en la que las variables crecen/decrecen como resultado de asignaciones o llamadas a métodos.

La implementación del método `analizar` para expresiones aritméticas plasmado en el código contenido en el Listado 3.90, calcula la complejidad temporal aportada por la expresión bajo el supuesto que toda expresión aritmética es ejecutada en un tiempo constante con la excepción de aquellas que contengan una llamada a una función o método, incluyendo el método de acceso [].

El conteo de las ocurrencias de un identificador en una expresión dada fue realizado a través de la estructura `apariciones_en_aritmetica`, la cual es vaciada al inicio de cada expresión aritmética. Debido a la naturaleza de la forma en la que se realizó el conteo, las

reglas que dependen del conteo de variables fueron modificadas para acomodar el hecho de que el identificador en el lado izquierdo del operador de asignación también es contado.

Con respecto al resto de las reglas, su implementación no difiere con lo especificado en la Sección 3.4.4 y la tasa de crecimiento de una variable es almacenada dentro de la estructura de datos asociativa `tasa_crecimiento_variable`, donde el nivel de anidamiento solo cambia al interactuar con estructuras iterativas.

```

1  cl_complejidad cl_analizador::analizar(cl_expresion_aritmetica* actual){
2      cl_complejidad complejidad;
3      if(actual->es_terminal && actual->es_llamada_funcion){
4          string id_llamada = actual->llamada_val->identificador;
5          if(actual->llamada_val->esta_contenida){
6              string id_cont = actual->identificador;
7              complejidad = complejidad + obtener_complejidad(id_llamada,
8              id_cont,tipado_de_variable[actual->identificador]);
9          }else{
10             complejidad = complejidad + obtener_complejidad(actual->
11             llamada_val);
12
13             if(actual->llamada_val->esta_contenida && modifica_tam_contenedor[
14             id_llamada]){
15                 tasa_crecimiento_variable[anidamiento_actual][id_llamada] = 1;
16             }
17         }
18         if(actual->es_terminal && actual->es_posicion_arreglo){
19             complejidad = complejidad + obtener_complejidad("[ ]", actual->
20             identificador,tipado_de_variable[actual->identificador]);
21             apariciones_en_aritmetica[actual->identificador]++;
22         }
23         if(actual->es_terminal && actual->es_identificador){
24             apariciones_en_aritmetica[actual->identificador]++;
25         }
26     }
27 }

```

```
22     }
23     if(actual->es_terminal) return complejidad;
24     //asume que esto se cumple
25     //if(actual->es_terminal == false)
26     complejidad = complejidad + analizar(actual->izquierda);
27     complejidad = complejidad + analizar(actual->derecha);
28
29     string identificador = actual->izquierda->identificador;
30     //Tasas de crecimiento, se asumen que en este solo pueden cambiar
dentro de una asignacion
31     //0->constante, 1->líneal, 2->logaritmica/exponencial
32     if(actual->operador >=2){
33         tasa_crecimiento_variable[anidamiento_actual][identificador] = 1;
34         apariciones_en_aritmetica.clear();
35     }
36     if(actual->operador ==2){
37         if(actual->derecha->es_terminal == false &&
apariciones_en_aritmetica[identificador]>=2){
38             tasa_crecimiento_variable[anidamiento_actual][identificador] =
2;
39         }
40     }
41     if(actual->operador ==3){
42         tasa_crecimiento_variable[anidamiento_actual][identificador] = 2;
43     }
44     if(actual->operador == 4){
45         if(actual->derecha->es_terminal == false && actual->derecha->
operador == 1){
46             tasa_crecimiento_variable[anidamiento_actual][identificador] =
2;
47         }
48         if(actual->derecha->es_terminal == false &&
```

```

    apariciones_en_aritmetica[identificador]>=3){
49         tasa_crecimiento_variable[anidamiento_actual][identificador] =
        2;
50     }
51     if(actual->derecha->es_terminal == true && actual->derecha->
    es_identificador == true){
52         tasa_crecimiento_variable[anidamiento_actual][identificador] =
        tasa_crecimiento_variable[anidamiento_actual][actual->derecha->
    identificador];
53     }
54 }
55
56 return complejidad;
57 }

```

Listado 3.90: Implementación del método `analizar` para la clase `cl_expresion_aritmetica`.

De acuerdo a lo discutido en la Sección 3.4.4, una expresión booleana no solo aporta complejidad temporal sino que también expresa la tasa en la que dos variables se aproximan una a la otra, esto es particularmente útil para estimar la complejidad que ciclos y funciones recursivas aportan, por lo que se agregó a la clase `cl_expresion_booleana` un objeto de tipo `cl_complejidad` junto con el atributo `bool es_logaritmica`. Fueron también creadas funciones para la creación de objetos de tipo `cl_complejidad` que expresan las clases  $O(1)$ ,  $O(\log(n))$  y  $O(n)$ .

En el código mostrado en el Listado 3.91, se puede apreciar la implementación del método `analizar` para las expresiones booleanas. La complejidad temporal aportada por una expresión booleana es igual a la suma de la complejidad aportada por las expresiones que la componen.

En cuanto a la tasa en la que se aproximan los identificadores que forman una expresión booleana, una expresión booleana formada por la unión de dos expresiones booleanas unidas

por un operador de comparación, como  $<$ ,  $<=$ ,  $=$ ,  $>$ ,  $>=$ , es considerada como una expresión con crecimiento logarítmico si alguno de los identificadores que la conforman tienen una tasa de crecimiento *exponencial*, denotada por el valor entero 2. Una expresión booleana resultante de la unión de dos expresiones mediante el operador  $||$  es logarítmica si ambas expresiones lo son, mientras que una expresión unida por el operador  $\&\&$  es logarítmica si cualquiera de los dos elementos que la componen lo es.

```
1 cl_complejidad cl_analizador::analizar(cl_expresion_booleana* actual){
2     cl_complejidad complejidad;
3     if(actual->es_terminal){
4         complejidad = complejidad + analizar(actual->valor);
5         return complejidad;
6     }else{
7         complejidad = complejidad + analizar(actual->izquierda);
8         complejidad = complejidad + analizar(actual->derecha);
9     }
10    if(actual->es_conector==3){
11        if(actual->izquierda->es_logaritmica){
12            complejidad = actual->izquierda->complejidad;
13            actual->es_logaritmica = actual->izquierda->es_logaritmica;
14        }else{
15            complejidad = actual->derecha->complejidad;
16            actual->es_logaritmica = actual->derecha->es_logaritmica;
17        }
18    }
19    //
20    if(actual->es_conector==2){
21        if(actual->izquierda->es_logaritmica == false){
22            complejidad = actual->izquierda->complejidad;
23            actual->es_logaritmica = actual->izquierda->es_logaritmica;
24        }else{
25            complejidad = actual->derecha->complejidad;
```

```

26         actual->es_logaritmica = actual->derecha->es_logaritmica;
27     }
28 }
29 if(actual->es_conector>=2){
30     return complejidad;
31 }
32 //asume que la desigualdad es en funcion de constantes numericas o
33 //es una expresion de la forma a<b, a>b, a<=b ...
34 string id_izq;
35 string id_der;
36
37 if(actual->izquierda->valor->es_constante){
38     //como una constante no puede ser un identificador, su valor en
39     //tasa_crecimiento sera 0 i.e. constante
40     id_izq = to_string((long long int)actual->izquierda->valor->
41     valor_numerico);
42 }else{
43     id_izq = actual->izquierda->valor->identificador;
44 }
45
46 if(actual->derecha->valor->es_constante){
47     //como una constante no puede ser un identificador, su valor en
48     //tasa_crecimiento sera 0 i.e. constante
49     id_der = to_string((long long int)actual->derecha->valor->
50     valor_numerico);
51 }else{
52     id_der = actual->derecha->valor->identificador;
53 }
54
55 //<
56 int tasa_maxima = tasa_crecimiento_variable[anidamiento_actual][id_izq

```

```

];
53     int tasa_maxima = max(tasa_maxima,tasa_crecimiento_variable[
anidamiento_actual][id_der]);
54     if(actual->es_conector == -1){
55         if(tasa_maxima==2){
56             actual->es_logaritmica = true;
57             actual->complejidad = crea_o_logaritmica(id_der);
58         }else{
59             actual->complejidad = crea_o_lineal(id_der);
60         }
61     }
62     //> ,==
63     if( actual->es_conector == 1 || actual->es_conector == 0){
64         if(tasa_maxima==2){
65             actual->es_logaritmica = true;
66             actual->complejidad = crea_o_logaritmica(id_izq);
67         }else{
68             actual->complejidad = crea_o_lineal(id_izq);
69         }
70     }
71
72     return complejidad;
73 }

```

Listado 3.91: Implementación del método `analizar` para la clase `cl_expresion_booleana`.

La implementación del método `analizar` para la estructura de control de flujo `if`, tiene como objetivo solamente calcular la complejidad aportada por ésta. Como se muestra en el código plasmado en el Listado 3.92, la complejidad aportada por esta estructura es igual a la suma de la complejidad resultante de evaluar la expresión booleana asociada como su argumento y de la suma de los elementos sintácticos contenidos tanto en el bloque de código principal, así como del bloque de código alternativo en caso de contar con uno.



```
1 cl_complejidad cl_analizador::analizar(cl_if* actual){
2     cl_complejidad complejidad;
3
4     complejidad = complejidad + analizar(actual->bloque_if);
5
6     if(actual->tiene_else){
7         complejidad = complejidad + analizar(actual->bloque_else);
8     }
9
10    complejidad = complejidad + analizar(actual->argumento);
11
12    return complejidad;
13 }
```

Listado 3.92: Implementación del método `analizar` para la clase `cl_if`.

De una manera similar, la complejidad temporal aportada por una estructura iterativa está expresada en función de la complejidad aportada por el bloque de código asociado a ésta y aquella de los argumentos.

En el código contenido en el Listado 3.93 se muestra la implementación de las reglas definidas en la Sección 3.4.4, donde antes de calcular la complejidad resultante de los elementos contenidos en el bloque de código, se incrementa la variable que denota el índice de anidamiento de ciclos actual y es limpiada la tabla que mantiene la tasa de crecimiento de las variables presentes en ese nivel. Una vez calculada la complejidad temporal de los contenidos del bloque de código, es analizada la complejidad del incremento (*step*) realizado en cada iteración, para después calcular la complejidad aportada por la expresión booleana que funge como argumento de continuación. Finalmente, la complejidad aportada por un ciclo es igual al producto de la suma de todas las complejidades ya enlistadas por la función que define la tasa en la que el argumento de continuación cambia a falso.

```
1 cl_complejidad cl_analizador::analizar(cl_ciclo* actual){
```

```
2   cl_complejidad complejidad;
3   anidamiento_actual++;
4   tasa_crecimiento_variable[anidamiento_actual].clear();
5
6   complejidad = complejidad + analizar(actual->bloque_codigo);
7
8   if(actual->step != NULL){
9       complejidad = complejidad + analizar(actual->step);
10  }
11
12  complejidad = complejidad + analizar(actual->argumento_continuacion);
13
14  complejidad = complejidad * actual->argumento_continuacion->
complejidad;
15
16  anidamiento_actual--;
17  return complejidad;
18 }
```

Listado 3.93: Implementación del método `analizar` para la clase `cl_ciclo`.

Una vez definido el método `analizar` para todos los elementos sintácticos que pueden estar presentes en la definición de una función no recursiva, es posible estimar su complejidad temporal. Como se muestra en el código plasmado en el Listado 3.94, se considera que la complejidad temporal aportada por una declaración de una función no recursiva es igual a la suma de la complejidad sintácticos contenidos en el bloque de código asociado a ella.

```
1 cl_complejidad cl_analizador::analizar(cl_declaracion_funcion* actual){
2     tipado_de_variable.clear();
3     tasa_crecimiento_variable.clear();
4
5     cl_complejidad complejidad;
6     cl_bloque_codigo* bloque_codigo = actual->bloque_codigo;
```

```
7 //La complejidad de una funcion es igual a la suma de las
instrucciones en su definicion
8 complejidad = complejidad + analizar(bloque_codigo);
9
10 complejidad_funcion[actual->identificador] = complejidad;
11 return complejidad;
12 }
```

Listado 3.94: Implementación del método `analizar` para la clase `cl_declaracion_funcion`.

Finalmente, fue implementado el método `analizar_rekursiva` como un método análogo al método `analizar`, donde fueron adaptados algunos aspectos de la implementación a la estructura de las funciones recursivas de cola.

También fueron modificadas las implementaciones del método `analizar` para la estructura `if` y las expresiones aritméticas, esto con el fin de facilitar la implementación de las reglas definidas en la Sección 3.4.4.

El método `analizar` para la estructura `if` fue modificado con el fin de identificar la condición que lleva a la ejecución de la instrucción `return` que representa el caso base, o dicho de otra manera la instrucción `return` que no realiza una llamada recursiva.

Como se muestra en el código plasmado en el Listado 3.95, en caso de que la estructura `if` cuente con una instrucción de tipo `return` y ésta no represente un paso recursivo, la condición que lleva a esta instrucción es almacenada en la variable global `retorno_pendientes`, la cual será utilizada. Para lograr esto, fue creada la función `contiene_llamada`, la cual de manera recursiva verifica si se encuentra una llamada a una función en particular.

```
1 bool contiene_llamada(cl_expresion_aritmetica* actual, string
identificador_objetivo){
2     if(actual->es_terminal){
3         if(actual->es_llamada_funcion && actual->es_llamada_funcion){
4             return (actual->llamada_val->identificador==
```

```

    identificador_objetivo);
5     }
6     return false;
7 }
8     return (contiene_llamada(actual->izquierda, identificador_objetivo) ||
    contiene_llamada(actual->derecha, identificador_objetivo));
9 }
10
11 cl_complejidad cl_analizador::analizar(cl_if* actual){
12     ...
13     if(actual->bloque_if->expresion_return != NULL){
14         if(contiene_llamada(actual->bloque_if->expresion_return,
    identificador_declaracion_funcion) == false){
15             retorno_pendientes = actual->argumento;
16         }
17     }
18     return complejidad;
19 }

```

Listado 3.95: Modificaciones a la implementación del método `analizar` de la clase `cl_if` para soportar llamadas recursivas de cola.

Como ya ha sido mencionado anteriormente, también fue modificada la implementación del método `analizar` para las expresiones aritméticas, pues es dentro de ellas donde puede ser identificada la tasa de crecimiento/decrecimiento de las variables utilizadas dentro de la llamada recursiva. Como se muestra en el código contenido en el Listado 3.96, esto se logró asignando a estas variables el nivel especial de anidación de `-1`, donde la tasa de crecimiento fue analizada por medio de la creación de una expresión aritmética de asignación ficticia, donde al argumento genérico le es asignado el argumento de la llamada.

```

1 cl_complejidad cl_analizador::analizar(cl_expresion_aritmetica* actual){
2     cl_complejidad complejidad;

```

```
3     if(actual->es_terminal && actual->es_llamada_funcion){
4         if(actual->llamada_val->esta_contenida){
5             ...
6         }else{
7             ...
8             //llamada recursiva
9             if(identificador_declaracion_funcion == actual->llamada_val->
identificador){
10
11                 int anidamiento_original = anidamiento_actual;
12                 anidamiento_actual = -1;
13                 cl_expresion_aritmetica* asignacion_por_llamada;
14                 asignacion_por_llamada->izquierda = new
cl_expresion_aritmetica;
15                 asignacion_por_llamada->izquierda->es_terminal = true;
16                 asignacion_por_llamada->izquierda->es_identificador = true
;
17                 asignacion_por_llamada->operador = 4;
18
19                 int ind = 0;
20                 cl_declaracion_funcion* decl_actual = definiciones[
identificador_declaracion_funcion];
21
22                 for(cl_expresion_aritmetica* argumento : actual->
llamada_val->argumentos){
23                     asignacion_por_llamada->izquierda->identificador =
decl_actual->argumentos[ind].identificador;
24                     asignacion_por_llamada->derecha = argumento;
25
26                     analizar(asignacion_por_llamada);
27                     ind++;
28                 }
```

```
29         anidamiento_actual = anidamiento_original;
30     }
31
32 }
33 ...
34 }
35 ...
36 return complejidad;
37 }
```

Listado 3.96: Modificaciones a la implementación del método `analizar` de la clase `cl_expression_aritmetica` para soportar llamadas recursivas de cola.

La implementación del método `analizar_recursiva` plasmada en el código mostrado en el Listado 3.97 ilustra la implementación de las reglas definidas en la Sección 3.4.4. En dicha implementación se tiene que la complejidad de una función recursiva es calculada asignando la complejidad ficticia de  $O(1)$  a ésta, esto con el fin de evitar comportamientos no esperados que influyan en la complejidad final a causa de las llamadas recursivas realizadas dentro de la definición de la función. De una manera similar a la complejidad obtenida por estructuras iterativas, la complejidad de una función recursiva de cola es expresada en función de la complejidad aportada por los elementos sintácticos contenidos en su bloque de código correspondiente a su definición, donde la expresión de continuación es representada por la expresión booleana `retorno_pendientes`, la cual es analizada con el valor especial de anidamiento de `-1` reservado para la tasa de crecimiento de los argumentos presentes en las llamadas recursivas. Finalmente, se tiene que la complejidad de una llamada recursiva es igual al producto de la complejidad aportada por el bloque de código multiplicada por la función de complejidad que expresa la tasa en la que la expresión booleana `retorno_pendientes` cambia su estado.

```
1 cl_complejidad cl_analizador::analizar_recursiva(cl_declaracion_funcion*
    actual){
```

```
2    tipado_de_variable.clear();
3    tasa_crecimiento_variable.clear();
4    complejidad_funcion[actual->identificador] = crea_o_constante();
5
6    cl_complejidad complejidad;
7    cl_bloque_codigo* bloque_codigo = actual->bloque_codigo;
8    //La complejidad de una funcion es igual a la suma de las
    instrucciones en su definicion
9    complejidad = complejidad + analizar(bloque_codigo);
10
11    int anidamiento_original = anidamiento_actual;
12    anidamiento_actual = -1;
13    analizar(retorno_pendientes);
14    anidamiento_actual = anidamiento_original;
15
16    complejidad = complejidad * retorno_pendientes->complejidad;
17
18    complejidad_funcion[actual->identificador] = complejidad;
19    return complejidad;
20 }
```

Listado 3.97: Definición del método `analizar_rekursiva`.

### Sobre la función `analizador_complejidad`

Una vez definida la clase que encapsula los métodos de análisis de complejidad, es definida la función con la que el módulo coordinador interactuará y es encargada de calcular la complejidad del programa en su totalidad.

La función `analizador_complejidad` descrita en el código plasmado en el Listado 3.98, recibe como argumento el ordenamiento topológico obtenido por el módulo de análisis de jerarquía y dependencia, el cual pasa al constructor de la clase `cl_analizador`. Después,

itera sobre el ordenamiento topológico, llamando el método de análisis correspondiente a cada declaración de acuerdo si es recursiva o iterativa, para después sumar la complejidad aportada si la función analizada se trata de la función `main`. Finalmente, imprime la complejidad temporal obtenida con el método definido dentro de la clase `cl_complejidad`.

```

1 void analizador_complejidad(vector<cl_declaracion_funcion*>
  orden_topologico){
2     cl_analizador analizador(orden_topologico);
3     cl_complejidad complejidad;
4     cl_complejidad comp_funcion;
5     for(cl_declaracion_funcion* declaracion: orden_topologico){
6         analizador.identificador_declaracion_funcion = declaracion->
identificador;
7         if(declaracion->es_recursiva){
8             comp_funcion = analizador.analizar_recursiva(declaracion);
9         }else{
10            comp_funcion = analizador.analizar(declaracion);
11        }
12        if(declaracion->identificador == "main"){
13            complejidad = complejidad + comp_funcion;
14        }
15    }
16    complejidad.imprimir_expresion();
17 }

```

Listado 3.98: Definición de la función `analizador_complejidad`.

### 3.5.5. Implementación del módulo coordinador y proceso de compilado

Durante la Sección 3.1, fue mencionado un módulo para coordinar la ejecución del resto de los módulos, éste fue creado dentro del archivo `coordinador.cpp` cuya implementación



está plasmada en el Listado 3.99.

Dentro del módulo coordinador, fue declarada a través de la palabra reservada **extern** la existencia de las funciones `yyparse` y `analizador_complejidad`, cuya definición se encuentra en archivos externos a `coordinador.cpp`.

La función `yyparse` invoca al proceso de análisis léxico y análisis sintáctico, mientras que `analizador_complejidad` invoca al proceso de análisis de complejidad central al módulo con el mismo nombre. El análisis de jerarquía y precedencia se encuentra encapsulado en la clase `analizador_precedencia`, por lo que es necesario crear un objeto de dicha clase para poder usar los métodos de análisis definidos en ella.

```
1 #include "nodos.h"
2 #include "analizadorPrecedencia.hpp"
3 extern cl_raiz* raiz;
4 extern int yyparse();
5 extern void analizador_complejidad(std::vector<cl_declaracion_funcion*>
   orden_topologico);
6
7 int main(int argc, char **argv)
8 {
9     yyparse();
10    analizador_precedencia analizadorPrecedencia(raiz);
11
12    bool tiene_dependencias_circulares;
13    tiene_dependencias_circulares = analizadorPrecedencia.esRekursivo();
14
15    if(tiene_dependencias_circulares){
16        cout<<"ERROR: El programa a analizar contiene dependencias
   circulares.\n";
17    }
18    analizador_complejidad(analizadorPrecedencia.orden_topologico);
19    return 0;
```

20 }

Listado 3.99: Implementación del módulo coordinador.

El código fuente discutido a lo largo de esta sección puede ser encontrado en el repositorio <https://github.com/MAlexxiT/automated-complexity-analysis/tree/main/src>.

## 3.6. Validación

A lo largo de la presente sección serán discutidos los cambios realizados de manera retroactiva a módulos considerados ya implementados. Estos cambios fueron realizados con el fin de mantener una mejor calidad dentro del producto o para eliminar un comportamiento problemático de un módulo para otro.

### Sobre los operadores `++` y `-`

La inclusión de los operadores de incremento y decremento fue omitida de forma accidental durante la fase de diseño de análisis léxico y de análisis sintáctico, esto fue corregido terminada la implementación del módulo de análisis sintáctico.

En el archivo que contiene las expresiones que definen a las palabras del lenguaje, fueron creados los patrones descritos en el código contenido en el Listado 3.100 y colocados antes de la definición de los *tokens* `sum` y `sub` para asegurar la precedencia de estas expresiones.

```
1 {SUM}{2} {  
2     return t_sumsum;  
3 }
```

Listado 3.100: Expresiones regulares correspondientes a los operadores `++` y `-`.

Dentro del archivo que contiene las reglas gramaticales del lenguaje, fueron declarados los *tokens* asignándoles el atributo simbólico `nada`, además fueron agregadas reglas de producción

al símbolo no terminal `expresionAritmetica`, ilustradas en el Listado 3.101, estas reglas tratan al operador como un incremento/decremento de una unidad denotado por `+=1` y `-=1`, lo cual permite su añadidura a las reglas del lenguaje con mínimos cambios.

```
1 %token<nada> t_sumsum t_subsub
2 expresionAritmetica:
3     ...
4     |expresionAritmetica t_sumsum {
5         //beginning of id related rules
6         $$= new cl_expresion_aritmetica;
7         $$->es_terminal = false;
8         $$->izquierda = $1;
9
10        $$->operador = 2;
11
12        $$->derecha = new cl_expresion_aritmetica;
13        $$->derecha->es_terminal = true;
14        $$->derecha->es_constante = true;
15        $$->derecha->valor_numerico = 1;
16    }
17    |expresionAritmetica t_subsub {
18        $$= new cl_expresion_aritmetica;
19        $$->es_terminal = false;
20        $$->izquierda = $1;
21
22        $$->operador = 2;
23
24        $$->derecha = new cl_expresion_aritmetica;
25        $$->derecha->es_terminal = true;
26        $$->derecha->es_constante = true;
27        $$->derecha->valor_numerico = 1;
28    }
```

29

...

Listado 3.101: Reglas de producción correspondientes a los operadores ++ y −.

### Sobre arreglos de múltiples dimensiones

De una manera similar a los operadores de incremento/decremento, se descubrió después de concluida la implementación del analizador sintáctico el hecho de que éste no era capaz de identificar el uso de arreglos de más de una dimensión como un elemento sintáctico válido.

La regla de producción implementada durante la fase de análisis sintáctico y descrita en el código 3.9 solo es capaz de reconocer arreglos de una sola dimensión, esto se debe a que los *tokens* correspondientes a los caracteres [], así como la expresión aritmética contenida entre ellos, fue especificada como una sola en lugar de una o más.

Este comportamiento fue corregido mediante la introducción de los símbolos no terminales `dimension` y `dimensiones`. Como se muestra en el código plasmado en el Listado 3.102, el símbolo `dimension` representa un par de corchetes y una expresión aritmética entre ellos, mientras que el símbolo `dimensiones` representa la presencia de cero o más símbolos `dimension`.

```

1 %type<nada> dimensiones dimension
2 ...
3 dimension:
4     t_opsqrb expresionAritmetica t_closqrb{ $$ = 0;}
5 ;
6 dimensiones:
7     /*empty*/ {}
8     |dimension {}
9     |dimensiones dimension { }
```

10 ;

Listado 3.102: Reglas de producción correspondientes a los símbolos `dimension` y `dimensiones`.

Una vez declarados los símbolos `dimension` y `dimensiones` junto con sus reglas de producción, es redefinida la gramática de manera en que su regla de producción esté escrita en función de estos nuevos símbolos. El resultado obtenido por estos cambios es ilustrado por el Listado 3.103.

```
1 expresionAritmetica:
2 ...
3     |expresionAritmetica dimension dimensiones {
4         $$= new cl_expresion_aritmetica;
5         $$→es_terminal = true;
6         $$→es_identificador = true;
7         $$→identificador = $1→identificador;
8         $$→es_posicion_arreglo = true;
9         delete $1;
10    }
11 ...
12 ;
```

Listado 3.103: Regla de producción relacionada con expresiones aritméticas formadas por accesos a posiciones en arreglos.

### Sobre estructuras `if` con más un `else`

En la Sección 3.4.2, donde se describe la gramática que define a las estructuras iterativas y de control de flujo, se menciona el hecho de que debido a la forma que tiene la gramática que define a las estructuras `if` existe la posibilidad de que a un `if` dado le sea asociado

más de un `else`. Esta posibilidad no fue abordada durante la implementación del módulo de análisis sintáctico, por lo que fue necesario modificar este módulo después de concluida su codificación. Dicha modificación, ilustrada en el Listado 3.104, consistió en una verificación donde se le informa al usuario de la existencia de este error en caso de intentar asignar un `else` a un `if` que ya tenga uno asociado.

```
1  ifStatement:
2  ...
3      | ifStatement t_else bloqueCodigo{
4          if($1→tiene_else){
5              yyerror("Un if tiene mas de un else");
6          }
7          $1→tiene_else = true;
8          $1→bloque_else = $3;
9          $$→tiene_break = $3→tiene_break;
10     }
11 ;
```

Listado 3.104: Verificación de la existencia de un `else` dentro de la regla del símbolo `ifStatement`.

# Capítulo 4

## Resultados y discusiones

A lo largo de este capítulo serán descritas una serie de pruebas funcionales orientadas a evaluar la capacidad del prototipo de estimar la complejidad temporal de manera precisa, además serán presentados y analizados los resultados de dichas pruebas.

### 4.1. Resultados

Las pruebas realizadas consisten en la evaluación de una serie de programas de C++ escritos por terceros y modificados con el fin de que cumplan con la estructura e instrucciones necesarias para ser analizables por el prototipo.

Los programas a evaluar fueron recolectados de soluciones a problemas de programación competitiva de los jueces en línea [www.codeforces.com](http://www.codeforces.com) y [www.cses.fi](http://www.cses.fi), priorizando aquellos problemas y soluciones que maximizan la diversidad del banco de pruebas en cuanto a la estructura, complejidad temporal o instrucciones utilizadas.

#### 4.1.1. Recolección y etiquetado de soluciones

La programación competitiva es un deporte donde los participantes resuelven problemas de desarrollo de algoritmos a través del desarrollo de programas en algún lenguaje de pro-

gramación. Los problemas dentro de esta disciplina suelen ser descritos siguiendo un formato donde se le asigna un tiempo y memoria límite a la solución del participante, un título único al problema, un enunciado donde es descrito el problema a través de una historia, la descripción del formato de entrada y del formato de salida; mantener el enunciado del problema fue considerado superfluo por el autor de este documento, por lo tanto solo se mantuvo una liga al enunciado del problema en un comentario dentro del encabezado de cada uno de los códigos recopilados, los cuales fueron anexados en la sección de apéndices.

Los programas escritos dentro del contexto de la programación competitiva suelen ser optimizados en cuanto a la longitud de su código fuente, así como en tiempo de ejecución, estas modificaciones no suelen influir en la complejidad temporal de la solución por lo que es posible revertirlas con el fin de mejorar su compatibilidad con el prototipo.

Fueron recolectados treinta y cinco códigos, los cuales fueron modificados a fin de mejorar su compatibilidad con el sistema, además fueron etiquetados de manera manual con su complejidad temporal real, expresada en términos de las variables definidas dentro del código. Una vez realizado esto, fueron alimentados al prototipo, el cual a su vez devolvió una complejidad temporal estimada, los resultados de este proceso pueden verse en las Tablas 4.1 y 4.2, donde las filas marcadas de rojo indican códigos para los cuales se considera diferentes las complejidades obtenidas de manera manual y automatizada, mientras que las filas marcadas de verde indican que las complejidades obtenidas son equivalentes a pesar de las diferencias en cuanto al polinomio que las representa.



Tabla 4.1: Comparación de la complejidad real con la complejidad obtenida por el prototipo sobre los códigos que conforman las pruebas principales.

Identificador	Complejidad real	Complejidad prototipo
CF-913A	$O(\log(n))$	$O(\log(n))$
CF-1490C	$O(10^4 * \log( is\_cube ) + t * \sqrt[3]{x})$	$O(1000000000000 * \log( is\_cube ) + x * t)$
CF-1612C	$O(t * \log(k))$	$O(t * \log(r))$
CF-1669C	$O(t * n^2)$	$O(t * n^2)$
CF-1702B	$O(ut * s)$	$O(ut * s * 3 * \log( used ))$
CF-1760C	$O(T * (n^2 + 200^2))$	$O(n^2 * T + n * T * 200 + p * T * 200)$
CF-1933C	$O(t * 400 * \log( ans ))$	$O(t * 20^2 * \log( ans ))$
CF-1937B	$O(t * n^2)$	$O(t * n^2)$
CF-102951B	$O(n * \log(n))$	$O( algoritmos  * \log( algoritmos ) + n)$
CF-103960A	$O( s )$	$O(tc *  s )$
CF-104375B	$O(1)$	$O(1)$
CSES-1636	$O(n * target)$	$O(target * n)$
CF-1450B	$O(nTest * n^2)$	$O(nTest * n^2)$
CF-1472B	$O(t * n)$	$O(t * n)$

<sup>1</sup> Los códigos correspondientes a las pruebas principales y complementarias pueden ser encontradas en el repositorio <https://github.com/MAlexxiT/automated-complexity-analysis/tree/main/test-data>.

Tabla 4.2: Comparación de la complejidad real con la complejidad obtenida por el prototipo sobre los códigos que conforman las pruebas complementarias.

Identificador	Complejidad real	Complejidad prototipo
CF-100187D	$O(\log(n))$	$O(\log(n))$
CF-100187L	$O(N)$	$O(N)$
CF-101502A	$O(t)$	$O(t)$
CF-101502F	$O(tt * n * \log(num) + tt * q)$	$O(tt * n * \log(num) + tt * q)$
CF-101908I	$O(n * m + l)$	$O(m + l + n * k)$
CF-102861A	$O(n + l)$	$O(l + n)$
CF-102861F	$O(n)$	$O(t * n * gr + t * pl * n + t * pr * n)$
CF-103061B	$O(s)$	$O(s)$
CF-103061J	$O(n)$	$O(n)$
CF-103274C	$O(ms * \log(\log(ms)) + q)$	$O(t * ms^2 + t * q)$
CF-104375D	$O((n + q) * \log n)$	$O(a * \log(a) + n + q * a + q * \log(a))$
CF-104375J	$O(n + q)$	$O(n + q)$
CF-104736F	$O(\sqrt{n} * (\log(n) + \log( res )))$	$O(n * \log(res) + n * \log(n) + t * n)$
CF-105164J	$O(N^2 * K)$	$O(N^2 * K)$
CF-1512F	$O(tt * n)$	$O(tt * n)$
CF-1676D	$O(n * m * T)$	$O(n * m * T)$
CF-1833D	$O(T * n)$	$O(i * T + id * T + n * T + a * T)$
CF-1840C	$O(ttt * n)$	$O(ttt * n^2)$
CF-1844D	$O(t * n)$	$O(t * n)$
CF-1909A	$O(n * T * \log(st))$	$O(n * T * \log(st))$
CF-1927D	$O(tc * n + tc * Q)$	$O(tc * n + tc * Q)$

## 4.2. Discusiones

De los catorce códigos que conforman las pruebas principales, la complejidad real coincidió con la complejidad estimada por el prototipo en ocho de ellos, donde la diferencia en el polinomio que representa la complejidad radica en el orden en el que fueron mostrados los factores o en la utilización de un factor equivalente, como es el caso de los factores 400 y  $20^2$  en el código CF-1933C.

Existen cinco códigos donde la complejidad real aparenta ser diferente a la complejidad estimada por el prototipo, es posible argumentar a favor de su equivalencia y a su vez detectar áreas de mejora.

Considere el fragmento de código correspondiente a CF-103960A, contenido en el Listado 4.1, el análisis de complejidad manual y el automatizado coinciden en que la complejidad que la función `uwo` aporta al código es de  $O(|s|)$ , sin embargo el prototipo no fue capaz de determinar que la variable `tc` representa la constante entera 1, lo cual provocó que la complejidad fuera expresada en términos de esta variable a pesar de solo haber sido realizada una sola llamada a `uwo`, sin embargo una vez aclarado esto, es posible defender la equivalencia de ambas complejidades.

```
1 void uwo(){
2     ...
3 }
4 int main(){
5     int tc=1;
6     //cin>>tc;
7     while(tc>0){
8         tc--;
9         uwo();
10    }
11    return 0;
```

12 }

Listado 4.1: Fragmento de código correspondiente a CF-103960A.

De manera similar, en el fragmento de código CF-1702B contenido en el Listado 4.2, es posible observar el hecho de que la cardinalidad de `used` nunca excede tres elementos, por lo que la expresión obtenida por el prototipo es equivalente a  $O(ut * s * 3 * \log(3))$ , lo cual a su vez es equivalente a  $O(ut * s)$  después de descartados los factores constantes.

```

1 ...
2 void solve()
3 {
4     string s;
5     ...
6     for(int i=0;i<s.size();ans++)
7     {
8         set<char>used;
9         while(used.size()<=3)
10        {
11            used.insert(s[i++]);
12        }
13        i--;
14    }
15    ...
16 }
17 ...

```

Listado 4.2: Fragmento de código correspondiente a CF-1702B.

En el código CF-1612C, la complejidad que el ciclo ilustrado en el Listado 4.3 aporta es de  $\log(2 \cdot k - 1)$ , donde  $2 \cdot k - 1$  es el valor inicial de la variable `r`, el prototipo fue incapaz de determinar esta equivalencia, por lo que expresó la complejidad en términos de `r`, lo cual a pesar de ser técnicamente correcto resulta ser una elección con menos significado comparado

con  $k$ .

```
1 ...
2 void solve() {
3     ...
4     long long int r = 2 * k - 1;
5     while (l < r) {
6         m = (l + r)/2;
7         if (emotes(m, k) >= x) {
8             r = m;
9         } else {
10            l = m;
11        }
12    }
13    ...
14 }
15 ...
```

Listado 4.3: Fragmento de código correspondiente a CF-1612C.

Algo similar sucede con el código CF-102951B, donde el fragmento plasmado en el Listado 4.4, muestra cómo la cardinalidad del vector `algoritmos` es de  $n$  elementos, lo cual el prototipo no fue capaz de concluir por lo que expresó la complejidad en términos de  $n$  y de la cardinalidad de `algoritmos`, a pesar de ser equivalentes.

```
1 void uwo(){
2     ...
3 }
4 int main(){
5 void solve() {
6     int n;
7     int a;
8     vector<int> algoritmos;
9     long long int x;
```

```
10  cin >> n >> x;
11  while (n>0) {
12      n--;
13      cin >> a;
14      algoritmos.push_back(a);
15  }
16  sort(algoritmos.begin(), algoritmos.end());
17  ...
18 }
```

Listado 4.4: Fragmento de código correspondiente a CF-102951B.

En el código CF-1760C, es posible argumentar por la equivalencia entre la complejidad real y la estimada por el prototipo en base a los valores que toma la variable de control  $p$ . Como se ilustra en el Listado 4.5, la variable  $t$ , sobre la cual depende la complejidad aportada por una estructura iterativa, crece de forma lineal en cuanto a la variable  $p$ , la cual a su vez crece de forma lineal en función del valor constante 200.

```
1  int main()
2  {
3      ...
4      while(T>0)
5      {
6          ...
7          for(int p=2;p<=200;p++)
8          {
9              ...
10             for(int j=1;j<=n;j++){
11                 cnt[a[j]%p]++;
12             }
13
14             ...
15             for(int t=0;t<p;t++){
```

```
16         ...
17     }
18
19     ...
20 }
21     ...
22 }
23 return 0;
24 }
```

Listado 4.5: Fragmento de código correspondiente a CF-1760C.

Finalmente, se tiene que el prototipo fue incapaz de calcular la complejidad de forma precisa para el código CF-1490C. Esto es atribuido al hecho de que los exponentes correspondientes a cada factor dentro de una complejidad fueron considerados como enteros, también es resultado de la falta de reglas de análisis de complejidad para la evaluación de la tasa de crecimiento de expresiones dentro de desigualdades. Como se muestra en el Listado 4.6, la complejidad aportada por las estructuras iterativas **for** depende de la tasa en que sus expresiones de continuación evalúan como falso, el prototipo consideró el crecimiento de **a** como lineal, sin embargo, éste falló en determinar el primer valor en que **a\*a\*a** evalúa mayor a **x** no es **x**, sino aproximadamente la raíz cubica de esta variable.

```
1 ...
2 int main () {
3     ...
4     while(t>0){
5         ...
6         for(long long int a = 1; (a * a * a) < x; a++){
7             ...
8         }
9         ...
10 }
```

```
11     return 0;  
12 }
```

Listado 4.6: Fragmento de código correspondiente a CF-1490C.

El resto de los códigos enlistados en la Tabla 4.2, fueron considerados como parte de las pruebas complementarias, esto debido a la similitud que presentan sus resultados comparados con los listados en la Tabla 4.1. Sin embargo, se decidió conservarlos en el documento con el fin de fortalecer las conclusiones obtenidas.



# Capítulo 5

## Conclusiones

A continuación, se discuten las conclusiones respecto al proyecto de investigación y son exploradas recomendaciones de trabajo a futuro.

### 5.1. Con respecto al objetivo general

De acuerdo a lo discutido en el Capítulo 4, se considera que el objetivo general de “desarrollar un sistema para la automatización del análisis de la complejidad temporal de programas en C++ basado en las técnicas de análisis de código comúnmente utilizadas en compiladores” se cumplió satisfactoriamente, llegando a las siguientes conclusiones:

- Se desarrolló un prototipo capaz de estimar de forma precisa la complejidad temporal de programas escritos en C++, cuyas estimaciones coinciden exactamente con la complejidad obtenida de forma manual en un 63 % de los códigos probados y obtiene estimaciones equivalentes al 25 % de los casos restantes.
- Los resultados obtenidos demuestran la capacidad del prototipo de estimar la complejidad temporal de códigos escritos en C++ en términos de los identificadores usados dentro de los mismos. A su vez también exponen áreas de oportunidad en cuanto a la extensión de las reglas de análisis de complejidad y la selección de los identificadores

usados para expresar la complejidad del código, pues el prototipo carece del contexto necesario para determinar cuáles identificadores resultan más significativos para el usuario final.

## 5.2. Recomendaciones para trabajo a futuro

Existen algunas áreas de oportunidad dentro de la propuesta descrita en el presente documento, de las cuales el autor del presente documento considera de mayor impacto las enlistadas a continuación.

1. Ampliar el subconjunto del estándar de C++ analizable por el sistema.
2. Mejorar la selección de las variables a través de las que se expresa la complejidad temporal.
3. Ampliar las reglas usadas para el cálculo de complejidad.
4. Desarrollar métodos alternativos para el análisis de la complejidad, posiblemente a través del uso de modelos de inteligencia artificial.

# Bibliografía

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
- [2] A. Asperti, “The intensional content of rice’s theorem,” *SIGPLAN Not.*, vol. 43, p. 113119, jan 2008.
- [3] J. Cohen, “Computer-assisted microanalysis of programs,” *Commun. ACM*, vol. 25, p. 724733, oct 1982.
- [4] R. Vaz, V. Shah, A. Sawhney, and R. Deolekar, “Automated big-o analysis of algorithms,” in *2017 International Conference on Nascent Technologies in Engineering (ICNTE)*, pp. 1–6, 2017.
- [5] I. Czibula, Z. Onet-Marian, and R.-F. Vida, “Automatic algorithmic complexity determination using dynamic program analysis,” in *Proceedings of the 14th International Conference on Software Technologies*, ICSOFT 2019, (Setubal, PRT), p. 186193, SCITEPRESS - Science and Technology Publications, Lda, 2019.
- [6] S. K. Singh and A. Singh, *Software testing*. Vandana Publications, 2012.
- [7] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*, vol. 4. John Wiley & Sons, 2007.

- [8] R. McLeod Jr and G. D. Everett, *Software Testing: Testing Across the Entire Software Development Life Cycle*. John Wiley & Sons, 2007.
- [9] M. Polo, P. Reales, M. Piattini, and C. Ebert, “Test automation,” *IEEE Software*, vol. 30, no. 1, pp. 84–89, 2013.
- [10] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998.
- [11] A. R. Meyer and D. M. Ritchie, “The complexity of loop programs,” in *Proceedings of the 1967 22nd national conference*, pp. 465–469, 1967.
- [12] G. Steele, *Common LISP: the language*. Elsevier, 1990.
- [13] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java language specification*. Addison-Wesley Professional, 2000.
- [14] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, “Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–28, 2017.
- [15] M. Grechanik, C. Fu, and Q. Xie, “Automatically finding performance problems with feedback-directed learning software testing,” in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 156–166, 2012.
- [16] M. Haverbeke, *Eloquent javascript: A modern introduction to programming*. No Starch Press, 2018.
- [17] S. Tsakiltidis, A. Miransky, and E. Mazzawi, “On automatic detection of performance bugs,” in *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)*, pp. 132–139, IEEE, 2016.

- [18] Y. D. Liang, *Introduction to programming using Python*. Pearson, 2013.
- [19] F. Demontiê, J. Cezar, M. Bigonha, F. Campos, and F. Magno Quintão Pereira, “Automatic inference of loop complexity through polynomial interpolation,” in *Programming Languages* (A. Pardo and S. D. Swierstra, eds.), (Cham), pp. 1–15, Springer International Publishing, 2015.
- [20] S. Zaman, B. Adams, and A. E. Hassan, “Security versus performance bugs: A case study on firefox,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR ’11, (New York, NY, USA), p. 93102, Association for Computing Machinery, 2011.
- [21] S. S. Skiena, *The algorithm design manual*, vol. 3. Springer, 1998.
- [22] L. Peter, “An introduction to formal languages and automata,” 2001.
- [23] S. Kandar, *Introduction to automata theory, formal languages and computation*. Pearson Education India, 2013.
- [24] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, vol. 2. Addison-wesley Reading, 2007.
- [25] S. Bergmann, *Compiler design: theory, tools, and examples*. Wm. C. Brown Publishers Dubuque, 1994.
- [26] S. R. Schach, *Object-Oriented and Classical Software Engineering Eighth Edition*. 2007.