17/06/2025

Técnicas de Compilación

INFORME FINAL

INTEGRANTE:
Carrizo Manuel Agustín

PROFESOR:
Ameri Lopez Lozano, Francisco

Informe TC - Carrizo Manuel

Introducción

En este informe se presenta el desarrollo de un compilador para un subconjunto del lenguaje C++, como trabajo final de la materia Técnicas de Compilación. El objetivo fue implementar todas las fases del proceso de compilación, desde el análisis léxico hasta la generación de código intermedio y la optimización, utilizando la herramienta ANTLR4.

El compilador fue desarrollado en Java y permite analizar programas que usan estructuras básicas de C++, como condicionales, bucles, funciones y operaciones aritméticas y lógicas. Todo el trabajo se hizo de manera individual, con control de versiones en GitHub para registrar el avance.

Análisis del problema

El problema planteado fue diseñar e implementar un compilador funcional para un subconjunto del lenguaje C++. Este subconjunto incluye tipos de datos como `int, char, double y void`, así como estructuras de control `(if-else, for, while, break y continue)`, declaraciones de variables y funciones, expresiones aritméticas y lógicas, llamadas a funciones, y sentencias de retorno y asignación.

El compilador debía estar hecho en Java y usar ANTLR4 para las fases de análisis léxico y sintáctico. Además, se requería que hiciera un análisis semántico completo, generara código intermedio en tres direcciones y aplicara al menos tres técnicas de optimización.

Diseño de la Solución

El compilador fue diseñado siguiendo una arquitectura modular, en donde cada fase del proceso de compilación está separada en diferentes clases y componentes. La herramienta ANTLR4 se utilizó para generar automáticamente los analizadores léxico y sintáctico a partir del archivo *MiLenguaje.g4*.

Arquitectura General

La solución se organiza en las siguientes partes:

1. **Gramática ANTLR (*MiLenguaje.g4*)**

   Define las reglas léxicas y sintácticas del subconjunto de C++. Esta gramática permite reconocer estructuras como declaraciones, expresiones, sentencias de control, funciones y llamadas.

2. **Clase Principal (*App.java*)**

   Es el punto de entrada del compilador. Se encarga de leer el archivo de entrada, ejecutar el análisis léxico y sintáctico, y generar el árbol de análisis. También inicia la construcción de la tabla de símbolos y lanza el generador de código.

```
public class App {
    public static void main(String[] args) {
        // Validación de argumentos y obtención de nombres de archivo
```

```
        // 1. Análisis léxico
        // 2. Análisis sintáctico
        // 3. Visualización del árbol sintáctico
        // 4. Análisis semántico
        // 5. Generación de código intermedio
        // 6. Optimización de código
        // Guardado de resultados
    }
    // Métodos auxiliares: guardarCodigoEnArchivo, generarImagenArbolSintactico
}
```

3. **Tabla de Símbolos (*TablaSimbolos.java*)**

   Esta clase almacena información sobre variables y funciones declaradas, incluyendo tipo, nombre y ámbito. Se usa durante el análisis semántico para detectar errores como variables no declaradas o usos incorrectos de tipos.

4. **Listener Semántico (*SimbolosListener.java*)**

   Extiende el listener generado por ANTLR y se utiliza para recorrer el árbol sintáctico y llenar la tabla de símbolos, además de realizar validaciones semánticas como el control de tipos y ámbitos.

5. **Generador de Código (*GeneradorCodigo.java*)**

   Se encarga de crear instrucciones en código intermedio de tres direcciones. Genera temporales, etiquetas y operaciones aritméticas/lógicas, así como estructuras de control.

6. **Visitor de Código (*CodigoVisitor.java*)**

   Recorre el árbol de sintaxis abstracta (AST) y traduce las expresiones del lenguaje fuente en instrucciones de código intermedio, utilizando el generador de código.

Implementación

La implementación del compilador se realizó en Java, usando ANTLR4 para generar los analizadores léxico y sintáctico. A continuación, se describen los componentes principales y su función dentro del sistema.

Gramática (*MiLenguaje.g4*)

Define tanto el análisis léxico como el sintáctico. Contiene las reglas para reconocer tipos de datos, declaraciones, funciones, expresiones, sentencias de control (if, while, for, etc.), operaciones aritméticas y más. Esta gramática permite construir el árbol de sintaxis abstracta (AST).

Clase principal (*App.java*)

Esta clase coordina el funcionamiento general del compilador. Realiza las siguientes tareas:

- Lee el archivo fuente de entrada.
- Ejecuta el análisis léxico y sintáctico usando ANTLR.

📖 Análisis Léxico

El analizador léxico es la primera fase del compilador. Su función principal es leer el código fuente como una secuencia de caracteres y convertirlo en una secuencia de tokens que serán utilizados posteriormente por el analizador sintáctico. Cada token representa una unidad significativa del lenguaje definido en el archivo MiLenguaje.g4 (palabras clave, identificadores, literales, operadores, etc.).

```
// Crear el lexer y capturar errores léxicos
List<String> erroresLexicos = new ArrayList<>();
MiLenguajeLexer lexer = new MiLenguajeLexer(input);
lexer.removeErrorListeners();
lexer.addErrorListener(new BaseErrorListener() {
   @Override
   public void syntaxError(Recognizer<?, ?> recognizer, Object offendingSymbol,
         int line, int charPositionInLine, String msg, RecognitionException e) {
      erroresLexicos.add("❌ ERROR LÉXICO en línea " + line + ":" + charPositionInLine + " - " + m
      throw new ParseCancellationException(msg);
   }
});

// Extraer los tokens
CommonTokenStream tokens = new CommonTokenStream(lexer);
tokens.fill();

// Mostrar tokens reconocidos
for (Token token : tokens.getTokens()) {
   if (token.getType() != Token.EOF) {
      String tokenName = MiLenguajeLexer.VOCABULARY.getSymbolicName(token.getType());
      System.out.printf("%-20s %-30s %-10d %-10d\n",
            tokenName, token.getText(), token.getLine(), token.getCharPositionInLine());
   }
}
```

Resultado esperado:

```
TIPO              LEXEMA                  LÍNEA     COLUMNA
-------------------------------------------------------------------------
INT               int                     1         0
ID                contador                1         4
IGUAL             =                       1         13
INTEGER           10                      1         15
```

📖 Análisis Sintáctico

El análisis sintáctico es la segunda fase del compilador. Su función es verificar que la secuencia de tokens generada por el analizador léxico siga la estructura gramatical del lenguaje. Es decir, comprueba que el código fuente esté correctamente escrito según las reglas de sintaxis definidas en el archivo MiLenguaje.g4.

```
// Crear el parser y capturar errores sintácticos
MiLenguajeParser parser = new MiLenguajeParser(tokens);
List<String> erroresSintacticos = new ArrayList<>();
parser.removeErrorListeners();
parser.addErrorListener(new BaseErrorListener() {
    @Override
    public void syntaxError(Recognizer<?, ?> recognizer, Object offendingSymbol,
            int line, int charPositionInLine, String msg, RecognitionException e) {
        erroresSintacticos.add("❌ ERROR SINTÁCTICO en línea " + line + ":" + charPositionInLine +
    }
});

// Analizar el programa según la regla inicial 'programa'
ParseTree tree = parser.programa();

// Verificar si hubo errores sintácticos
if (!erroresSintacticos.isEmpty()) {
    erroresSintacticos.forEach(System.out::println);
} else {
    System.out.println("✅ Análisis sintáctico completado sin errores.");
    System.out.println("Representación textual del árbol sintáctico:");
    System.out.println(tree.toStringTree(parser)); // Muestra el árbol como texto
}
```

Resultado esperado:

```
✅ Análisis sintáctico completado sin errores.
Representación textual del árbol sintáctico:
(programa (sentencia int x = 5 ;) ...)
```

Muestra el árbol:

```
generarImagenArbolSintactico(tree, parser);
private static void generarImagenArbolSintactico(ParseTree tree, Parser parser) {
      try {
         JFrame frame = new JFrame("Árbol Sintáctico");
         JPanel panel = new JPanel();

         TreeViewer viewer = new TreeViewer(Arrays.asList(parser.getRuleNames()), tree);
         viewer.setScale(1.5);

         panel.add(viewer);

         JScrollPane scrollPane = new JScrollPane(panel);
         scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDE
         scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
```

```
            frame.add(scrollPane);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(800, 600);
            viewer.open();

        } catch (Exception e) {
            System.err.println("\u001B[31m❌ Error al mostrar árbol sintáctico: " + e.getMessage() + "\
        }
    }
```

- Llama al *SimbolosListener* para hacer el análisis semántico.

📖 Análisis Semántico con SimbolosListener

El componente SimbolosListener implementa la fase de análisis semántico del compilador. Esta fase se encarga de verificar que el programa tenga sentido más allá de la sintaxis: valida el uso correcto de variables, funciones, tipos de datos y ámbitos.

Trabaja sobre el árbol sintáctico (AST) generado en la fase anterior, recorriéndolo nodo a nodo para construir y mantener una tabla de símbolos con toda la información relevante del programa (identificadores, tipos, categorías, ámbito, etc.).

- Construye la tabla de símbolos mientras recorre el AST.
- Garantiza que todas las **declaraciones y usos de identificadores sean válidos**.
- Detecta errores **semánticos críticos** (como variables no declaradas o tipos incompatibles) y genera **advertencias** para problemas menores.

  Es una pieza clave para asegurar que el código fuente no solo esté bien escrito (sintaxis), sino que también **tenga sentido lógico**.

```
// === ANÁLISIS SEMÁNTICO ===
    SimbolosListener listener = new SimbolosListener();
    ParseTreeWalker walker = new ParseTreeWalker();
    walker.walk(listener, tree);

    TablaSimbolos tabla = listener.getTablaSimbolos();
    tabla.imprimir();

    List<String> erroresSemanticos = listener.getErrores();
    if (!erroresSemanticos.isEmpty()) {
        System.out.println("\n" + RED + "=== ERRORES SEMÁNTICOS ===" + RESET);
        erroresSemanticos.forEach(e -> System.out.println(RED + "❌ " + e + RESET));
    } else {
        System.out.println(GREEN + "\n✅ Análisis semántico completado sin errores." + RESET);
    }

    List<String> advertencias = listener.getAdvertencias();
    if (!advertencias.isEmpty()) {
        System.out.println("\n" + YELLOW + "=== ADVERTENCIAS ===" + RESET);
```

```
    advertencias.forEach(w -> System.out.println(YELLOW + "⚠️ " + w + RESET));
}
```

- Invoca al *CodigoVisitor* para generar el código intermedio.

📖 Generación de Código Intermedio con CodigoVisitor

El componente CodigoVisitor es el encargado de recorrer el árbol sintáctico (AST) generado durante el análisis y traducir las instrucciones del programa a un código intermedio en tres direcciones. Este código intermedio es una representación más cercana a un lenguaje ensamblador, pensada para ser fácilmente optimizable y servir de base para la generación de código final.

```
// === GENERACIÓN DE CÓDIGO INTERMEDIO ===
System.out.println("\n=== 5. GENERACIÓN DE CÓDIGO INTERMEDIO ===\n");
System.out.println("  🎯 Iniciando recorrido del AST con CodigoVisitor...");

CodigoVisitor visitor = new CodigoVisitor(tabla);
visitor.visit(tree);

GeneradorCodigo generador = visitor.getGenerador();

System.out.println("  📝 Código de tres direcciones generado:");
generador.imprimirCodigo();

if (generador.getTiposVariables() != null) {
    generador.imprimirTipos();
}
generador.imprimirEstadisticas();

String codigoIntermedioPath = baseName + "_codigo_intermedio.txt";
guardarCodigoEnArchivo(generador.getCodigo(), codigoIntermedioPath);
System.out.println(GREEN + "✅ Código intermedio guardado en: " + codigoIntermedioPath + R
```

- Generar el código intermedio.

📖 Optimización de Código Intermedio

Una vez generado el **código intermedio**, el compilador aplica una fase de **optimización** para simplificar y mejorar la eficiencia del código antes de pasarlo a la etapa final de compilación.

El objetivo es producir un código optimizado que:

- Sea más eficiente en tiempo de ejecución.
- Elimine instrucciones innecesarias o redundantes.
- Reduzca el uso de recursos como registros o memoria.

En nuestro compilador, esto se realiza mediante la clase Optimizador, que recibe el código intermedio y aplica una serie de transformaciones locales y globales.

```
// === 6. OPTIMIZACIÓN DE CÓDIGO ===
  Optimizador optimizador = new Optimizador(generador.getCodigo());
  List<String> codigoOptimizado = optimizador.optimizar();

  String codigoOptimizadoPath = baseName + "_codigo_optimizado.txt";
  guardarCodigoEnArchivo(codigoOptimizado, codigoOptimizadoPath);
  System.out.println(GREEN + "✅ Código optimizado guardado en: " + codigoOptimizadoPath + R

  // (Opcional) Imprimir en consola el código optimizado
  optimizador.imprimirCodigoOptimizado();
```

Análisis Semántico (*SimbolosListener.java + TablaSimbolos.java*)

- El *SimbolosListener* recorre el AST y verifica que el programa sea semánticamente correcto. Controla:
- Declaración de variables y funciones.
- Ámbitos válidos.
- Compatibilidad de tipos.
- Errores semánticos con mensajes claros.
- La *TablaSimbolos* almacena la información necesaria para estas validaciones, como tipo, nombre y ámbito de cada símbolo.

Generación de Código (*CodigoVisitor.java + GeneradorCodigo.java*)

El *CodigoVisitor* recorre el árbol de manera controlada y llama a *GeneradorCodigo* para producir el código intermedio. Este código se construye usando:

- Temporales para almacenar resultados parciales.
- Etiquetas para saltos condicionales.
- Instrucciones de tres direcciones para representar operaciones.

También se generan instrucciones para estructuras de control, llamadas a funciones y asignaciones.

Optimización de Código (*Optimizador.java*)

El *Optimizador* toma como entrada el código intermedio generado y aplica una serie de **transformaciones** para simplificarlo y mejorarlo. Este proceso produce un **código optimizado**, más eficiente y fácil de traducir a ensamblador.

El optimizador trabaja con:

- **Eliminación de código muerto**: quita instrucciones cuyas variables no son utilizadas posteriormente.
- **Propagación de constantes**: sustituye variables por valores constantes cuando es posible.
- **Plegado de constantes**: evalúa operaciones con literales en tiempo de compilación.
- **Simplificación de saltos**: reorganiza saltos y elimina etiquetas innecesarias.

Al final, se genera un **nuevo archivo** con el código optimizado `<nombre_archivo>_codigo_optimizado.txt`

Ejemplo y prueba

Ejemplo:

```
void main() {
   int a[3];
   a[0] = 10;
   a[1] = 20;
   int uno;
   uno = a[1];
}
```

Salida esperada:

```
=== ANÁLISIS LÉXICO ===
TIPO            LEXEMA                  LÍNEA     COLUMNA
-----------------------------------------------------------------
VOID             void                    1         0
ID              main                    1         5
PA               (                      1         9
PC               )                      1         10
LA               {                      1         12
INT              int                    2         4
ID              a                       2         8
CA               [                      2         9
INTEGER             3                       2         10
CC               ]                      2         11
PYC              ;                      2         12
ID              a                       3         4
CA               [                      3         5
INTEGER             0                       3         6
CC               ]                      3         7
IGUAL              =                       3         9
INTEGER             10                      3         11
PYC              ;                      3         13
ID              a                       4         4
CA               [                      4         5
INTEGER             1                       4         6
CC               ]                      4         7
```

```
IGUAL              =                    4        9
INTEGER            20                     4        11
PYC                ;                    4        13
INT                int                  5        4
ID                 uno                  5        8
PYC                ;                    5        11
ID                 uno                  6        4
IGUAL              =                    6        8
ID                 a                    6        10
CA                 [                    6        11
INTEGER            1                      6        12
CC                 ]                    6        13
PYC                ;                    6        14
LC                 }                    7        0

? Análisis léxico completado sin errores.

=== ANÁLISIS SINTÁCTICO ===
? Análisis sintáctico completado sin errores.
//Se mostraría el arbol en una pestaña nueva

=== TABLA DE SÍMBOLOS ===
NOMBRE        TIPO      CATEGORÍA      LÍNEA     COLUMNA    ÁMBITO        PARÁMETROS
-------------------------------------------------------------------------------------------
main          void      funcion        1         5          global
a             int[]     variable       2         8          main
uno           int       variable       5         8          main

? Análisis semántico completado sin errores.

=== 5. GENERACIÓN DE CÓDIGO INTERMEDIO ===

  ? Iniciando recorrido del AST con CodigoVisitor...
? GENERADOR: Iniciado y listo para trabajar
? VISITOR: Iniciado con tabla de símbolos
? VISITOR: Iniciando recorrido del programa
? VISITOR: Procesando una sentencia...
? VISITOR: Encontré función → main
? GENERADOR: Colocando etiqueta func_main
? GENERADOR: Generé → func_main:
? VISITOR: Cambiando al ámbito de main
? VISITOR: Procesando cuerpo de la función...
? VISITOR: Procesando bloque con 5 sentencias
? VISITOR: Declaración de variable int a
? VISITOR: Encontré número → 0
? VISITOR: Encontré número → 10
? VISITOR: Asignación a arreglo → a[0] = 10
? GENERADOR: Generé arreglo → a[0] = 10
```

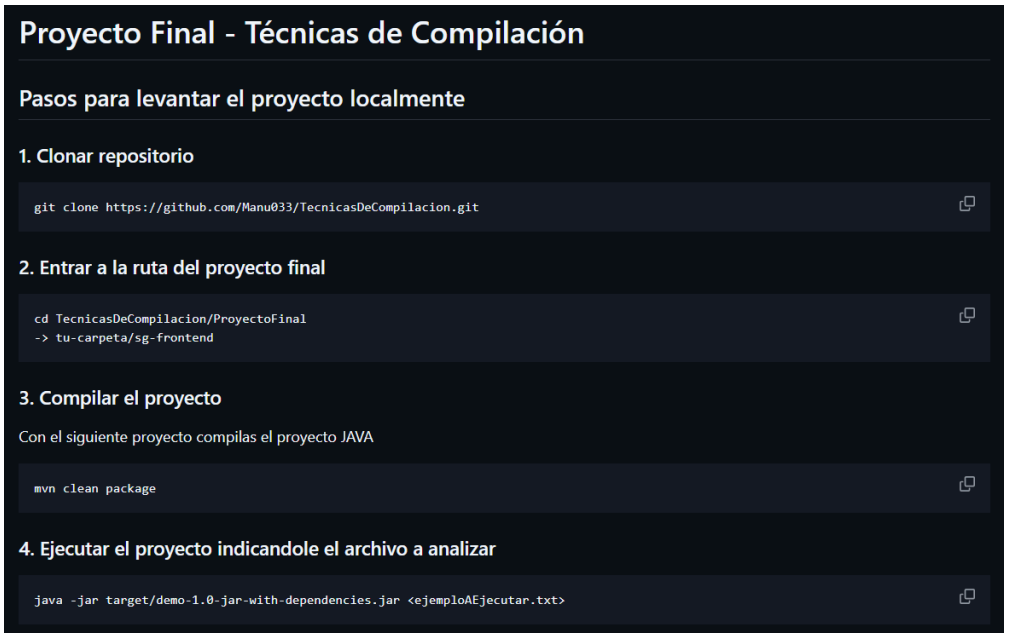
```
? VISITOR: Encontré número → 1
? VISITOR: Encontré número → 20
? VISITOR: Asignación a arreglo → a[1] = 20
? GENERADOR: Generé arreglo → a[1] = 20
? VISITOR: Declaración de variable int uno
? VISITOR: Encontré número → 1
? VISITOR: Expresión arreglo → a[1]
? GENERADOR: Generé carga → t0 = a[1]
? VISITOR: Asignación simple → uno = t0
? GENERADOR: Generando asignación uno = t0
? GENERADOR: Generé → uno = t0
? VISITOR: Función main completada
? VISITOR: Programa completado
  ? Código de tres direcciones generado:

? === CÓDIGO DE TRES DIRECCIONES ===
  0: func_main:
  1: a[0] = 10
  2: a[1] = 20
  3: t0 = a[1]
  4: uno = t0
Total instrucciones: 5

? ESTADÍSTICAS:
  - Temporales creados: 1
  - Etiquetas creadas: 0
  - Instrucciones totales: 5
  ? Archivo guardado con 5 instrucciones
? Código intermedio guardado en: ejemploInforme_codigo_intermedio.txt
  ? Archivo guardado con 5 instrucciones
? Código optimizado guardado en: ejemploInforme_codigo_optimizado.txt

=== CÓDIGO OPTIMIZADO ===
0: func_main:
1: a[0] = 10
2: a[1] = 20
3: t0 = 20
4: uno = 20

=== 7. RESUMEN DE COMPILACIÓN ===
  ? Archivo procesado: ejemploInforme.txt
  ? Tokens analizados: 36
  ? Símbolos en tabla: 3
  ? Instrucciones generadas: 5
  ? Instrucciones optimizadas: 5
  ? Archivo código intermedio: ejemploInforme_codigo_intermedio.txt
  ? Archivo código optimizado: ejemploInforme_codigo_optimizado.txt
```

Manual de usuario

Requisitos Previos

Antes de comenzar, asegúrese de contar con lo siguiente:

- **Java 8 JDK** instalado y configurado en la variable JAVA_HOME.
- **Maven** (versión 3.x) instalado para compilar el proyecto.
- **Git** para clonar el repositorio.
- Sistema operativo: Windows, macOS o Linux.
- Memoria mínima recomendada: 4 GB de RAM.



Link del repositorio: https://github.com/Manu033/TecnicasDeCompilacion.git

Dificultades encontradas y Soluciones aplicadas

Durante la generación del código intermedio, las asignaciones y accesos a elementos de arreglo (`a[índice]`) no se estaban reconociendo correctamente, por lo que todas las referencias a arrays quedaban como `null` o se trataban como variables simples.

- **Problema**: el `CodigoVisitor` solo manejaba literales y variables, ignorando la alternativa de expresión de arreglo en la gramática, y el generador no tenía instrucciones específicas para cargar o almacenar en un array.
- **Solución**:
  1. **Extensión de la gramática**: confirmamos que la regla `expresion` incluye la etiqueta `#expAccesoArreglo` para `ID '[' expresion ']'`.

2. **Visitor especializado**: implementamos `visitExpAccesoArreglo`, que genera un temporal con `genLoadArray(nombre, índice)` y devuelve ese temporal como valor de la expresión.
3. **Asignación a array**: en `visitAsignacion` detectamos si el contexto contiene corchetes y llamamos a `genAsignacionArray(nombre, índice, valor)` para emitir la instrucción correcta `array[indice] = valor`.

Con estos cambios, las operaciones sobre arrays (`a[0]=10`, `x = a[1]`) producen temporales y accesos explícitos en el código de tres direcciones, solucionando el `null` y permitiendo optimización posterior.

Conclusión

El desarrollo de este compilador fue una experiencia muy útil para poner en práctica los conceptos aprendidos en la materia. Se logró implementar todas las fases principales del proceso de compilación y trabajar con una herramienta profesional como ANTLR4. Además el uso de GitHub ayudó a tener una mejor organización y seguir buenas prácticas de desarrollo. Si bien hubo desafíos, pude resolverlos y obtener un compilador funcional para un subconjunto del lenguaje C++.

Referencias Bibliográficas

- Oracle. Java SE 8 Documentation.
- ANTLR64 Documentation
- Material de estudio Técnicas de Compilación - Francisco Ameri