Desarrollo de un Compilador para un Subconjunto de C++

Integrantes: Nicolás Moresco, Joaquín Vélez

Materia: Técnicas de Compilación

Profesor: Francisco Ameri

Fecha:17 de Julio de 2025

1. Introducción

En este trabajo se presenta el diseño e implementación de un compilador completo para un subconjunto del lenguaje C++. El objetivo principal es recorrer todas las fases clásicas de compilación —análisis léxico, sintáctico, semántico, generación de código intermedio y optimización— aplicando ANTLR4 para la generación automática de analizadores y una arquitectura de visitor/generador para la producción de código de tres direcciones.

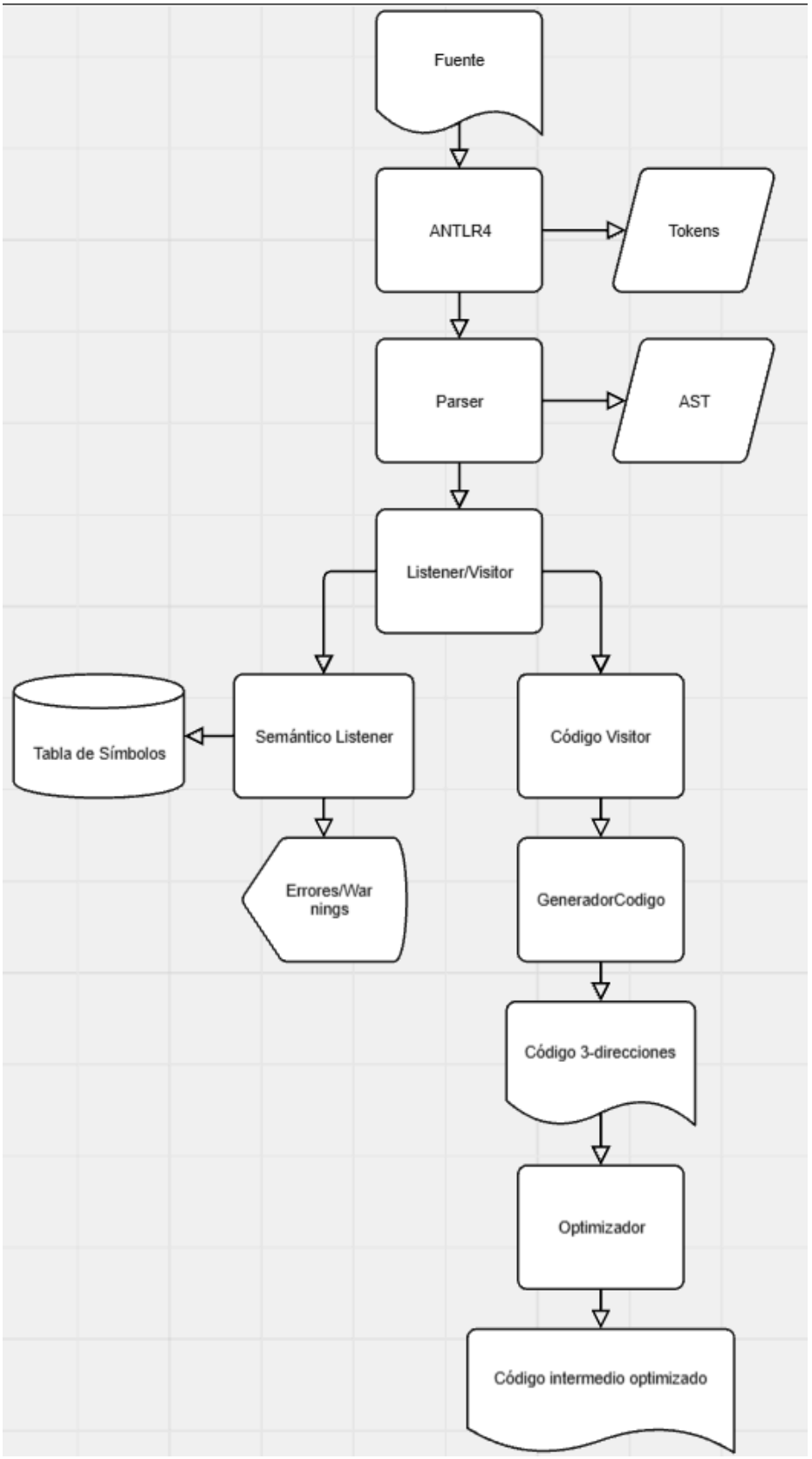
2. Análisis del Problema

Subconjunto de C++ implementado:

- **Tipos básicos:** `int`, `char`, `double`, `void`.
- **Estructuras de control:** `if-else`, `for`, `while`, `break`, `continue`.
- **Funciones:** Declaración, parámetros, llamadas y valores de retorno.
- **Operadores:** Aritméticos (`+`, `-`, `*`, `/`, `%`), lógicos (`&&`, `||`, `!`), de comparación (`<`, `<=`, `>`, `>=`, `==`, `!=`).
- **Expresiones:** Literales, variables, llamadas, paréntesis.

3. Diseño de la Solución

3.1 Arquitectura General



3.2 Fases de Compilación

1. **Análisis Léxico:** Definición de tokens en `MiniLenguajeLexer.g4`.
2. **Análisis Sintáctico:** Gramática en `MiniLenguajeParser.g4` y construcción de AST.
3. **Análisis Semántico:**
    - Tabla de símbolos con ámbitos anidados.
    - Verificación de tipos, declaración antes de uso, parámetros, retornos.
    - Detección de warnings (variables sin inicializar, bucles infinitos).
4. **Generación de Código Intermedio:** Visitor que recorre el AST y emite instrucciones de tres direcciones.
5. **Optimización de Código:**
    - Eliminación de código muerto.
    - Propagación de constantes.
    - Simplificación de expresiones.
    - Eliminación de asignaciones redundantes.

4. Implementación

4.1 Detalles Técnicos

- **Lenguaje:** Java 1.8
- **Gestión de build:** Maven, con plugin ANTLR4.
- **Estructura de carpetas:**

```
src/main/antlr4/com/compilador/
  ├─ MiniLenguajeLexer.g4
  └─ MiniLenguajeParser.g4
src/main/java/com/compilador/
  ├─ App.java
  ├─ CodigoVisitor.java
  ├─ GeneradorCodigo.java
  ├─ Optimizador.java
  └─ semantico/
      ├─ SimbolosListener.java
      └─ TablaSimbolos.java
```

4.2 Gramática ANTLR4

- Definición de reglas para declaraciones, sentencias, expresiones.
- Reconocimiento de literales, identificadores, comentarios y espacios en blanco.

4.3 Tabla de Símbolos

- Estructura = `Map<String, List<Simbolo>>` por ámbito.
- `Simbolo` almacena nombre, tipo, categoría (variable, función, parámetro), estado de inicialización y uso.
- Métodos: `entrarAmbito`, `salirAmbito`, `agregar`, `buscar`, `buscarEnAmbitoExacto`.

4.4 Algoritmos de cada fase

- **Semántica:** Recorrido `enter`/`exit` de reglas, detección de errores al momento.
- **Generación:** Visitor que para cada nodo produce:
  - Temporales con `gen.newTemp()`
  - Etiquetas con `gen.newLabel()`
  - Instrucciones en lista interna.
- **Optimización:** Recorrido iterativo de la lista de instrucciones con transformaciones de patrones.

4.5 Técnicas de Optimización

1. **Eliminación de código muerto:** Seguimiento de etiquetas y saltos para descartar instrucciones inalcanzables.
2. **Propagación de constantes:** Rastreo de asignaciones a literales y sustitución en usos posteriores.
3. **Simplificación de expresiones:** Cálculo en tiempo de compilación de operaciones constantes (`2+3`→`5`, `4>1`→`1`).
4. **Eliminación de sentencias redundantes:** Quitar `x = x` y saltos innecesarios.

5. Ejemplos y Pruebas

5.1 Caso 1: Función máxima

Código:

```
int calculaMax(int a, int b) {
    int resultado;
    if (a > b) resultado = a;
    else resultado = b;
    return resultado;
}
```

C3D inicial:

```
0: func_calculaMax:
1: t0 = a > b
2: if !t0 goto L1
3: resultado = a
4: goto L2
5: L1:
6: resultado = b
7: L2:
8: return resultado
```

C3D optimizado (constantes, muerto): igual, sin cambios.

5.2 Caso 2: Propagación y simplificación

Código:

```
int f() {
   int x = 2;
   int y = x;
   int z = y + 3;
   return z;
}
```

C3D inicial:

```
t0 = 2
```

x = t0
y = x
t1 = y + 3
z = t1
return z

C3D optimizado:

x = 2
y = 2
z = 5
return 5

6. Conclusiones

Se alcanzaron los objetivos: un compilador funcional que recorre todas las fases, genera código intermedio legible, detecta errores semánticos y aplica optimizaciones. La separación Visitor/Generador y el uso de ANTLR4 permitieron un desarrollo modular y mantenible.