Proyecto Final - TC

Asignatura:
Técnicas de Compilación

Profesor:
Francisco, Ameri Lopez Lozano

Escrito por:
Mora Colodrero, Jonathan

Github:
https://github.com/JoniMora/TC.git

Fecha: 26/06/2025

UBP UNIVERSIDAD Blas Pascal

INTRODUCCIÓN

El presente trabajo final documenta la extensión y mejora de un compilador para un subconjunto del lenguaje C, desarrollado utilizando la herramienta ANTLR4. Partiendo de una base funcional establecida en trabajos previos, el objetivo principal fue robustecer las fases de análisis léxico, sintáctico y semántico, e implementar una etapa de generación de código intermedio de tres direcciones (TAC) que incluye optimizaciones básicas, culminando en una salida de consola clara y enriquecida visualmente.

OBJETIVO

El objetivo central de este proyecto fue construir un compilador capaz de:

1. Analizar Léxicamente un archivo de código fuente en C, tokenizando el input y reportando errores léxicos.
2. Analizar Sintácticamente, verificando la estructura gramatical del programa y construyendo un Árbol de Sintaxis Abstracta (AST), con reporte de errores sintácticos.
3. Realizar un Análisis Semántico Profundo, construyendo una tabla de símbolos detallada, gestionando los ámbitos de visibilidad y detectando errores semánticos (ej., variables no declaradas, incompatibilidad de tipos, funciones con uso incorrecto) y advertencias (ej., código muerto, variables/funciones no utilizadas).
4. Generar Código Intermedio (TAC) de tres direcciones a partir del AST, representando el programa de manera más abstracta y fácil de optimizar.
5. Optimizar el Código Intermedio mediante técnicas como la propagación y plegado de constantes, la simplificación de expresiones y la eliminación de código muerto.
6. Proporcionar una Salida de Consola Clara y Personalizada, utilizando códigos de color ANSI para mejorar la legibilidad y la identificación de las diferentes fases del proceso de compilación, así como los resultados de la generación y optimización del código.

DESARROLLO DEL PROYECTO

DEFINICIÓN DE LA GRAMÁTICA (.G4)

La gramática, definida en `Compilador.g4` con ANTLR4, soporta operadores aritméticos, comparativos y lógicos, junto con estructuras de control (`if-else, while, for`), declaraciones de variables y funciones. Un punto clave fue la precisión en la definición del bucle `for` para asegurar la correcta generación de su código intermedio, incluyendo su sección de actualización y el manejo de `break/continue`.

ANÁLISIS LÉXICO

Gestionado por `CompiladorLexer`, esta fase transforma el código fuente en tokens, reportando errores léxicos. Una mejora clave es la personalización de la salida de la tabla de tokens, donde cada tipo de token se visualiza con un color específico en la consola, facilitando la inspección y depuración.

ANÁLISIS SINTÁCTICO

El `CompiladorParser` valida la estructura del código según la gramática, construyendo el AST y reportando errores sintácticos detalladamente. Su éxito se indica con un mensaje claro en verde.

LISTENER: ANÁLISIS SEMÁNTICO (TABLA DE SÍMBOLOS Y MANEJO DE ERRORES)

El `SimbolosListener` construye la `TablaSimbolos`, gestionando ámbitos y detectando errores semánticos (ej., variables no declaradas, incompatibilidad de tipos, uso incorrecto de funciones). También emite advertencias por código subóptimo (ej., variables/funciones no usadas). La salida de la `TablaSimbolos` ha sido mejorada visualmente, presentando la información con colores específicos para tipos, categorías y estados de inicialización/uso.

```
ERROR SINTÁCTICO en línea 6:17 - extraneous input ''' expecting {'(', '!', 'true', 'false', ID, INTEGER, DECIMAL, CHARACTER, STRING_LITERAL}
ERROR SINTÁCTICO en línea 6:20 - extraneous input ''' expecting ';'
ERROR SINTÁCTICO en línea 9:25 - extraneous input '€' expecting ';'
ERROR SINTÁCTICO en línea 12:8 - no viable alternative at input 'int2'
```

La clase `Caminante`, un visitor adicional, realiza verificaciones más profundas, como la detección de bucles `while(true)` o condiciones constantes en `if/for`. Los errores (❌ `rojo`) y advertencias (⚠️ `amarillo`) detectados se integran en el reporte final.

```
Advertencia en línea 26: Condición verdadera constante en bucle `while`. Posible bucle infinito detectado.
Advertencia en línea 31: Condición falsa constante en bucle `while`. Este bucle nunca se ejecutará.
Advertencia en línea 32: Variable 'bucleNuncaEjecutadoVar' declarada pero no usada.
Advertencia en línea 23: Variable 'dentroElse' declarada pero no usada.
```

Tabla de Símbolos coloreada: A continuación, una captura de la tabla de símbolos generada, destacando los colores que indica el tipo, categoría y estado de uso/inicialización de cada símbolo.

```
=== TABLA DE SIMBOLOS ===
NOMBRE          TIPO        CAT.            LÍNEA    COLUMNA    ÁMBITO          INICIALIZADA USADA        PARÁMETROS
-------------------------------------------------------------------------------------------------------------------
a               int         variable        6        8          bloque_1        true         true
b               int         variable        7        8          bloque_1        true         true
c               int         variable        8        8          bloque_1        true         false
d               int         variable        9        8          bloque_1        true         false
x               int         parametro       1        17         cuadrado        true         true
cuadrado        int         funcion         1        4          global          false        false        (int)
main            void        funcion         5        5          global          false        false
```

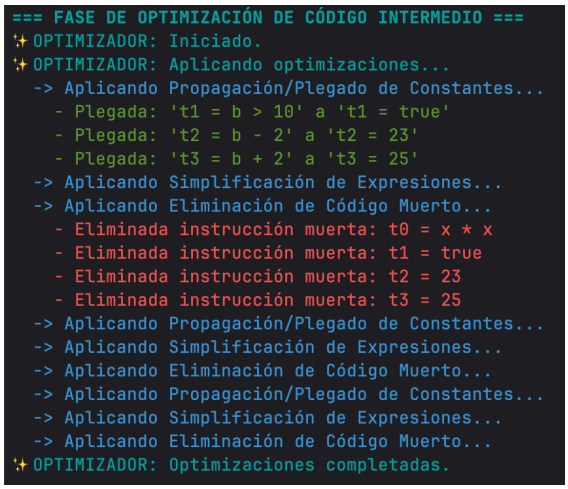
VISITOR: GENERACIÓN DEL CÓDIGO DE TRES DIRECCIONES (TAC)

El `CodigoVisitor` utiliza `GeneradorCodigo` para transformar el AST en instrucciones TAC. Esto incluye el manejo de expresiones, control de flujo con etiquetas y saltos, y una implementación robusta del bucle `for`. El código TAC generado se imprime con un esquema de colores para funciones, asignaciones, control de flujo, etiquetas y llamadas, permitiendo una visualización clara de la estructura intermedia del programa.

Ejemplo de Código de Tres Direcciones (Original): Aquí se muestra una porción del código intermedio generado antes de cualquier optimización.

```
=== CÓDIGO DE TRES DIRECCIONES (Original) ===
 0: func_cuadrado:
 1: t0 = x * x
 2: return t0
 3: end_func_cuadrado
 4: func_main:
 5: c = 2
 6: d = 3
 7: a = 5
 8: b = 25
 9: t1 = b > 10
10: if !t1 goto L0
11: t2 = b - 2
12: b = t2
13: goto L1
14: L0:
15: t3 = b + 2
16: b = t3
17: L1:
18: end_func_main
Total instrucciones: 19
```

FASE DE OPTIMIZACIÓN DEL CÓDIGO INTERMEDIO

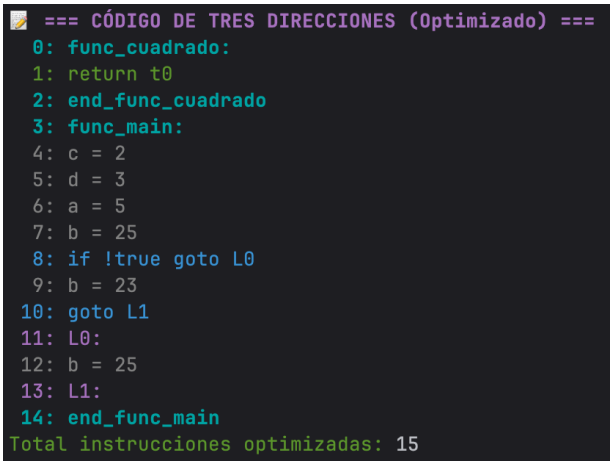


La clase `OptimizadorCodigo` aplica múltiples pasadas de optimización:

- ***Propagación y Plegado de Constantes***: Reemplaza variables con valores constantes y evalúa expresiones con literales conocidos.
- ***Simplificación de Expresiones***: Reduce expresiones a su forma más simple (ej., `x + 0` a `x`).
- ***Eliminación de Código Muerto***: Elimina asignaciones a variables cuyo valor no se utiliza posteriormente.

Esta fase destaca por su salida detallada y coloreada, mostrando en verde las optimizaciones aplicadas y en rojo las instrucciones eliminadas. El código optimizado se presenta con el mismo esquema de colores que el original, facilitando la comparación y comprensión de las mejoras.

Ejemplo de Código de Tres Direcciones (Optimizado): Una captura que muestre el mismo fragmento de código intermedio después de aplicar las optimizaciones, resaltando las diferencias y mejoras.



CONCLUSIÓN

Este proyecto ha permitido la implementación de un compilador integral para C, destacando por su gramática robusta en ANTLR4 y sus fases de análisis semántico y generación/optimización de código intermedio. La correcta implementación del bucle `for` en TAC fue un logro técnico significativo.

La personalización de la salida en consola con colores y emojis ha sido una mejora clave, transformando la experiencia de depuración y haciendo el proceso de compilación más intuitivo y visualmente atractivo. Este trabajo no solo ha cumplido sus objetivos técnicos, sino que también ha brindado una comprensión práctica profunda del diseño y funcionamiento de los compiladores.