UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERIAS

SEMINARIO DE SOLUCIÓN DE PROBLEMAS DE TRADUCTORES DE LENGUAJES II

Sección D02

Proyecto Final

Compilador en C++

Alumno: Oscar Etienne Orozco Guzman

Profesor: Michel Emanuel López Franco

Fecha: 01/Dic/2025

Introducción

Este proyecto consta del diseño e implementación de un compilador en lenguaje de alto nivel C++ que abarca las fases de análisis léxico, análisis sintáctico, análisis semántico y generación de código intermedio. El método de desarrollo fue de manera modular con la creación de archivos principales y headers.

El objetivo es ofrecer una visión clara y técnica, explicando el funcionamiento de cada fase, la interacción entre ellas y las pruebas realizadas.

En este proyecto se optó por mantener separados los archivos destino y headers para la mantenibilidad, escalabilidad y claridad en el código.

Fases del compilador

1. **Análisis Léxico:** Convierte la secuencia de caracteres del código fuente en una secuencia de tokens.

2. **Análisis Sintáctico:** Organiza los tokens en una estructura jerárquica (AST) según las reglas gramaticales del lenguaje.

3. **Análisis Semántico:** Verifica la coherencia lógica y de tipos de datos del programa, utilizando la información del AST y la tabla de símbolos.

4. **Generación de código intermedio:** Traduce el AST en una representación intermedia (IR), como instrucciones de tres direcciones, facilitando la generación de código máquina siguiente o la interpretación.

Analizador Léxico

Es la primera fase del compilador y su objetivo principal es transformar la secuencia de caracteres del código fuente en una secuencia de tokens, que son las unidades léxicas significativas para el lenguaje de cualquier programa.

Los tokens pueden representar identificadores, palabras clave, operadores, números, símbolos, entre otros.

Este analizador elimina los espacios en blanco y comentarios, asocia a cada token información relevante, la posición del archivo fuente y el tipo de token. Esta información es fundamental para la generación de mensajes de error precisos y para un análisis posterior.

Los componentes léxicos reconocidos en este proyecto son:

- **Identificadores:** Nombres de variables o funciones definidos por un patrón de letras, dígitos y guion bajo, comenzando con una letra.
- **Números Reales**
- **Operadores aritméticos (+, -, *, /).**
- **Operadores relacionales (==, !=, <, >, <=, >=).**
- **Operadores lógicos (&&, ||, !).**
- **Símbolos de puntuación.**
- **Palabras reservadas (int, float, if, while, return, else).**

El analizador léxico clasificará el texto ingresado en los siguientes tipos según sea su estructura:

- Palabra clave.
- Identificador.
- Operador.
- Entero.
- Puntuación.

Cada token incluye información adicional como el lexema y la posición en el archivo fuente.

Análisis Sintáctico

Es la segunda fase del compilador y se encarga de organizar la secuencia de tokens en una estructura jerárquica que refleja la gramática del lenguaje.

En este proyecto el “parser” está implementado con un enfoque recursivo descendente, donde cada regla gramatical se traduce en una función recursiva.

Cada función del parser corresponde a un no terminal de la gramática. El parser obtiene los tokens y construye un árbol sintáctico abstracto (AST) al aplicar las reglas de producción.

Durante el análisis sintáctico el parser construye un AST, donde cada nodo representa una construcción sintáctica relevante del programa. Este AST omite detalles sintácticos innecesarios (paréntesis, punto y coma) y solo se centra en la estructura lógica del código.

Esta parte es la responsable de reportar errores sintácticos, indicando la posición y naturaleza del error.

Análisis Semántico

Esta parte del proyecto se encarga de validar que el programa fuente tiene sentido desde un punto de vista lógico y de tipos.

Utiliza el AST y la tabla de símbolos para realizar comprobaciones como:

- **Declaración de variables.**
- **Compatibilidad de tipos.**
- **Uso correcto de variables.**
- **Validación de funciones.**
- **Conversión de tipos.**

Durante el análisis semántico se recorren los nodos del AST y se realizan las comprobaciones necesarias. Si se detecta un error el analizador sintáctico lo reporta con información precisa sobre la ubicación y el problema.

Generación de Código intermedio:

Esta parte del proyecto es la que traduce el AST ya validado semánticamente en una representación intermedia (IR) que facilita la optimización y la posterior traducción a código máquina.

Se utiliza el código de tres direcciones, que es una notación donde cada instrucción tiene como máximo dos operandos y un resultado y puede involucrar variables temporales y etiquetas para el control del flujo.

Estas son algunas instrucciones:

- **Asignación binaria: t1 = a + b**
- **Asignación unaria: t2 = - a**
- **Copia: x = y**
- **Saltos condicionales: if x relop y goto L1**
- **Saltos incondicionales: goto L2**
- **Etiquetas: label L1**
- **Llamadas y retornos: call f, return 0**

Las variables temporales **t1** y **t2** son generadas automáticamente para almacenar resultados intermedios.

El archivo **ir.cpp** implementa las funciones para recorrer el AST y generar la secuencia de instrucciones de tres direcciones. El archivo ir.h declara las estructuras y funciones utilizadas.

El generador de código intermedio utiliza un contador para crear nombres únicos de temporales y etiquetas y mantiene una lista de las instrucciones generadas.

Conclusión

El desarrollo de un compilador modular en C++ que abarca las fases de análisis léxico, sintáctico, semántico y generación de código intermedio es un ejercicio que combina teoría y práctica. La organización en carpetas Sources y Headers, junto con la separación clara de responsabilidades entre archivos facilita la mantenibilidad y extensibilidad del proyecto.

Cada fase cumple un rol esencial en la transformación del código fuente en una representación intermedia eficiente y correcta. El análisis léxico identifica tokens y elimina detalles irrelevantes, el análisis sintáctico construye el AST que refleja la estructura lógica del programa, el análisis semántico valida la coherencia y los tipos, y la generación de código intermedio produce instrucciones de tres direcciones listas para la optimización o traducción final.

Las pruebas exhaustivas y la corrección de errores, como la clasificación de ‘;’ en el lexer, demuestran la importancia de la validación continua y la atención al detalle en el desarrollo de compiladores. El manejo adecuado de tipos y coerciones asegura la robustez y portabilidad del código.

Este proyecto constituye una base sólida para la exploración de optimizaciones, generación de código máquina y soporte para características avanzadas del lenguaje.