Diseño e Implementación de un Compilador

Este documento presenta el proyecto final del curso, titulado "Diseño e Implementación de un Compilador". El proyecto fue desarrollado de manera individual por Raynick Rosario, matrícula 1-21-0012. El objetivo principal es aplicar los conocimientos adquiridos durante el curso para construir un compilador funcional utilizando el lenguaje de programación Python.

El desarrollo de compiladores es una disciplina fundamental en las ciencias de la computación. Permite entender en profundidad el funcionamiento interno de los lenguajes de programación y el proceso que transforma código fuente en instrucciones ejecutables por una máquina. En este proyecto se desarrolla un compilador sencillo, desde la etapa de análisis léxico hasta la generación de código final, con el propósito de asimilar prácticamente los conceptos teóricos.

Capítulo 1: Descripción del Proyecto

El desarrollo de software depende en gran medida de los compiladores, que traducen lenguajes de alto nivel a lenguaje de máquina. Sin embargo, el proceso que ocurre entre escribir el código y su ejecución es muchas veces una "caja negra" para los estudiantes. Este proyecto busca resolver ese desconocimiento mediante la implementación práctica de un compilador.

Se propone el desarrollo de un compilador escrito en Python para un lenguaje imperativo simplificado inspirado en Python. Este lenguaje de entrada tendrá una sintaxis clara y sencilla, permitiendo el enfoque en el funcionamiento interno del compilador. El compilador recorre todas las etapas clásicas: analisis lexico, analisis sintactico, analisis semantico, generacion de codigo intermedio y final, y optimizacion basica.

Objetivo General:

Construir un compilador funcional escrito en Python que permite traducir programas escritos en un lenguaje imperativo sencillo a una representación ejecutable.

Objetivos Específicos:

- Comprender y aplicar cada etapa del proceso de compilación.

- Implementar un analizador léxico utilizando expresiones regulares.

- Diseñar e implementar un analizador sintáctico basado en una gramática definida.

- Realizar la verificación semántica básica del código fuente.

- Generar una representación intermedia del programa.

- Producir una salida final interpretable por la máquina virtual definida.

- Incorporar técnicas simples de optimizacion de codigo.

- Documentar detalladamente cada fase del desarrollo del compilador.

Capítulo 2: Marco Teórico

2.1.1 Definición de un compilador: Un compilador es un programa que traduce el código fuente escrito en un lenguaje de programación de alto nivel a un lenguaje de bajo nivel, normalmente código máquina o lenguaje intermedio, para que pueda ser ejecutado por una computadora.

2.1.2 Estructura de un Compilador: La estructura de un compilador incluye varias fases: analisis lexico, analisis sintactico, analisis semantico, generacion de codigo intermedio, optimizacion de codigo, y generacion de codigo final. También incluye componentes auxiliares como las tablas de símbolos y el entorno de ejecución.

2.2 Análisis Léxico: Esta fase convierte la secuencia de caracteres del código fuente en una secuencia de tokens. Cada token representa una unidad significativa del lenguaje (palabras clave, identificadores, operadores, etc.).

2.3 Análisis Sintáctico: Verifica que la secuencia de tokens siga la estructura gramatical del lenguaje. El resultado es un árbol sintáctico (o AST) que representa la estructura jerárquica del código.

2.3.1 Análisis sintáctico descendente: Es un tipo de análisis que intenta construir el árbol sintáctico desde la raíz hasta las hojas, aplicando reglas de producción desde el símbolo inicial hacia los símbolos terminales.

2.4 Analisis Semantico: Asegura que las estructuras sintacticamente correctas tambien sean semántica o contextualmente correctas. Por ejemplo, que las variables están declaradas antes de usarse.

2.4.1 Traducción dirigida por la sintaxis: Consiste en asociar reglas semánticas a las producciones gramaticales para guiar la traducción del programa.

2.4.2 Atributos: Son valores asociados a los símbolos del lenguaje que se utilizan durante el análisis semántico para llevar información adicional.

2.4.3 Notaciones para asociar reglas semánticas

2.4.3.1 Definición dirigida por la sintaxis: Utiliza atributos sintetizados y heredados para representar la semántica.

2.4.3.2 Esquema de traducción: Representa la combinación de una gramática con reglas semánticas escritas en un lenguaje de dirección de acciones.

2.5 Generación de Código Intermedio: Traduce el árbol sintáctico o la representación intermedia a un código intermedio mas cercano al lenguaje maquina, como el codigo de tres direcciones.

2.6 Generación de Código: Final Convierte el código intermedio en instrucciones específicas para una máquina virtual o lenguaje de salida.

2.7 Tablas de Símbolos y Tipos: Contienen información sobre los identificadores utilizados en el programa, como nombre, tipo, ámbito y valor.

2.8 Entorno de Ejecución: Define la organización de la memoria en tiempo de ejecución, incluyendo pila, montículo y variables globales.

2.9 Optimización de Código: Aplica técnicas para mejorar el rendimiento del código, reduciendo instrucciones innecesarias o mejorando la eficiencia de ejecución.

2.9 Manejo de Errores: Incluye la detección, reporte y recuperación de errores durante el proceso de compilación.

2.10 Herramientas de Construcción de Compiladores: Existen herramientas que ayudan en la construcción de compiladores como Flex, Bison, ANTLR y frameworks en Python como PLY o Lark.

Capitulo 3: Desarrollo

3.1 Analizador Léxico

3.1.1 Autómata del análisis Léxico El autómata finito para el análisis léxico se encarga de reconocer patrones como identificadores, números, operadores y palabras clave. Está diseñado como un autómata determinista que recorre caracter por caracter la entrada y clasifica las secuencias válidas como tokens.

3.1.2 Tabla de Símbolos Es una estructura que guarda los identificadores encontrados en el código fuente junto con información adicional como tipo, valor y ámbito. Se utiliza en fases posteriores del compilador.

3.1.3 Implementación del análisis léxico Se utilizó el módulo `re` de Python para construir expresiones regulares que identifican los tokens. Alternativamente, se podría usar una herramienta como `PLY` (Python Lex-Yacc).

3.2 Analizador Sintáctico

3.2.1 Definición de Gramáticas Libres del Contexto La gramática utilizada define un lenguaje imperativo sencillo. Algunos ejemplos de reglas gramaticales:

- `programa -> declaracion_list`
- `declaracion_list -> declaracion declaracion_list | ε`
- `declaracion -> asignacion | condicion`
- `asignacion -> ID = expresion`
- `condicion -> if expresion then declaracion_list end`

3.2.2 Tipo de Analizador Sintáctico (LL, LR o ninguno) Se optó por un analizador LL recursivo descendente por su simplicidad y facilidad de implementación manual en Python.

3.2.3 Autómata del análisis Sintáctico El autómata se representa implícitamente mediante funciones recursivas que siguen las reglas de la gramática. Cada función representa una no terminal.

3.2.3 Implementación del análisis Sintáctico La implementación consiste en funciones recursivas que validan la estructura de los tokens generados por el analizador léxico, construyendo un árbol sintáctico.

3.3 Reglas del Lenguaje El lenguaje admite operaciones aritméticas básicas, condicionales (`if`), y asignaciones. No se incluyen funciones ni estructuras complejas.

3.4 Generador de Código Intermedio El árbol sintáctico se recorre para producir una representación intermedia en forma de instrucciones tipo tres direcciones:

t1 = 5

t2 = 3

t3 = t1 + t2

x = t3

3.5 Manejo de Errores El compilador detecta errores léxicos (tokens inválidos), sintácticos (estructura incorrecta) y semánticos (uso de variables no declaradas). Se utiliza una estructura `try-except` para reportar errores sin detener la ejecución completa.

3.6 Optimización de Código a Compilar Se implementan optimizaciones simples como:

- Eliminación de código muerto.
- Simplificación de expresiones constantes.

3.7 Creación del Proyecto

3.7.1 Pasos para crear el proyecto

- Definir la gramática del lenguaje.
- Implementar el analizador léxico.
- Implementar el analizador sintáctico.
- Definir las reglas semánticas.
- Generar el código intermedio.
- Implementar la optimización.
- Crear una interfaz simple de ejecución.

3.7.2 Explicación del funcionamiento del proyecto El compilador recibe como entrada un archivo de texto con código fuente. Pasa por todas las fases de compilación hasta generar una salida que representa instrucciones ejecutables en una máquina virtual simple.

3.7.3 Despliegue de la aplicación El proyecto puede ejecutarse desde la línea de comandos. Se planea empaquetarlo como módulo Python ejecutable para facilitar su uso.

Conclusiones

El desarrollo de este compilador permitió comprender a fondo el proceso de traducción del código fuente. Cada etapa representa un desafío técnico y conceptual, y su implementación en Python facilitó la experimentación y aprendizaje. Se logró cumplir con los objetivos planteados, y el resultado es una herramienta educativa útil para seguir profundizando en temas de compiladores.

Bibliografías

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition).
- Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J., & Langendoen, K. (2012). Modern Compiler Design.
- Documentación oficial de Python: https://docs.python.org/3/
- PLY (Python Lex-Yacc): https://www.dabeaz.com/ply/