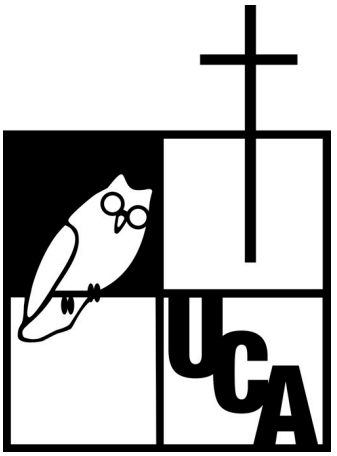
Universidad Centroamericana
“José Simeón Cañas”



Materia: Teoría de Lenguaje de Programación - Sección 1

Catedrático: Ing. Jaime Roberto Climaco

Tema: Documentación compilador lenguaje C y definición de gramáticas

Integrantes:	Andres Emilio Puente Cruz	00287919
	Oscar Alexander Juarez Gonzalez	00126320
	Victor Rafael Valenzuela Cortez	00022120
	Carlos Misael Perez Perez	00202118
	Xavier Alessandro Quiñonez del Cid	00048219
	Mario Ernesto Mayen Castro	00220618

Grupo 02

Índice

Introducción

¿Qué es un compilador?
Un compilador es una herramienta fundamental en la informática que permite traducir código fuente escrito en un lenguaje de programación a un código de máquina ejecutable. Este documento describe un compilador diseñado para el lenguaje de programación C, que incluye tres componentes principales: el analizador léxico, el parser (analizador sintáctico) y el analizador semántico.

El proceso completo abarca desde la identificación de tokens hasta el análisis semántico, verificando el correcto uso de variables, compatibilidad de tipos, y más, para garantizar la generación de un árbol sintáctico y una tabla de símbolos coherentes. En este documento se detalla la implementación de cada componente, destacando su rol y funcionamiento dentro del sistema, además de ofrecer ejemplos prácticos y explicaciones claras sobre su operación.

Propósito del compilador
El propósito de este proyecto es desarrollar un compilador modular para el lenguaje C que abarque las siguientes funciones principales:

1. ***Identificar y procesar tokens:*** Descomponer el código fuente en componentes mínimos como identificadores, operadores y delimitadores, asegurando un análisis preciso a nivel léxico.
2. ***Estructuración lógica:*** Construir un árbol sintáctico basado en las reglas gramaticales del lenguaje C, que refleje la jerarquía y relaciones entre las instrucciones del programa.
3. ***Validación semántica:*** Implementar verificaciones exhaustivas para garantizar que las reglas del lenguaje sean respetadas, como:

    3.1. Declaración y uso correcto de variables.
    3.2. Compatibilidad de tipos en asignaciones y operaciones.
    3.3. Reporte de advertencias para optimización del código, como variables no utilizadas.

El objetivo es que este compilador no solo sirva como herramienta de traducción, sino que actúe como un sistema integral de validación y depuración, ayudando a los desarrolladores a escribir código más robusto y eficiente.

Definición de gramáticas

Gramática para declaraciones, instrucciones, y estructuras básicas de control y funciones.

La siguiente gramática proporciona la definición de aspectos fundamentales del lenguaje C relacionados con **declaraciones, instrucciones, y estructuras básicas de control y funciones**.

Gramática Principal

1. **S -> INSTRUCCION _INSTRUCCION**
    - `S` es el símbolo inicial que representa el programa completo.
    - `INSTRUCCION` representa una instrucción en el programa.
    - `_INSTRUCCION` representa una lista de instrucciones adicionales.
2. **_INSTRUCCION -> INSTRUCCION _INSTRUCCION**
    - `_INSTRUCCION` puede ser una instrucción seguida de más instrucciones.
3. **_INSTRUCCION -> ''**
    - `_INSTRUCCION` también puede ser vacío, indicando el final de la lista de instrucciones.

Instrucciones

1. **INSTRUCCION -> identificador EXPRESION punto_coma**
    - Una instrucción puede ser un identificador seguido de una expresión y un punto y coma.
2. **INSTRUCCION -> RETORNO punto_coma**
    - Una instrucción puede ser una declaración de retorno seguida de un punto y coma.
3. **INSTRUCCION -> TIPO identificador DECLARACION**
    - Una instrucción puede ser una declaración de variable o función, comenzando con un tipo y un identificador.
4. **INSTRUCCION -> void identificador parentesis_de_inicio PARAMETROS parentesis_de_cierre FUNCION_COLA**
    - Una instrucción puede ser la declaración de una función que no retorna valor (`void`), con parámetros y un cuerpo de función.

Declaraciones

1. **DECLARACION -> ASIGNACION _DECLARACION_CONT punto_coma**
    - Una declaración puede ser una asignación seguida de más declaraciones y un punto y coma.
2. **DECLARACION -> parentesis_de_inicio PARAMETROS parentesis_de_cierre FUNCION_COLA**
    - Una declaración puede ser una función con parámetros y un cuerpo de función.
3. **FUNCION_COLA -> punto_coma**

    - El cuerpo de una función puede ser un punto y coma, indicando una función vacía.
4. **FUNCION_COLA -> llave_de_inicio _BLOQUE llave_de_cierre**
    - El cuerpo de una función puede ser un bloque de código encerrado entre llaves.

Declaraciones Continuas

1. **_DECLARACION_CONT -> coma DECLARACION_CONT _DECLARACION_CONT**
    - Una lista de declaraciones puede ser separada por comas.
2. **_DECLARACION_CONT -> "**
    - La lista de declaraciones puede ser vacía.
3. **DECLARACION_CONT -> identificador ASIGNACION**
    - Una declaración continua puede ser un identificador seguido de una asignación.

Asignaciones

1. **ASIGNACION -> asignacion EXPRESION**
    - Una asignación puede ser un operador de asignación seguido de una expresión.
2. **ASIGNACION -> "**
    - Una asignación puede ser vacía.

Parámetros

1. **PARAMETROS -> TIPO identificador _PARAMETROS**
    - Los parámetros de una función pueden ser un tipo y un identificador, seguidos de más parámetros.
2. **PARAMETROS -> "**
    - Los parámetros pueden ser vacíos.
3. **_PARAMETROS -> coma TIPO identificador _PARAMETROS**
    - Los parámetros adicionales pueden ser separados por comas.
4. **_PARAMETROS -> "**
    - Los parámetros adicionales pueden ser vacíos.

Tipos

1. **TIPO -> int**
2. **TIPO -> float**
3. **TIPO -> char**
    - Los tipos pueden ser `int`, `float` o `char`.

Retorno

1. **RETORNO -> return EXPRESION**
    - Una declaración de retorno puede ser la palabra clave `return` seguida de una expresión.

Expresiones

1. **EXPRESION -> ASIGNACION**
    - Una expresión puede ser una asignación.
2. **EXPRESION -> LLAMAR_FUNCION**
    - Una expresión puede ser una llamada a función.
3. **LLAMAR_FUNCION -> parentesis_de_inicio ARGUMENT parentesis_de_cierre**
    - Una llamada a función puede ser un identificador seguido de paréntesis con argumentos.

Argumentos

1. **ARGUMENT -> EXPRESION _ARGUMENT**
    - Un argumento puede ser una expresión seguida de más argumentos.
2. **ARGUMENT -> ''**
    - Un argumento puede ser vacío.
3. **_ARGUMENT -> coma EXPRESION _ARGUMENT**
    - Los argumentos adicionales pueden ser separados por comas.
4. **_ARGUMENT -> ''**
    - Los argumentos adicionales pueden ser vacíos.

Expresión Vacía

1. **EXPRESION -> e**
    - Una expresión puede ser vacía.

Segunda Parte de la Gramática

1. **INSTRUCCION -> _INSTRUCCION INSTRUCCION**
    - Una instrucción puede ser una lista de instrucciones seguida de una instrucción.
2. **INSTRUCCION -> ''**
    - Una instrucción puede ser vacía.
3. **_INSTRUCCION -> DECLARACION punto_coma**
    - Una instrucción puede ser una declaración seguida de un punto y coma.
4. **_INSTRUCCION -> identificador ID punto_coma**
    - Una instrucción puede ser un identificador seguido de un ID y un punto y coma.
5. **_INSTRUCCION -> RETURN_I punto_coma**
    - Una instrucción puede ser una declaración de retorno seguida de un punto y coma.

Identificadores y Llamadas a Función

1. **ID -> D_INIT**
    - Un ID puede ser una inicialización de declaración.
2. **ID -> FUNTION_CALL**
    - Un ID puede ser una llamada a función.

Tipos

1. **TIPO -> int**
2. **TIPO -> float**
3. **TIPO -> char**
4. **TIPO -> void**
    - Los tipos pueden ser `int`, `float`, `char` o `void`.

Retorno

1. **RETURN_I -> return EXPRESION**
    - Una declaración de retorno puede ser la palabra clave `return` seguida de una expresión.

Llamadas a Función

1. **FUNTION_CALL -> (ARGUMENT)**
    - Una llamada a función puede ser un identificador seguido de paréntesis con argumentos.

Argumentos

1. **ARGUMENT -> EXPRESION A_ARGUMENT**
    - Un argumento puede ser una expresión seguida de más argumentos.
2. **ARGUMENT -> ''**
    - Un argumento puede ser vacío.
3. **A_ARGUMENT -> , EXPRESION A_ARGUMENT**
    - Los argumentos adicionales pueden ser separados por comas.
4. **A_ARGUMENT -> ''**
    - Los argumentos adicionales pueden ser vacíos.

Declaraciones

1. **DECLARACION -> TIPO identificador D_INIT**
    - Una declaración puede ser un tipo seguido de un identificador y una inicialización.
2. **D_INIT -> = EXPRESION**
    - Una inicialización puede ser un operador de asignación seguido de una expresión.
3. **D_INIT -> ''**
    - Una inicialización puede ser vacía.

Gramática para declaraciones de variables, inicializaciones, y declaraciones de funciones, así como la especificación de tipos y parámetros.

La siguiente gramática proporciona la definición de aspectos fundamentales del lenguaje C relacionados con declaraciones de variables, inicializaciones, y declaraciones de funciones, así como la especificación de tipos y parámetros.

Gramática Principal

1. **PROGRAM -> A PROGRAM**
    - `PROGRAM` es el símbolo inicial que representa el programa completo.
    - `A` representa una declaración de tipo seguida de una declaración global.
    - `PROGRAM` puede ser una lista de declaraciones.
2. **PROGRAM -> "**
    - `PROGRAM` también puede ser vacío, indicando el final del programa.

Declaraciones

1. **A -> TypeSpecifier B**
    - `A` es una declaración de tipo seguida de un identificador y una declaración global.
2. **B -> identificador GlobalDeclaration**
    - `B` es un identificador seguido de una declaración global.

Declaraciones Globales

1. **GlobalDeclaration -> VariableDeclaration**
    - Una declaración global puede ser una declaración de variable.
2. **GlobalDeclaration -> VariableInit**
    - Una declaración global puede ser una inicialización de variable.
3. **GlobalDeclaration -> FunctionDeclaration**
    - Una declaración global puede ser una declaración de función.

Declaraciones de Variables

1. **VariableDeclaration -> punto_coma**
    - Una declaración de variable puede ser un punto y coma, indicando una declaración sin inicialización.
2. **VariableInit -> coma B**
    - Una inicialización de variable puede ser una coma seguida de otra declaración.

Declaraciones de Funciones

1. **FunctionDeclaration -> (ParameterList) punto_coma**
    - Una declaración de función puede ser un identificador seguido de paréntesis con una lista de parámetros y un punto y coma.

Especificadores de Tipo

1. **TypeSpecifier -> int**
2. **TypeSpecifier -> float**
3. **TypeSpecifier -> char**
    - Los especificadores de tipo pueden ser `int`, `float` o `char`.

Lista de Parámetros

1. **ParameterList -> Parameter ParameterRest**
    - La lista de parámetros puede ser un parámetro seguido de más parámetros.
2. **ParameterList -> void**
    - La lista de parámetros puede ser `void`, indicando que la función no tiene parámetros.
3. **ParameterList -> "**
    - La lista de parámetros puede ser vacía.

Parámetros

1. **Parameter -> TypeSpecifier Identifier**
    - Un parámetro puede ser un especificador de tipo seguido de un identificador.
2. **ParameterRest -> coma Parameter ParameterRest**
    - Los parámetros adicionales pueden ser separados por comas.
3. **ParameterRest -> "**
    - Los parámetros adicionales pueden ser vacíos.

Gramática para declaraciones globales, asignaciones, parámetros, y bloques de código.

La siguiente gramática proporciona la definición de aspectos fundamentales del lenguaje C relacionados con declaraciones globales, asignaciones, parámetros, y bloques de código.

Gramática Principal

1. **PROGRAMA -> _GLOBAL**
    - `PROGRAMA` es el símbolo inicial que representa el programa completo.
    - `_GLOBAL` representa una lista de declaraciones globales.
2. **_GLOBAL -> GLOBAL _GLOBAL**
    - `_GLOBAL` puede ser una declaración global seguida de más declaraciones globales.
3. **_GLOBAL -> "**
    - `_GLOBAL` también puede ser vacío, indicando el final de las declaraciones globales.

Declaraciones Globales

1. **GLOBAL -> TIPO identificador DECLARACION_GLBL**
    - `GLOBAL` puede ser una declaración de tipo seguida de un identificador y una declaración global.
2. **GLOBAL -> void identificador parentesis_de_inicio PARAMETROS parentesis_de_cierre FUNCIONG_COLA**
    - `GLOBAL` puede ser una declaración de función que no retorna valor (`void`), con parámetros y un cuerpo de función.

Declaraciones Globales Detalladas

1. **DECLARACION_GLBL -> ASIGNACION _ASIGNACION_CONST punto_coma**
    - Una declaración global puede ser una asignación seguida de más asignaciones y un punto y coma.
2. **DECLARACION_GLBL -> parentesis_de_inicio PARAMETROS parentesis_de_cierre FUNCIONG_COLA**
    - Una declaración global puede ser una función con parámetros y un cuerpo de función.
3. **FUNCIONG_COLA -> punto_coma**
    - El cuerpo de una función puede ser un punto y coma, indicando una función vacía.
4. **FUNCIONG_COLA -> llave_de_inicio BLOQUE llave_de_cierre**
    - El cuerpo de una función puede ser un bloque de código encerrado entre llaves.

Asignaciones Continuas

1. **_ASIGNACION_CONST -> coma ASIGNACION_CONST _ASIGNACION_CONST**
    - Una lista de asignaciones puede ser separada por comas.
2. **_ASIGNACION_CONST -> "**
    - La lista de asignaciones puede ser vacía.
3. **ASIGNACION_CONST -> identificador ASIGNACION**
    - Una asignación continua puede ser un identificador seguido de una asignación.

Expresiones Constantes

1. **EXPRESION_CONST -> constante_entera**
2. **EXPRESION_CONST -> constante_character**
3. **EXPRESION_CONST -> constante_flotante**
    - Las expresiones constantes pueden ser enteras, caracteres o flotantes.

Asignaciones

1. **ASIGNACION -> asignacion EXPRESION_CONSTANTE**
    - Una asignación puede ser un operador de asignación seguido de una expresión constante.
    - Hola bb

2. **ASIGNACION -> "**
    - Una asignación puede ser vacía.

Parámetros

1. **PARAMETROS -> TIPO identificador _PARAMETROS**
    - Los parámetros de una función pueden ser un tipo y un identificador, seguidos de más parámetros.
2. **PARAMETROS -> "**
    - Los parámetros pueden ser vacíos.
3. **_PARAMETROS -> coma TIPO identificador _PARAMETROS**
    - Los parámetros adicionales pueden ser separados por comas.
4. **_PARAMETROS -> "**
    - Los parámetros adicionales pueden ser vacíos.

Tipos

1. **TIPO -> int**
2. **TIPO -> float**
3. **TIPO -> char**
    - Los tipos pueden ser `int`, `float` o `char`.

Bloques de Código

1. **BLOQUE -> b**
    - Un bloque de código puede ser representado por `b`.

Gramática para definición de estructuras de control y bloques de código

La siguiente gramática proporciona la definición de estructuras de control y bloques de código en el lenguaje C, incluyendo instrucciones condicionales y bucles.

Gramática Principal

1. **BLOQUE -> INSTRUCCION_B BLOQUE**
    - `BLOQUE` es el símbolo inicial que representa un bloque de código.
    - `INSTRUCCION_B` representa una instrucción dentro del bloque.
    - `BLOQUE` puede ser una lista de instrucciones.
2. **BLOQUE -> "**
    - `BLOQUE` también puede ser vacío, indicando el final del bloque de código.

Instrucciones en el Bloque

1. **INSTRUCCION_B -> INSTRUCCION**
    - `INSTRUCCION_B` puede ser una instrucción simple.
2. **INSTRUCCION_B -> condicion_if parentesis_de_inicio EXPRESION parentesis_de_cierre INSTRUCCION_C BLOQUE_ELSE**

    - `INSTRUCCION_B` puede ser una instrucción `if` con una expresión condicional, seguida de una instrucción compuesta y un bloque `else`.
3. **INSTRUCCION_B -> bucle_while parentesis_de_inicio EXPRESION parentesis_de_cierre INSTRUCCION_C**
    - `INSTRUCCION_B` puede ser un bucle `while` con una expresión condicional, seguido de una instrucción compuesta.
4. **INSTRUCCION_B -> bucle_do INSTRUCCION_C bucle_while parentesis_de_inicio EXPRESION parentesis_de_cierre punto_coma**
    - `INSTRUCCION_B` puede ser un bucle `do-while` con una instrucción compuesta y una expresión condicional.
5. **INSTRUCCION_B -> bucle_for parentesis_de_inicio INSTRUCCION identificador EXPRESION punto_coma identificador EXPRESION parentesis_de_cierre INSTRUCCION_C**
    - `INSTRUCCION_B` puede ser un bucle `for` con inicialización, condición y actualización, seguido de una instrucción compuesta.

Instrucciones Compuestas

1. **INSTRUCCION_C -> condicion_if parentesis_de_inicio EXPRESION parentesis_de_cierre INSTRUCCION_C condicion_else INSTRUCCION_C**
    - `INSTRUCCION_C` puede ser una instrucción `if-else` anidada.
2. **INSTRUCCION_C -> bucle_while parentesis_de_inicio EXPRESION parentesis_de_cierre INSTRUCCION_C**
    - `INSTRUCCION_C` puede ser un bucle `while` anidado.
3. **INSTRUCCION_C -> bucle_do INSTRUCCION_C bucle_while parentesis_de_inicio EXPRESION parentesis_de_cierre punto_coma**
    - `INSTRUCCION_C` puede ser un bucle `do-while` anidado.
4. **INSTRUCCION_C -> bucle_for parentesis_de_inicio INSTRUCCION identificador EXPRESION punto_coma identificador EXPRESION parentesis_de_cierre INSTRUCCION_C**
    - `INSTRUCCION_C` puede ser un bucle `for` anidado.
5. **INSTRUCCION_C -> INSTRUCCION**
    - `INSTRUCCION_C` puede ser una instrucción simple.

Bloques Else

1. **BLOQUE_ELSE -> condicion_else COLA_ELSE**
    - `BLOQUE_ELSE` puede ser una instrucción `else` seguida de una cola de instrucciones.
2. **BLOQUE_ELSE -> ''**
    - `BLOQUE_ELSE` también puede ser vacío, indicando que no hay bloque `else`.

Colas Else

1. **COLA_ELSE -> condicion_if parentesis_de_inicio EXPRESION parentesis_de_cierre COLA_ELSE**
    - `COLA_ELSE` puede ser una instrucción `if` anidada dentro de un `else`.

2. **COLA_ELSE -> INSTRUCCION**
    - `COLA_ELSE` puede ser una instrucción simple dentro de un `else`.

Instrucciones Simples

1. **INSTRUCCION -> i**
    - `INSTRUCCION` puede ser una instrucción simple representada por `i`.
2. **INSTRUCCION -> llave_de_inicio BLOQUE llave_de_cierre**
    - `INSTRUCCION` puede ser un bloque de código encerrado entre llaves.

Expresiones

1. **EXPRESION -> o**
    - `EXPRESION` puede ser una expresión representada por `o`.
2. **EXPRESION -> "**
    - `EXPRESION` también puede ser vacía.

Segunda Parte de la Gramática

1. **programa -> bloque**
    - `programa` es el símbolo inicial que representa el programa completo.
2. **bloque -> "{" lista_instrucciones "}**
    - `bloque` representa un bloque de código encerrado entre llaves.
3. **lista_instrucciones -> instruccion lista_instrucciones**
    - `lista_instrucciones` puede ser una instrucción seguida de más instrucciones.
4. **lista_instrucciones -> ε**
    - `lista_instrucciones` también puede ser vacío, indicando el final de la lista de instrucciones.

Instrucciones

1. **instruccion -> if_statement**
    - `instruccion` puede ser una instrucción `if`.
2. **instruccion -> while_statement**
    - `instruccion` puede ser una instrucción `while`.
3. **instruccion -> do_while_statement**
    - `instruccion` puede ser una instrucción `do-while`.
4. **instruccion -> for_statement**
    - `instruccion` puede ser una instrucción `for`.
5. **instruccion -> instruccion**
    - `instruccion` puede ser una instrucción simple.

Instrucciones If

1. **if_statement -> "if" "(" exp_bool ")" bloque**

    - `if_statement` puede ser una instrucción `if` con una expresión booleana y un bloque de código.
2. **if_statement -> "if" "(" exp_bool ")" bloque "else" bloque**
    - `if_statement` puede ser una instrucción `if-else` con una expresión booleana y dos bloques de código.

Instrucciones While

1. **while_statement -> "while" "(" exp_bool ")" bloque**
    - `while_statement` puede ser una instrucción `while` con una expresión booleana y un bloque de código.

Instrucciones Do-While

1. **do_while_statement -> "do" bloque "while" "(" exp_bool ")" ";"**
    - `do_while_statement` puede ser una instrucción `do-while` con un bloque de código y una expresión booleana.

Instrucciones For

1. **for_statement -> "for" "(" instruccion ";" exp_bool ";" instruccion ")" bloque**
    - `for_statement` puede ser una instrucción `for` con inicialización, condición, actualización y un bloque de código.

Gramática para expresiones aritméticas y lógicas

La siguiente gramática proporciona la definición de expresiones aritméticas y lógicas, incluyendo operadores de suma, resta, multiplicación, división, módulo, y operadores lógicos.

Gramática Principal

1. **Expr -> Term Expr'**
    - `Expr` es el símbolo inicial que representa una expresión completa.
    - `Term` representa un término en la expresión.
    - `Expr'` representa el resto de la expresión después del término inicial.
2. **Expr' -> PlusExpr'**
    - `Expr'` puede ser una expresión de suma.
3. **Expr' -> MinusExpr'**
    - `Expr'` puede ser una expresión de resta.
4. **Expr' -> LogicalExpr**
    - `Expr'` puede ser una expresión lógica.
5. **Expr' -> ''**
    - `Expr'` también puede ser vacío, indicando el final de la expresión.

Expresiones de Suma y Resta

1. **PlusExpr' -> mas Term Expr'**

    - `PlusExpr'` representa una suma, seguida de un término y el resto de la expresión.
2. **MinusExpr' -> menos Term Expr'**
    - `MinusExpr'` representa una resta, seguida de un término y el resto de la expresión.

Términos

1. **Term -> Factor Term'**
    - `Term` representa un término que consiste en un factor seguido de más términos.
2. **Term' -> MultTerm'**
    - `Term'` puede ser una expresión de multiplicación.
3. **Term' -> DivTerm'**
    - `Term'` puede ser una expresión de división.
4. **Term' -> ModTerm'**
    - `Term'` puede ser una expresión de módulo.
5. **Term' -> ''**
    - `Term'` también puede ser vacío, indicando el final del término.

Expresiones de Multiplicación, División y Módulo

1. **MultTerm' -> multiplicacion Factor Term'**
    - `MultTerm'` representa una multiplicación, seguida de un factor y el resto del término.
2. **DivTerm' -> division Factor Term'**
    - `DivTerm'` representa una división, seguida de un factor y el resto del término.
3. **ModTerm' -> modulo Factor Term'**
    - `ModTerm'` representa una operación de módulo, seguida de un factor y el resto del término.

Factores

1. **Factor -> constante_entera**
    - `Factor` puede ser una constante entera.
2. **Factor -> constante_flotante**
    - `Factor` puede ser una constante flotante.
3. **Factor -> OpenExpr**
    - `Factor` puede ser una expresión entre paréntesis.
4. **OpenExpr -> (Expr)**
    - `OpenExpr` representa una expresión encerrada entre paréntesis.

Expresiones Lógicas

1. **LogicalExpr -> operador_y Expr**
    - `LogicalExpr` representa una expresión lógica con el operador `y`.

2. **LogicalExpr -> operador_o Expr**
    - `LogicalExpr` representa una expresión lógica con el operador `o`.

Componentes del Compilador

3.1 Analizador Léxico

El analizador léxico es el primer componente del compilador, encargado de descomponer el código fuente en unidades significativas conocidas como tokens. Cada token representa un tipo de componente básico del lenguaje de programación, como palabras clave, identificadores, operadores y delimitadores.

➢ ***Implementación (Archivo: c_lexer.py)***

- Se define un conjunto de expresiones regulares para identificar cada tipo de token, como operadores (**+**, **-**, *****, */*), constantes (Números enteros y flotantes), identificadores (Nombres de las variables o el nombre de las funciones) y delimitadores (Paréntesis, punto y coma (**;**), llaves (**{ }**), etc).

- Además, el analizador léxico incluye un manejo de errores léxicos. Si encuentra caracteres no válidos o símbolos que no pertenecen al lenguaje, muestra advertencias para alertar al programador.

- También gestiona comentarios y cadenas de texto, asegurándose de que no interfieran con el proceso de análisis léxico.

Ejemplo de tokens generados:

Un código como int x = 10; generaría tokens para “**Int**”, “**x**”, “**=**”, “**10**”, y “**;**”.

- ***int*** (palabra clave)
- ***x*** (identificador)
- **=** (operador de asignación)
- ***10*** (constante numérica)
- ***;*** (delimitador)

3.2 Parser (Analizador Sintáctico)

El parser toma los tokens generados por el analizador léxico y construye un árbol sintáctico, que es una representación jerárquica de la estructura lógica del programa, siguiendo las reglas de la gramática del lenguaje C.

➢ ***Implementación (Archivo: c_parser.py)***

- El parser utiliza una tabla de producciones que define las reglas gramaticales del lenguaje C. Estas reglas determinan cómo se deben combinar los tokens para formar instrucciones y expresiones válidas.

- Además, el parser gestiona los errores sintácticos, tales como la falta de un punto y coma (;) al final de una declaración o el uso incorrecto de

delimitadores. En caso de encontrar un error, el parser tiene la capacidad de recuperarse y continuar el análisis en busca de más errores.

- El resultado de este análisis es la construcción de un árbol sintáctico. Este árbol refleja la organización y jerarquía de los elementos del programa, mostrando las relaciones entre instrucciones, expresiones y operadores.

Ejemplo: Para un código como int x = 10;
El parser construiría un árbol con la siguiente estructura:

- ***Nodo raíz:*** ASIGNACIÓN
  - ***Nodo hijo 1:*** IDENTIFICADOR (x)
  - ***Nodo hijo 2***: CONSTANTE (10)
  - ***Nodo hijo 3:*** DELIMITADOR (;)

3.3 Analizador Semántico
El analizador semántico valida las reglas lógicas del programa. Se asegura de que las variables sean declaradas antes de su uso, que las operaciones sean compatibles con los tipos de datos, y que no existan variables sin usar.

➢ ***Implementación (Archivo: c_semantic.py)***
- ***Declaración y uso de variables:*** Se asegura de que todas las variables hayan sido declaradas antes de ser utilizadas. Si una variable es utilizada sin haber sido previamente declarada, se genera un error.

- ***Compatibilidad de tipos:*** Verifica que las operaciones sean tipo-compatibles. Por ejemplo, no se puede sumar un número entero a una cadena de texto sin que esto genere un error de tipo.

- ***Advertencias de optimización:*** Detecta variables no usadas en el código, generando advertencias que ayudan a optimizar el código y eliminar posibles errores lógicos.

Ejemplo: Para un código como el siguiente:

```
int x = 5;
y = x + 3;
```

El analizador semántico generaría un error porque la variable y se usa sin haber sido previamente declarada. El mensaje de error podría ser algo como:

"Error: Variable 'y' usada pero nunca declarada."

Esto es importante porque permite al programador detectar errores de programación comunes, como el uso de variables no declaradas, lo cual podría llevar a comportamientos inesperados en tiempo de ejecución.

Ejecución del Compilador

La ejecución del compilador implica integrar todos los componentes previos (analizador léxico, parser y analizador semántico) para analizar un programa completo. El objetivo es transformar el código fuente escrito en C en una estructura que pueda ser entendida por la máquina, garantizando que cumpla con las reglas del lenguaje y no contenga errores lógicos o sintácticos.

1. ***Flujo de Trabajo (Archivo: c_compiler.py):***
   El proceso de ejecución del compilador sigue una serie de pasos secuenciales, donde cada componente del compilador realiza una tarea específica para garantizar la correcta traducción del código fuente.

   1.1. ***Análisis Léxico:***
   - 1.1.1. El compilador comienza el proceso con el análisis léxico, que descompone el código fuente en tokens.
   - 1.1.2. Cada token es una unidad mínima significativa, como palabras clave, operadores, identificadores, constantes y delimitadores.
   - 1.1.3. Los tokens identificados son almacenados en una lista, que luego será pasada al siguiente componente del compilador para su análisis sintáctico.
   - 1.1.4. Durante este proceso, el compilador también detecta caracteres no válidos y genera advertencias de errores léxicos cuando corresponde.

   1.2. ***Análisis Sintáctico:***
   - 1.2.1. Una vez que el código fuente ha sido tokenizado, el parser toma estos tokens y construye un árbol sintáctico.
   - 1.2.2. Este árbol refleja la estructura jerárquica de las instrucciones y expresiones del programa según las reglas gramaticales del lenguaje C.
   - 1.2.3. El parser valida que la sintaxis del código fuente siga las reglas definidas, detectando errores como la falta de un punto y coma, paréntesis desbalanceados o uso incorrecto de delimitadores.
   - 1.2.4. En caso de errores sintácticos, el parser no solo los reporta, sino que también intenta recuperarse para seguir analizando el resto del código, permitiendo al programador identificar y corregir múltiples problemas en un solo análisis.

   1.3. ***Análisis Semántico:***
   - 1.3.1. Después de generar el árbol sintáctico, el analizador semántico revisa el código para asegurarse de que todas las operaciones sean lógicas y que se respeten las reglas semánticas del lenguaje.
   - 1.3.2. Este paso es crucial para la verificación de variables: el analizador semántico garantiza que todas las variables sean declaradas antes de su uso, que las operaciones entre tipos de datos sean compatibles y que no haya variables no utilizadas en el código.

1.3.3. Genera advertencias cuando detecta que una variable fue declarada pero no utilizada, ayudando a optimizar el código y mejorar su eficiencia.

2. Ejemplo de Ejecución: Dado un archivo test/test1.c, el compilador realiza lo siguiente:

2.1. **Generación de tokens:** El código int x = 10; y = x + 3; es descompuesto en tokens:

- "**int**", "**x**", "**=**", "**10**", "**;**", "**y**", "**=**", "**x**", "**+**", "**3**", "**;**".

2.2. **Construcción del Árbol Sintáctico:** El parser organiza los tokens en un árbol sintáctico:

- Para int x = 10;
  - **Nodo raíz:** ASIGNACIÓN
  - **Hijos:** TIPO (int), IDENTIFICADOR (x), CONSTANTE (10).

- Para y = x + 3;
  - **Nodo raíz:** ASIGNACIÓN
  - **Hijos:** IDENTIFICADOR (y), OPERACIÓN (con hijos x, 3, +).

2.3. **Ejecución del Análisis Semántico:** El analizador semántico detecta un error:

➢ *"Error: Variable 'y' usada pero nunca declarada."*

Si no se encuentran errores, se pueden generar advertencias sobre variables no usadas.

Conclusión

El compilador desarrollado integra un análisis exhaustivo en tres niveles: léxico, sintáctico y semántico. Esto no solo garantiza que el código fuente sea válido y estructurado correctamente, sino que también ofrece herramientas avanzadas para detectar y resolver errores comunes de programación.

Al validar el correcto uso de variables y la compatibilidad de tipos, el analizador semántico mejora significativamente la calidad del código generado, ofreciendo advertencias útiles para optimizar el desarrollo. La estructura modular del compilador permite futuras expansiones y mejoras, asegurando su adaptabilidad a nuevas necesidades. Este documento refleja un sistema funcional y completo que abarca todos los componentes necesarios para un compilador eficiente y práctico.