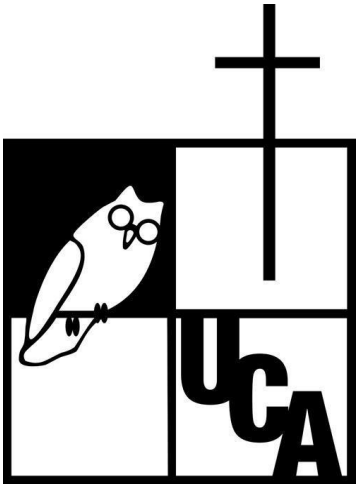
**UNIVERSIDAD CENTROAMERICANA "JOSÉ SIMEÓN CAÑAS"**




# TEORÍAS DE LENGUAJES DE PROGRAMACIÓN

## Catedrático: Jaime Roberto Clímaco

### Analizador Sintáctico y Analizador de Errores para un lenguaje C

**Integrantes:**

| | |
|---|---|
| Aníbal Ernesto Hernández García | 00401117 |
| Roberto Carlos Orellana Chávez | 00163618 |
| José Roberto Nasser Sánchez | 00015218 |
| Javier Alexander Villatoro Jurado | 00199919 |

# INTRODUCCIÓN

En el contexto de la ingeniería de software, los compiladores desempeñan un papel fundamental en la transformación de código fuente legible por humanos a un formato ejecutable por máquinas. Este proyecto se centra en el diseño y desarrollo de un compilador para el lenguaje C, explorando en detalle los componentes esenciales que intervienen en este proceso.

Un aspecto crítico en la compilación es el análisis sintáctico, llevado a cabo por el parser. Este componente es esencial para verificar que el código fuente adhiera estrictamente a las reglas gramaticales del lenguaje C. Funcionando como un validador sintáctico, el parser garantiza la corrección estructural del código, identificando y señalando cualquier desviación de las normas del lenguaje.

Complementariamente, el analizador de errores desempeña un papel crucial en la detección y notificación de cualquier anomalía en el código fuente. Al identificar errores sintácticos y semánticos, este componente facilita la depuración y mejora del código, contribuyendo significativamente a la calidad y fiabilidad del software desarrollado en C.

Este documento presenta los resultados obtenidos en el desarrollo de un compilador para C, con un énfasis especial en el diseño e implementación del parser y del analizador de errores. A través de un enfoque riguroso en estos componentes, se busca desarrollar un compilador robusto y eficiente que facilite el proceso de desarrollo de software en el entorno C.

# DESARROLLO

## ESTRUCTURA BÁSICA DEL PARSER

```
#importaciones y conf global
from lexer import tokens, lexer
from parse_table import *
from collections import defaultdict
import time
import sys
```

**from lexer import tokens, lexer:** Esta línea de código en Python está utilizando la instrucción import para incorporar funcionalidades de otro módulo (archivo Python) llamado lexer hacia el módulo actual.

**from parse_table import *:** Incorpora al entorno actual todas las especificaciones definidas en el módulo parse_table. Esto engloba funciones, clases y otras variables declaradas en este módulo.

**from collections import defaultdict:** Incorpora la clase defaultdict del módulo collections. Esta clase representa un tipo de diccionario que permite asignar un valor por defecto a claves no existentes

**import time:** Incorpora el módulo time, el cual ofrece un conjunto de funciones diseñadas para manipular y gestionar información temporal.

**import sys:** Incorpora el módulo sys, el cual facilita el acceso a variables y funciones internas del intérprete de Python, permitiendo una interacción más profunda con el entorno de ejecución.

```
stack = ["EOF", 0]
```

Esto inicializa la **variable stack** como una lista que contiene dos elementos: la cadena "EOF" y el número 0.

## Función  base para la implementación del parser

```
def parse(code):
    # Abre y lee un archivo de código fuente.
    lexer.input(code)
    tok=lexer.token()
    x=stack[-1] #primer elemento de la pila
    # Bucle principal de análisis.
    while True:
        if x is not None and tok is not None:
        #Manejo de Tokens y Pila
            if x == tok.type:
                symbol_table_insert(tok.value, tok.type, tok.lineno, tok.lexpos)
                stack.pop()
                x=stack[-1]
                tok=lexer.token()
        #Manejo de errores
            if x in tokens and x != tok.type:
                print("Error: se esperaba ", x)
                print("Se encontro: ", tok.type)
                print("En posición:", tok.lexpos)
                print("En linea:", tok.lineno)
                tok = panic_mode_recovery(x, tok)
                continue
            if x not in tokens:
                if tok is None:
                    print("Analisis sintactico completado con exito")
                    return #aceptar
                print("van entrar a la tabla:")
                print(x)
                print(tok.type)
                print("En linea:", tok.lineno)
```

```
            print("Con valor: ", tok.value)
            celda=search_in_table(x,tok.type)
            #Manejo de Errores y Recuperación
            if  celda is None:
                expected_tokens = find_expected_tokens(x)
                print("Error: se esperaba uno de", expected_tokens, "pero se encontró", tok.type)
                print("En posición:", tok.lexpos)
                print("En linea:", tok.lineno)
                print("celda: ", celda)
                print("Entrando en modo pánico...")
                tok = panic_mode_recovery(expected_tokens, tok)
                if tok is None or tok.type == 'EOF':
                    print("No se pudo recuperar del error.")
                    return 0
                # Reanuda el análisis después de la recuperación
                x = stack[-1]
                continue
            else:
                stack.pop()
                agregar_pila(celda)
                print(stack)
                print("/--------------------------------------------------------------------------------/")
                x=stack[-1]
    else:
        print("Analisis sintactico completado con exito")
        return 0;
    #Manejo de No Terminales
```

**lexer.input(code):** Inicialice el analizador léxico para que procese el código fuente suministrado.

**tok=lexer.token():** Obtiene el primer token del código fuente usando el analizador

**léxico. x=stack[-1]:** Inicializa la variable x con el primer elemento de la pila (stack).

**El bucle principal while True:** Indica que el análisis continuará hasta que se alcance un estado de aceptación o se detecte un error irrecuperable.

**if x is not None and tok is not None:** Se asegura de que tanto el elemento superior de la pila (x) como el token actual (tok) no sean nulos.

**Manejo de Tokens y Pila:** Verifica el tipo de token actual contra el símbolo de pila superior. En caso de coincidencia, ejecuta la acción y actualiza el estado. De lo contrario, reporta error y activo modo de recuperación.

Manejo de Errores y Recuperación: Al detectar una discrepancia entre el token actual y el símbolo en la cima de la pila, se genera un error y se inicia el proceso de recuperación. Si el elemento superior no es un símbolo, se consulta la tabla para determinar la acción a realizar.

Si no se encuentra en la tabla, se imprime un mensaje de error y se intenta recuperar en el modo de pánico.

En el caso de que tok sea None, se imprime un mensaje indicando que el análisis sintáctico se ha completado con éxito.

```
def barra_de_progreso(duracion, longitud_barra=50):
    for i in range(longitud_barra + 1):
        porcentaje = int((i / longitud_barra) * 100)
        barra = '#' * i + '-' * (longitud_barra - i)
        sys.stdout.write(f"\r[{barra}] {porcentaje}%")
        sys.stdout.flush()
        time.sleep(duracion / longitud_barra)
    print()
```

**def barra_de_progreso(duracion, longitud_barra=50):** Este mecanismo de depuración es fundamental para detener la ejecución del programa en puntos críticos, facilitando la identificación de errores en código extenso.

**for i in range(longitud_barra + 1):** Inicia un bucle que recorre desde 0 hasta longitud_barra, inclusive.

**porcentaje = int((i / longitud_barra) * 100):** Calcular el porcentaje de progreso dividiendo el valor actual del índice i entre la longitud total de la barra

**barra = '#' * i + '-' * (longitud_barra - i):** Construye la representación de la barra de progreso utilizando el carácter '#' para las partes completadas y '-' para las partes restantes.

**sys.stdout.write(f"\r[{barra}] {porcentaje}%"):** Escribe en la salida estándar (stdout) una línea que sobrescribe la línea anterior, mostrando la barra de progreso actual y el porcentaje completado.

**sys.stdout.flush():** Vacía el almacenamiento temporal de salida, asegurando una exhibición inmediata de la información en el monitor.

**time.sleep(duracion / longitud_barra):** Pausa la ejecución del programa durante un breve período, calculado para que la barra de progreso avance gradualmente a lo largo del tiempo total especificado.

**print():** Después de completar la barra de progreso, imprime una línea nueva para que el siguiente contenido en la consola comience en una nueva línea.

### Implementación de sistema de recuperación de errores del parser

```
def panic_mode_recovery(recovery_tokens, tok):
    # Bucles que buscan un token de recuperacion y ajustan la pila.
    while tok is not None and tok.type not in recovery_tokens:
        tok = lexer.token()
        print(f"Buscando {recovery_tokens}")
    while stack and (stack[-1] not in recovery_tokens and stack[-1] in tokens):
        stack.pop()

    if stack and tok is not None:
        barra_de_progreso(1)
        print(f"Recuperación exitosa, próximo token: {tok.type}, próximo en la pila: {stack[-1]}")
        return tok
    else:
        barra_de_progreso(1)
        print("La pila está vacía después de la recuperación del modo pánico.")
        return tok
def search_in_table(no_terminal, terminal):
    for i in range(len(table)):
        if( table[i][0] == no_terminal and table[i][1] == terminal):
            return table[i][2] #retorno la celda
```

La función **panic_mode_recovery** es una implementación de la recuperación en modo pánico para el análisis sintáctico. Aquí hay una descripción de cómo funciona:

**while tok is not None and tok.type not in recovery_tokens:** En este bucle, se busca un token de recuperación (**recovery_tokens**) hasta que se encuentra un token que está en la lista de tokens de recuperación o hasta que tok sea None.

**while stack and (stack[-1] not in recovery_tokens and stack[-1] in tokens):** En esta iteración, se depura la pila descartando elementos hasta que el elemento superior coincida con un token de recuperación o hasta que la pila quede vaciada.

**if stack and tok is not None:** Después de los bucles, se verifica si la pila no está vacía y tok no es None. Si es así, se muestra un mensaje indicando que la recuperación fue exitosa, junto con información sobre el próximo token y el próximo elemento en la pila.

**else:** Si la pila está vacía después de la recuperación del modo pánico, se muestra un mensaje indicando que la pila está vacía.

**return tok:** La función devuelve el token después del intento de recuperación.

**La función search_in_table:** Explora una tabla en busca de una celda que coincida con los criterios especificados por el no terminal y el terminal. En caso de éxito, devuelve el valor asociado a esa celda.

```
def search_in_table(no_terminal, terminal):
    for i in range(len(table)):
        if( table[i][0] == no_terminal and table[i][1] == terminal):
            return table[i][2] #retorno la celda

def agregar_pila(produccion):
    for elemento in reversed(produccion):
        if elemento != 'empty': #la vacía no la inserta
            stack.append(elemento)
```

Las funciones **search_in_table y agregar_pila** están diseñadas para ser utilizadas en un analizador sintáctico, y parecen estar relacionadas con la búsqueda de información en una tabla y la manipulación de una pila durante el proceso de análisis.

**search_in_table(no_terminal, terminal)** se define de la siguiente manera.

**Parámetros:** no_terminal es el símbolo no terminal y terminal es el símbolo terminal.

**Función:** Itera sobre una tabla (que no está definida en el código proporcionado) y busca una entrada que coincida tanto con el símbolo no terminal como con el símbolo terminal.

**Retorna:** Si se encuentra una coincidencia, devuelve el valor de la celda correspondiente en la tabla.

**agregar_pila(produccion):**

**Parámetros:** producción es una lista que representa una producción gramatical.

**Función:** Itera sobre los elementos de la producción (en orden inverso) y los agrega a la pila, excepto si el elemento es 'empty'.

**Acción:** Agrega los elementos de la producción a la pila.

```
def find_expected_tokens(no_terminal):
    expected = []
    for row in table:
        if row[0] == no_terminal and row[2] is not None:
            expected.append(row[1])
    return expected
```

La función **find_expected_tokens** devuelve una lista de símbolos terminales esperados después de un símbolo no terminal dado. Aquí está una descripción detallada:

Parámetro:

**no_terminal**: Es el símbolo no terminal para el cual se desea encontrar los símbolos terminales esperados.

Funcionamiento:

Itera sobre cada fila de la tabla (cuya definición no está incluida en el código proporcionado).

Comprueba si la primera columna de la fila coincide con el símbolo no terminal proporcionado (**row[0] == no_terminal**).

Si hay una coincidencia y la tercera columna (row[2]) no es None, agrega el símbolo terminal correspondiente a la lista de símbolos esperados.

**Retorno:** Devuelve la lista de símbolos terminales esperados después del símbolo no terminal dado.

**Implementación de la tabla de símbolos de las gramáticas**

```python
# Tabla de símbolos
symbol_table = defaultdict(list)

# Insertar en la tabla de símbolos
def symbol_table_insert(name, var_type, line, pos):
    symbol_table[name].append({"type": var_type, "line": line, "pos": pos})

# Mostrar la tabla de símbolos
def symbol_table_print():
    print("Nombre                          Tipo            Linea    Posicion")
    print("---------------------------------------------------------------------")
    for name, entries in symbol_table.items():
        for entry in entries:
            print(f"{name:<35}{entry['type']:10}{entry['line']:10}{entry['pos']:10}")
```

utilizando un diccionario predeterminado (**defaultdict**) de listas. Aquí hay una explicación de cada parte:

**symbol_table = defaultdict(list):**

Crea un diccionario predeterminado (**defaultdict**) llamado **symbol_table** donde los valores por defecto son listas vacías. Esto significa que, si intentas acceder a una clave que aún no existe en el diccionario, se creará automáticamente con una lista vacía como valor.

**symbol_table_insert(name, var_type, line, pos):**

Esta función se utiliza para insertar entradas en la tabla de símbolos.

**Parámetros:**

**name:** Nombre del símbolo.

**var_type:** Tipo del símbolo.

**line:** Número de línea donde se encuentra el símbolo.

**pos:** Posición del símbolo en la línea.

La función anexa un diccionario que encapsula datos sobre el símbolo a la lista asociada con la clave "name" dentro del diccionario "symbol_table". Cada elemento de dicha lista representa una instancia de uso del símbolo.

**symbol_table_print():**

Esta función tiene como propósito generar una representación legible de la tabla de símbolos.

Imprime una cabecera que describe las columnas de la tabla (Nombre, Tipo, Línea, Posición).

Luego, itera a través del diccionario symbol_table, e imprime cada entrada en un formato tabular.

```
# Buscar en la tabla de símbolos
def symbol_table_search(name):
    if name in symbol_table:
        for info in symbol_table[name]:
            print(f"{name} - {info}")
    else:
        print(f"No se encontró '{name}' en la tabla de símbolos.")
```

La función **symbol_table_search** es una función para buscar y mostrar información asociada con un símbolo específico en la tabla de símbolos. Aquí está una explicación de su funcionamiento:

**Parámetro:**

**name:** Nombre del símbolo que se desea buscar en la tabla de símbolos.

**Funcionamiento:**

Verifica si el nombre del símbolo (name) está presente como una clave en el diccionario **symbol_table.**

Si el símbolo está presente, itera a través de las entradas asociadas con ese nombre en la lista (puede haber múltiples entradas para el mismo nombre).

Imprime información asociada con cada entrada, que generalmente incluirá detalles como el tipo de símbolo, la línea y la posición.

**Salida:**

Si el símbolo se encuentra en la tabla de símbolos, se imprimirá la información asociada con todas sus entradas.

Si el símbolo no se encuentra, se imprime un mensaje indicando que no se encontró en la tabla de símbolos.

```
# Borrar de la tabla de símbolos
def symbol_table_delete(name):
    if name in symbol_table:
        del symbol_table[name]
        print(f"'{name}' eliminado de la tabla de símbolos.")
    else:
        print(f"No se pudo eliminar '{name}' porque no se encuentra en la tabla de símbolos.")

symbol_table_print()
```

La función **symbol_table_delete** es una función para eliminar un símbolo y todas sus entradas asociadas de la tabla de símbolos. Aquí está una explicación de su funcionamiento:

Parámetro:

**name:** Nombre del símbolo que se desea eliminar de la tabla de símbolos.

**Funcionamiento:**

Verifica si el nombre del símbolo (name) está presente como una clave en el diccionario **symbol_table.**

Si el símbolo está presente, se utiliza para eliminar la entrada completa asociada con ese nombre de la tabla de símbolos.

Imprime un mensaje indicando que el símbolo fue eliminado.

**Salida:**

En el caso de que el símbolo esté registrado en la tabla de símbolos, se procederá a su eliminación y se emitirá un mensaje de confirmación al usuario. De lo contrario, se informará al usuario que la operación no pudo realizarse debido a la ausencia del símbolo en la tabla.

El fragmento de código final, **symbol_table_print()**, invoca la función **symbol_table_print** para imprimir la tabla de símbolos después de posiblemente haber eliminado un símbolo. Este tipo de operación podría ser útil durante el desarrollo del programa para verificar el estado actual de la tabla de símbolos.

## ESTRUCTURA BÁSICA DEL PARSER_TABLE

Se utiliza para la representación de las reglas gramaticales y la tabla de análisis sintáctico utilizadas en el parser.

**Definición de constantes:**


Se establecen valores numéricos constantes para representar distintos tipos de construcciones lingüísticas, tales como instrucciones, declaraciones de funciones y operaciones. Esta convención facilita la referencia a dichas construcciones en las reglas gramaticales, mejorando así la legibilidad del código.

**Table:**
Una tabla de análisis sintáctico es una estructura de datos bidimensional que representa las posibles derivaciones de una gramática. Cada fila corresponde a una regla de producción, y cada columna a un símbolo terminal del lenguaje. Las celdas de la tabla contienen referencias a otras producciones o a acciones a realizar durante el análisis. La producción principal, encargada de procesar instrucciones a nivel global y declaraciones, constituye el punto de partida del análisis.

```
table = [
    # 0
    [instruction, 'SEMICOLON', None],
    [instruction, 'COMMA', None],
    [instruction, 'WHILE', None],
    [instruction, 'LPAREN', None],
    [instruction, 'RPAREN', None],
    [instruction, 'LBRACE', None],
    [instruction, 'RBRACE', None],
    [instruction, 'READ', None],
    [instruction, 'QUOTES', None],
    [instruction, 'WRITE', None],
    [instruction, 'BOOL', None],
    [instruction, 'LOGIC', None],
    [instruction, 'ASSIGN', None],
    [instruction, 'INT', [func_or_var_dec, instruction]],
    [instruction, 'FLOAT', [func_or_var_dec, instruction]],
    [instruction, 'CHAR', [func_or_var_dec, instruction]],
    [instruction, 'RELATIONAL', None],
    [instruction, 'OP', None],
    [instruction, 'MOD', None],
    [instruction, 'ID', ['ID', vasign_func_call, 'SEMICOLON', instruction]],
    [instruction, 'STRING', None],
    [instruction, 'NUMBER', None],
    [instruction, 'RETURN', None],
    [instruction, 'IF',None ],
    [instruction, 'ELSE', None],
    [instruction, 'EOF', None],
      You, 3 days ago • ajustes parser st …
```

**ESTRUCTURA BÁSICA DEL MAIN**

El propósito principal de la función main() es cargar un archivo, en este caso "code.c", que será la entrada para el parser y luego imprimir información sobre la tabla de símbolos resultante.

Las importaciones son la función parse del archivo parse, el cual analiza el código del archivo, y **symbol_table_print** se encarga de la impresión de la tabla de símbolos.

```python
from lexer import lexer
from parse import parse, symbol_table_print
def main():
    file_path = "code.c"
    try:
        f = open(file_path, 'r')
        code = f.read()
        parse(code)
        symbol_table_print()
    except FileNotFoundError:
        print(f"No se pudo encontrar el archivo: {file_path}")

if __name__ == "__main__":
    main()      You, 3 weeks ago • lexer+main+conn …
```

## ESTRUCTURA BÁSICA DEL LEXER

```
import ply.lex as lex

# Lista de nombres de tokens
tokens = [
    'INCLUDE', 'STRING', 'ID', 'NUMBER', 'QUOTES',
    'READ', 'WRITE',
    'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE',
    'SEMICOLON', 'COMMA', 'ASSIGN',
    'RELATIONAL', 'OP', 'LOGIC',
    'EOF',
    # Palabras reservadas
    'IF', 'ELSE', 'FOR', 'WHILE', 'IF', 'ELSE', 'DO', 'VOID', 'RETURN', 'INT', 'FLOAT', 'CHAR',
]

# Palabras reservadas
reserved = {
    'if': 'IF',
    'else': 'ELSE',
    'for': 'FOR',
    'while': 'WHILE',
    'if': 'IF',
    'else': 'ELSE',
    'do': 'DO',
    'void': 'VOID',
    'return': 'RETURN',
    'int': 'INT', 'float': 'FLOAT', 'char': 'CHAR',
    'scanf' : 'READ',
    'printf':'WRITE',
}

# Reglas para expresiones regulares simples
t_LPAREN = r'\('; t_RPAREN = r'\)'; t_LBRACE = r'\{'; t_RBRACE = r'\}'
```

```
t_SEMICOLON = r';'; t_COMMA = r','; t_ASSIGN = r'='; t_QUOTES = r'\"' ; t_EOF = r'\$'

def t_INCLUDE(t):
    r'\#include[ ]*<[^>]+>'  # Esta parte reconoce el formato #include <nombre>
    t.lexer.lineno += t.value.count('\n')  # Incrementa el número de línea según los saltos de línea en el comentario
    pass  # include ignorados

# Token para cadenas de caracteres
def t_STRING(t):
    r'\"([^\\\n]|(\\.))*\"'
    #print(f"STRING token: {t.value}, Line: {t.lineno}")
    return t

# Reglas más complejas para ID y NUMBER
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')  # Palabras reservadas
    #print(f"ID token: {t.value}, Line: {t.lineno}")
    return t

# Token para comentarios de una línea y multilínea
def t_COMMENT(t):
    r'\/\/.*|\/\*[\s\S]*?\*\/'
    t.lexer.lineno += t.value.count('\n')  # Incrementa el número de línea según los saltos de línea en el comentario
    pass  # Los comentarios son ignorados
```

```
# Token para números (enteros y flotantes)
def t_NUMBER(t):
    r'\d+(\.\d+)?'
    t.value = float(t.value) if '.' in t.value else int(t.value)
    #print(f"NUMBER token: {t.value}, Line: {t.lineno}")
    return t

# Token para operadores
def t_OP(t):
    r'(\+)|(\-)|(\*)|(\/)|(\%)'
    return t

# Token para operadores logicos
def t_LOGIC(t):
    r'(\>=)|(\<=)|(\==)|(\!=)|(\<)|(\>)'
    return t

# Token para relaciones && ||
def t_RELATIONAL(t):
    r'(\&{2})|(\|{2})'
    return t
```

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
# Caracteres ignorados
t_ignore = ' \t'

def t_error(t):
    #print(f"Illegal character {t.value[0]}")
    t.lexer.skip(1)

lexer = lex.lex(debug=1)
```

## Estruraracion de Lexer para el parcer a utilizar

Tokens Definidos: Palabras clave del lenguaje como IF, ELSE, FOR, WHILE, DO, VOID, RETURN, INT, FLOAT, CHAR, READ, WRITE.

Operadores y delimitadores como LPAREN, RPAREN, LBRACE, RBRACE, SEMICOLON, COMMA, ASSIGN, RELATIONAL, OP, LOGIC.

Tokens adicionales como INCLUDE, STRING, ID, NUMBER, QUOTES, EOF.

Expresiones Regulares: Se utilizan expresiones regulares para definir el patrón de los tokens. Por ejemplo, t_STRING para cadenas, t_ID para identificadores, t_NUMBER para números, etc.

Manejo de Palabras Reservadas: Se define un diccionario llamado reserved que asigna palabras clave del lenguaje a sus respectivos tokens.

Manejo de Comentarios: Se proporciona una regla para manejar comentarios de una línea (//) y comentarios multilínea (/* ... */), pero se ignoran (no se generan tokens).

Manejo de Errores: Se define una función t_error para manejar caracteres no válidos, simplemente saltándolos.

Números y Operadores: Se manejan números enteros y flotantes, así como operadores aritméticos y lógicos.

Lexer Object: Se crea un objeto lexer al final del código utilizando lex.lex().

# RESULTADOS

**Resultado de la Compilación:**

Análisis sintáctico exitoso.

No se han detectado errores sintácticos.

El análisis sintáctico es ahora completamente exitoso.

Nuestro compilador C ha superado con éxito las pruebas de análisis sintáctico, demostrando una precisa interpretación del código fuente y un eficaz manejo de errores.

Uno de los aspectos destacados de nuestro logro radica en la eficiencia del analizador de errores. Este componente es crucial para proporcionar notificaciones precisas a los desarrolladores, facilitando la identificación y corrección eficiente de cualquier error presente en el código fuente analizado.

Este logro consolida la solidez de nuestro analizador sintáctico y de errores, impulsándonos a continuar desarrollando un compilador C completo y robusto.

**Errores obtenidos durante el proceso de creación del analizador lexicográfico**

Errores de Reconocimiento de Caracteres

Reconocimiento de caracteres incorrecto: El analizador sintáctico está manejando los caracteres como cadenas de tamaño 1 en lugar de caracteres individuales. Por ejemplo, se está tratando "a" en lugar de 'a'.

Errores de Reglas de Producción

Operaciones aritméticas no permitidas en ciertas reglas: Hay operaciones aritméticas que no están incluidas en las reglas de producción de condicionales de if, else, y while. Esto provoca que el analizador entre en modo de recuperación de errores cuando se encuentran tales operaciones en el código a analizar.

Nota: Por motivo de decisión no se empleó estas para los resultados de nuestro sistema a compilar.

Errores de Token EOF

Falta de manejo correcto del token EOF: El analizador no maneja un token para el final del archivo de manera correcta, ya que se lo salta cuando llega a este. Como resultado, el parser solo finaliza cuando detecta el fin del archivo, sin incluir el token ‘EOF’ en la tabla de símbolos.