**UNIVERSIDAD DE GUADALAJARA**

**CENTRO UNIVERSITARIO DE TONALÁ**

**Logotipo

Descripción generada automáticamente**

**Materia: Traductores de Lenguaje.**

**Tema: Análisis sintáctico de variables y constantes.**

**Integrantes:**

// Celestino Martinez Cristopher

// Ramirez Rivas Luis Fernando

// Rivas Tejeda Hector Manuel

**Fecha:** 03 de febrero del 2025

**Lenguaje asignado C**

#include <stdio.h>

// Celestino Martinez Cristopher

// Ramirez Rivas Luis Fernando

// Rivas Tejeda Hector Manuel

int main() {

    // Imprimir nombres completos en orden alfabético

    printf("TRADUCTORES DE LENGUAJES:\n");

    printf("Actividad 2\n");

    int entero = 1;

    float flotante = 2.5;

    bool booleano = true;

    String nombre = "nombre";

    int entero2 = 123x;  // Identificador no válido

    float flotante = y;   // Tipo no válido

    bool booleano = true // Falta punto y coma

    String cadena = 123;   // Identificador no válido

    a = 10;  // Asignación de valor a 'a'

    if (a > b) { // Comparación de 'a' y 'b'

        printf("%s\n", saludo); // Imprimir el saludo

    }

    // Imprimir nombres completos en orden alfabético al final

    printf("\nIntegrantes del equipo:\n");

    printf("1. Celestino Martinez Cristopher\n");

    printf("2. Ramirez Rivas Luis Fernando\n");

    printf("3. Rivas Tejeda Hector Manuel\n");

    return 0;

}

Este código está diseñado para simular parte de una actividad que involucra la implementación de un lenguaje de programación, donde se deben declarar variables de diferentes tipos (entero, flotante, booleano y cadena) y luego verificar la sintaxis de las declaraciones de las variables y la gestión de errores en caso de que haya problemas.

Aquí te explico la relación del código con los puntos de la actividad:

**1. Definición de la Gramática:**

El código incluye ejemplos de la declaración de variables con distintos tipos, como int, float, bool, y String. La gramática que has definido para la actividad incluiría las reglas para declarar estos tipos de variables correctamente:

* declaracion\_variable -> tipo IDENTIFICADOR ';'
* tipo -> 'entero' | 'flotante' | 'booleano' | 'cadena'
* IDENTIFICADOR -> [letra | '\_'][letra | digito | '\_']\*

Por ejemplo, las declaraciones int entero = 1; y float flotante = 2.5; son correctas según las reglas de gramática, pero las declaraciones como int entero2 = 123x; y float flotante = y; no lo son, debido a que los identificadores y los valores no son válidos según la gramática que defines.

**2. Implementación del Analizador Sintáctico:**

En la actividad se menciona que deben implementar un analizador sintáctico que verifique las declaraciones de variables. Este código contiene algunas declaraciones válidas y algunas con errores que el analizador sintáctico debería identificar:

* int entero = 1; (válida)
* float flotante = 2.5; (válida)
* int entero2 = 123x; (error, identificador no válido)
* float flotante = y; (error, tipo no válido)
* bool booleano = true; (válida, pero en C se usa bool solo si se incluye stdbool.h, no es un tipo por defecto)
* String cadena = 123; (error, tipo incorrecto y asignación incorrecta)

El analizador sintáctico debe verificar que:

* Los tipos sean válidos (entero, flotante, booleano, cadena).
* Los identificadores no empiecen con números.
* Falte el punto y coma (;).

**3. Gestión de Errores Sintácticos:**

El código presenta varios errores que deberían ser reportados por el manejador de errores:

* int entero2 = 123x; -> Error: "Identificador no válido".
* float flotante = y; -> Error: "Tipo no válido".
* bool booleano = true; -> Error de compilación, pues el tipo bool requiere stdbool.h en C.
* String cadena = 123; -> Error: "Identificador no válido" (se espera un string y no un número).

El manejador de errores debe capturar estos errores y mostrar mensajes de error detallados, como los mencionados en los ejemplos de la actividad:

* "Error: Identificador inválido en la línea X, columna Y."
* "Error: Tipo no válido en la línea X, columna Y."

**Main**

from lexer import lexer

def main():

    # Leer el archivo de prueba

    with open('C:/Users/user/Desktop/traductores/Lex-main/tests/example.c', 'r') as f:

        code = f.read()

    # Alimentar el lexer

    lexer.input(code)

    # Mostrar tokens generados

    print("Tokens encontrados:")

    while True:

        tok = lexer.token()

        if not tok:

            break

        print(tok)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Explicación del Código**

1. **Importación del Lexer**:

from lexer import lexer

Aquí se importa un objeto lexer desde un módulo llamado lexer. Este lexer es probablemente una instancia de un analizador léxico generado por una herramienta como **PLY** (una implementación de Lex y Yacc en Python).

1. **Función main**:  
   La función main es el punto de entrada del programa. Aquí se realiza lo siguiente:
   * **Lectura del archivo de código fuente**:

with open('C:/Users/user/Desktop/traductores/Lex-main/tests/example.c', 'r') as f:

code = f.read()

Se abre el archivo example.c en modo lectura ('r') y se lee todo su contenido en la variable code.

* + **Alimentar el lexer**:

python

Copy

lexer.input(code)

El contenido del archivo (code) se pasa al lexer para que lo procese.

* + **Mostrar los tokens generados**:

print("Tokens encontrados:")

while True:

tok = lexer.token()

if not tok:

break

print(tok)

Se entra en un bucle infinito donde se llama repetidamente a lexer.token() para obtener los tokens generados por el lexer. Cuando no hay más tokens (tok es None), el bucle se detiene. Cada token se imprime en la consola.

1. **Ejecución del script**:

python

Copy

if \_\_name\_\_ == '\_\_main\_\_':

main()

Este bloque asegura que la función main se ejecute solo cuando el script se ejecuta directamente (no cuando se importa como módulo).

**¿Qué hace este código?**

Este script toma un archivo de código fuente en C (example.c) y lo procesa con un lexer para extraer los tokens. Los tokens son las unidades básicas del lenguaje, como palabras clave, identificadores, números, operadores, etc. El lexer identifica estos tokens según las reglas definidas en su implementación.

**Lexer**

import ply.lex as lex

# Definir los tokens

tokens = (

    'TYPE', 'IDENTIFIER', 'INT\_LITERAL', 'FLOAT\_LITERAL', 'STRING\_LITERAL',

    'BOOL\_LITERAL', 'CONST', 'ASSIGN', 'SEMICOLON'

)

# Definir las expresiones regulares para los tokens

t\_TYPE = r'int|float|bool|char'

t\_CONST = r'const'

t\_ASSIGN = r'='

t\_SEMICOLON = r';'

t\_INT\_LITERAL = r'\d+'

t\_FLOAT\_LITERAL = r'\d+\.\d+'

t\_STRING\_LITERAL = r'\"[^\"]\*\"'

t\_BOOL\_LITERAL = r'true|false'

# Para los identificadores (solo letras y números, pero no puede empezar con un número)

def t\_IDENTIFIER(t):

    r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

    if t.value in ('int', 'float', 'bool', 'char', 'const', 'true', 'false'):

        t.type = 'TYPE'  # Cambiar tipo si se encuentra una palabra reservada

    return t

# Ignorar espacios y tabulaciones

t\_ignore = ' \t'

# Manejo de saltos de línea

def t\_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)

# Manejo de errores

def t\_error(t):

    print(f"Error: Caracter ilegal '{t.value[0]}' en línea {t.lineno}")

    t.lexer.skip(1)

# Crear el lexer

lexer = lex.lex()

# Para los identificadores (solo letras y números, pero no puede empezar con un número)

def t\_IDENTIFIER(t):

    r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

    if t.value in ('int', 'float', 'bool', 'char', 'const', 'true', 'false'):

        t.type = 'TYPE'  # Cambiar tipo si se encuentra una palabra reservada

    return t

# Ignorar espacios y tabulaciones

t\_ignore = ' \t'

# Manejo de saltos de línea

def t\_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)

# Manejo de errores

def t\_error(t):

    print(f"Error: Caracter ilegal '{t.value[0]}' en línea {t.lineno}")

    t.lexer.skip(1)

# Crear el lexer

lexer = lex.lex()

Este código utiliza la librería ply (Python Lex-Yacc) para implementar un lexer (analizador léxico) en Python. El propósito de este lexer es analizar una cadena de texto y dividirla en tokens según las reglas definidas. Aquí está la explicación de cómo funciona y cómo se relaciona con la actividad que mencionaste previamente:

**1. Definición de Tokens:**

El lexer está configurado para reconocer los siguientes tokens:

* **'TYPE'**: Representa los tipos de datos (int, float, bool, char).
* **'IDENTIFIER'**: Representa los identificadores (nombres de variables o constantes).
* **'INT\_LITERAL'**: Representa literales enteros (números sin decimales).
* **'FLOAT\_LITERAL'**: Representa literales flotantes (números con decimales).
* **'STRING\_LITERAL'**: Representa literales de cadenas (secuencia de caracteres entre comillas).
* **'BOOL\_LITERAL'**: Representa valores booleanos (true o false).
* **'CONST'**: Representa la palabra clave const para declarar constantes.
* **'ASSIGN'**: Representa el operador de asignación =.
* **'SEMICOLON'**: Representa el punto y coma ; al final de las declaraciones.

**2. Expresiones Regulares para los Tokens:**

Cada token tiene una expresión regular asociada que define su formato:

* t\_TYPE: Coincide con los tipos de datos int, float, bool, char.
* t\_CONST: Coincide con la palabra clave const.
* t\_ASSIGN: Coincide con el signo de igual =.
* t\_SEMICOLON: Coincide con el punto y coma ;.
* t\_INT\_LITERAL: Coincide con secuencias de dígitos (números enteros).
* t\_FLOAT\_LITERAL: Coincide con números con un punto decimal (números flotantes).
* t\_STRING\_LITERAL: Coincide con una cadena entre comillas dobles.
* t\_BOOL\_LITERAL: Coincide con las palabras clave true o false.

**3. Reglas para los Identificadores:**

En la función t\_IDENTIFIER, se especifica que los identificadores deben comenzar con una letra o guion bajo ([a-zA-Z\_]), seguido de letras, números o guiones bajos. Además, la función verifica si el identificador es una palabra reservada como int, float, bool, char, const, true, o false, en cuyo caso el tipo del token se cambia a TYPE.

**4. Ignorar Espacios y Tabulaciones:**

Con t\_ignore = ' \t', se indica que los espacios y las tabulaciones no deben ser procesados como tokens, simplemente se omiten.

**5. Manejo de Saltos de Línea:**

La función t\_newline actualiza el número de línea cuando se encuentran saltos de línea (\n), lo que es útil para mostrar el número de línea en los mensajes de error.

**6. Manejo de Errores:**

En la función t\_error, se captura cualquier carácter que no coincida con los tokens definidos. Si se encuentra un carácter no válido, se imprime un mensaje de error indicando el carácter ilegal y la línea en la que ocurrió.

**7. Creación del Lexer:**

Finalmente, el lexer se crea mediante lexer = lex.lex(), lo que genera el lexer basado en las reglas definidas.

**Parser**

import ply.yacc as yacc

from lexer import tokens

valid\_types = ['int', 'float', 'bool', 'char']  # Tipos válidos

# Declaración de una variable o constante

def p\_declaration(p):

    '''

    declaration : variable\_decl

                | constant\_decl

    '''

    pass

# Declaración de una variable

def p\_variable\_decl(p):

    '''

    variable\_decl : TYPE IDENTIFIER SEMICOLON

                  | TYPE IDENTIFIER ASSIGN literal SEMICOLON

    '''

    tipo = p[1]  # El tipo de la variable (ej. "float", "bool", etc.)

    nombre = p[2]  # El nombre de la variable

    linea = p.lineno(1)  # Línea donde ocurre

    # 🛑 Error 1: Comprobar si el tipo es válido

    if tipo not in valid\_types:

        print(f"Error: Tipo no válido '{tipo}' en línea {linea}. Los tipos válidos son 'int', 'float', 'bool', 'char'.")

        return

    # 🛑 Error 2: Comprobar si el identificador es válido

    if not nombre.isidentifier():  # Verifica si el identificador es válido

        print(f"Error: Identificador no válido '{nombre}' en línea {linea}.")

        return

    # 🛑 Error 3: Comprobar la asignación y si el valor es un literal

    if len(p) > 4:  # Si hay una asignación

        valor\_asignado = p[4]

        if not isinstance(valor\_asignado, str):  # Solo se aceptan literales

            print(f"Error: Asignación inválida a '{nombre}' en línea {linea}. Solo se pueden asignar valores literales.")

            return

    print(f"Declaración válida de variable en línea {linea}")

# Declaración de una constante

def p\_constant\_decl(p):

    '''

    constant\_decl : CONST TYPE IDENTIFIER ASSIGN literal SEMICOLON

    '''

    tipo = p[2]  # El tipo de la constante

    nombre = p[3]  # El nombre de la constante

    linea = p.lineno(1)

    # 🛑 Error 1: Comprobar si el tipo es válido

    if tipo not in valid\_types:

        print(f"Error: Tipo no válido '{tipo}' en línea {linea}. Los tipos válidos son 'int', 'float', 'bool', 'char'.")

        return

    # 🛑 Error 2: Comprobar si el identificador es válido

    if not nombre.isidentifier():

        print(f"Error: Identificador no válido '{nombre}' en línea {linea}.")

        return

    print(f"Declaración válida de constante en línea {linea}")

# Literal (puede ser int, float, string, etc.)

def p\_literal(p):

    '''

    literal : INT\_LITERAL

            | FLOAT\_LITERAL

            | STRING\_LITERAL

            | BOOL\_LITERAL

    '''

    pass

# Error de sintaxis

def p\_error(p):

    if p:

        # Verificar errores específicos relacionados con la declaración

        if p.type == 'IDENTIFIER':

            print(f"Error: Identificador no válido '{p.value}' en línea {p.lineno}")

        elif p.type == 'TYPE' and p.value not in valid\_types:

            print(f"Error: Tipo no válido '{p.value}' en línea {p.lineno}. Los tipos válidos son 'int', 'float', 'bool', 'char'.")

        elif p.type == 'SEMICOLON':

            print(f"Error: Falta ';' al final de la declaración en línea {p.lineno}")

        else:

            print(f"Error de sintaxis en '{p.value}' en línea {p.lineno}")

    else:

        print("Error de sintaxis al final del archivo")

parser = yacc.yacc()

# Función para analizar código

def analizar\_codigo(codigo):

    parser.parse(codigo)

    # 🛑 Error 3: Comprobar la asignación y si el valor es un literal

    if len(p) > 4:  # Si hay una asignación

        valor\_asignado = p[4]

        if not isinstance(valor\_asignado, str):  # Solo se aceptan literales

            print(f"Error: Asignación inválida a '{nombre}' en línea {linea}. Solo se pueden asignar valores literales.")

            return

    print(f"Declaración válida de variable en línea {linea}")

# Declaración de una constante

def p\_constant\_decl(p):

    '''

    constant\_decl : CONST TYPE IDENTIFIER ASSIGN literal SEMICOLON

    '''

    tipo = p[2]  # El tipo de la constante

    nombre = p[3]  # El nombre de la constante

    linea = p.lineno(1)

    # 🛑 Error 1: Comprobar si el tipo es válido

    if tipo not in valid\_types:

        print(f"Error: Tipo no válido '{tipo}' en línea {linea}. Los tipos válidos son 'int', 'float', 'bool', 'char'.")

        return

    # 🛑 Error 2: Comprobar si el identificador es válido

    if not nombre.isidentifier():

        print(f"Error: Identificador no válido '{nombre}' en línea {linea}.")

        return

    print(f"Declaración válida de constante en línea {linea}")

# Literal (puede ser int, float, string, etc.)

def p\_literal(p):

    '''

    literal : INT\_LITERAL

            | FLOAT\_LITERAL

            | STRING\_LITERAL

            | BOOL\_LITERAL

    '''

    pass

# Error de sintaxis

def p\_error(p):

    if p:

        # Verificar errores específicos relacionados con la declaración

        if p.type == 'IDENTIFIER':

            print(f"Error: Identificador no válido '{p.value}' en línea {p.lineno}")

        elif p.type == 'TYPE' and p.value not in valid\_types:

            print(f"Error: Tipo no válido '{p.value}' en línea {p.lineno}. Los tipos válidos son 'int', 'float', 'bool', 'char'.")

        elif p.type == 'SEMICOLON':

            print(f"Error: Falta ';' al final de la declaración en línea {p.lineno}")

        else:

            print(f"Error de sintaxis en '{p.value}' en línea {p.lineno}")

    else:

        print("Error de sintaxis al final del archivo")

parser = yacc.yacc()

# Función para analizar código

def analizar\_codigo(codigo):

    parser.parse(codigo)

Este código implementa un analizador sintáctico (parser) utilizando la librería ply.yacc en Python, que es complementaria a la librería ply.lex que has utilizado anteriormente para el lexer. Este parser toma los tokens generados por el lexer y verifica si las declaraciones de variables y constantes son correctas según las reglas que has definido.

Aquí está el análisis y los puntos importantes de este código:

**1. Definición de Tipos Válidos:**

Se definen los tipos válidos (int, float, bool, char) en la lista valid\_types. Estos se utilizan para validar las declaraciones de variables y constantes.

**2. Declaración de Variables y Constantes:**

El parser tiene una regla de producción para manejar declaraciones de variables y constantes. Ambas reglas pasan por una validación adicional para asegurar que la sintaxis sea correcta y que los tipos y identificadores sean válidos.

**- Declaración de una variable:**

Se pueden declarar variables de dos formas:

* TYPE IDENTIFIER ;
* TYPE IDENTIFIER = literal ;

Para cada caso, el parser valida:

* **Tipo válido**: Si el tipo de la variable no es uno de los tipos válidos, muestra un error.
* **Identificador válido**: Si el identificador (nombre de la variable) no es un identificador válido (es decir, no sigue las reglas para los nombres de variables en Python), muestra un error.
* **Asignación válida**: Si hay una asignación (=), verifica que el valor asignado sea un literal (de tipo INT\_LITERAL, FLOAT\_LITERAL, STRING\_LITERAL, o BOOL\_LITERAL).

**- Declaración de una constante:**

Las constantes se deben declarar como CONST TYPE IDENTIFIER = literal ;. El parser valida lo mismo que en el caso de las variables, pero también verifica que la palabra clave const esté presente.

**3. Reglas de Literales:**

El parser tiene una regla p\_literal que maneja los literales. Aunque no está implementada completamente, esta regla debería permitir que los literales de tipo entero, flotante, cadena y booleano se reconozcan correctamente en las declaraciones.

**4. Manejo de Errores:**

La función p\_error captura los errores de sintaxis:

* Si el token actual es un IDENTIFIER, verifica si es válido.
* Si el token es un TYPE y no es un tipo válido, se genera un error.
* Si falta un SEMICOLON al final de la declaración, también se muestra un error.
* En cualquier otro caso, se genera un mensaje de error genérico con el token que causó el error.

**5. Creación del Parser:**

El parser se crea con yacc.yacc().

**6. Función analizar\_codigo:**

La función analizar\_codigo recibe el código como una cadena y lo pasa al parser para su análisis.

**7. Errores Específicos y Validación:**

Los errores específicos que se manejan son:

* **Error de tipo inválido**: Si el tipo de la declaración no es uno de los tipos válidos.
* **Error de identificador no válido**: Si el identificador no es un nombre de variable válido.
* **Error de asignación inválida**: Si se intenta asignar algo que no es un literal.

**Salida**Tokens encontrados:

Error: Caracter ilegal '<' en línea 1

Error: Caracter ilegal '.' en línea 1

Error: Caracter ilegal '>' en línea 1

Error: Caracter ilegal '/' en línea 3

Error: Caracter ilegal '/' en línea 3

Error: Caracter ilegal '/' en línea 4

Error: Caracter ilegal '/' en línea 4

Error: Caracter ilegal '/' en línea 5

Error: Caracter ilegal '/' en línea 5

Error: Caracter ilegal '(' en línea 7

Error: Caracter ilegal ')' en línea 7

Error: Caracter ilegal '{' en línea 7

Error: Caracter ilegal '/' en línea 8

Error: Caracter ilegal '/' en línea 8

Error: Caracter ilegal 'Ã' en línea 8

Error: Caracter ilegal '©' en línea 8

Error: Caracter ilegal '(' en línea 9

Error: Caracter ilegal ')' en línea 9

Error: Caracter ilegal '(' en línea 10

Error: Caracter ilegal ')' en línea 10

Error: Asignación inválida a 'entero' en línea 12. Solo se pueden asignar valores literales.

Error de sintaxis en 'float' en línea 13

Error: Tipo no válido 'true' en línea 14. Los tipos válidos son 'int', 'float', 'bool', 'char'.

Error: Identificador no válido 'x' en línea 16

Error: Caracter ilegal '/' en línea 16

Error: Caracter ilegal '/' en línea 16

Error: Caracter ilegal 'Ã' en línea 16

Error: Caracter ilegal '¡' en línea 16

Error: Identificador no válido 'y' en línea 17

Error: Caracter ilegal '/' en línea 17

Error: Caracter ilegal '/' en línea 17

Error: Caracter ilegal 'Ã' en línea 17

Error: Caracter ilegal '¡' en línea 17

Error: Tipo no válido 'true' en línea 18. Los tipos válidos son 'int', 'float', 'bool', 'char'. **Github**

[cristopherstrange25/Tokkens-y-Exp.-Reguares: Actividad 2: Definición de tokens y expresiones regulares.](https://github.com/cristopherstrange25/Tokkens-y-Exp.-Reguares/tree/main)

**Video**

<https://youtu.be/Zw6Ms9GDgak?si=nfkU-OzzXlQmzKFO>