Centro Universitario de Ciencias Exactas e Ingeniería

TRADUCTORES DE LENGUAJES ll

PROYECTO FINAL PT 1

JULIO ESTEBAN VALDES LOPEZ

Juan Pablo Hernández Orozco

219294285

Objetivo:

El propósito de esta práctica es diseñar e implementar un analizador sintáctico descendente recursivo para un lenguaje muy sencillo de declaraciones y expresiones aritméticas. El sistema debe:

- Leer un archivo fuente plano con comentarios, declaraciones y expresiones.
- Tokenizar el contenido (fase léxica) identificando palabras reservadas, identificadores, literales y símbolos.
- Aplicar una gramática libre de recursión por la izquierda que respete precedencia y asociatividad.
- Chequeo semántico básico: permitir declaración de variables con y sin inicialización, y detectar usos de variables no declaradas.
- Reportar si el programa es válido o indicar con claridad el tipo de error (léxico, sintáctico o semántico).
- Ofrecer una interfaz gráfica (Tkinter) para cargar el archivo, mostrar su contenido y ver el resultado del análisis.

(implementar un analizador sintáctico descendente de la gramática del lenguaje de programación corregido en la práctica 3.
El programa recibe un archivo fuente de entrada. De salida solo nos va indicar si el programa fuente es válido de acuerdo a la gramática ó, en caso contrario mostrar el mensaje de error correspondiente.)

Introducción:

En el recorrido de construcción de compiladores o intérpretes, el **parser** (analizador sintáctico) es responsable de verificar que la secuencia de tokens producida por el léxico forme oraciones válidas según la gramática del lenguaje. Un método sencillo y muy didáctico para lenguajes simples es el **parser descendente recursivo**, que implementa cada regla de la gramática como un método que “consume” tokens.

Sin embargo, las gramáticas que aparecen de forma natural suelen incluir **recursión por la izquierda**, lo que provoca llamadas infinitas en un parser recursivo. Además, para un lenguaje elemental de expresiones, hay que manejar correctamente la **precedencia** (multiplicación/división > suma/resta) y la **asociatividad** (por la izquierda, generalmente).

En esta práctica, se diseñó una gramática transformada que:

- Elimina recursión por la izquierda
- Hace explícita la precedencia y asociatividad
- Permite extensiones futuras (como la incorporación de if, while, etc.)

Combinado con un **lexer** capaz de saltar comentarios (# …), reconocer palabras reservadas (var, if, …) y llevar una **tabla de símbolos** para validar declaraciones de variables, el sistema final no sólo comprueba la sintaxis, sino también un nivel inicial de semántica. Todo ello se integra en una pequeña aplicación gráfica construida con Tkinter, lo que facilita la interacción al cargar archivos de prueba y visualizar errores con mensajes precisos.

Desarrollo:

Explicación del problema

El enunciado propone un lenguaje muy limitado, con sólo dos tipos de sentencias:

1. **Declaración de variable** — opcionalmente con inicialización (var x; ó var x = 5;).
2. **Asignación** (x = expr;).

Y una regla para expresiones aritméticas con operadores +, -, *, /, literales enteros y uso de paréntesis. Además, se deben ignorar **comentarios** que comienzan con # hasta el final de línea.

Los principales retos son:

Recursión por la izquierda: gramáticas como

expr → expr PLUS term

| expr MINUS term

| term

producen bucles infinitos en un parser recursivo.

- Ambigüedad: sin reglas de precedencia, a + b * c podría interpretarse mal.
- Chequeo semántico: al asignar o usar una variable, debemos verificar que haya sido “declarada” antes. En el ejemplo c = i + a;, si i no existe en la tabla de símbolos, debe lanzarse un error.

- Manejo de tokens especiales: diferenciar var como palabra reservada de un identificador común, y saltar correctamente comentarios y espacios.
- Experiencia de usuario: al usar una GUI, los errores deben mostrarse de forma clara y no técnica, para que cualquier alumno entienda si falló la sintaxis o la semántica.

Explicación de la solución

La arquitectura propuesta se divide en tres capas:

1. **Lexer (análisis léxico)**
    - Lee carácter a carácter del archivo fuente.
    - **Ignora** espacios, tabulaciones y saltos de línea.
    - Cuando ve #, avanza hasta el fin de línea para saltarse comentarios.
    - Agrupa secuencias de letras y dígitos en **tokens**:
        - Consulta un diccionario de **palabras reservadas** (var, if, else, …) para asignarles un tipo especial.
        - Cualquier otra secuencia alfanumérica se considera un **identificador**.
    - Reconoce literales enteros completos.
    - Genera tokens para operadores y símbolos (+, -, *, /, (,), =, ;).
    - Al llegar al final, emite un token EOF.
2. **Parser (análisis sintáctico y chequeo semántico)**
    - Se implementa como un parser **descendente recursivo**, donde cada regla gramatical es un método.
    - **Gramática transformada** para evitar recursión por la izquierda:
        - expr → term expr_rest
        - expr_rest → (PLUS|MINUS) term expr_rest | ε
        - term → factor term_rest
        - term_rest → (TIMES|DIVIDE) factor term_rest | ε
    - El parser mantiene una **tabla de símbolos** (un set de nombres) para:
        - **Declarar** variables al procesar var x; o var x = expr;.

        - **Verificar** antes de usar un identificador en una expresión o asignación, disparando un error semántico si no existe.
    - Cada sentencia sigue la regla general:
        - Si comienza con var: es una **declaración** (con o sin inicializador).
        - Si comienza con un identificador: es una **asignación**.
    - Tras procesar todas las sentencias, exige el token EOF para confirmar que no quedó texto sobrante.

3. **Interfaz gráfica (Tkinter)**
    - Ventana principal con:
        - **Botón** para seleccionar y cargar un archivo .txt.
        - **Área de texto** que muestra el contenido del archivo.
        - **Botón** “Analizar programa” que invoca al parseador.
        - **Etiqueta** que muestra el resultado: “Programa válido.” o “Error: descripción precisa”.
    - Los errores léxicos, sintácticos o semánticos se capturan como excepciones y su mensaje se presenta al usuario.

Este enfoque modular permite futuras extensiones — agregar más palabras reservadas, nuevas estructuras de control, expresiones booleanas, etc. — sin alterar la interfaz ni el flujo general.

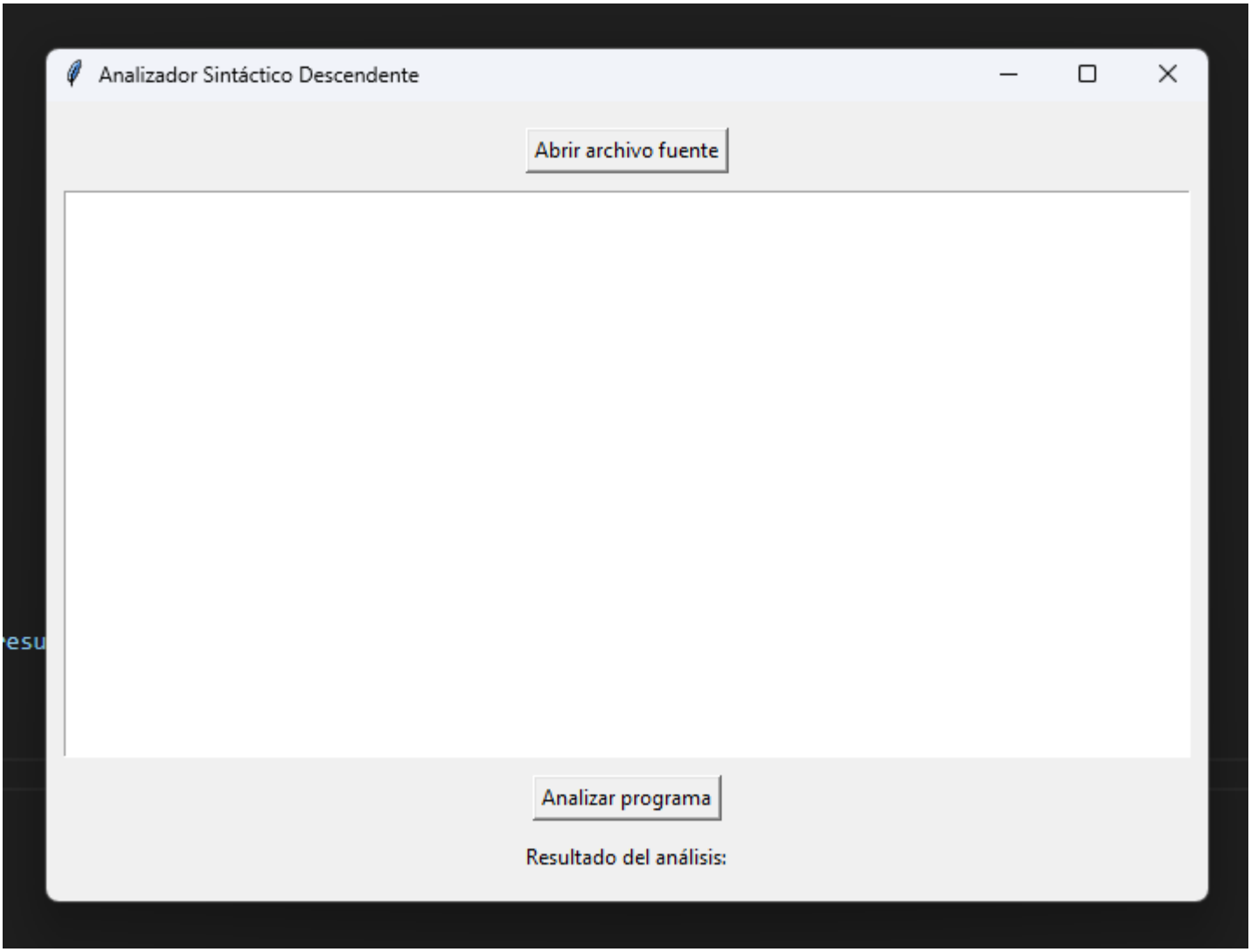

Tabla de casos de prueba

Caso	Código de prueba	Resultado esperado	Descripción
1	`var x;`	Programa válido.	Declaración simple sin inicialización.
2	`var x = 5;`	Programa válido.	Declaración con inicializador.
3	`var x; x = 10;`	Programa válido.	Asignación a variable previamente declarada.
4	`x = 5;`	Error: Variable 'x' no declarada	Uso de variable sin declaración previa.
5	`var y = (3 + 2) * 4;`	Programa válido.	Expresión con paréntesis y operadores de precedencia.

6	`var x`	Error: Se esperaba token SEMICOLON	Falta punto y coma al final de la declaración.
7	`var z = 10 / (2 - 2);`	Programa válido (aunque semántico: división por cero no detectada)	División por cero en tiempo de ejecución; sintácticamente válido.
8	`# Esto es un comentario\nvar a;`	Programa válido.	Comentario ignorado, declaración válida en segunda línea.
9	`var a; var b = a + 2;`	Programa válido.	Uso de variable en inicialización de otra.
10	`var n = 1 + 2 * (3 - 4) ;`	Programa válido.	Combinación de sumas, restas y multiplicaciones en una misma línea.
11	`var x = -5;`	Error: Se esperaba '(', número o identificador	Uso de un operador unario no soportado.
12	`var a = (1 + (2 * (3 + 4)));`	Programa válido.	Expresión con paréntesis anidados múltiples niveles.
13	`var a = 007;`	Programa válido.	Entero con ceros a la izquierda, interpretado como 7.
14	`var a = ;`	Error: Se esperaba '(', número o identificador	Falta expresión después del '='.
15		Error: Se esperaba '(', número o identificador	Falta operando después del operador de multiplicación.
16	`var a = 5 5;`	Error: Se esperaba token PLUS o MINUS	Falta operador entre dos literales numéricos.
17	`var _temp = 10;`	Programa válido.	Identificador que comienza con guion bajo.
18	`var a = b;`	Error: Variable 'b' no declarada	Uso de variable no declarada en inicializador.
19	`var a; var a;`	Programa válido.	Redeclaración de variable ignorada (tabla de símbolos usa conjunto).
20	`# comentario sin newline`	Programa válido.	Comentario hasta EOF, sin sentencias, válido sintácticamente.
21	`var a = (1 + 2;`	Error: Se esperaba token RPAREN	Paréntesis de cierre faltante en la expresión.
22	`var x = 2147483647;`	Programa válido.	Número entero grande dentro de límites de Python.
23	`var x = 3 / 0;`	Programa válido (aunque semántico: división por cero no detectada)	División por cero en tiempo de ejecución; sintácticamente válido.

24	`var x = ;`	Error: Se esperaba '(', número o identificador	Falta expresión tras '=' (caso análogo al caso 14).
25	`var a; a = ();`	Error: Se esperaba '(', número o identificador	Paréntesis vacío sin contenido válido.

Conclusión:

La implementación de un **parser descendente recursivo** para un lenguaje de expresiones aritméticas y declaraciones simples ilustra de manera efectiva conceptos clave de compilación: eliminación de recursión por la izquierda, manejo de precedencia y asociatividad, análisis léxico, y una incipiente comprobación semántica mediante una tabla de símbolos.

La solución modular, dividida en un **Lexer** robusto y un **Parser** con métodos que reflejan las reglas gramaticales, facilita la comprensión y futuras extensiones. El uso de **Tkinter** aporta una interfaz básica que permite a cualquier usuario cargar archivos de prueba y visualizar mensajes de error claros, acelerando ciclos de validación y depuración.

Si bien el parser ya cubre →

- Comentarios (# …)
- Declaraciones con y sin inicialización (var x;, var x = expr;)
- Asignaciones normales (x = expr;)
- Expresiones con prioridad y asociatividad

quedan abiertas oportunidades para enriquecer la práctica:

1. **Nuevos tipos de sentencias**: condicionales if … else, bucles while, bloques { … }.
2. **Chequeos semánticos avanzados**: detección de división por cero, tipos de datos diferentes a enteros, análisis de flujos de control, detección de variables definidas pero no usadas.
3. **Generación de un árbol de sintaxis abstracta (AST)** para facilitar posteriores etapas de optimización o generación de código.
4. **Internacionalización** de mensajes de error, o personalización vía archivos de configuración.

En suma, este ejercicio no solo refuerza la teoría de compiladores, sino que aporta una base práctica sobre la que construir lenguajes más complejos, intérpretes y compiladores educativos. Con una cobertura de casos de prueba exhaustiva y una arquitectura clara, el sistema está bien preparado para escalar y servir como prototipo didáctico en cursos de teoría de la computación y construcción de compiladores.