Centro Universitario de Ciencias Exactas e Ingeniería

TRADUCTORES DE LENGUAJES ll

PROYECTO FINAL PT 2

JULIO ESTEBAN VALDES LOPEZ

Juan Pablo Hernández Orozco

219294285

Objetivo:

Diseñar e implementar un analizador sintáctico descendente recursivo para un lenguaje elemental de declaraciones y expresiones aritméticas, **incorporando además** generación de código intermedio en formato de tres direcciones. El sistema debe ofrecer una interfaz gráfica en Tkinter que permita tanto validar la sintaxis y semántica básica (declaración y uso de variables) como guardar fácilmente el resultado del TAC en un archivo .txt.

Introducción:

- Un parser descendente mapea cada regla gramatical a un método que “consume” tokens y construye la estructura sintáctica.
- Muchas gramáticas iniciales contienen recursión por la izquierda, lo cual causa bucles infinitos en llamadas recursivas.
- Para resolverlo, transformamos la gramática y explicitamos la precedencia (*/ > +–) y asociatividad (izquierda) en reglas separadas.
- Además, se añade un primer nivel de chequeo semántico comprobando que ninguna variable se use sin haber sido declarada previamente.

Desarrollo:

Análisis Léxico (Lexer):

- Recorre carácter a carácter el archivo fuente, saltando espacios y comentarios (# …).
- Agrupa secuencias alfanuméricas y guiones bajos en identificadores o palabras reservadas (var, int, if, else, while, …).
- Reconoce literales enteros completos y símbolos (+, -, *, /, (,), =, ;).
- Emite un token EOF al final, para asegurar que no queden caracteres sin procesar.

Análisis Sintáctico y Semántico (Parser):

Se implementa un **parser recursivo** con métodos para cada símbolo no terminal:

expr → term expr_rest

expr_rest → (PLUS|MINUS) term expr_rest | ε

term → factor term_rest

term_rest → (TIMES|DIVIDE) factor term_rest | ε

factor → (Expr) | INTEGER | ID

Mantiene una tabla de símbolos (un conjunto) donde añade variables tras var x; o int x = expr;.

Al encontrar un identificador en expresión o en asignación, valida que esté previamente declarado, lanzando un error semántico si no lo está.

Generación de Código de 3 Direcciones:

Cada operación aritmética o literal se descompone en cuádruplas (operador, operando1, operando2, resultado).

Se generan variables temporales automáticas (t0, t1, …) para almacenar resultados intermedios.

La interfaz añade un botón “Generar y guardar 3-direcciones” que:

Invoca al parser para producir la lista de cuádruplas.

Muestra el TAC en la ventana.

Abre un diálogo para guardar el resultado en un archivo .txt.

Explicación del problema

El enunciado plantea un lenguaje de programación extremadamente simple, con dos tipos de sentencias: declaraciones de variables (con o sin inicialización) y asignaciones. A primera vista, estas construcciones parecen triviales, pero presentan varias dificultades al implementar un analizador sintáctico recursivo:

Recursión por la izquierda: Una gramática natural para expresiones aritméticas como

expr → expr PLUS term

expr → expr MINUS term

expr → term

provoca llamadas infinitas en un parser recursivo, ya que expr se invoca a sí mismo antes de consumir ningún token.

Precedencia y asociatividad: Sin reglas explícitas de precedencia, la expresión a + b * c podría interpretarse como (a + b) * c en lugar de a + (b * c). Además, todas las operaciones deben asociarse por la izquierda.

Chequeo semántico básico: Además de la sintaxis, se debe garantizar que ninguna variable se use sin haber sido declarada previamente. Por ejemplo, en

x = y + 3;

si y no fue declarada antes, el parser debe reportar un error semántico claro.

Manejo de comentarios y espacios: El lexema # comentario debe ignorarse hasta el final de línea, sin que afecte al conteo de líneas ni al flujo de tokens. Los espacios, tabulaciones y saltos de línea también deben descartarse adecuadamente.

Interfaz de usuario: Todo este proceso debe integrarse en una GUI que permita al usuario cargar un archivo, visualizar su contenido y obtener un mensaje preciso sobre la validez del programa o el tipo de error (léxico, sintáctico o semántico).

Explicación de la solución

Para abordar estos retos, la implementación se divide en tres módulos claramente diferenciados, cada uno responsable de una fase de compilación:

Para abordar estos retos, la implementación se divide en tres módulos claramente diferenciados, cada uno responsable de una fase de compilación:

Análisis Léxico (Lexer)

Recorre carácter a carácter el archivo fuente, descartando espacios y comentarios. Agrupa secuencias de letras, dígitos y guiones bajos en identificadores o en tokens de palabra reservada (var, int, if, etc.). Reconoce literales enteros y símbolos (+, –, *, /, (,), =, ;) y emite un token EOF al final.

Análisis Sintáctico y Semántico (Parser)

Se utiliza un parser descendente recursivo donde cada regla gramatical es un método:

expr → term expr_rest

expr_rest → (PLUS|MINUS) term expr_rest | ε

term → factor term_rest

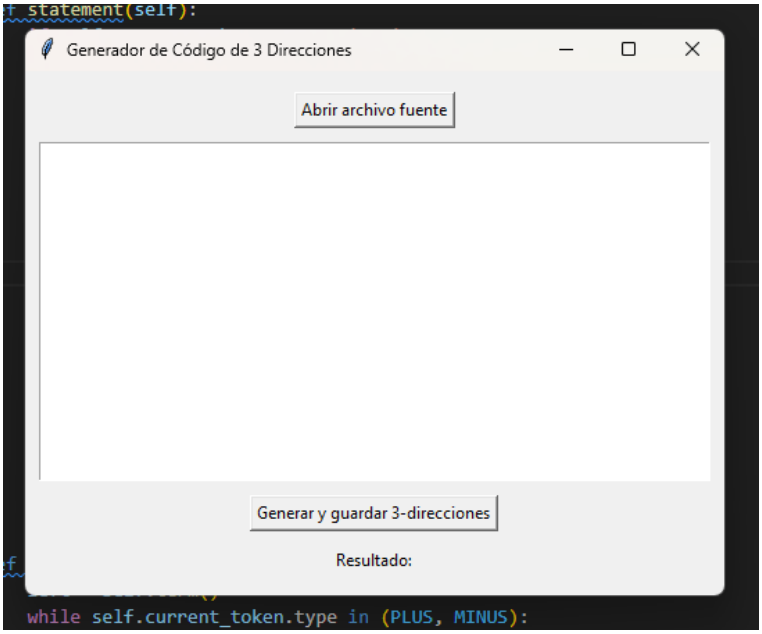
term_rest → (TIMES|DIVIDE) factor term_rest | ε

factor → (Expr) | INTEGER | ID

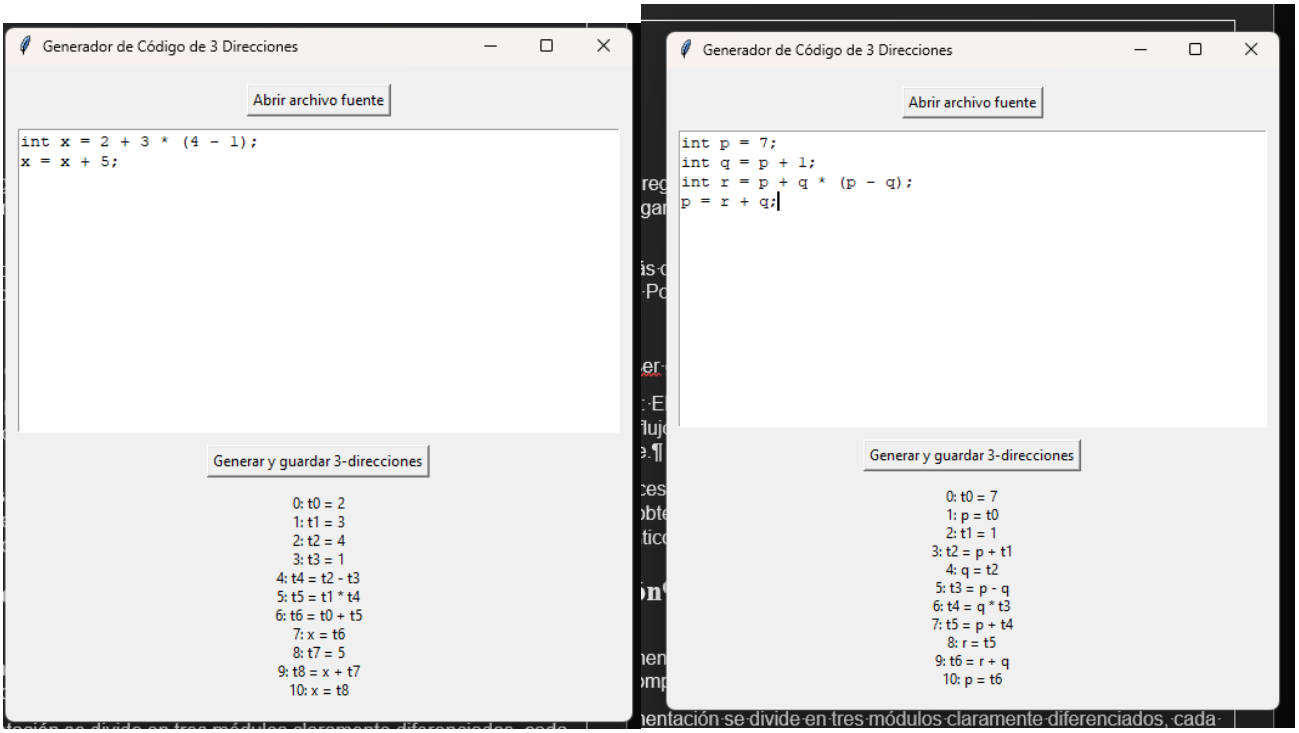
Esta transformación elimina la recursión por la izquierda y hace explícita la precedencia y asociatividad. Se mantiene una tabla de símbolos (un set de nombres) para declarar variables en var x; o int x = expr; y para verificar antes de usar un identificador. Si se detecta un uso sin declaración, se lanza un error semántico con un mensaje claro.

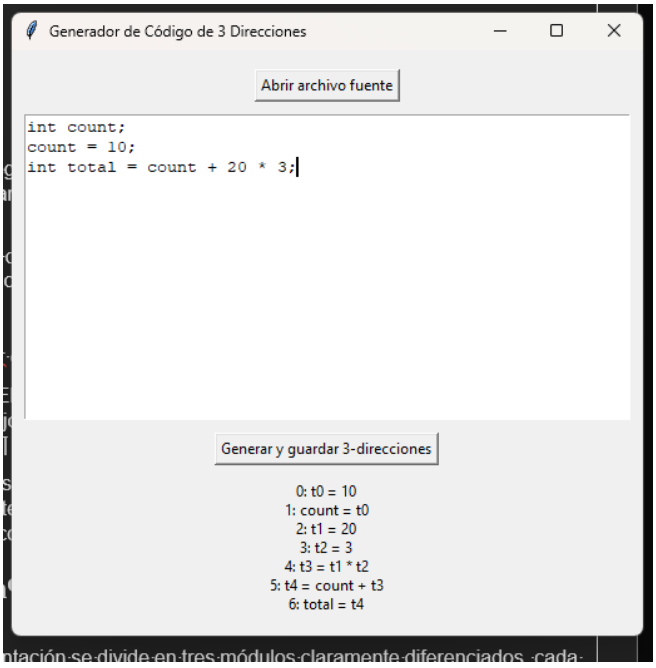
Generación de Código de 3 Direcciones

Durante el parsing de expresiones y asignaciones se emiten cuádruplas (operador, operando1, operando2, resultado) que registran cada paso del cálculo. Se generan variables temporales automáticas (t0, t1, …) para almacenar resultados intermedios de literales y subexpresiones. La GUI incorpora un botón “Generar y guardar 3-direcciones” que muestra en pantalla la lista de cuádruplas y permite guardarlas en un archivo .txt, integrando así análisis y documentación en un solo flujo de trabajo.



Casos de prueba





Conclusión:

La ampliación para producir código intermedio en tres direcciones enriquece el parser, proporcionando una representación detallada de cada paso de cálculo. Esta estructura facilita enormemente posteriores fases de compilación —optimización o generación de código de máquina— y demuestra

la modularidad del diseño (Lexer → Parser → TAC). La GUI en Tkinter ofrece una experiencia accesible, con mensajes claros de error y un flujo de guardado de resultados, sentando una base sólida para futuras extensiones (AST, control de flujo, manejo de tipos, detección de errores semánticos más avanzados).