

# Integración de IA generativa en entornos de compilación

Universidad Nacional Autónoma de México

Escalante Galván Alan

Frías Domínguez Oscar Fernando

Victoria Morales Ricardo Maximiliano

## Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Desarrollo</b>	<b>3</b>
2.1. IA generativa, LLMs y su integración en Compiladores . . . . .	3
2.2. Compilador con LLMs . . . . .	6
2.2.1. Probar recursividad izquierda y su eliminación . . . . .	7
2.2.2. Prueba LL(1) . . . . .	8
2.2.3. Flujo General . . . . .	9
<b>3. Código</b>	<b>11</b>
3.1. Relación con el tema . . . . .	11
3.2. Explicación . . . . .	12
3.3. Ejecuciones . . . . .	14
<b>4. Conclusiones</b>	<b>15</b>
<b>Referencias</b>	<b>15</b>

# 1. Introducción

A lo largo de la historia de la computación, los compiladores han desempeñado un papel fundamental en el desarrollo de software y continúan siendo herramientas esenciales para los programadores. Un compilador es un programa que actúa como traductor, recibiendo como entrada un programa escrito en un lenguaje de alto o medio nivel (denominado programa o código fuente) y transformándolo en una representación equivalente en un lenguaje de más bajo nivel, tradicionalmente ensamblador o código máquina. En los compiladores modernos, este proceso suele realizarse de manera gradual mediante la generación de uno o más códigos intermedios, lo cual facilita la optimización y la portabilidad del código.

El proceso de compilación se estructura en distintas fases bien definidas, que comúnmente se agrupan en dos grandes etapas:

- **Fase de análisis:** incluye el análisis léxico, el análisis sintáctico y el análisis semántico, cuyo objetivo es verificar la corrección del programa fuente y construir una representación interna del mismo.
- **Fase de síntesis:** comprende la generación del código intermedio, la optimización del código y la generación del código objeto o código máquina.

Para llevar a cabo una traducción correcta, los compiladores dependen de una sintaxis y una semántica bien definidas, propias del lenguaje de programación que procesan. Asimismo, incorporan mecanismos de detección y manejo de errores que permiten identificar problemas léxicos, sintácticos, semánticos y, en algunos casos, errores lógicos, facilitando así el desarrollo y depuración del software.

En años recientes, la incorporación de técnicas de inteligencia artificial en el proceso de desarrollo de software ha abierto nuevas oportunidades para la generación y optimización automática de código. Diversos estudios han demostrado que el uso de aprendizaje automático presenta un alto potencial para mejorar las decisiones de optimización realizadas por los compiladores. No obstante, la generación automática de código para lenguajes como **C**, **C++**, **Fortran** o **Python**, especialmente en programas de gran complejidad, continúa siendo un desafío significativo. Este problema se acentúa en aplicaciones que hacen uso intensivo de bibliotecas especializadas, flujos de control no triviales o extensiones específicas del dominio, donde el código generado automáticamente suele presentar un rendimiento inferior al del código escrito manualmente [1].

Dentro de este contexto, los modelos de lenguaje de gran escala (Large Language Models, LLMs) han emergido como una de las tecnologías más prometedoras. Estos modelos, entrenados con grandes volúmenes de datos textuales, son capaces de comprender y generar código

de manera coherente, así como de realizar tareas de traducción automática entre distintos niveles de abstracción. En particular, se ha explorado su uso para traducir código de alto nivel a representaciones de bajo nivel, dividiendo programas complejos en bloques más manejables e incorporando mecanismos de autocorrección que mejoran la calidad del resultado final [2].

Actualmente, una parte importante de la investigación en el área de compiladores se centra en la integración de técnicas de inteligencia artificial, especialmente LLMs, con el objetivo de mejorar la optimización del código, la detección de errores y la eficiencia del proceso de compilación. En este proyecto se analizará el impacto de la inteligencia artificial generativa en los compiladores modernos, se estudiará el uso de LLMs para la optimización del código y se desarrollará una API de carácter demostrativo que permita ilustrar de manera clara los conceptos y objetivos abordados.

## 2. Desarrollo

### 2.1. IA generativa, LLMs y su integración en Compiladores

La inteligencia artificial generativa es una rama de la inteligencia artificial que emplea técnicas de aprendizaje automático para crear contenido nuevo a partir de grandes conjuntos de datos de entrenamiento. A diferencia de las herramientas tradicionales, como los motores de búsqueda (por ejemplo, Google) o los sistemas de preguntas y respuestas, que se limitan a recuperar información existente, las herramientas de IA generativa utilizan conocimiento previo para producir contenido original. Dicho contenido puede presentarse en diversas formas, como texto, imágenes, video, código, representaciones en tres dimensiones o audio [3].

El desarrollo de estos modelos inicia en la fase de entrenamiento, en la cual se emplean redes neuronales para identificar patrones y estructuras dentro de los datos existentes, con el objetivo de generar contenido nuevo y original. A lo largo de esta fase, modelos como GPT-3 o Stable Diffusion son entrenados para comprender y manipular información lingüística o visual en función de los datos proporcionados.

Una vez concluido el entrenamiento inicial, los modelos pasan a la fase de ajuste fino (*fine-tuning*). En esta etapa, los modelos previamente entrenados se perfeccionan para mejorar su desempeño en tareas o dominios específicos, lo cual resulta fundamental para alinearlos con los requisitos de los usuarios y aumentar su precisión y relevancia en aplicaciones prácticas [3].

Posteriormente, durante la fase de uso o inferencia, los desarrolladores emplean los mode-

los de IA generativa para producir contenido nuevo. Este proceso puede implicar la generación de texto, la creación de imágenes a partir de descripciones textuales o la producción de datos sintéticos utilizados para entrenar otros modelos de inteligencia artificial [3].

La IA generativa cuenta con una amplia variedad de aplicaciones [1], entre las que destacan:

- Generación automática de código.
- Optimización y transformación de programas.
- Traducción entre lenguajes de programación y niveles de abstracción.
- Detección y corrección de errores.
- Generación de documentación y explicaciones.
- Soporte a sistemas de compilación y herramientas de desarrollo.

Estas aplicaciones resultan de particular interés en el contexto de este trabajo, al abordar la integración de la IA generativa en los compiladores, aunque cabe señalar que su campo de aplicación se extiende a numerosos dominios adicionales.

Dentro del ámbito de la inteligencia artificial, los modelos de lenguaje de gran tamaño (*Large Language Models*, LLMs) representan una clase de sistemas capaces de reconocer, interpretar y generar texto, entre otras tareas. Estos modelos se entrenan con enormes volúmenes de datos, de ahí el calificativo de “grandes”, y se basan principalmente en arquitecturas de aprendizaje profundo, como los modelos transformadores [4].

En términos generales, un LLM puede entenderse como un programa informático que ha sido entrenado con una cantidad suficiente de ejemplos como para modelar el lenguaje humano u otros tipos de datos complejos. Los LLMs pueden entrenarse para realizar diversas tareas; uno de los usos más conocidos es su aplicación como sistemas de IA generativa, donde, a partir de una indicación o pregunta, producen texto como respuesta. Un ejemplo representativo es ChatGPT, que puede generar ensayos, poemas u otros tipos de texto en función de la entrada del usuario.

Desde un punto de vista técnico, los LLMs se fundamentan en el aprendizaje automático, el cual constituye un subconjunto de la inteligencia artificial que se basa en entrenar modelos

mediante grandes cantidades de datos para que identifiquen patrones sin intervención humana directa [4].

Antes de la aparición de los LLMs modernos, ya existían líneas de investigación orientadas a aprovechar técnicas de aprendizaje automático para la generación de casos de prueba. En particular, se exploraba la sustitución de generadores de pruebas basados en reglas o gramáticas por modelos generativos aprendidos, capaces de producir fragmentos de código de manera estocástica [5]. La principal ventaja de estos enfoques radica en la reducción significativa del esfuerzo humano requerido para la construcción manual de generadores de programas aleatorios.

Con la introducción de los LLMs, la capacidad de los modelos para generar y razonar sobre código ha mejorado notablemente, dando lugar a múltiples líneas de investigación que los aplican a distintos dominios del desarrollo de software. En la actualidad, los conjuntos de datos de entrenamiento continúan creciendo, lo que permite a los LLMs ofrecer respuestas cada vez más precisas y contextualizadas.

No obstante, diversos estudios señalan que estos modelos presentan limitaciones importantes, como la restricción en la cantidad de tokens de entrada, la falta de memoria a largo plazo y la ausencia de garantías formales sobre la corrección del código generado [5]. A pesar de ello, los LLMs han demostrado ser capaces de generar y traducir código entre distintos lenguajes de programación, aprovechando el hecho de que el código fuente puede modelarse como texto con una sintaxis y semántica bien definidas.

En los últimos años, el desarrollo de la inteligencia artificial generativa ha abierto nuevas líneas de investigación en el diseño y la optimización de compiladores. A diferencia de los enfoques tradicionales, basados en reglas estáticas y heurísticas predefinidas, las técnicas de aprendizaje automático permiten a los compiladores incorporar mecanismos adaptativos capaces de aprender patrones a partir de grandes volúmenes de datos [1].

La integración de la IA generativa en los compiladores no busca sustituir las fases clásicas del proceso de compilación, sino complementarlas. En particular, se ha explorado su aplicación como apoyo en tareas de optimización de código, transformación de programas y análisis semántico avanzado, automatizando decisiones que en los compiladores tradicionales requieren un ajuste manual cuidadoso por parte de expertos [1].

Los LLMs han sido propuestos como herramientas de apoyo para la traducción de código entre distintos niveles de abstracción, desde lenguajes de alto nivel hasta representaciones intermedias o de bajo nivel. Un ejemplo representativo es el trabajo presentado en *LEGO-Compiler*, donde se propone un modelo de compilación neuronal que emplea LLMs para dividir programas complejos en bloques más manejables, facilitando su traducción e incorporando mecanismos de autocorrección durante la generación de código [2].

Asimismo, los LLMs han sido utilizados como analizadores inteligentes capaces de interpretar reportes de optimización generados por compiladores clásicos y sugerir modificaciones al código fuente con el objetivo de mejorar el rendimiento o reducir el consumo de recursos. Este enfoque híbrido combina la robustez de los compiladores tradicionales con la flexibilidad de los modelos generativos, permitiendo una toma de decisiones más informada durante el proceso de compilación [6].

No obstante, diversos estudios coinciden en que la integración de la IA generativa en los compiladores aún enfrenta desafíos significativos, como la falta de garantías formales de corrección, la posible degradación del rendimiento en ciertos escenarios y la dificultad de aplicar estos modelos en programas altamente especializados o dependientes del dominio [1]. Por ello, la mayoría de las propuestas actuales consideran a la IA generativa como un complemento a los compiladores tradicionales, y no como un reemplazo total de los mismos.

En este proyecto se analiza el impacto de la inteligencia artificial generativa y de los modelos de lenguaje de gran tamaño en el proceso de compilación, explorando su aplicación en tareas de optimización y traducción de código. Asimismo, se propone el desarrollo de una API de carácter demostrativo que permita ilustrar de manera práctica el uso de estas técnicas dentro de un flujo de compilación simplificado.

## 2.2. Compilador con LLMs

En esta sección analizaremos con mayor profundidad la implementación del análisis sintáctico de un compilador con LLMs, descrita en el artículo “*Enhancing XML-based Compiler Construction with Large Language Models: A Novel Approach*” [7], con el propósito de analizar un ejemplo concreto.

En este ejemplo, se implementa el análisis sintáctico para gramáticas  $LL(1)$ . El flujo de este programa es recibir una gramática libre de contexto, probar si la gramática tiene recursión izquierda, eliminar recursión por la izquierda, generar los conjuntos de First y Follow, y finalmente generar la tabla de análisis  $LR(1)$ . La gramática debe ser libre de contexto como

precondición del programa, y éste debe enviar error si deduce que la entrada no es una gramática LR(1).

En este ejemplo se hace uso del formato de archivo XML para codificar la entrada (gramática), es decir, la gramática libre de contexto se escribe en forma de archivo XML. Cada producción se representa como una elemento del archivo XML. Las ventajas de usar el formato de XML son varias: para validar entradas correctas, pues se propone un formato específico para gramáticas; para manipular más fácilmente a las gramáticas, en cuanto a la eliminación de recursión izquierda y conflictos; y para servir como entrada de una LLM.

En general es más fácil manipular archivos XML por sus utilidades estándar, es decir, porque se pueden leer, consultar, transformar, y validar documentos de tal tipo sin tener que hacer otro programa (como un parser) que manipule la información en estructuras de datos.

Entonces, en concreto, los pasos del análisis sintáctico tratado son los de las siguientes subsecciones.

### 2.2.1. Probar recursividad izquierda y su eliminación

Con un LLM se prueba si la gramática contiene recursión por la izquierda, y en caso de que la tenga, la elimina. El LLM construye una nueva gramática sin recursión izquierda, pero manteniendo la misma semántica.

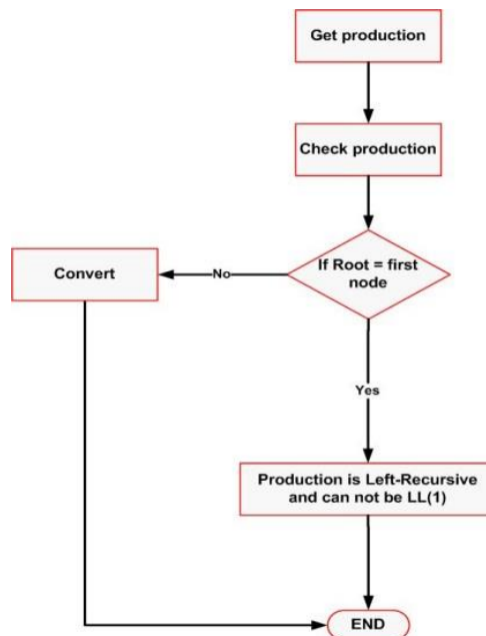


Figura 1: Flujo clásico de testeo de recursión izquierda. [7]

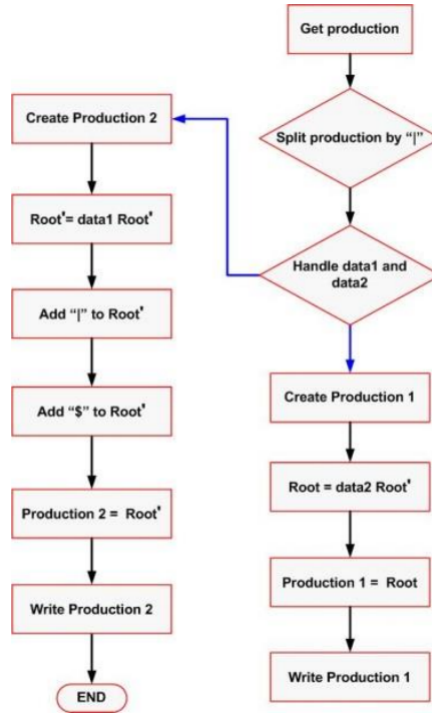


Figura 2: Flujo clásico de eliminación de recursión izquierda. [7]

### 2.2.2. Prueba LL(1)

Con un LLM se calculan los conjuntos de First y Follow, y con otro se construye la tabla de análisis correspondiente. En caso de que surga algún conflicto, se decide que la gramática no es LL(1) (esa es la técnica para verificar si la gramática es o no LL(1)).

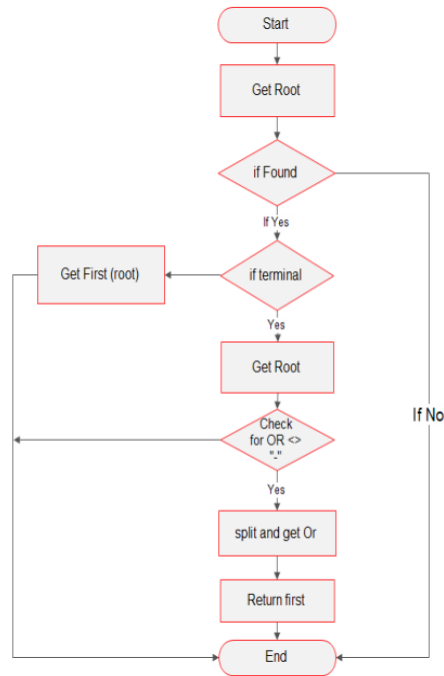


Figura 3: Flujo clásico de cálculo del conjunto first. [7]

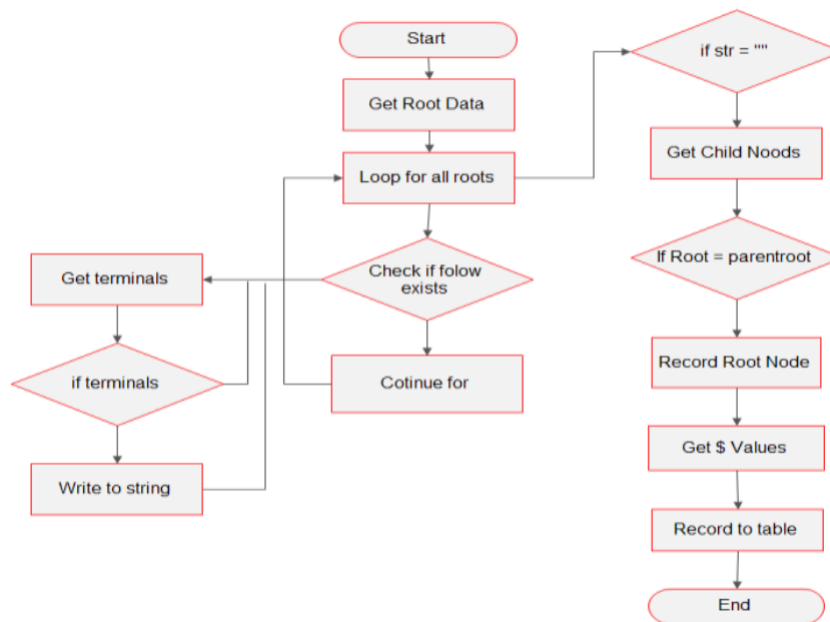


Figura 4: Flujo clásico de cálculo del conjunto follow. [7]

### 2.2.3. Flujo General

En general, todo el proceso que se realiza para el análisis sintáctico se resume con el siguiente diagrama.

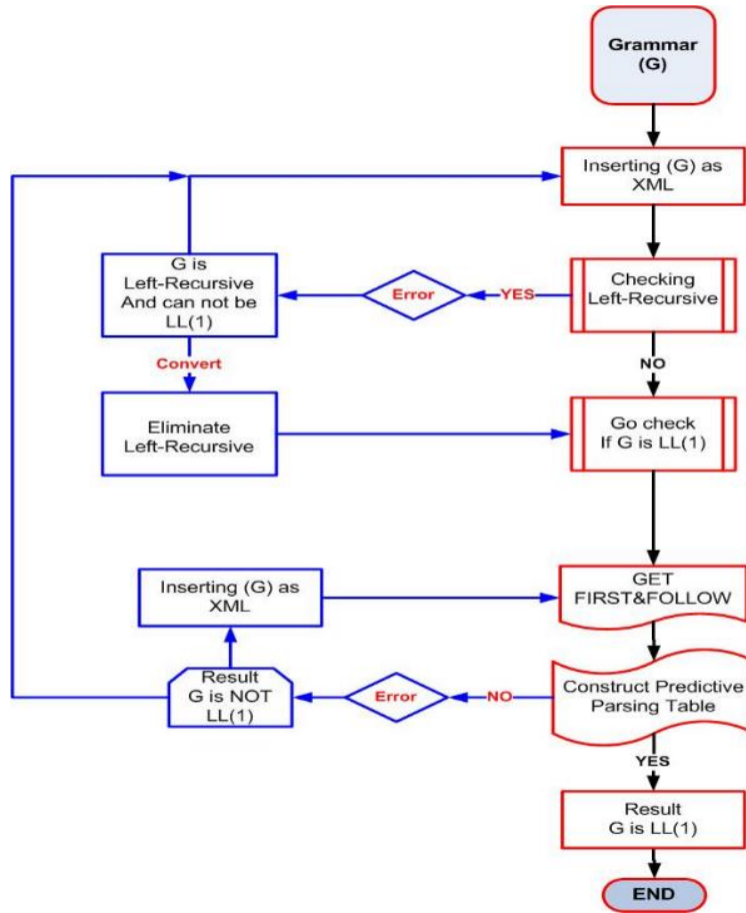


Figura 5: Flujo completo. [7]

Los procesos y cálculos de los diagramas anteriores se automatizan con LLMs, no se hace de la forma tradicional. No se implementan los algoritmos clásicos de análisis sintáctico, sino que todo este proceso se logra con el entrenamiento de LLMs.

Y aunque aparentemente esto no tiene ninguna utilidad nueva, sí existen varias ventajas:

- Optimización en la eliminación de recursión izquierda, debido a la rapidez y facilidad de manipular archivos XML por un LLM.
- Posibilidad de aplicar transformaciones extras como factorización por la izquierda y reorganización de la gramática sin tener que implementar un algoritmo convencional complejo.
- Posibilidad de obtener explicaciones claras y amenas dadas por el LLM sobre las transformaciones realizadas sobre la gramática.
- Debido a la alta capacidad de los LLMs en proceso de lenguaje natural, los LLMs pueden detectar patrones de recursividad indirecta compleja, o problemas muy difíciles

de encontrar por algoritmos clásicos.

## 3. Código

### 3.1. Relación con el tema

Con el objetivo de ejemplificar cómo la inteligencia artificial generativa puede apoyar la escritura y transformación automatizada de código en compiladores modernos, se desarrolló un prototipo en forma de una API simplificada. Este prototipo simula el comportamiento de un modelo de lenguaje aplicado al análisis y optimización de código, centrándose en la detección de patrones redundantes, ineficiencias comunes y elementos no utilizados dentro de programas escritos en Python.

La API implementa un análisis estático del código fuente mediante la identificación de patrones sintácticos previamente definidos. Para ello, el sistema procesa un archivo de entrada que contiene el código a analizar y aplica un conjunto de reglas basadas en expresiones regulares que permiten detectar asignaciones redundantes, operaciones innecesarias (como sumas con cero o multiplicaciones por uno), condiciones booleanas simplificables, así como variables, funciones, clases e importaciones que no son utilizadas a lo largo del programa.

Desde la perspectiva de los compiladores, este prototipo puede entenderse como un módulo de apoyo a la fase de optimización y transformación de código. Las sugerencias generadas por la API son conceptualmente equivalentes a las decisiones que un compilador moderno o una herramienta de análisis asistida por IA podría tomar para mejorar la calidad del código fuente antes de su traducción a una representación intermedia o a código máquina.

Aunque el sistema no emplea directamente un modelo de lenguaje de gran tamaño, su diseño simula el rol que estos modelos pueden desempeñar dentro del flujo de compilación. En particular, la API reproduce el comportamiento de un sistema generativo al analizar el contexto del código y producir recomendaciones automáticas de refactorización, emulando la capacidad de los LLMs para reconocer patrones sintácticos y semánticos aprendidos durante el entrenamiento.

En conjunto, la API desarrollada demuestra que la integración de técnicas inspiradas en la inteligencia artificial generativa puede facilitar la automatización de tareas tradicionalmente

manuales dentro del proceso de compilación. Esto refuerza la hipótesis de que los modelos generativos, reales o simulados, pueden actuar como herramientas auxiliares en compiladores modernos, mejorando los procesos de optimización y transformación de código sin sustituir los fundamentos teóricos del diseño de compiladores.

### 3.2. Explicación

El prototipo desarrollado emplea expresiones regulares (Regex) como mecanismo principal para identificar patrones de código redundante o susceptibles de optimización. Dichos patrones se dividen en dos categorías: aquellos que se analizan de manera individual por línea y aquellos que requieren examinar el contenido completo del archivo fuente.

En el primer grupo se incluyen patrones aplicados línea por línea, los cuales permiten detectar operaciones aritméticas innecesarias, como sumas o restas con cero, así como multiplicaciones o divisiones por uno. Este tipo de análisis corresponde a optimizaciones locales que pueden realizarse sin considerar el contexto global del programa. En la Figura 1 se muestran algunos ejemplos de estos patrones utilizados durante el análisis.

```
end_comment = r'(?:\$*(?:#.?)?)$'
patrones_linea["incremento"] = r'^\$*([A-Za-z_]\w*)\$*=\$*([A-Za-z_]\w*)\$*+ \$*([?d+])\$*' + end_comment
patrones_linea["decremento"] = r'^\$*([A-Za-z_]\w*)\$*=\$*([A-Za-z_]\w*)\$*- \$*([?d+])\$*' + end_comment
patrones_linea["multiplicacion"] = r'^\$*([A-Za-z_]\w*)\$*=\$*([A-Za-z_]\w*)\$*.*\$*([?d+])\$*' + end_comment
patrones_linea["division"] = r'^\$*([A-Za-z_]\w*)\$*=\$*([A-Za-z_]\w*)\$*/\$*([?d+])\$*' + end_comment
```

El segundo grupo de patrones se aplica sobre el código en su totalidad, con el objetivo de identificar elementos declarados que no son utilizados a lo largo del programa. Entre estos se incluyen variables, funciones y otras estructuras que, aunque están definidas, no participan en la ejecución del código. Este tipo de análisis se asocia con optimizaciones globales, ya que requiere una visión completa del programa. Algunos de estos patrones se ilustran en la Figura 2.

```
patrones_completo["funcion_inutil"] = r'^(?!(?:.*(def)\s*([A-Za-z_]\w*)(\.(.*)).*|(def)\s*([A-Za-z_]\w*)(\.(.*)).*|(def)\s*([A-Za-z_]\w*)(\.(.*)).*$
```

Durante la ejecución del programa, el sistema abre el archivo de entrada `codigo.txt` y recorre cada línea del código fuente. Cada línea es comparada con los patrones definidos para el análisis local y, en caso de coincidencia, se genera una sugerencia que indica la línea correspondiente y la posible optimización o refactorización aplicable. Este proceso se muestra en la Figura 3.

```

with open("codigo.txt", "r") as archivo:
    completo = archivo.read()
    linea = 0
    respuesta = []
    lineas = completo.splitlines()
    for codigo in lineas:
        linea += 1
        for caso, regex in patrones_linea.items():
            for flag in re.finditer(regex, codigo.strip()):
                match caso:
                    case "incremento":
                        var1 = flag.group(1)
                        var2 = flag.group(2)
                        var3 = flag.group(3)
                        if var1 == var2:
                            respuesta.append(f"En la linea {linea} considera utilizar {var1} += {var3} en lugar de {var1} = {var2} + {var3}")

```

El análisis global del código se realiza de manera similar, con la diferencia de que los patrones definidos para este propósito se aplican sobre el contenido completo del archivo. Este enfoque permite verificar si ciertos elementos se declaran pero nunca se utilizan en el resto del programa, lo cual puede indicar código innecesario o errores de diseño. Un ejemplo de este proceso se presenta en la Figura 4.

```

match caso:
    case "variable_inutil":
        var = flag.group(1)
        respuesta.append(f"La variable {var} en la linea {linea} se asigna pero nunca se utiliza en el código")
    case "funcion_inutil":
        func = flag.group(8)
        respuesta.append(f"La función {func} en la linea {linea} no se utiliza en el código")
    case "lista_inutil":
        lista = flag.group(1)
        respuesta.append(f"La lista {lista} en la linea {linea} no se utiliza en el código")

```

Adicionalmente, la detección de importaciones no utilizadas se realiza de forma independiente, debido a la diversidad de formas en que estas pueden declararse en Python, como `import`, `import x as y` o `from x import y`. Este análisis permite identificar dependencias innecesarias que pueden eliminarse para mejorar la claridad y mantenibilidad del código. La Figura 5 muestra un ejemplo de este procedimiento.

```

for m in re.finditer(r'^\s*import\s+([A-Za-z_]\w*)(?:\s+as\s+([A-Za-z_]\w*))?', completo, re.M):
    module = m.group(1)
    alias = m.group(2) or None
    identifier = alias if alias else module
    start_pos = m.start()
    line_no = completo[:start_pos].count('\n') + 1
    imports.append({'identifier': identifier, 'line': line_no})

```

Finalmente, una vez completado el análisis, todas las sugerencias generadas se imprimen en la consola y se almacenan en el archivo de salida `salida.txt`. En caso de que no se

detecten oportunidades de mejora, el sistema emite un mensaje indicando que el código analizado no presenta optimizaciones evidentes. Un ejemplo de la salida generada se muestra en la Figura 6.

```
if respuesta:
    with open("salida.txt", "w", encoding="utf-8") as archivo_salida:
        for sugerencia in respuesta:
            print(sugerencia)
            archivo_salida.write(sugerencia + "\n")
else:
    print("Felicitaciones! No se encontraron mejoras.")
```

### 3.3. Ejecuciones

Ejemplo parcial del archivo de prueba utilizado:

```
# Useful variable and use
used_var = 1
print(used_var) # usage of used_var

# Several unused variables
unused_var1 = 100
unused_var2 = 'not used'
_temp_unused = None

# Line-level inefficiencies (examples repeated)
x = x + 1
x = x + 1 # repeated intentionally
y = y - 2
y = y - 2
z = z * 3
w = w / 4
m = m % 5
pwr = pwr ** 2
redund = redund
add_zero = add_zero + 0
sub_zero = sub_zero - 0
mul_one = mul_one * 1
div_one = div_one / 1
mod_one = mod_one % 1
pow_one = pow_one ** 1
zero_plus = 0 + zero_plus
zero_minus = 0 - zero_minus
one_mul = 1 * one_mul
one_div = 1 / one_div
```

Visualización al ejecutarlo en la línea de comando mediante “python compilador.py”:

```
C:\Users\alane\Documents\Homework\Compiladores\python compilador.py
En la línea 19 considera utilizar x += 1 en lugar de x = x + 1
En la línea 20 considera utilizar x += 1 en lugar de x = x + 1
En la línea 21 considera utilizar y -= 2 en lugar de y = y - 2
En la línea 22 considera utilizar y -= 2 en lugar de y = y - 2
En la línea 23 considera utilizar z *= 3 en lugar de z = z * 3
En la línea 24 considera utilizar w /= 4 en lugar de w = w / 4
En la línea 25 considera utilizar m %= 5 en lugar de m = m % 5
En la línea 26 considera utilizar pwr ** 2 en lugar de pwr = pwr ** 2
En la línea 27 la asignación redund = redund es redundante
En la línea 28 considera utilizar add_zero += 0 en lugar de add_zero = add_zero + 0
En la línea 29 considera eliminar la suma con 0 en add_zero = add_zero + 0
En la línea 30 considera utilizar sub_zero -= 0 en lugar de sub_zero = sub_zero - 0
En la línea 31 considera eliminar la resta con 0 en sub_zero = sub_zero - 0
En la línea 32 considera utilizar mul_one *= 1 en lugar de mul_one = mul_one * 1
En la línea 33 considera eliminar la multiplicación por 1 en mul_one = mul_one * 1
```

Documento “salida.txt” generado por el programa:

```

En la línea 19 considera utilizar x += 1 en lugar de x = x + 1
En la línea 20 considera utilizar x += 1 en lugar de x = x + 1
En la línea 21 considera utilizar y -= 2 en lugar de y = y - 2
En la línea 22 considera utilizar y -= 2 en lugar de y = y - 2
En la línea 23 considera utilizar z += 3 en lugar de z = z + 3
En la línea 24 considera utilizar w /= 4 en lugar de w = w / 4
En la línea 25 considera utilizar m *= 5 en lugar de m = m * 5
En la línea 26 considera utilizar per **= 2 en lugar de per = per ** 2
En la línea 27 la asignación redund = redund es redundante
En la línea 28 considera utilizar add_zero += 0 en lugar de add_zero = add_zero + 0
En la línea 29 considera eliminar la suma con 0 en add_zero = add_zero + 0
En la línea 29 considera utilizar sub_zero -= 0 en lugar de sub_zero = sub_zero - 0
En la línea 29 considera eliminar la resta con 0 en sub_zero = sub_zero - 0
En la línea 30 considera utilizar mul_one *= 1 en lugar de mul_one = mul_one * 1
En la línea 30 considera eliminar la multiplicación por 1 en mul_one = mul_one * 1
En la línea 31 considera utilizar div_one /= 1 en lugar de div_one = div_one / 1

```

## 4. Conclusiones

La inteligencia artificial y el machine learning están cambiando rápidamente muchas áreas de las ciencias de la computación. Y el uso de modelos grandes de lenguaje o LLMs dentro del área de lenguajes de programación, y en particular de compiladores, no es la excepción. En este reporte entendimos que los LLMs sirven para extender la funcionalidad de un compilador, pero no para sustituirlo. Las redes neuronales no son perfectas, siempre tienen un margen de error y pueden alucinar. También aprendimos que es conveniente utilizar herramientas o formatos de texto (como XML) para optimizar el funcionamiento de LLMs, puesto que éste tipo de redes neuronales son mejores en el procesamiento de texto bien estructurado.

Finalmente, logramos entender y programar la idea de cómo una API de LLM puede ayudar al usuario y/o a un sistema computacional a mejorar y optimizar código para una mayor rapidez y eficiencia en la ejecución de éste.

## Referencias

- [1] S. P. Velaga, “Ai-assisted code generation and optimization,” *International Journal of Innovations in Engineering Research and Technology (IJIERT)*, vol. 7, no. 6, 2020, survey sobre generación y optimización de código asistida por inteligencia artificial.
- [2] S. Zhang, Y. Chen, H. Wang, and Z. Su, “Lego-compiler: Enhancing neural compilation through translation composability,” *arXiv preprint arXiv:2505.20356*, 2025, arXiv:2505.20356. [Online]. Available: <https://arxiv.org/abs/2505.20356>
- [3] L. Latorre, E. Rego, I. Cerrato, J. D. Zarate, and L. De Leo, “Reporte de tecnología: Ia generativa,” Reporte de Tecnología, Tech. Rep., 2024, supervisor: Mariana Gutiérrez. Colaboradores: José Daniel Zarate, Eduardo Rego, Rodrigo Villamayor.
- [4] Cloudflare, Inc., “¿qué es un modelo de lenguaje grande (llm)?” <https://www.cloudflare.com/es-es/learning/ai/what-is-large-language-model/>, 2024, recurso educativo en línea sobre modelos de lenguaje de gran escala y su funcionamiento.

- [5] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 283–294.
- [6] P. Pirkelbauer and K. Sagonas, “Compilergpt: Leveraging large language models for analyzing and acting on compiler optimization reports,” *arXiv preprint arXiv:2506.06227*, 2025, arXiv:2506.06227. [Online]. Available: <https://arxiv.org/abs/2506.06227>
- [7] I. A. Zahid and S. S. Joudar, “Enhancing xml-based compiler construction with large language models: A novel approach,” *Mesopotamian Journal of Big Data*, vol. 2024, p. 23–39, Mar. 2024. [Online]. Available: <https://journals.mesopotamian.press/index.php/bigdata/article/view/343>