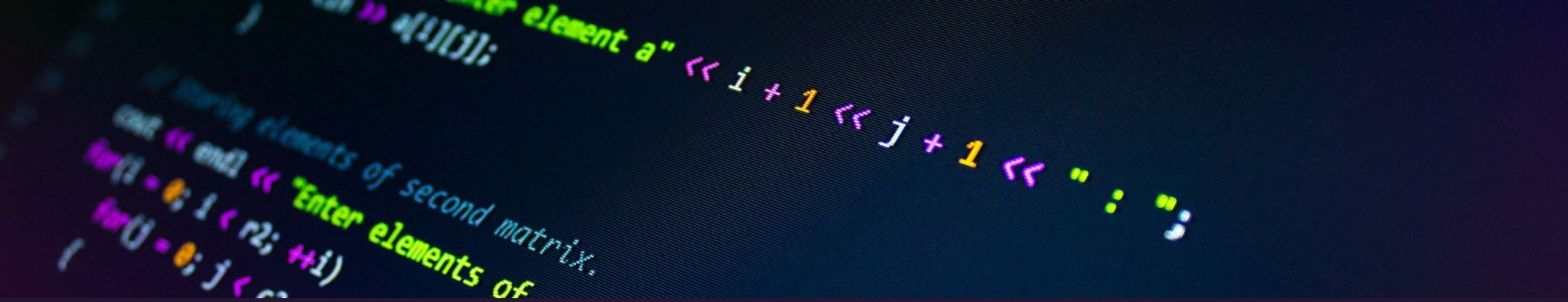
LENGUAJES & AUTÓMATAS II

Unidad III. "Optimización"

Instituto Tecnológico de Saltillo

CATEDRÁTICO RESPONSABLE: Espinoza Arzola Jesús Alberto

PRESENTADO POR: Aleksandra Flores



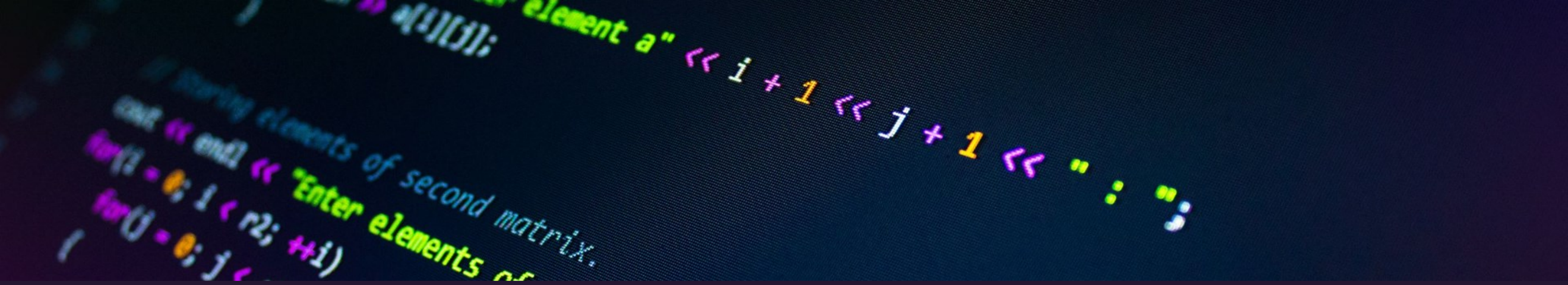
Objetivo de la unidad

Comprender la importancia de la optimización en los lenguajes y autómatas

Desarrollar habilidades para aplicar métodos y técnicas de optimización en diferentes contextos.

La optimización desde diferentes perspectivas

Incluyendo la minimización del tiempo de ejecución, la reducción del uso de memoria, o la maximización de la utilización del procesador. Cada uno de estos objetivos puede requerir diferentes técnicas y estrategias.



Objetivos específicos de los costos en la optimización

1. **Optimización del tiempo de desarrollo**: En ocasiones, la mejora obtenida puede no verse reflejada en el programa final, pero sí ser perjudicial para el equipo de desarrollo. Por lo tanto, se busca minimizar el tiempo de desarrollo.
2. **Optimización del tiempo de ejecución**: La optimización de una pequeña mejora tal vez tenga una pequeña ganancia en tiempo o en espacio, pero sale muy costosa en tiempo en generarla. Por lo tanto, se busca minimizar el tiempo de ejecución.

1. **Optimización del uso de memoria**: La memoria es uno de los recursos más importantes de la computadora. Por lo tanto, se busca minimizar el uso de memoria.
2. **Optimización del uso de registros**: Los registros del procesador se emplean para controlar instrucciones en ejecución, manejar direccionamiento de memoria y proporcionar capacidad aritmética1. Por lo tanto, se busca minimizar el uso de registros.

Introducción

Definición de Optimización

La optimización se refiere a un proceso que busca mejorar la forma en que un programa utiliza los recursos. La finalidad de la optimización de código es producir un código objeto lo más eficiente posible.
Las optimizaciones se realizan en base al alcance ofrecido por el compilador.

La optimización es un componente esencial en el estudio de los lenguajes y autómatas, y es el enfoque principal de la Unidad 3 en Lenguajes y Autómatas II. Este proceso busca mejorar la eficiencia del código generado por un compilador, lo que puede tener un impacto significativo en el rendimiento de un programa.

Tipos de Optimización

1

LOCALES

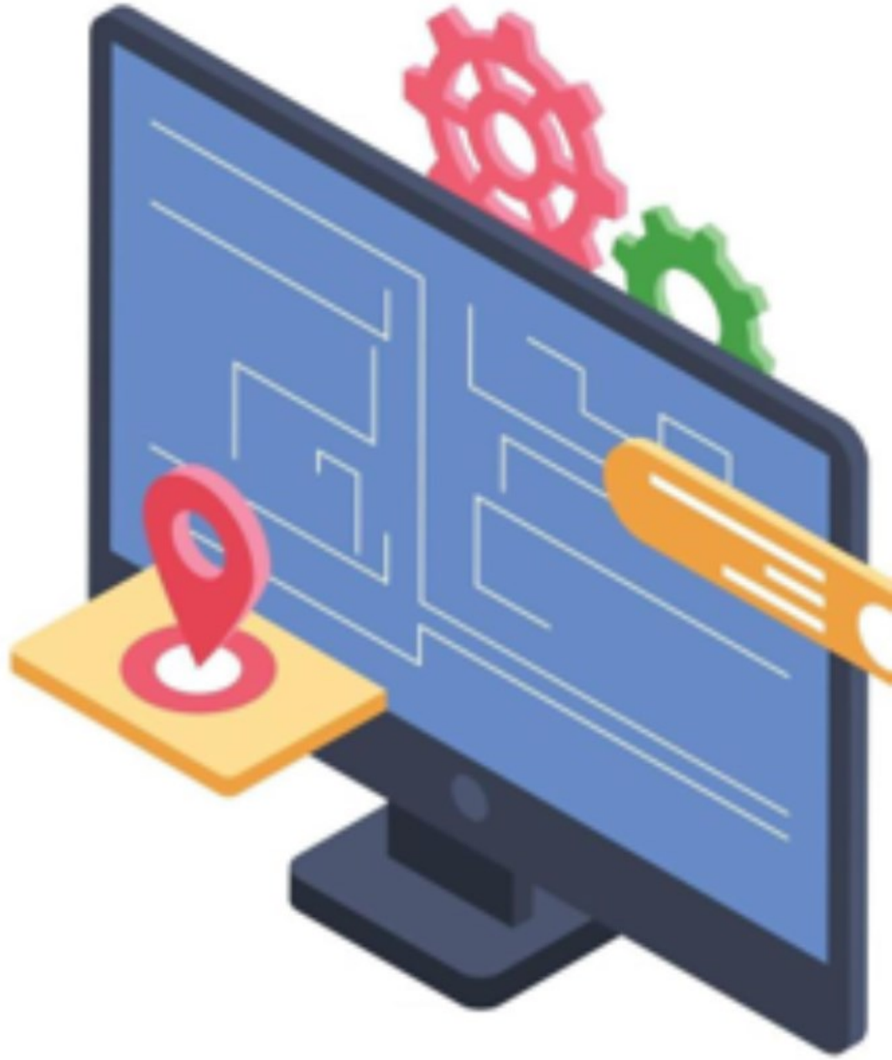
La optimización local se realiza sobre módulos del programa. En la mayoría de las ocasiones a través de funciones, métodos, procedimientos, clases, etc. La característica de las optimizaciones locales es que solo se ven reflejados en dichas secciones.

La optimización local sirve cuando un bloque de programa o sección es crítico por ejemplo: E/S, la concurrencia, la rapidez y confiabilidad de un conjunto de instrucciones. Como el espacio de soluciones es más pequeño la optimización local es más rápida.

2

CICLOS

Los ciclos son una de las partes más esenciales en el rendimiento de un programa dado que realizan acciones repetitivas, y si dichas acciones están mal realizadas, el problema se hace N veces más grandes. La mayoría de las optimizaciones sobre ciclos tratan de encontrar elementos que no deben repetirse en un ciclo.



ocal Optimization

Tipos de Optimización

1

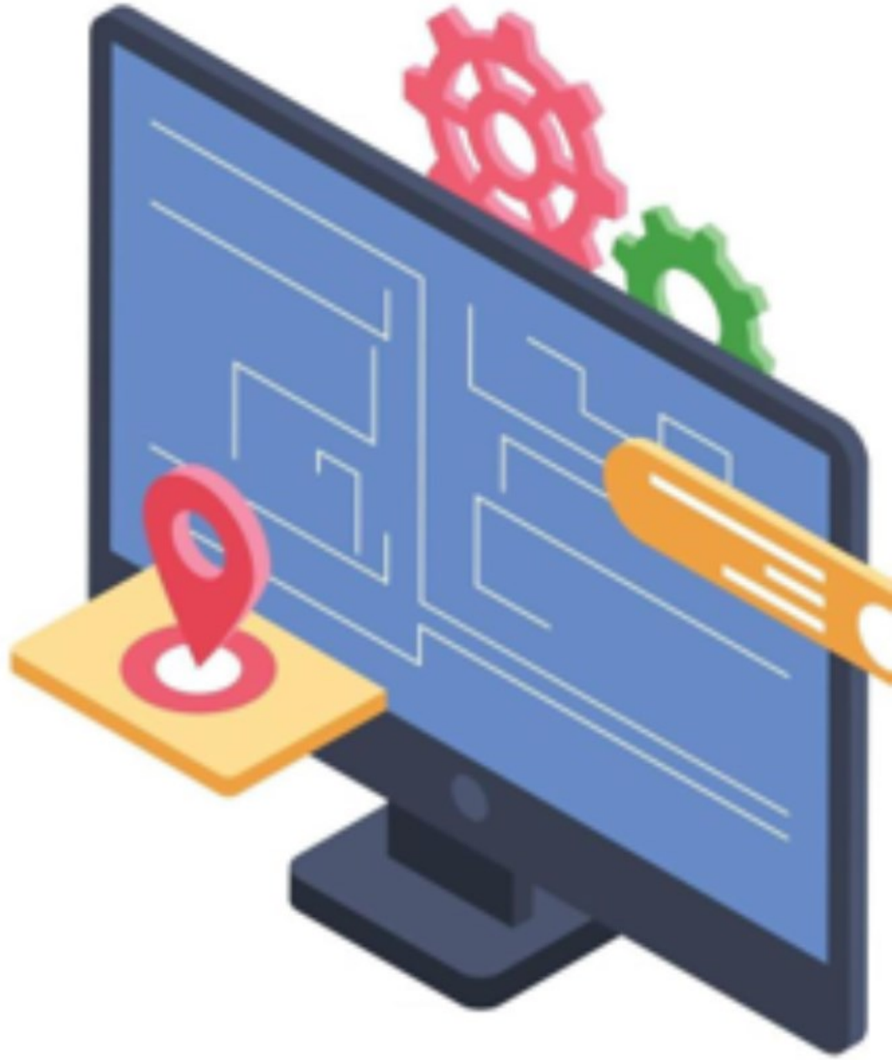
GLOBALES

La optimización global se da con respecto a todo el código. Este tipo de optimización es más lenta pero mejora el desempeño general de todo programa. Las optimizaciones globales pueden depender de la arquitectura de la máquina. En algunos casos es mejor mantener variables globales para agilizar los procesos (el proceso de declarar variables y eliminarlas toma su tiempo) pero consume más memoria. Algunas optimizaciones incluyen utilizar como variables registros del CPU, utilizar instrucciones en ensamblador.

2

DE MIRILLA

La optimización de mirilla trata de estructurar de manera eficiente el flujo del programa, sobre todo en instrucciones de bifurcación como son las decisiones, ciclos y saltos de rutinas. La idea es tener los saltos lo más cerca de las llamadas, siendo el salto lo más pequeño posible. Instrucciones de bifurcación Interrumpen el flujo normal de un programa, es decir que evitan que se ejecute alguna instrucción del programa y salta a otra parte del programa.



ocal Optimization

Ejemplos de Optimización

LOCALES

Las optimizaciones locales se realizan sobre el bloque básico.

- ***Optimizaciones locales***
  - Folding
  - Propagación de constantes
  - Reducción de potencia
  - Reducción de subexpresiones comunes

CICLOS

Sea el ejemplo:

```
while(a == b) {
int c = a;
c = 5;
 ...; }
```

GLOBALES

DE MIRILLA

Por ejemplo: el “break”

```
Switch (expresión que estamos evaluando)
{
Case 1: cout << “Hola” ;
Break;
Case 2: cout << “amigos”;
Break;
}
```

Ejemplos de Optimización

LOCALES

- ***Bloque básico***
    - Un bloque básico es un fragmento de código que tiene una única entrada y salida, y cuyas instrucciones se ejecutan secuencialmente.
      Implicaciones:
        - Si se ejecuta una instrucción del bloque se ejecutan todas en un orden conocido en tiempo de compilación.
        - La idea del bloque básico es encontrar partes del programa cuyo análisis necesario para la optimización sea lo más simple posible.

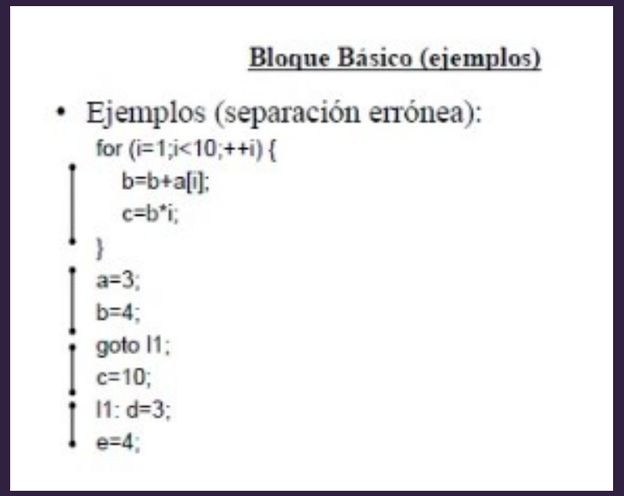
CICLOS

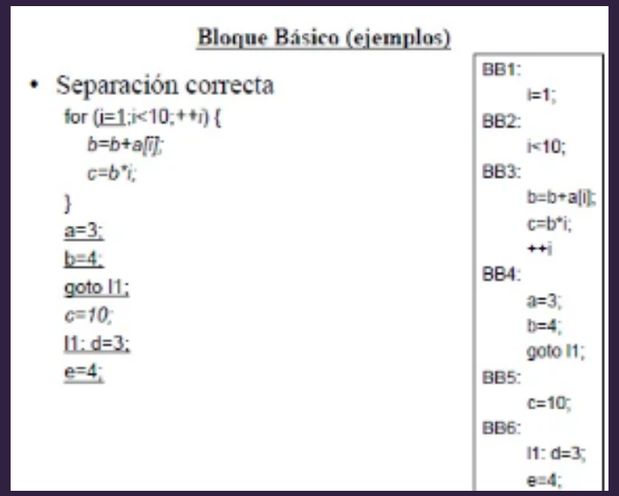
GLOBALES

DE MIRILLA

Ejemplos de Optimización

LOCALES





Ensamblamiento (Folding)

- El ensamblamiento es remplazar las expresiones por su resultado cuando se pueden evaluar en tiempo de compilación (resultado constante).
  - Ejemplo: A=2+3+A+C -> A=5+A+C
- Estas optimizaciones permiten que el programador utilice cálculos entre constantes representados explícitamente sin introducir ineficiencias.

Ejemplos de Optimización

LOCALES

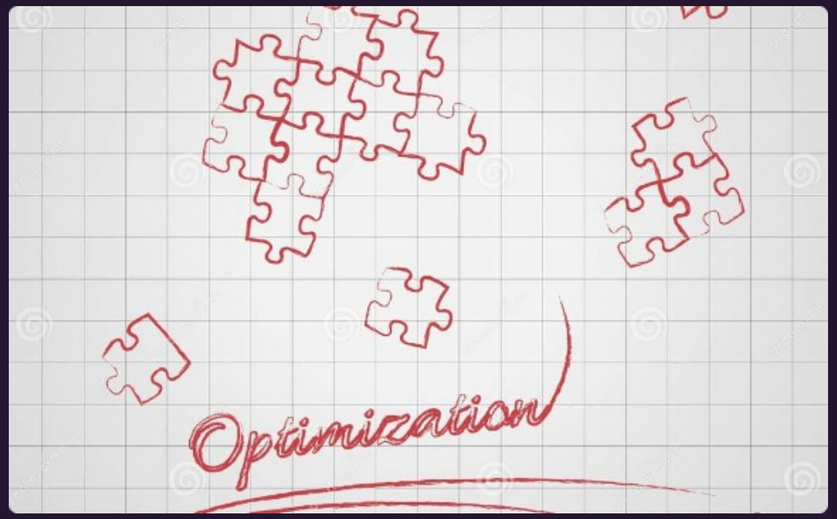
Implementación del Folding

- Implementación del floding durante la generación de código realizada conjuntamente con el análisis sintáctico.
  - Se añade el atributo de constante temporal a los símbolos no terminales y a las variables de la tabla de símbolos.
  - Se añade el procesamiento de las constantes a las reglas de análisis de expresiones.
    - Optimiza: 2+3+b -> 5+b
- **Hay una suma de constantes (2+3)+b**
  - No optimiza: 2+b+3 -> 2+b+3
- **No hay una suma de constantes (2+b)+3**

Implementación del Folding

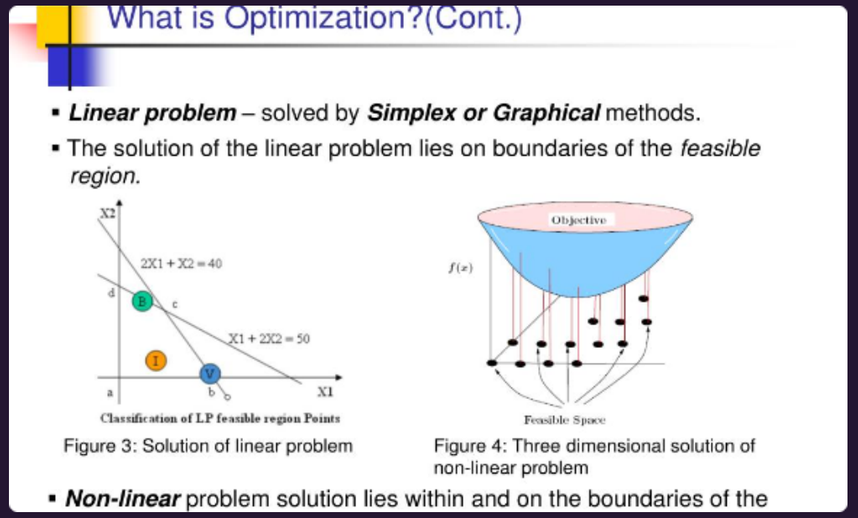
- **Implementación posterior a la generación de código**
  - Buscar partes del árbol donde se puede aplicar la propiedad conmutativa:
- Sumas/restas: como la resta no es conmutativa se transforma en sumas: a+b-c+d -> a+b+(-c)+d
- Productos/divisiones: como la división no es conmutativa se transforma en productos: a*b/c*e -> a*b*(1/c)*e
  - Buscar las constantes y operarlas
  - Reconstruir el árbol.

COSTOS



Resumen de la Importancia de los COSTOS en la optimización

Los costos son el factor más importante a tomar en cuenta a la hora de optimizar ya que en ocasiones la mejora obtenida puede verse no reflejada en el programa final pero si ser perjudicial para el equipo de desarrollo. La optimización de una pequeña mejora tal vez tenga una pequeña ganancia en tiempo o en espacio pero sale muy costosa en tiempo en generarla. Pero en cambio si esa optimización se hace por ejemplo en un ciclo, la mejora obtenida puede ser N veces mayor por lo cual el costo se minimiza y es benéfico la mejora.



Sub-subtemas a tratar en el contexto de COSTOS dentro de la optimización:

1. Costo de ejecución (Memoria, registros, pilas)
2. Criterios para mejorar el código
3. Herramientas para el análisis del flujo de datos