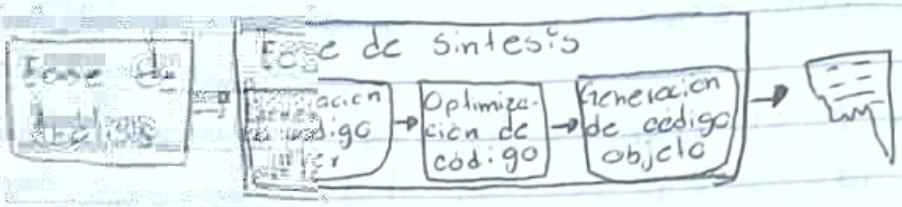
Optimización de Código



La optimización de código es la segunda parte de la fase general de sintesis del objeto en producir una versión del código que sea significativamente mejor que el código original logrando un mejor desempeño general del programa ejecutado. Para lograrlo se realizan una serie de transformaciones que van sustituyendo ciertas secciones del código por otras que le ayuden a mejorar el desempeño del mismo, estas mejoras pueden incluir un mejor uso del almacenamiento y la aceleración en la ejecución de los programas.

No todos los compiladores implementan esta fase, ya que implica un sobresfuerzo que no siempre valdrá la pena. A los compiladores que si implementan la fase de optimización se les llama compiladores optimizadores

Criterios para la desición de aplicar optimización a un código

Dicho de una manera sencilla, las mejores transformaciones son las que producen el

mayor beneficio con el menor esfuerzo, por lo que para decidir si deberá o no aplicarse un proceso de optimizcaión a un código deben verificarse los siguientes 3 criterios

1. Preservar el significado: Debemos garantizar que la aplicación de las transformaciones para optimizar el código, no debe cambiar el resultado producido originalmente por el programa
2. Aceleración del programa: El objetivo de la optimizacion es mejorar el desempeño de los programas por lo que será sumamente importante estimar si el proceso de optimización aplicado a un programa especifico conseguirá acelerar en una cantidad mensurable la velocidad de ejecución del mismo
3. Debe valer la pena: Implica valorar la relación entre esfuerzo y el beneficio de aplicar la optimizacion de un programa en especifico. Por ejemplo, un proyecto escolar que será utilizado por la acreditación de una asignatura o requisito académico, arrojará un balance negativo entre esfuerzo y beneficio, lo mismo que una app desarrollada para resolver una tarea concreta que no ha de repetirse seguido. En cambio, un sistema que se mantendrá en ejecución continua será importante asegurarnos que el desempeño sea óptimo

Optimización de código

código puede realizarse
al o desde un ámbito
bloques específicos de
un proceso completo de
digo incluye la aplicación
local de las técnicas

Cualquier transformación
re el código deberá
icado del mismo, es decir,
la misma función y
resultados independiente-
bios realizados durante

incipales técnicas de
digo
que preservan la función
dificaciones que se
digo sin cambiar la
le cumple. Las técnicas
or esto son: - la
bexpresiones comunes,
e código inactivo y
de constantes

bloque 1

(1) i := m - 1
(2) j := n
(3) t_1 := 4 * n
(4) v := a[t_1]

bloque 2

(5) i := i + 1
(6) t_2 := 4 * i
(7) t_3 := a[t_2]
(8) if t_3 < v goto (5)

bloque 3

(9) j := j - 1
(10) t_4 := 4 * j
(11) t_5 := a[t_4]
(12) if t_5 > v goto (9)
(13) if i >= j goto (23)

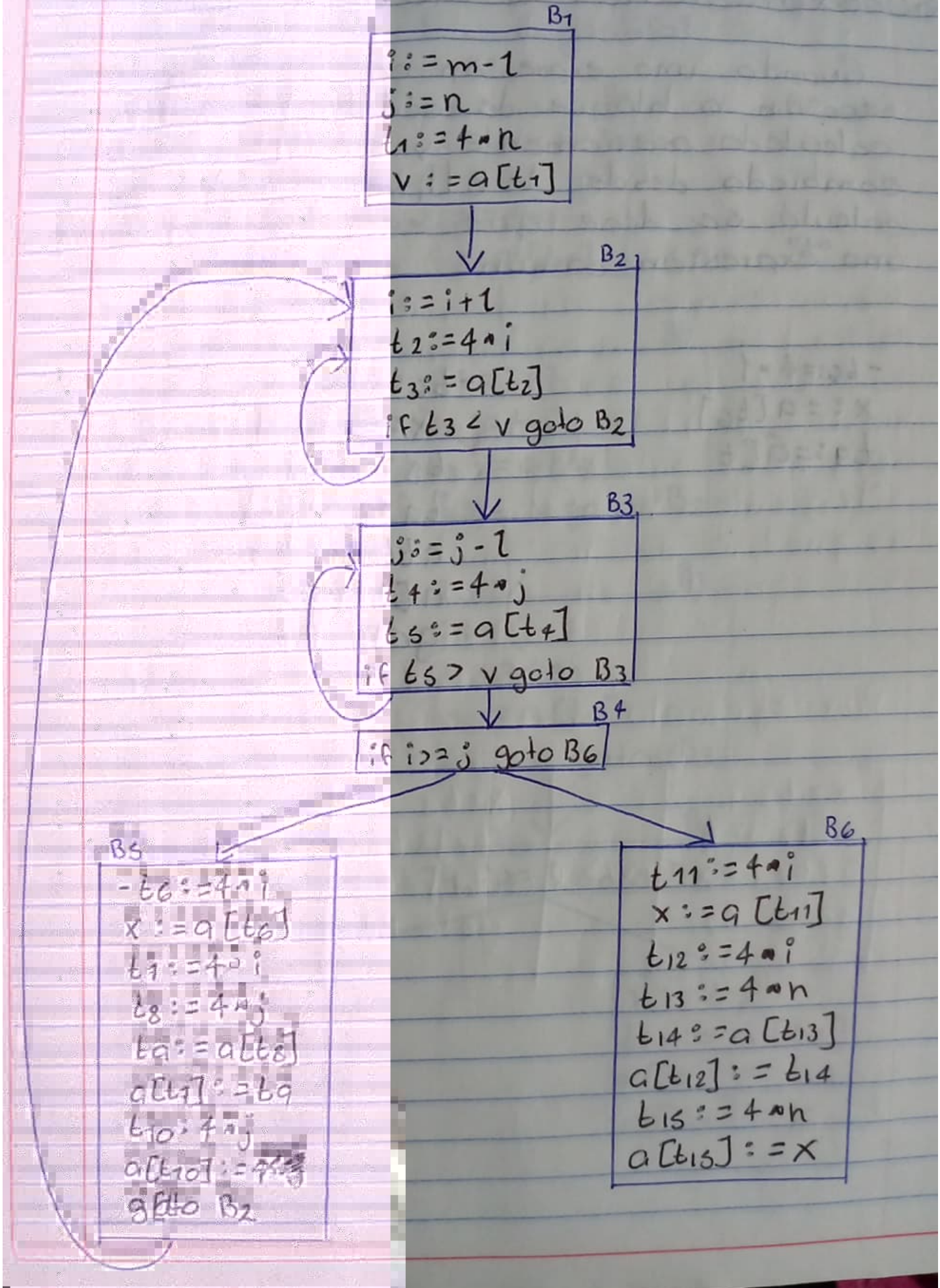
bloque 4

(14) t_6 := 4 * i
(15) x := a[t_6]
(16) t_7 := 4 * i
(17) t_8 := 4 * j
(18) ta := a[t_8]
(19) a[t_7] := ta
(20) t_{10} := 4 * j
(21) a[t_{10}] := x
(22) goto (5)

bloque 5

(23) t_{11} := 4 * i
(24) x := a[t_{11}]
(25) t_{12} := 4 * i
(26) t_{13} := 4 * n
(27) t_{14} := a[t_{13}]
(28) a[t_{12}] := t_{14}
(29) t_{15} := 4 * n
(30) a[t_{15}] := x

Lo primero que debe realizarse antes de comenzar con el proceso de optimización es organizar el código identificando los distintos bloques que conforman al programa completo


B1
i := m-1
j := n
t1 := 4*n
v := a[t1]
B2
i := i+1
t2 := 4*i
t3 := a[t2]
if t3 < v goto B2
B3
j := j-1
t4 := 4*j
t5 := a[t4]
if t5 > v goto B3
B4
if i >= j goto B6
B5
t6 := 4*i
x := a[t6]
t7 := 4*i
t8 := 4*j
t9 := a[t8]
a[t7] := t9
t10 := 4*j
a[t10] := x
goto B2
B6
t11 := 4*i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x

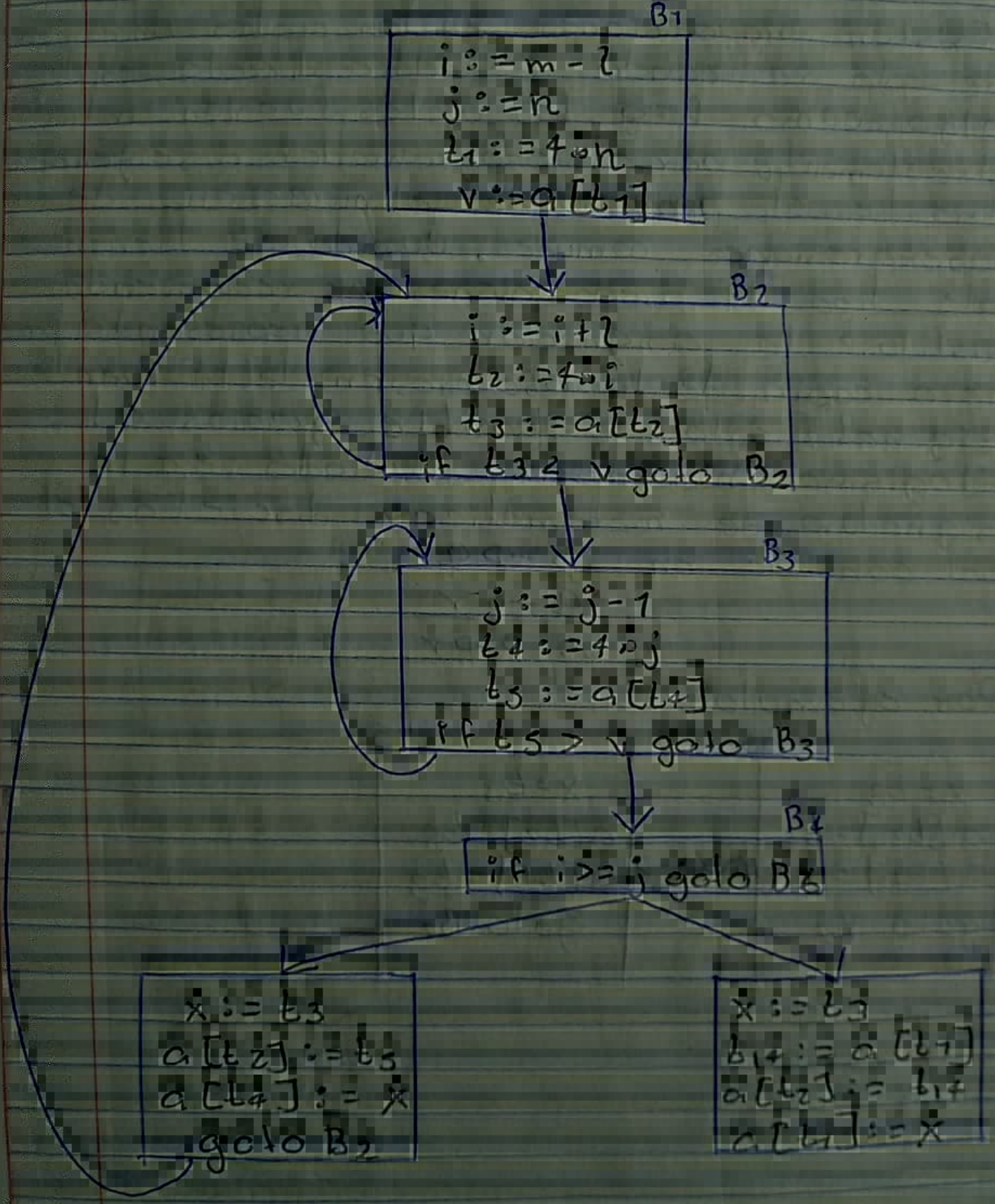
SubExpresiones comúnes

Cuando una expresión contenide en una seeeión o bloque de código ha sido calculado previamente y su valor no ha cambiado desde la última vez que se calculó se dice que esa expresión es una sub expresión común

```
-t6:=4*i
x:=a[t6]
t9:=a[t
```

```
-t6:=4*i
x:=a[t6]
t8:=4*j
t9:=a[t8]
a[t6]:=t9
a[t8]:=x
goto B2
```

Después de eliminación de subexpresiones de forma global

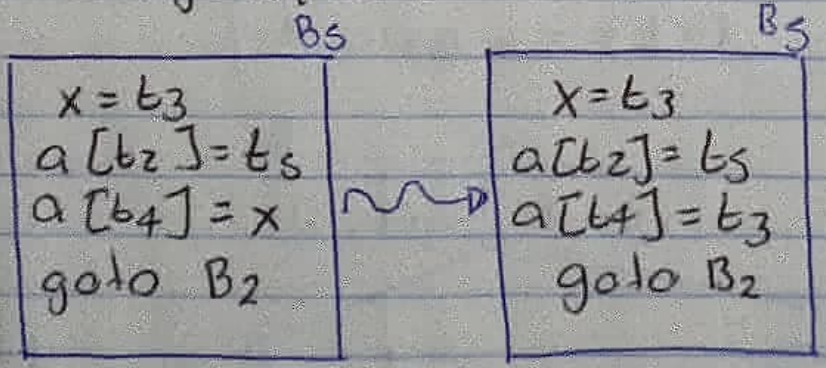


Propagacion de Copias

Existen proposiciones en las cuales únicamente se esta realizando una copia directa, es decir algo como $f=g$ las cuales fueron introducidas directamente por el programador o segeneraren durante el proceso de eliminación de subexpresiones comunes. Cuando se tienen este tipo de preposiciones se recomiende sustituir la asignacion que representa una mera copia de la subexpresion de la fuente original, esto es posible gracias al silogismo hipotético estudiado en la asignatura de matemáticas discretas el cual consiste en

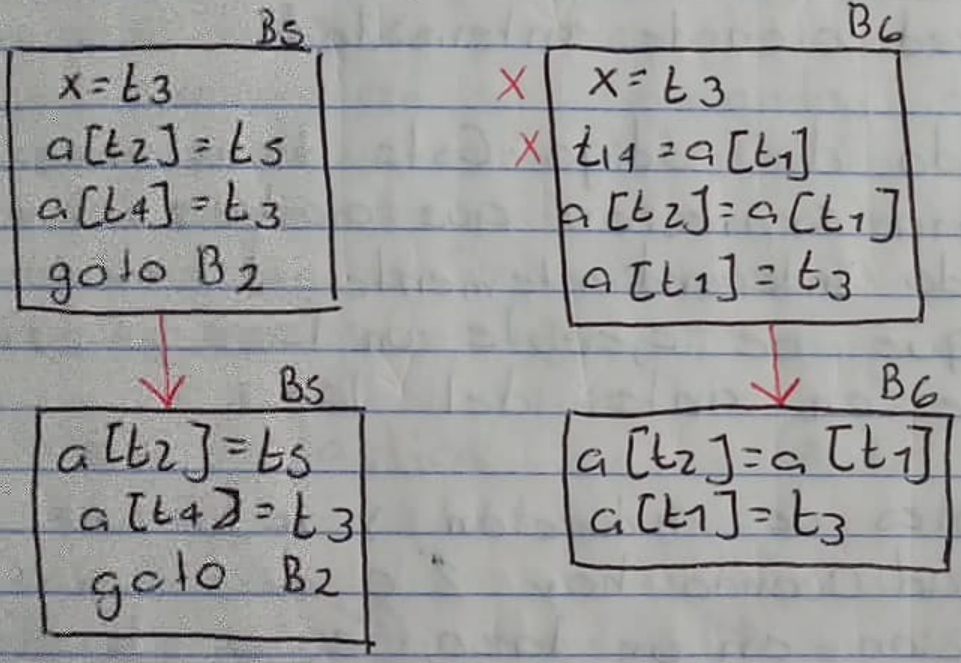
$$\begin{array}{l} p \to q \\ \underline{q \to r} \\ \therefore\ p \to r \end{array}$$

Por ejemplo



Eliminación de código inactivo

Se considera que una proposición o instrucción se encuentra inactiva en un determinado punto si el resultado de su ejecución no es utilizado posteriormente a la ejecución o cálculo de la misma, es decir, si no contribuye al cumplimiento del objetivo del programa a partir de ese punto



Recordemos que todas las técnicas de optimización deben verificarse tanto en el ámbito global como local, nunca debe eliminarse una instrucción si no se ha comprobado que se trata efectivamente de código inactivo en el contexto global del programa ~~que~~ ya que pudiéramos provocar, por ignorancia o desconocimiento el no respetar el primer gran criterio para la optimización de código

Optimización de lazos

El tiempo de ejecución de un programa puede mejorar notablemente si se disminuye la cantidad de instrucciones que forman parte de un lazo, incluso si fuera necesario aumentar el número de instrucciones. Hay 3 importantes para la optimización de lazos: el traslado de código, la eliminación de variables de inducción y la reducción de intensidad.

Traslado de código: Esta transformación toma una expresión que produce el mismo resultado independientemente del número de veces que se ejecute un lazo y coloca la expresión antes del lazo.

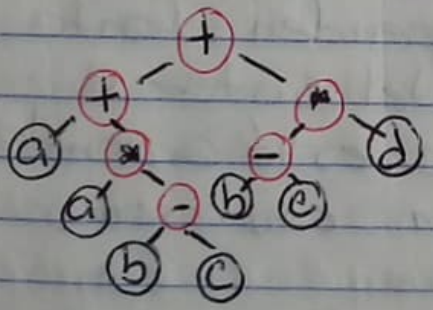
Variables de inducción y reducción de intensidad: Cuando hay 2 o más variables de inducción en un lazo, es posible eliminar todas menos una, mediante el proceso de eliminación de variables de inducción

Optimización de bloques básicos

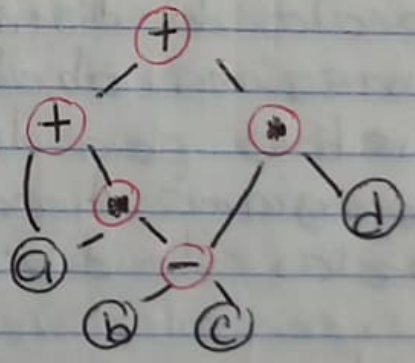
Muchas transformaciones que preservan la estructura se pueden implementar mediante bloques básicos con una técnica conocida como GDA (Grafo Dirigido Asíclico).
Un grafo dirigido asiclico puede equipararse a un árbol sintáctico en el cual se tiene un nodo para toda subexpresión de la expresión; un nodo interior representa operador y sus hijos representan sus operandos, con la diferencia de que en un GDA una subexpresión común se representa como un nodo que tiene más de un padre. Por ejemplo

$$a \underset{5}{+} a \underset{3}{*} (b \underset{1}{-} c) \underset{6}{+} (b \underset{2}{-} c) \underset{4}{*} d$$

Árbol Sintáctico



GDA



Si las expresiones se repiten, se pueden reutilizar

Como podemos observar un GDA nos permite visualizar fácilmente las subexpresiones comunes que pueden ser utilizadas, recordando que para ser consideradas subexpresiones

comunes el resultado de su cálculo no debe variar entre una y otra (optimización) utilización.