

OPTIMIZACIÓN DE CODIGO

JUAN DAVID VELÁSQUEZ HENAO, MSc, PhD

Profesor Titular


Departamento de Ciencias de la Computación y la Decisión


Facultad de Minas


Universidad Nacional de Colombia, Sede Medellín

 jdvelasq@unal.edu.co

 [@jdvelasquezh](https://twitter.com/jdvelasquezh)

 <https://github.com/jdvelasq>

 <https://goo.gl/prkJAq>

 <https://goo.gl/vXH8jy>

OBJETIVOS Y REQUISITOS DE LA OPTIMIZACIÓN DEL CÓDIGO

Busca mejorar la utilización de recursos del programa.

Tiempo de ejecución.

Tamaño del código.

Uso de memoria.

Uso de otros recursos

La optimización no debe afectar lo que hace el programa.

PRINCIPALES OPTIMIZACIONES.

Asignación de registros:

- Ubicación de las variables más usadas en los registros del procesador.

Optimizaciones tempranas:

- Simplificación de expresiones con constantes (*constant folding*).
- Simplificación algebraica.
- Propagación de copias.
- Propagación de constantes (variables que no cambian de valor, *constant propagation*).

Eliminación de redundancias y operaciones innecesarias:

- Eliminación de subexpresiones comunes (evaluaciones repetitivas).
- Reubicación de código invariante en ciclos.
- Eliminación de verificación innecesaria de límites.
- Eliminación de código muerto.

Optimización de subrutinas:

- Optimización Tail-Call y Tail-Recursion.
- Integración de funciones.
- Expansión en línea.

Otras:

- Linealización (código espagueti).
- Simplificación de if con elementos vacíos.
- Simplificación de ciclos.
- Optimización de la memoria.
- Optimización de saltos.

Ejemplo de *tail-recursion*.

Procedimiento recursivo por la cola:

```
int gdc(int u, int v)
{
    if (v == 0) return u;
    return gcd(v, u % v);
}
```

Procedimiento no recursivo por la cola:

```
int fact(int n)
{
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

Procedimiento original:

```
int gcd(int u, int v)
{
    if (v == 0) return u;
    return gcd(v, u % v);
}
```

Procedimiento optimizado:

```
int gcd(int u, int v)
{
begin:
    if (v == 0) return u;
    else
    {
        int t1 = u, t2 = u % v;
        u = t1; v = t2;
        goto begin;
    }
}}
```

CLASIFICACIÓN DE LAS OPTIMIZACIONES

Según el momento en que se realiza la optimización:

A nivel de código fuente (en el árbol sintáctico) – independiente de la máquina.

A nivel del lenguaje intermedio.

A nivel de código destino (*peephole optimization*) – dependiente de la máquina.

Según el lugar donde se realiza la optimización:

Local (aplicada a cada bloque básico de forma aislada).

Global (dentro de un procedimiento).

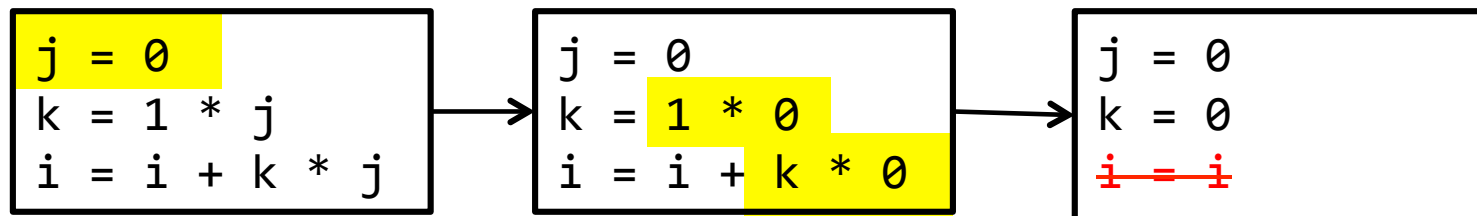
Interprocedural (a nivel de todo el programa)

EJECUCIÓN DE LA OPTIMIZACIÓN.

Cada optimización modifica una pequeña porción del código.

La aplicación de una optimización permite que otra optimización sea ejecutada posteriormente.

El proceso de optimización se repite hasta que no sea posible aplicar ninguna otra optimización.



BLOQUE BÁSICO Y GRÁFICA DE CONTROL DE FLUJO

Son bloques de instrucciones que tienen una única entrada (al inicio) y una sola salida (al final).

Un bloque básico se identifica así:

La primera instrucción inicia un bloque básico.

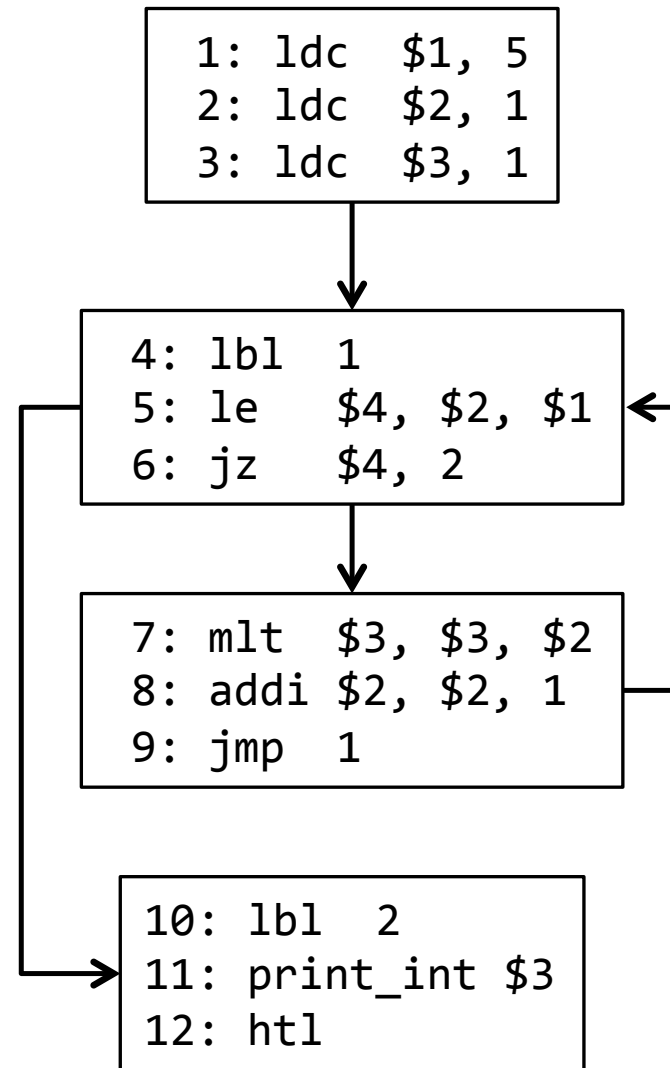
Cada etiqueta (lbl) que es el objetivo de un salto inicia un bloque básico.

La instrucción que sigue un salto inicia un bloque básico.

El conjunto de bloques básicos y las flechas indicando los saltos forman la gráfica de control de flujo.

EJEMPLO

Código	Bloque
1: ldc \$1, 5	1
2: ldc \$2, 1	1
3: ldc \$3, 1	1
4: lbl 1	2
5: le \$4, \$2, \$1	2
6: jz \$4, 2	2
7: mlt \$3, \$3, \$2	3
8: addi \$2, \$2, 1	3
9: jmp 1	3
10: lbl 2	4
11: print_int \$3	4
12: htl	4



ASPECTOS PRÁCTICOS

En la práctica sólo se usa un pequeño grupo de optimizaciones que produzcan el máximo beneficio:

Algunas optimizaciones son muy difíciles de implementar.

Algunas optimizaciones son muy costosas en tiempo de compilación.

Algunas optimizaciones dan muy pocas ganancias.

Muchas optimizaciones cumplen los tres puntos anteriores.

CLASIFICACIÓN DE LAS INSTRUCCIONES

Antes y después de la optimización se presentan los siguientes datos:

total de instrucciones =

+ # total de lecturas de memoria

+ # total de escrituras a memoria

+ # total de saltos

+ # otras instrucciones

OPTIMIZACIÓN LOCAL

Optimizaciones que se realizan sobre bloques básicos durante la compilación.

En la práctica se aplican sobre código de tres direcciones.

Eliminación de instrucciones. (estas instrucciones se pueden eliminar directamente)

$x := x + 0$

$x := x * 1$

$x := x / 1$

$x := x$

Simplificación de instrucciones.

$t1 := t1 * 0 \quad \rightarrow \quad t1 := 0$

Simplificación de expresiones constantes (en tiempo de compilación):

Si

$x := y \text{ op } z \quad \& \quad x := \text{constante} \quad \& \quad y := \text{constante}$

Entonces

El valor de x se puede calcular en tiempo de compilación

Eliminación de bloques básicos no alcanzables.

Elimine todos los bloques básicos no alcanzables por un salto condicional o incondicional. (¿cómo? – plantee el algoritmo.)

Eliminación de subexpresiones comunes.

$x := y + z$		$x := y + z$
\dots	\rightarrow	\dots
$w := y + z$		$w := x$

Eliminación de propagación de copias. (Se reemplaza por la variable original).

$a := x + y$		$a := x + y$
$b := a$	\rightarrow	$b := a$
$w := 2 * b$		$w := 2 * a$

Eliminación de código muerto.

$x := expr$	Si x no se usa en ninguna otra parte del programa
	\rightarrow se puede eliminar.

Ejemplo.

Original

```
y  := a + b
x  := b
z  := a + x
```

Paso 1.

```
y  := a + b
x  := b
z  := a + b    ; propagación de copias
```

Paso 2.

```
y  := a + b
x  := b
z  := y        ; eliminación de subexpresiones comunes
```

Paso 3.

```
y  := a + b
x  := b      ; eliminación de código muerto
z  := y
```


OPTIMIZACIÓN “PEEPHOLE”

Optimizaciones que se realizan a nivel de código assembler.

“Peephole” es una secuencia corta de instrucciones.

CODIGO DE TRES DIRECCIONES.

Considere el siguiente fragmento de código de un programa:

```
x := x + 1;
```

En código de tres direcciones sería:

```
mem2reg $1 1  
loadc   $2 1  
add     $1 $1 $2
```

Sin embargo, existe instrucciones que permiten la operación directa con una constante:

$$\text{addi } R_{\text{dest}} \ R_{\text{src}} \ n \quad \rightarrow \quad R_{\text{dest}} := R_{\text{src}} + n$$

Más aún:

$$\text{addi } R_{\text{dest}} \ R_{\text{src}} \ 0 \quad \rightarrow \quad \text{mov } R_{\text{dest}} \ R_{\text{src}}$$

Ejemplos.

```
...
jz  L1
jmp L2
L1:
<bloque 1>
L2:
<bloque 2>
...
→
...
jnz L2
<bloque 1>
L2:
<bloque 2>
...
```

`addi $1 $1 n; addi $1 $1 m → addi $1 $1 n + m`

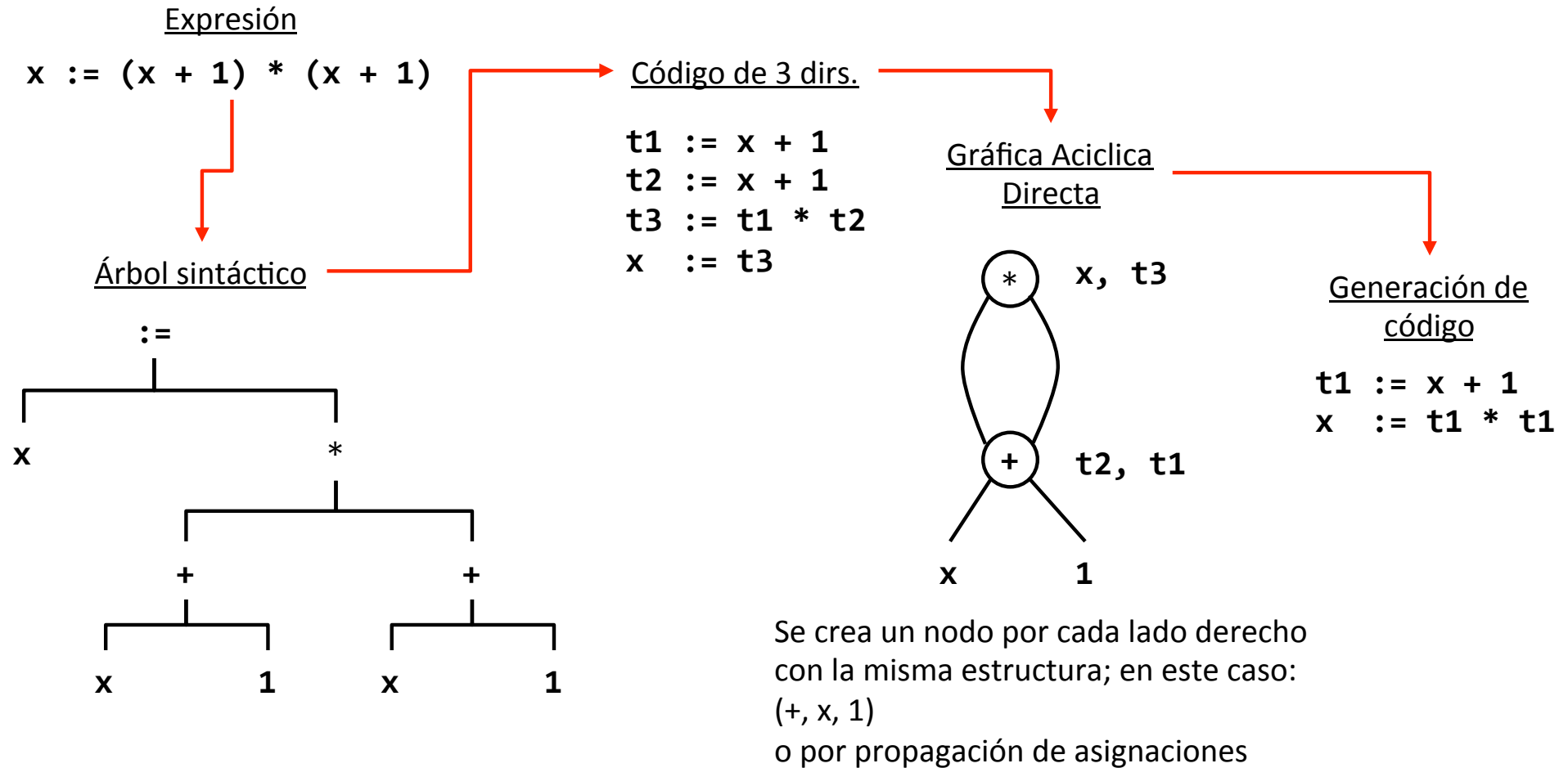
`addi $1 $1 0 → mov $1 $1 → se elimina`

<pre> . . . jmp L1 . . . L1: jmp L2 <bloque 1> L2: <bloque 2> . . . </pre>	→	<pre> . . . jmp L2 . . . L1: jmp L2 <bloque 1> L2: <bloque 2> . . . </pre>
--	---	--

<pre> jmp L1 . . . L1: <cond> jnz L2 L3: . . . </pre>	<pre> <cond> jnz L2 . . . L3: . . . </pre>
---	--

ANALISIS DE FLUJO DE DATOS

GRAFICA ACICLICA DIRECTA*



* Permite la eliminación de subexpresiones comunes, almacenamientos redundantes y reorganización de código.

ANÁLISIS DEL PROXIMO USO.

Determina sobre que porciones de código una variable no cambia de valor (se comporta como una constante).

La expresión

$$a : = x + y$$

Es una definición de a ; es un uso de x y de y .

TAIL-CALL Y TAIL-RECURSION

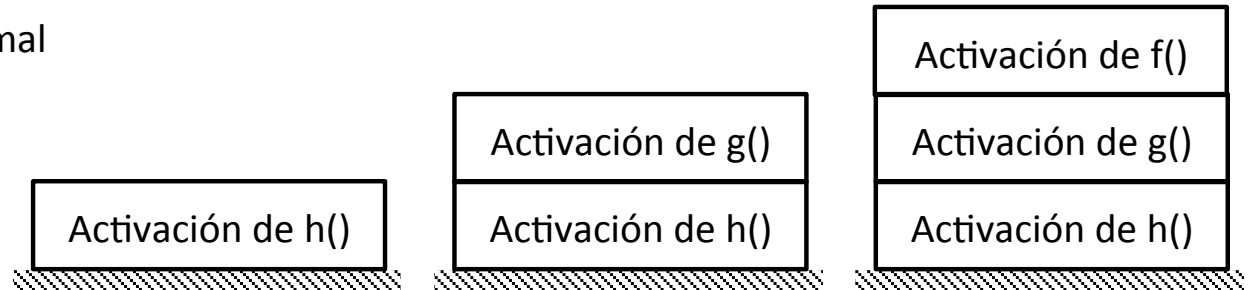

```
function f(x)
{
...
return z;
}
```

```
function g(x)
{
...
return f(z);
}
```

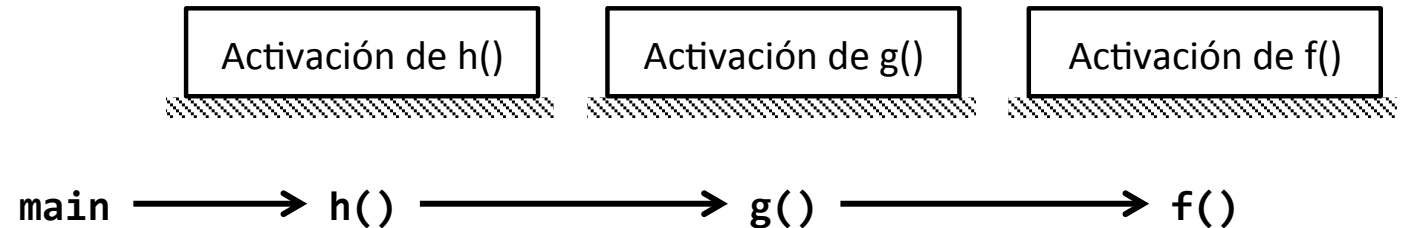
```
function h(x)
{
...
return g(z);
}
```

h(3)

Proceso normal



Proceso optimizado: se reemplaza la activación (no se apila)



OPTIMIZACIÓN DE CODIGO

Juan David Velásquez H., MSc, PhD

Profesor Titular

Departamento de Ciencias de la Computación y la Decisión

E-mail: jdvelasq@unal.edu.co

URL: www.docentes.unal.edu.co/jdvelasq