Compiladores e Intérpretes Optimización

Sebastian Gottifredi

Universidad Nacional del Sur
Departamento de Ciencias e Ingeniería de la Computación
2018

Repaso

Repaso

- Para entender y controlar la **estructura** de un programa fuente hay que analizar si sigue las reglas de **sintaxis** del lenguaje
- El **Analizador Léxico** es el encargado armar los **tokens**
- Para expresar las reglas de sintaxis del lenguaje utilizamos **gramáticas libres de contexto**
- El **analizador sintáctico** es el encargado de **reconocer** si un programa sigue esas expresadas por la gramática

Repaso

- El análisis semántico es el encargado validar y entender el **significado** del programa
- Para esto el analizador semántico debe:
  - Recolectar, entender y controlar todas las entidades declaradas

    Chequeo de Declaraciones
  - Recolectar, entender y controlar todas las sentencias asociadas a las entidades recolectadas

    Chequeo de Sentencias

Repaso

- Para las tareas del análisis semántico y optimización suele ser necesaria una **representación alternativa del fuente** estructuralmente adecuada para trabajar con toda la **información que recolecto** el compilador
- Estas formas son conocidas como

Representaciones Intermedias (RI)

  - Son **representaciones alternativas** del fuente que aun **no son código maquina**

Repaso

- **Basadas en Grafos**: codifican la información recopilada en una estructura de grafo. Los algoritmos de están asociados al recorrido de componentes como nodos, arcos, listas o arboles.

Arboles Sintácticos Abstractos (AST)

Grafos de Flujo de Control

Grafos Dirigidos Aciclicos (DAG)

Grafos de Dependencias

Grafos de Llamadas

Repaso

- **Lineal**: se asemeja al código para alguna maquina abstracta o virtual. Los algoritmos hacen una simple iteración sobre una secuencia lineal de operaciones/instrucciones.

Tres Direcciones

Basados en Pila

Dos Direcciones

Optimización

Optimización

- La **optimización** busca que la **ejecución** del programa resultante **mejore** el **uso de los recursos**:
  - Tiempo de ejecución
  - Espacio en memoria/disco
  - Temperatura,
  - Uso de la red, etc
- La **optimización** no debe alterar el resultado esperado de la computación.
  - Es decir las respuestas del sistema en términos lógicos deberían ser las mismas que sin optimizar

Tipos de Optimizaciones

- La **optimización** puede dividirse en **dos grandes tipos**:
- Optimización sobre **código intermedio** o de alto nivel

  Considera la estructura del programa fuente, sus operaciones y como es el flujo de control dentro de sus unidades
- Optimización sobre **código maquina** o de bajo nivel

  Considera las características de la arquitectura, mircoprocesador, memoria

Optimizaciones de Código Intermedio

- Hay **tres niveles de granularidad** con los que se puede encarar la optimización en un compilador
  - **Nivel Local:** estudian como mejorar bloques de sentencias sin cambios de flujo de control interno (bloques básicos)
  - **Nivel Globales:** estudian como mejorar el cuerpo completo de una unidad de manera aislada
  - **Nivel Interproceso:** estudia como mejorar el código considerando la interacción entre las unidades del programa
- Los primeros **dos niveles** son los **mas comunes**

Optimizaciones de Código Intermedio

- En la practica **no todas las optimizaciones**, por mas ganancia que otorguen, **son aplicadas**
- ¿Por qué?
  - Algunas son **difíciles de implementar**
  - Otras son **muy costosas** en tiempo de **compilación**
  - Otras tienen **beneficios circunstanciales**
- La idea es usar técnicas de optimización que **maximicen la ganancia** por el **menor costo posible**

Grafo de Flujo de Control

- Las **optimizaciones Locales y Globales** se llevan se realizan analizando el **grafo de control de flujo**
- En este grafo los **nodos** representan **computaciones secuenciales** sin cambios de flujo de control mas allá de la ejecución secuencia de sus instrucciones
- Los **arcos salientes/entrantes** de/a un nodo representan **cambio de flujo de control**
- Los nodos son llamados **Bloques Básicos**

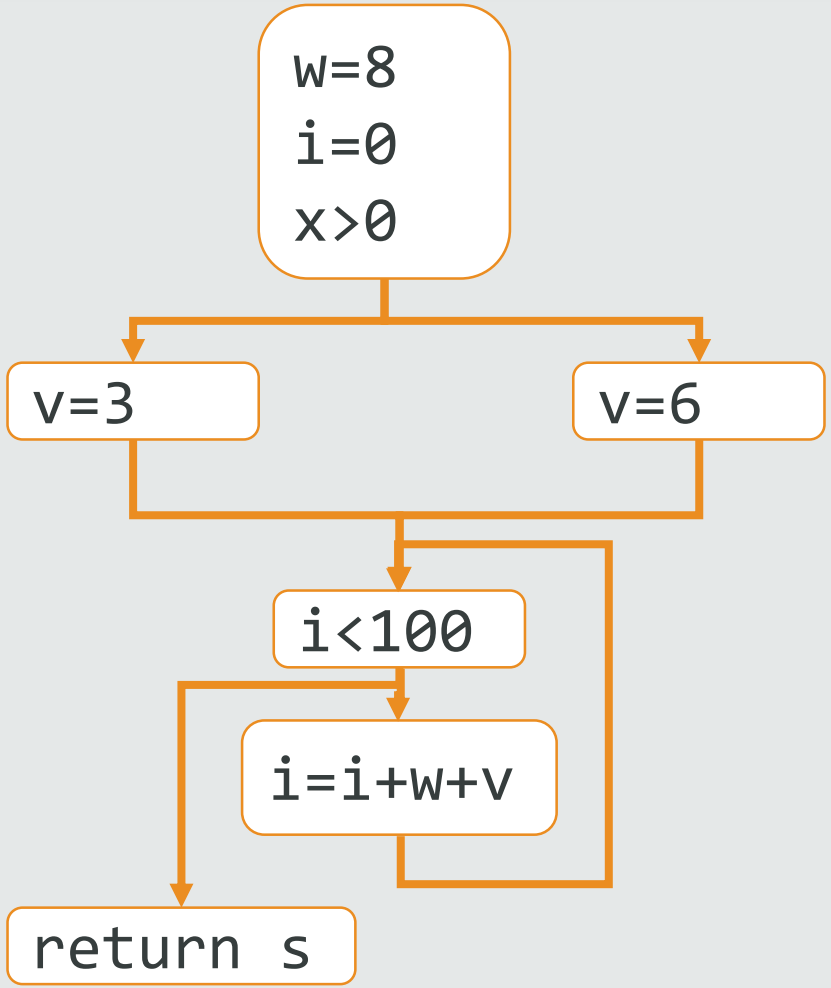
Grafo de Flujo de Control

- **Bloque básico** es una secuencia de instrucciones que:
  - **No** hay **cambios de flujo** de control **salientes** desde el **medio del bloque**
  - **No** hay **cambios de flujo** de control **entrantes** al **medio del bloque**
  - Son **maximales** respecto a la cantidad de instrucciones
- Los **arcos entrantes** representan **de donde puede venir** el flujo de control antes de empezar la computación del bloque básico
- Los **arcos salientes a donde va** el flujo de control al terminar la computación del bloque básico

Grafo de Flujo de Control

- Por ejemplo

```
int m1(int x){
   w=8
   i=0
   if(x>0) v=3
   else v=6
   while(i<100) i=i+w+v
   return i
}
```



Optimizaciones Locales

Optimizaciones Locales

- Operan sobre los **bloques básicos** de código
- **No** necesita analizar **todo el cuerpo de una unidad**
  - Trabaja **optimizando bloques básicos** de manera aislada
- Se vale de la **naturaleza secuencial de ejecución** que van a tener las instrucciones del bloque básico
- La idea en general es **reusar todo lo posible** y **eliminar lo no usado**
- Es la **forma mas simple** de optimización

Eliminación de sub-expresiones comunes

- La idea es **reutilizar valores** previamente calculados para **eliminar sub expresiones** por completo
- Para esto cada vez que se realiza una **asignación** se guarda el **valor calculado** en un **temporal**
- Estos **temporales reemplazaran una subexpresion** con el mismo valor virtual que el de otra que ya hayamos calculado

```
a = c+d
b = c+d
```

```
a = c+d
t1= a
b = t1
```

Eliminación de sub-expresiones comunes

- Por ejemplo:

```
x = 10+a
y = x+b+d
y = 10
z = (x+b+d)*(y+a)
```

```
x = 10+a
t1= x
y = x+b+d
t2= y
y = 10
t3= y
z = (t2)*t1
```

Eliminación de su

Esta técnica introduce muchos temporales y asignaciones a esos temporales!...
Vamos que mucho de esto se puede eliminar ☺

- Por ejemplo:

```
x = 10+a
y = x+b+d
y = 10
z = (x+b+d)*(y+a)
```

```
x = 10+a
t1= x
y = x+b+d
t2= y
y = 10
t3= y
z = (t2)*t1
```

Podemos reemplazar aun cuando la variable sobre la que se hizo el calculo cambio su valor!

Podemos reemplazar aun cuando la subexpresion usa distintas variables

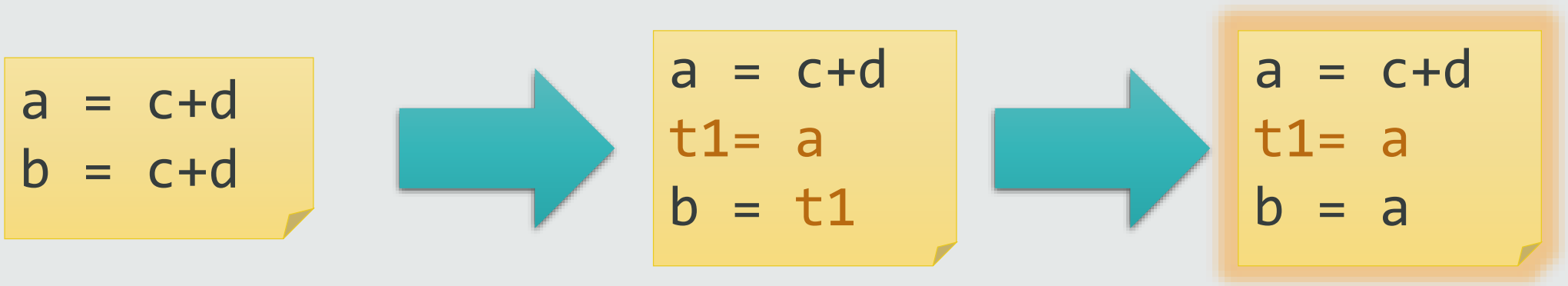
Eliminación de sub-expresiones comunes

- Para **implementar** recorriendo el bloque básico **almacenando los valores virtuales** producidos por cada asignación

- Cuando se **procesa una expresión** se controla por cada expresión si se **corresponde** con algún **valor virtual previamente calculado**, de ser así se **reemplaza**

Propagación por Copia

- Esta técnica se aplica **después de la anterior**
- Busca usar la **menor cantidad** de **temporales** posibles:
  - Si la **variable** de la que estamos usando un temporal **no cambio su valor "en el camino"**, entonces **en vez** de usar el **temporal** directamente **uso la variable**



Propagación por Copia

```
x = 10+a
t1= x
y = x+b+d
t2= y
y = 10
t3= y
z = (t2)*t1
```

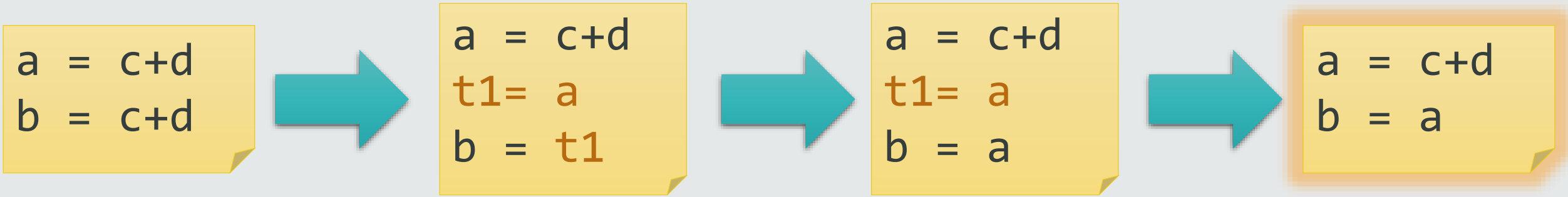
```
x = 10+a
t1= x
y = x+b+d
t2= y
y = 10
t3= y
z = (t2)*x
```

Propagación por Copia

- Esta técnica se **implementa recorriendo** el código del bloque básico **en orden**
- Se va llevando **una estructura de mapeo bidireccional** de cada **variable** con el **ultimo temporal asociado**
- Cuando nos **encontramos** con un **temporal t** en una expresión, vemos **si t esta asociado a una variable v** en el mapeo, en caso **afirmativo reemplazamos t por v**

Eliminación de Código Muerto

- Al usar las dos técnicas anteriores **nos quedan muchos temporales no utilizados**
- La **eliminación de código muerto** básicamente elimina todas las **asignaciones de temporales** que **no** son **utilizadas**



Eliminación de Código Muerto

```
x = 10+a
t1= x
y = x+b+d
t2= y
y = 10
t3= y
z = (t2)*x
```

```
x = 10+a
y = x+b+d
t2= y
y = 10
z = (t2)*x
```

Eliminación de Código Muerto

- Esta técnica se **implementa recorriendo** en **orden inverso** el bloque básico
- Se van **recolectando los temporales** que son **utilizados**
- Si nos topamos con la **asignación de un temporal** que **no** fue **utilizado**, se **elimina** la instrucción

Simplificación Algebraica

- La idea esa **usar propiedades** básicas de **algebra y lógica** para **reducir expresiones**
- Por ejemplo:

```
a = a * 0
a = b * 1
x = r + 0
w = m / 1
j = 0 - x
x = x + -y
```

```
a = 0
a = b
x = r
w = m
j = -x
x = x - y
```

Simplificación Algebraica

- La idea esa **usar propiedades** básicas de **algebra y lógica** para **reducir expresiones**
- Por ejemplo:

```
a = true || w
x = false && y
o = y || false
w = true && v
```

```
a = true
x = false
o = y
w = v
```

Simplificación Algebraica

- Otro tipo de **simplificaciones** pueden involucrar:
  - Transformar **potencias** en **multiplicaciones**
  - Transformar **multiplicaciones** en **sumas**

- En general si se conoce la **maquina** sobre la que se va a **generar el código** se puede **tomar ventaja** de estas otras **simplificaciones**.

Optimizaciones Globales

Optimizaciones Globales

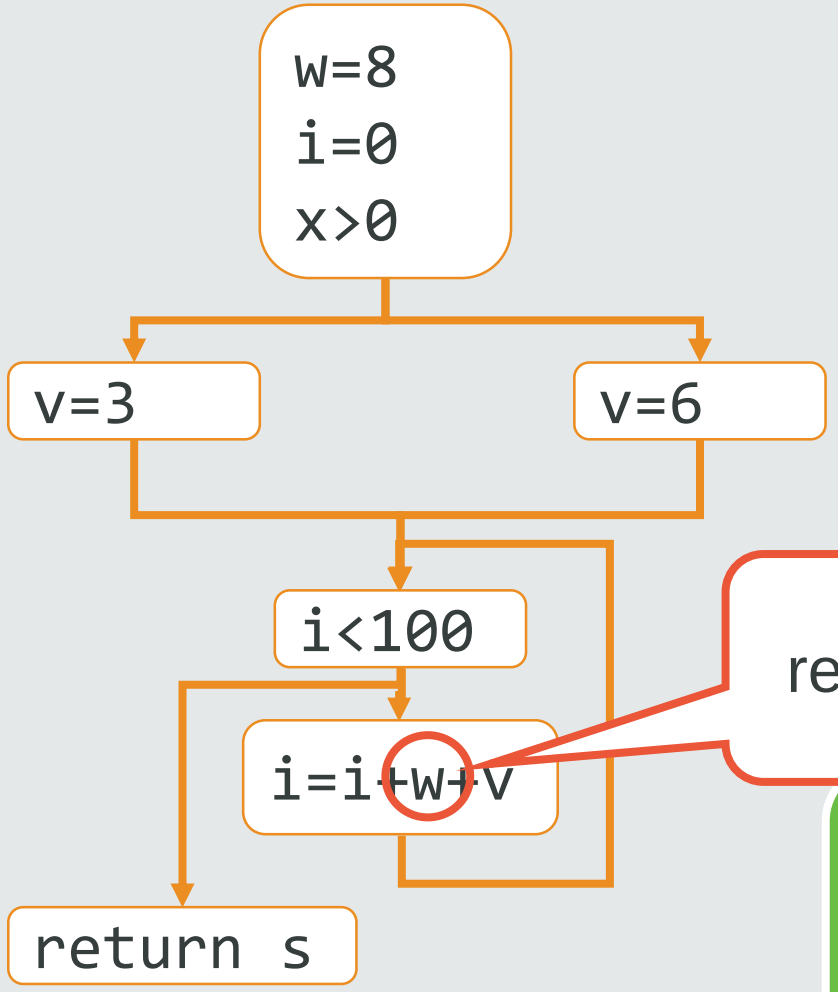
- Estos métodos usan el **contexto completo del método/función**
- Para ellos se construye el **grafo de flujo de control**
- Se analiza como ciertas **acciones** en **un bloque básico** pueden **ahorrarnos computación** en otros bloque básicos
- En general **primero se analiza** todo el flujo de control y **después** se **aplican** las correspondientes **transformaciones**

Propagación de Constantes

- La idea es **reemplazar accesos a variables** por los **valores constantes** que tienen **asignados**
- Para esto **recorreremos** los bloques básicos (en orden) **buscando** lugares donde **podamos reemplazar** variables por **valores ya calculados**
- En esta técnica debemos **considerar** los **cambios en el flujo de control!!!**

Propagación de Constantes

- Por ejemplo

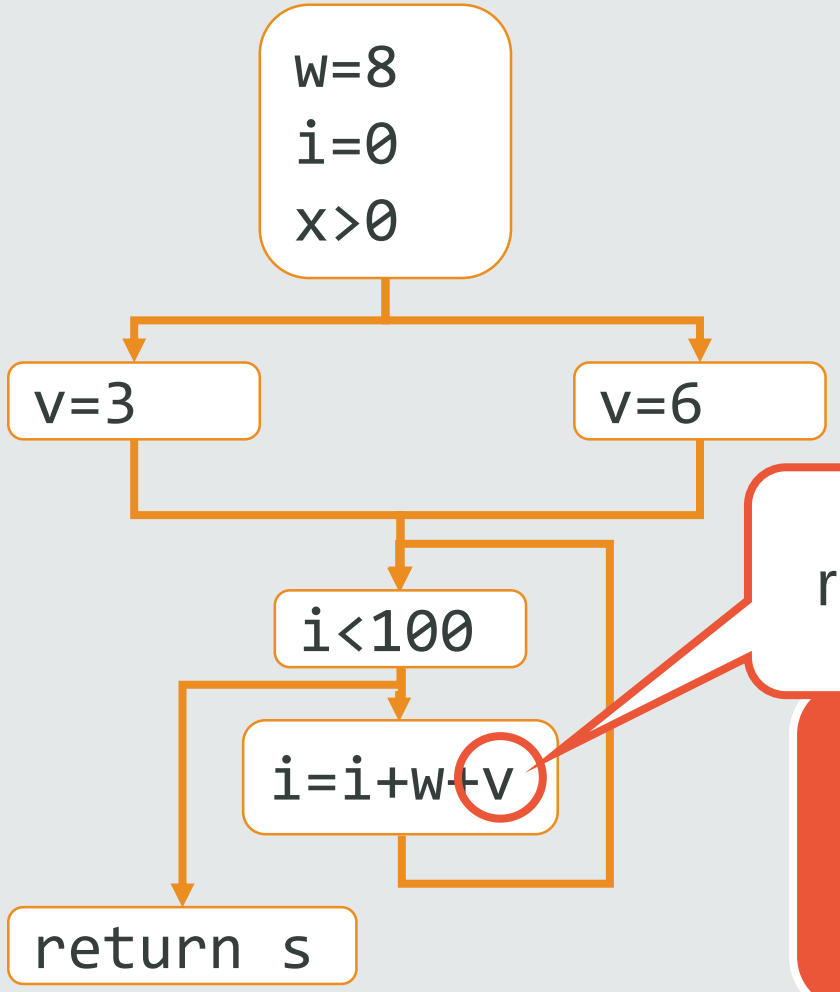


¿Podemos reemplazar w por un valor constante?

SI por 8! Por que en todos los caminos de flujo de control hasta este punto w se mantiene con 8

Propagación de Constantes

- Por ejemplo



¿Podemos reemplazar v por un valor constante?

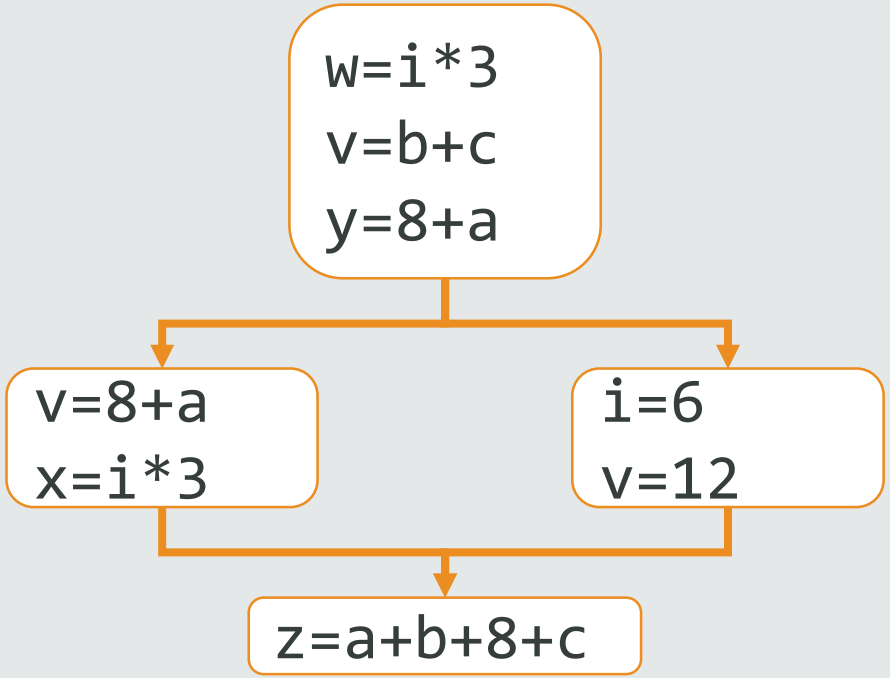
No!!! Por que en un camino toma valor 3 y en otro toma valor 6

Reutilización de Expresiones

- Esta técnica busca reutilizar **expresiones previamente calculadas**
- Es **similar** a la **técnica local** de eliminación de sub-expresiones comunes pero se aplica sobre todo el **cuerpo de la función**
- Buscaremos **reemplazar subexpresiones** completas por **variables**
- La idea **reusar** una expresión siempre y cuando podamos **asegurar** que las **variables en** la **expresión no cambiaron** entre su calculo y el punto de reúso

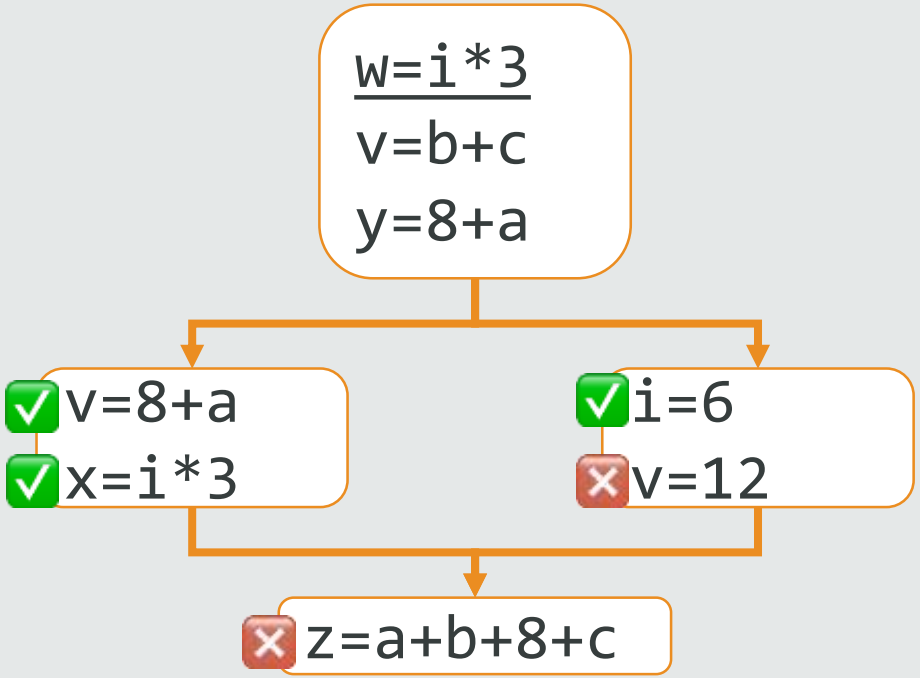
Reutilización de Expresiones

- Por ejemplo:



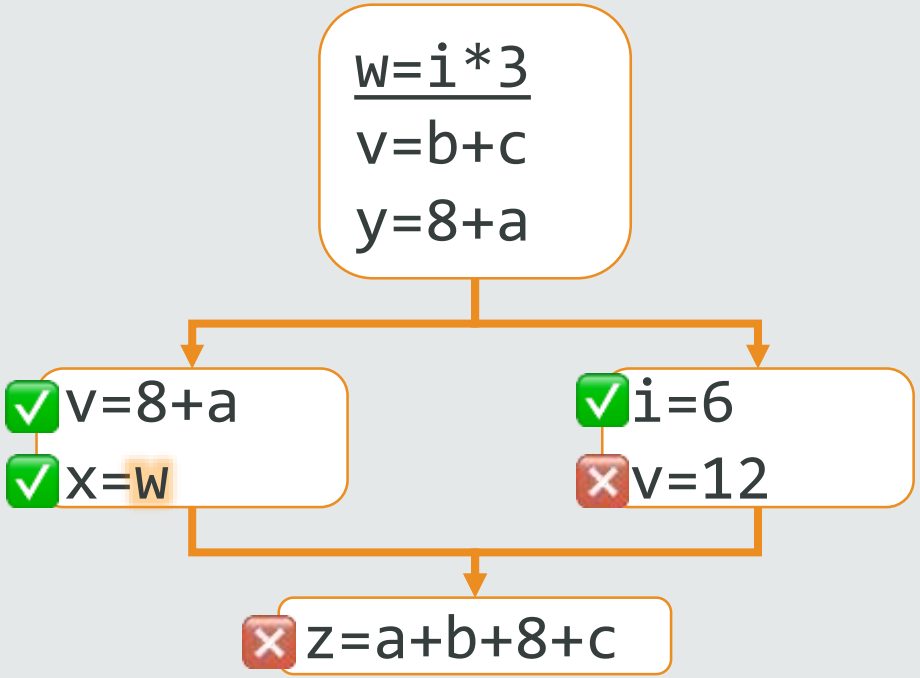
Reutilización de Expresiones

- Por ejemplo:



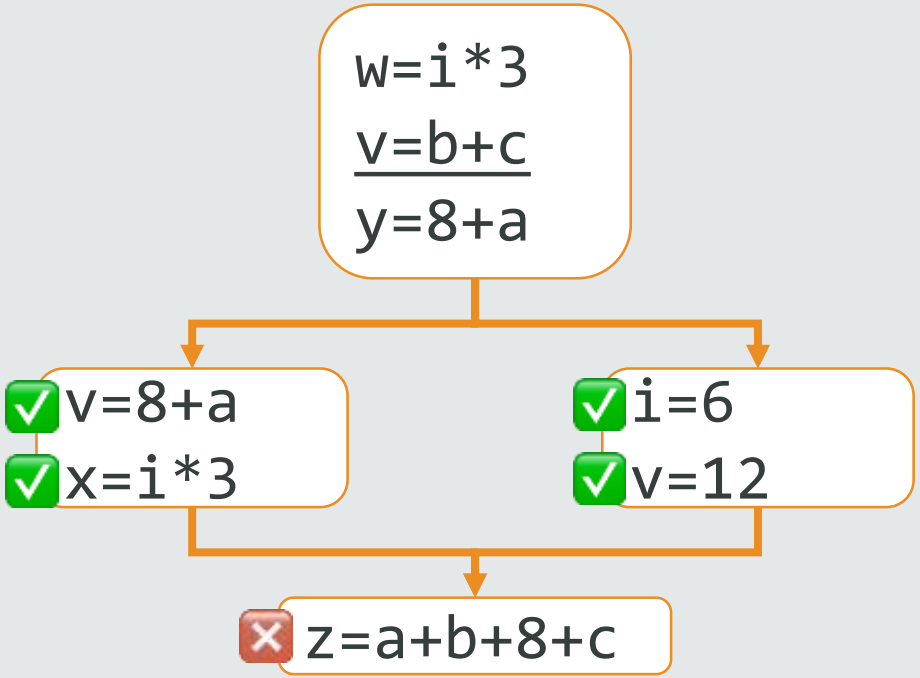
Reutilización de Expresiones

- Por ejemplo:



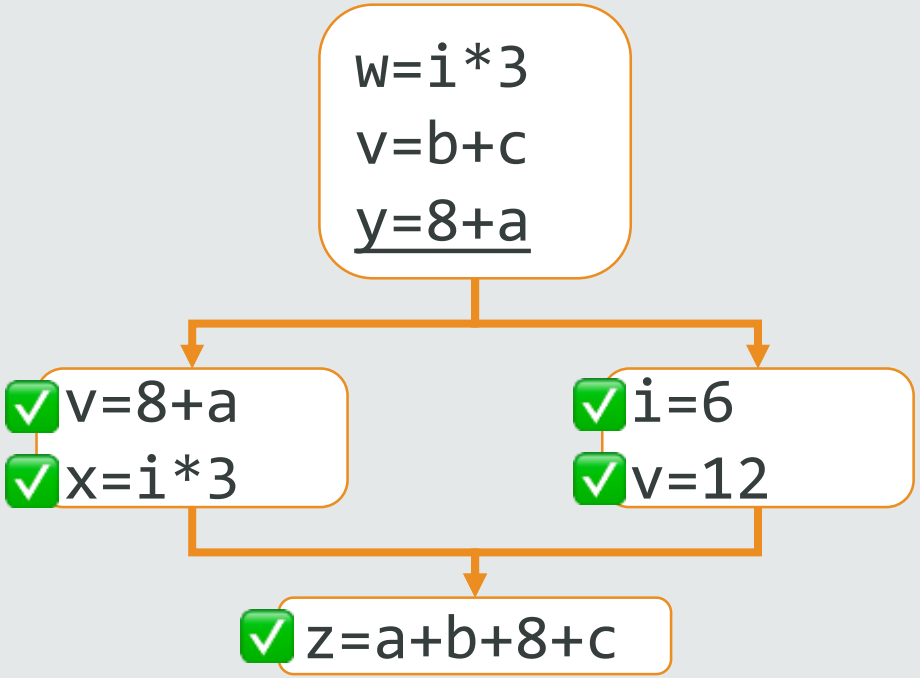
Reutilización de Expresiones

- Por ejemplo:



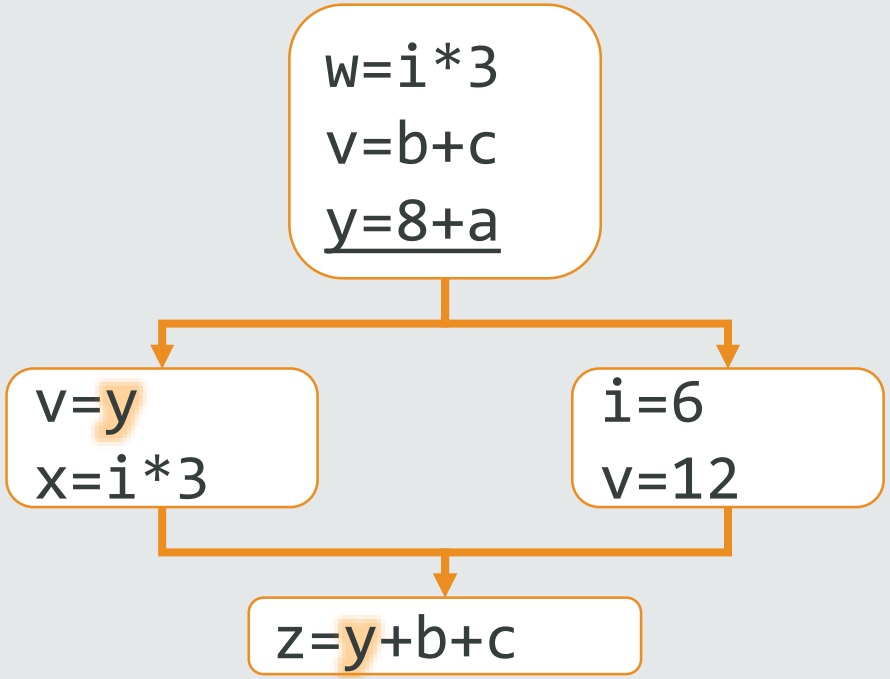
Reutilización de Expresiones

- Por ejemplo:



Reutilización de Expresiones

- Por ejemplo:



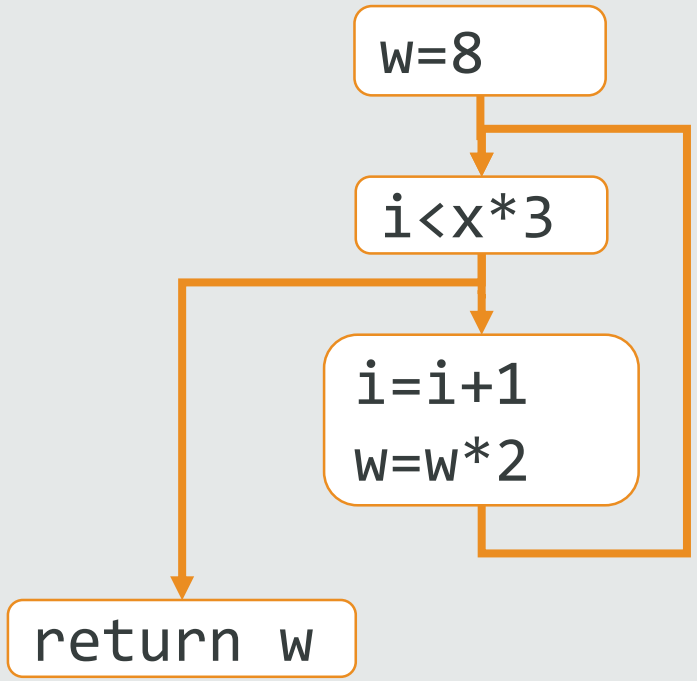
Relocación de Código

- Busca **reubicar código** en sentencias de **repetición**
- En los ciclos suele haber **código** que **no es afectado** por la **repetición**
- La idea de la técnica es identificar ese código y de ser posible **moverlo fuera del bucle**
- Un ejemplo usual suele estar vinculado a una parte del código para el **computo de la expresión de evaluación** del ciclo

Relocación de Código

- Por Ejemplo:

```
int m1(int x){
   w=8
   while(i<x*3){
      i = i+1
      w = w*2
   }
   return w
}
```



Relocación de Código

- Por Ejemplo:

```
int m1(int x){
   w=8
   while(i<x*3){
      i = i+1
      w = w*2
   }
   return w
}
```