Tema 12 | Optimización de Código

Contenidos

Bibliografía básica

31-May-2006

[Aho90] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman
Compiladores. Principios, técnicas y herramientas. Addison-Wesley Iberoamericana 1990.

[Trem85] J Tremblay, P.G. Sorenson
The theory and practice of compiler writing. McGraw-Hill, 1985.

12.1 Función del Optimizador de Código

- La optimización de código no es un problema fácil, y depende de cual sea el nivel deseado de optimización.

- Toda optimización conlleva la transformación de un código en otro más eficiente.

- ***Requisitos que debe cumplir toda optimización*.**
  - **Ser mesurable la optimización obtenida mediante la transformación.**
  - **El esfuerzo de transformar para optimizar debe compensar con la optimización conseguida.**
  - **Mantener el significado de en la transformación de optimización**.

- ***¿Cómo se puede mejorar el rendimiento?.***
  - Analizando los Flujos de Control.
  - Analizando los Flujos de Datos.

12.2 Tipos de Optimizadores de Código (1)

- **Peephole.**
  - Consiste en obtener un lenguaje transformado con un número menor de instrucciones.
- **Independiente de la Máquina.**
- **Dependiente de la máquina.**

12.2 Tipos de Optimizadores de Código (2)

Optimización Independiente de la Máquina.

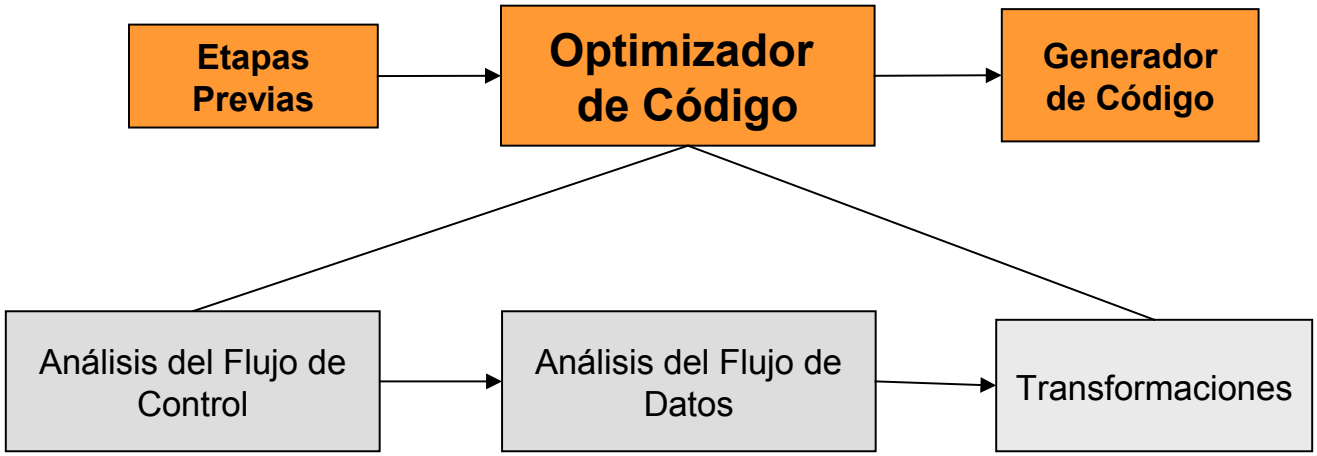
- ***REDUCCIÓN SIMPLE***:
  - **Evaluar expresiones cuando son conocidos sus operandos en tiempo de compilación.**
- ***REACONDICIONAMIENTO DE LAS INSTRUCCIONES***:
  - **Reducir el número de temporales.**
- ***ELIMINAR REDUNDANCIAS.***
- ***REDUCCIÓN DE POTENCIA***:
  - **Sustituir operaciones software por operaciones Hardware:Ejemplo la potencia de 2, se puede sustituir por desplazamientos.**
- ***REDUCIR LA EJECUCIÓN ITERATIVA DE BLOQUES INNECESARIOS***:
  - **Sacar bloques fuera de los lazos.**

12.2 Tipos de Optimizadores de Código (3)

Optimización Dependiente de la Máquina:

- Reasignar los registros de internos de la máquina adecuadamente.
- Usar operaciones especiales de la máquina para expresar operaciones complejas.
- Utilizar los recurso y periféricos de la máquina más adecuados para cada momento de ejecución.

12.3 Organización de los Compiladores Optimizados



12.4 Principales Fuentes de Optimización (1)

Las transformaciones puede ser:

- Locales al Bloque Básico (ya estudiadas en la fase generación de código)
- Globales.

Tipo de transformaciones:

- Transformaciones que preservan la función *(a nivel global y local).*
- Transformaciones que mejoran el código (*a nivel global).*

12.4 Principales Fuentes de Optimización (2)

Transformaciones que preservan la función (a nivel local):

- Propagación de copia.
- Eliminación de código inactivo.
- Cálculo previo de constantes.
- Eliminar subexpresiones comunes
- Optimización de constantes.

12.4 Principales Fuentes de Optimización (3)

Transformaciones que preservan la función (a nivel global):

- Optimización de los Bloques Básicos:
  - *Simplificación Algebraica.*
  - *Mediante GDAs.*

12.4 Principales Fuentes de Optimización (4)

Transformaciones que mejoran código (a nivel global)

Se expresan en base a los siguientes conceptos:

- **Alcance:** Una expresión alcanza un punto ***P*** (está disponible) si todo camino desde el punto inicial hasta el punto ***P***, es evaluada y no hay asignaciones sobre los operandos en dicha expresión hasta el punto ***P***.
- **Punto:** Representa la adyacencia entre dos posiciones.
- **Camino:** Representa una secuencia de puntos.
- **Activación y desactivación de variables:** Cuando una variable es evaluada se dice que está activada hasta que otra vez se le vuelva asignar valor. En este punto se dice que la variable se desactiva, aunque de nuevo se vuelve a activar.

12.4.1 Transformaciones que Mejoran Código (1)

ELIMINACIÓN DE SUBEXPRESIONES COMUNES:

Conociendo las expresiones que alcanzan un determinado punto ***P***, se conoce el conjunto de subexpresiones comunes al punto ***P***.

PROPAGACIÓN DE COPIA:

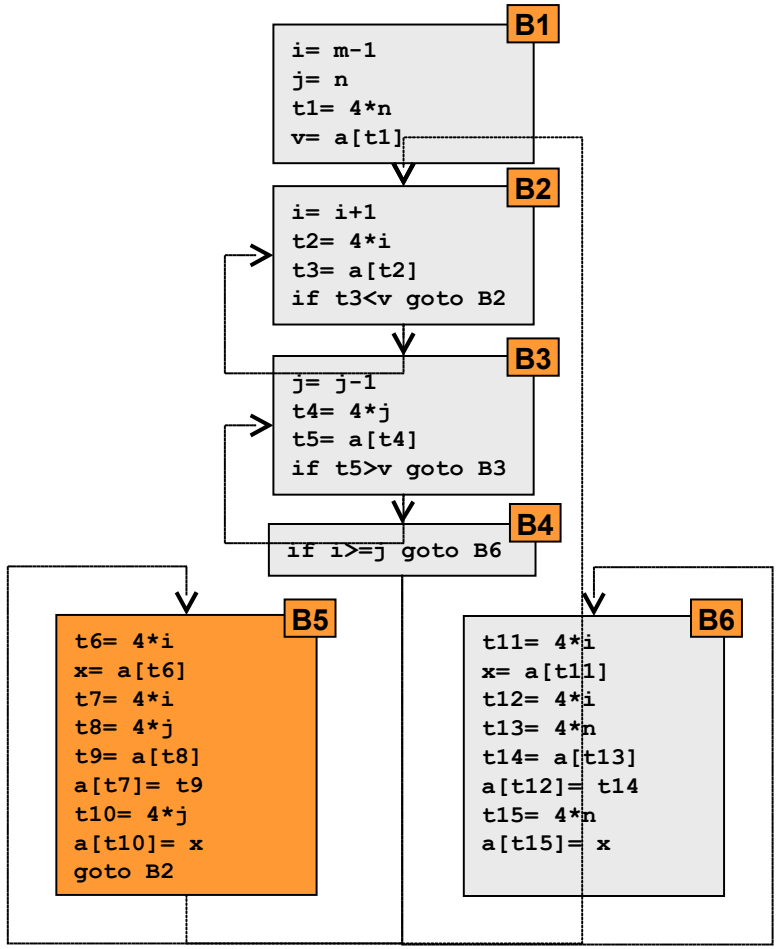
(Es un efecto colateral de la técnica de generación de código).

Para sustituir una variable **X** por **Y**, cuando aparece **X= Y** se debe cumplir:

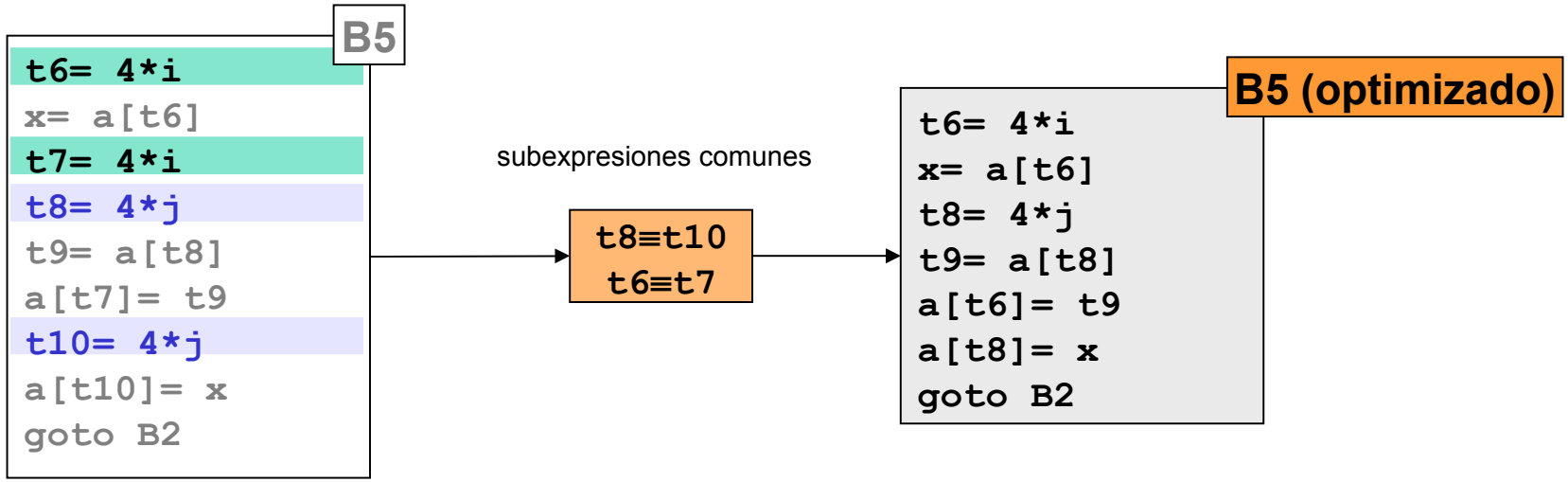
1. La proposición **X= Y** sea la única definición de **X**, y
2. Si en los demás caminos no hay nueva asignación de **Y**.

12.4.1 Transformaciones que Mejoran Código (2)

```
(1)   i= m-1
(2)   j= n
(3)   t1= 4*n
(4)   v= a[t1]
(5)   i= i+1
(6)   t2= 4*i
(7)   t3= a[t2]
(8)   if t3<v goto 5
(9)   j= j-1
(10)  t4= 4*j
(11)  t5= a[t4]
(12)  if t5>v goto 9
(13)  if i>=j goto 23
(14)  t6= 4*i
(15)  x= a[t6]
(16)  t7= 4*i
(17)  t8= 4*j
(18)  t9= a[t8]
(19)  a[t7]= t9
(20)  t10= 4*j
(21)  a[t10]= x
(22)  goto 5
(23)  t11= 4*i
(24)  x= a[t11]
(25)  t12= 4*i
(26)  t13= 4*n
(27)  t14= a[t13]
(28)  a[t12]= t14
(29)  t15= 4*n
(30)  a[t15]= x
```



12.4.1 Transformaciones que Mejoran Código (3)



Además:

```
t8= 4*j
t9= a[t8]
a[t8]= x
```

pueden sustituirse por:

```
t9= a[t4]
a[t4]= x
```

utilizando **t4** obtenido en **B3**

B3

```
j= j-1
t4= 4*j
t5= a[t4]
if t5>v goto B3
```

12.4.1 Transformaciones que Mejoran Código (4)

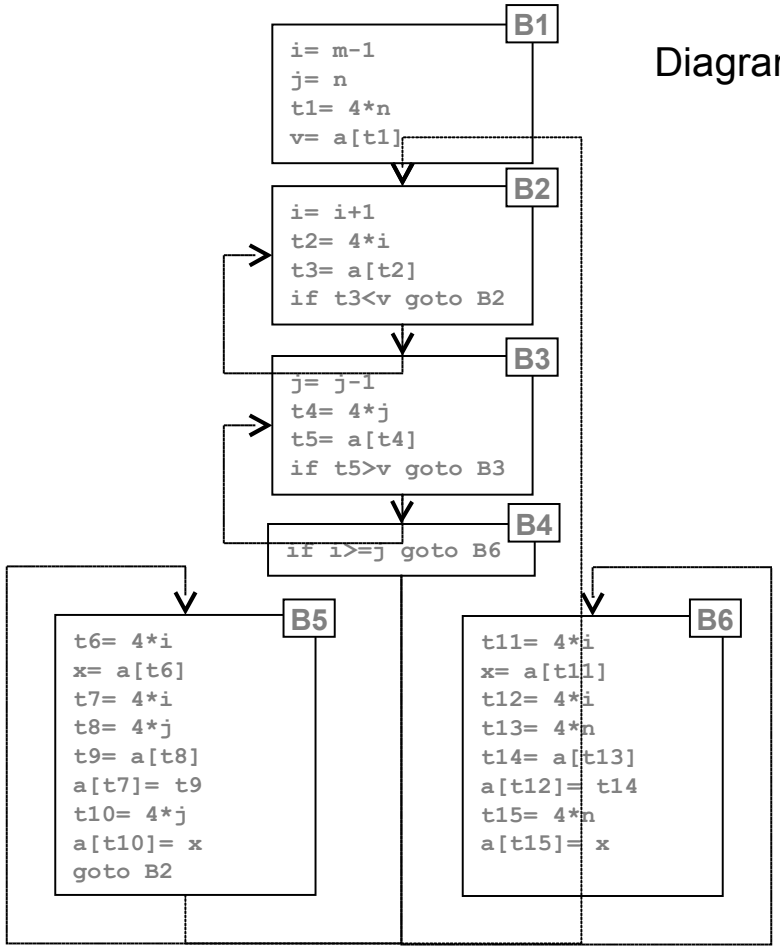
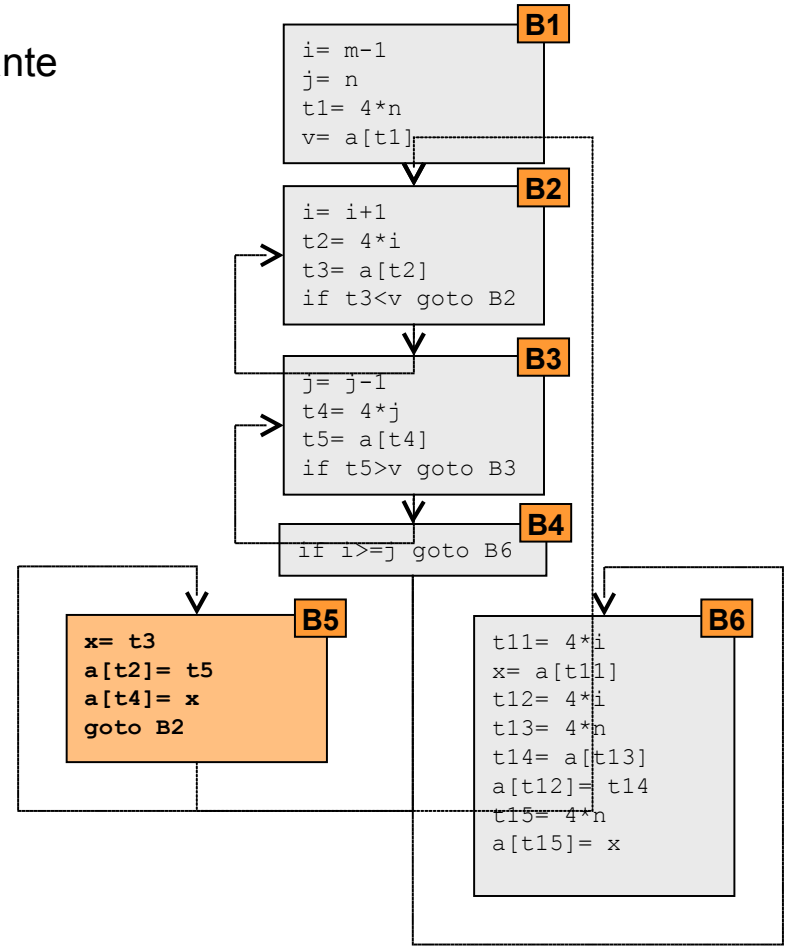


Diagrama de flujo resultante



12.4.1 Transformaciones que Mejoran Código (5)

Eliminación de código inactivo.

Una variable es activa si se le asigna valor y es usada posteriormente. Si una variable tiene un valor asignado pero no es utilizada después, entonces, es considerada como código inactivo.

Optimización de Lazos.

Existen tres técnicas para optimizar lazos:

- Traslado de código
- Eliminación de variables de inducción.
- Reducción de intensidad.

12.4.1 Transformaciones que Mejoran Código (6)

***Traslado de código*:**

Calcular los invariantes del ciclo y sacarlos fuera del mismo, formando parte de lo que se denomina *Preencabezamiento*.

***Eliminación de variables de inducción y Reducción de intensidad*:**

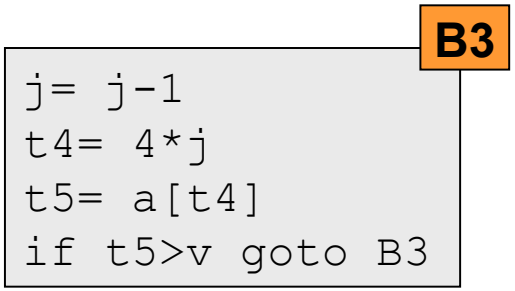
Una variable es inducida por otra cuando ambas aparecen dependientes, tal que los cambios de una influyen en los cambios de la otra.

12.4.1 Transformaciones que Mejoran Código (7)

*Ejemplo de **reduccion de intensidad:***

En el bloque B3

```
j= j-1
t4= 4*j
```



Puede ser reescrito de la forma:

```
j= j-1
t4= t4-4
```

con una previa inicialización de **`t4= 4*j`**, resultando también una reducción de intensidad

12.4.1 Transformaciones que Mejoran Código (8)

ANÁLISIS DE LOS LAZOS DESDE EL GRAFO DE FLUJO.

Analizar los lazos que aparecen en los grafos de flujo nos permite delimitarlos y conocer si son susceptibles de ser simplificados.

Se introducen los conceptos de

- *Nodo dominante,*
- *Lazos naturales,* y
- *Lazos reducibles.*

12.4.1 Transformaciones que Mejoran Código (9)

Nodo Dominante:

El nodo D domina al nodo N, que lo representaremos como D dom N, si todo camino desde el nodo inicial del grafo de flujo a N pasa por el nodo D.

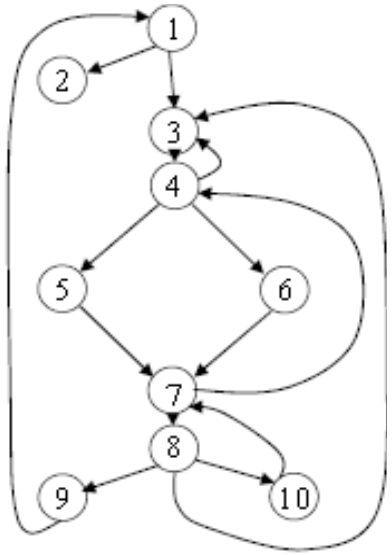
Todo nodo se domina a sí mismo.
El nodo entrada de un lazo domina a todos los nodos del lazo.

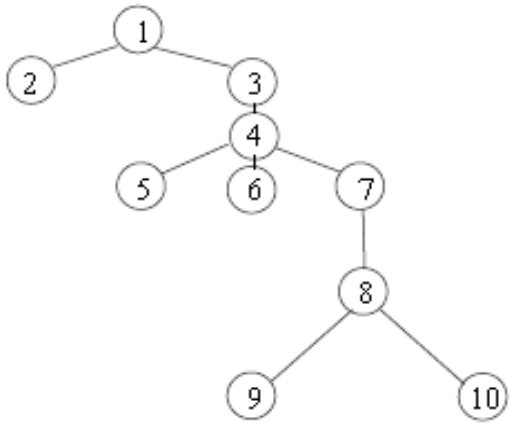
***Árbol de Dominación*:**

Es un árbol que parte de un nodo raíz y cada nodo domina a los nodos que cuelgan de él.

12.4.1 Transformaciones que Mejoran Código (10)

Ejemplo de árbol de dominación:





12.4.1 Transformaciones que Mejoran Código (11)

Lazos Naturales:

Siempre la cabeza debe dominar a la cola y además deben tener las propiedades siguientes:

- Solo deben tener un punto de entrada, llamado ***Encabezamiento.***
- Debe existir alguna forma de iterar.

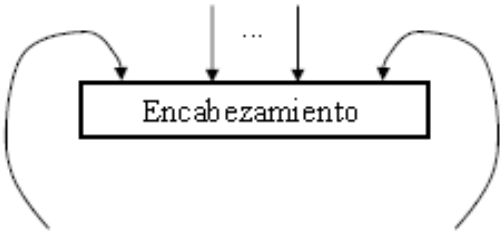
Lazos internos:

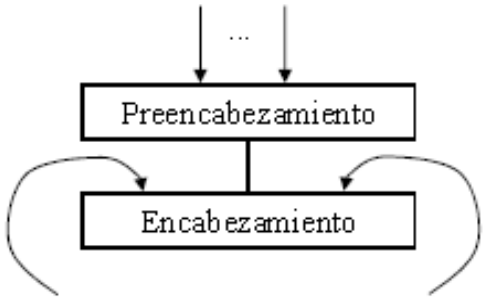
Lazos anidados en otro lazo externo. Puede ocurrir que uno o más lazos internos tenga el mismo encabezamiento.

12.4.1 Transformaciones que Mejoran Código (12)

Preencabezamiento:

Bloque básico colocado delante del encabezamiento y donde son introducidas todas las proposiciones invariantes del lazo.





12.4.1 Transformaciones que Mejoran Código (13)

Grafo de Flujo Reducible:

Si se puede particionar el conjunto de aristas en dos subconjuntos disjuntos: Conjunto de *aristas de avance*, y Conjunto de *aristas de retroceso.*

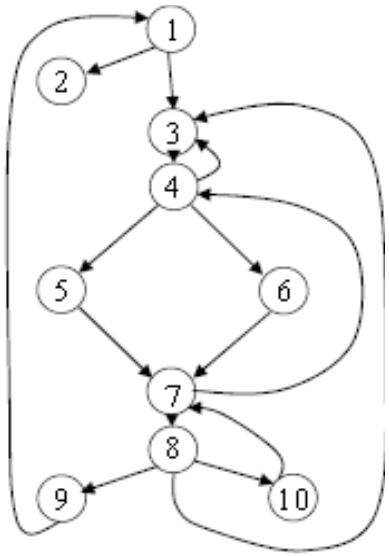
Aristas de Avance:

Las aristas que alcanzan cada nodo desde un nodo inicial.

Aristas de Retroceso:

Las aristas que alcanzan cada nodo desde nodos dominados. Cuando las cabezas dominan a sus colas

12.4.1 Transformaciones que Mejoran Código (14)



El conjunto de aristas de retroceso son:

(7,4) (4,3) (10,7) (8,3) (9,1)

donde 4 dom 7, 3 dom 4, 7 dom 10, 3 dom 8 y 1 dom 9

La presencia de aristas de retroceso indican la presencia de lazos.

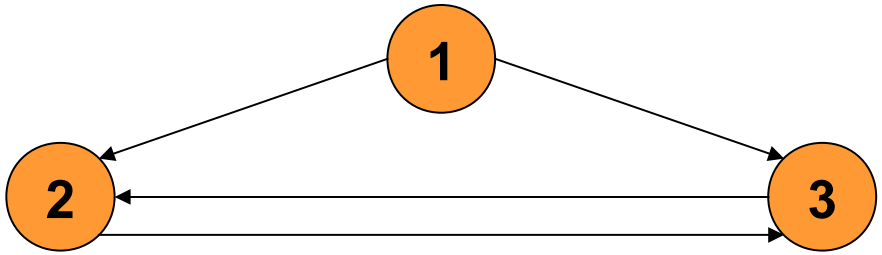
La arista de retroceso `(N,D)`, define un lazo natural formado por el conjunto de nodos que pueden alcanzar N sin ir desde D, ya que D resulta ser el encabezamiento del lazo.

12.4.1 Transformaciones que Mejoran Código (15)

Un grafo de flujos es reducible:

Si las aristas de avance forman un grafo acíclico en el que cada nodo puede alcanzarse desde el nodo inicial del grafo, y
Si las aristas de retroceso constan sólo de las aristas cuya cabezas dominan a las colas.

Ejemplo de Grafos no reducibles:



12.5 Análisis Global del Flujo de Datos (1)

Analiza el grafo de ***control de flujo*** para obtener información que permite

- Detectar posibles fuentes de optimizaciones.
- Facilitar la realización de la transformación de código.

El flujo de datos se realiza estableciendo y resolviendo, para diversos puntos del programa, sistemas de ecuaciones que van a relacionar la siguiente información:

ent[*S*] → Información que entra en *S*.
sal[*S*] → Representa la información que se produce al final de *S*.
desact[*S*] →Información que se desactiva en el interior de *S*.
gen[*S*] → Información que se genera en el interior de *S*.

12.5 Análisis Global del Flujo de Datos (2)

Siendo *S* una sentencia, un bloque básico o conjunto de bloques.

Tipos de ecuaciones:

Transferencia:

- $sal[S] = f(ent[S])$ *análisis hacia delante*.
- $ent[S] = g(sal[S])$ *análisis hacia atrás*.

Reglas de confluencia: *unión* o *intersección*.

Solución para las ecuaciones:

- Método iterativo.
- Mediante traducción dirigida por sintaxis.

12.5 Análisis Global del Flujo de Datos (3)

Información a calcular en las ecuaciones de flujo:

- Alcance de una definición.
- Expresión disponible.
- Variables activas.
- Cadenas de uso.
- Expresiones muy ocupadas.

Concepto de ***PUNTO:*** En un bloque básico existe un *punto*:

- Antes de la primera sentencia.
- Otro después de la última.
- Uno entre cada dos sentencias adyacentes.

12.5 Análisis Global del Flujo de Datos (4)

En el grafo de flujo aparece el concepto de ***camino*** entre dos puntos.

Definición de una ***variable***

- Una definición de una variable es una sentencia que asigna o puede asignar valor a una variable.

- Una definición de una variable se desactiva a lo largo de un camino si existe una nueva definición de la variable.

12.5 Análisis Global del Flujo de Datos (5)

Alcance de una definición

- Una definición **d** alcanca un punto **p**, si existe un camino desde el punto que sigue a **d** hasta **p**, y **d** no se desactiva.
- Sistema de ecuaciones.

Ecuaciones de transferencia:

$$Sal(B_i) = (Ent(B_i) - Desact(B_i)) \cup Gen(B_i)$$

Reglas de confluencia:

$$Ent(B_i) = \bigcup_{P \in \mathrm{Pred}(B_i)} Sal(P) \qquad \text{donde } 1 \le i \le n.$$

$Gen[B]$ { **definiciones producidas en *B* y que no son desactivadas posteriormente en el mismo bloque** }

$Desact[B]$ { **definiciones desactivadas en *B*** }

12.5 Análisis Global del Flujo de Datos (6)

Cálculo de los conjuntos *Gen* y *Desact* para un bloque.

$Gen[B]$ { **definiciones producidas en *B* y que no son desactivadas posteriormente en el mismo bloque** }

$Desact[B]$ { **definiciones desactivadas en *B*** }

$S \rightarrow$ def1 : a := b op c

$Gen[s] = \{def1\}$
$Desact[s] = Da - \{def1\}$

donde *Da* es el conjunto de todas las definiciones de a que existe en el programa.

$S \rightarrow S1 ; S2$

$$Gen\ [S] = Gen\ [S2] \cup (Gen\ [S1] - Desact\ [S2])$$
$$Desact\ [S] = Desact\ [S2] \cup (Desact\ [S2] - Gen\ [S2])$$

12.5.1 Alcance de una Definición (1)

Solución de las ecuaciones:

Información de entrada: Grafo de flujo, con n bloques básicos, para cada bloque básico:

$Gen(B_i)$ para $1 \leq i \leq n$
$Desact(B_i)$ para $1 \leq i \leq n$

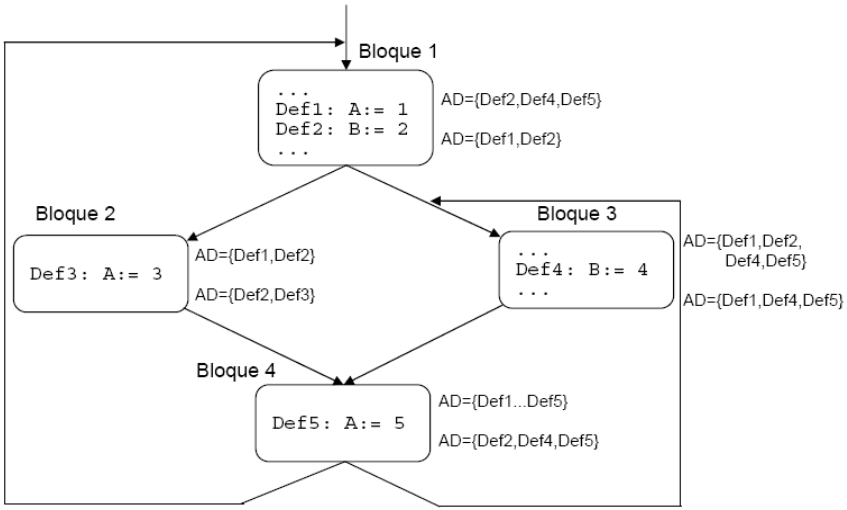
Algoritmo:

```
Para i=1,n hacer
     Ent(Bi)= {};
     Sal(Bi)= Gen(Bi);
Fin para
Mientras hay cambio en los conjuntos calculados hacer
     Para i=1,n hacer
          Ent(Bi)= Unión en P de Sal(P);
          Sal(Bi)= (Ent(Bi) ∪ Desact(Bi)) ∪ Gen(Bi)
     Fin para
Fin mientras
```

12.5.1 Alcance de una Definición (2)

	Bloque 1	Bloque 2	Bloque 3	Bloque 4
	Ent={} Sal={Def1,Def2}	Ent={} Sal={Def3}	Ent={} Sal={Def4}	Ent={} Sal={Def5}
Iteración 1	Ent={Def5} Sal={Def1,Def2}	Ent={Def1,Def2} Sal={Def2,Def3}	Ent={Def1,Def2,Def3} Sal={Def1,Def4,Def5}	Ent={Def1...Def5} Sal={Def2,Def4, Def5}
Iteración 2	Ent={Def2,Def4,Def5} Sal={Def1,Def2}	Ent={Def1,Def2} Sal={Def2,Def3}	Ent={Def1,Def2,Def4,Def5} Sal={Def1,Def4,Def5}	Ent={Def1...Def5} Sal={Def2,Def4, Def5}
Iteración 3	Ent={Def2,Def4, Def5} Sal={Def1,Def2}	Ent={Def1,Def2} Sal={Def2,Def3}	Ent={Def1,Def2,Def4, Def5} Sal={Def1,Def4,Def5}	Ent={Def1...Def5} Sal={Def1,Def4, Def5}

Gen(Bloque1) = {Def1,Def2}
Desact(Bloque1) = {Def3,Def4,Def5}
Gen(Bloque2) = {Def3} *Desact*(Bloque2) = {Def1,Def5}
Gen(Bloque3) = {Def4} *Desact*(Bloque3) = {Def2}
Gen(Bloque4) = {Def5} *Desact*(Bloque4) = {Def1,Def3}



12.5.1 Alcance de una Definición (3)

Aplicaciones del alcance de una definición

Detectar el uso de una variable antes de una definición.

1. Crear un bloque básico *D* que contiene una definición de cada variable que se usa en el programa.
2. Situar dicho bloque antes del bloque inicial.
3. Calcular el alcance de las definiciones.
4. Comprobar para cada bloque *B*i, si existe una definición de *D* que alcanza el uso de alguna variable, en ese caso, la variable está siendo usada antes de ser definida.

12.5.1 Alcance de una Definición (4)

Aplicaciones del alcance de una definición (2)

- **Cadenas de uso y definición.**
  Una forma de estructurar los alcances de las definiciones es mediante listas, que indican para cada uso de una variable las definiciones que lo alcanza. En ese caso recibe el nombre cadenas de uso y definición.

Cálculo de las cadenas de uso y definición:

1. Calcular los alcances de las definiciones.
2. Para cada bloque B_i y cada uso de una variable x
   - Comprobar que x no es redefinida dentro del bloque antes de ser usada. En dicho caso, la cadena de uso y definición contendrá la variable y las definiciones que alcanzan el principio del bloque, es decir, las definiciones de la variable contenidas en $ent[B_i]$.
   - Si la variable es redefinida antes de su uso, la cadena asociada contiene la variable y la última definición que hay dentro del bloque antes de su uso.

12.5.2 Expresiones Disponibles (1)

Una expresión ***x* op *y*** se dice disponible en un punto *p* del programa, si a lo largo de un camino que acaba en *p*, se ha calculado y no ha cambiado el valor de *x* o el de *y*.

Un bloque desactiva una expresión ***x* op *y***, si hay una definición de *x* o *y*, y además no se recalcula ***x* op *y***.

Sistema de ecuaciones:

Ecuaciones de transferencia:

$$Sal(B_i) = (Ent(B_i) - Desact(B_i)) \cup Gen(B_i)$$

Reglas de confluencia:

$$Ent(B_i) = \bigcap_{P \in \mathrm{Pred}(B_i)} Sal(P) \qquad \text{donde } 1 \le i \le n.$$

$Gen[B]$ { **expresiones calculadas en *B* y que no son desactivadas posteriormente en el mismo bloque** }

$Desact[B]$ { **expresiones desactivadas en *B*** }

12.5.2 Expresiones Disponibles (2)

Cálculo de los conjuntos *Gen* y *Desact* para un bloque.

$S \rightarrow$ def1 : a := b op c

$Gen\ [s] = \{b\ op\ c\}$
$Desact\ [s] = Ua$

donde *Ua* es el conjunto de todas las expresiones que tiene en su parte derecha *a*.

$S \rightarrow S1\ ;\ S2$

$Gen\ [S] = Gen\ [S2] \cup (Gen\ [S1] - Desact\ [S2])$
$Desact\ [S] = Desact\ [S2] \cup (Desact\ [S2] - Gen\ [S2])$

$Gen[B]$ { **expressiones calculadas en *B* y que no son desactivadas posteriormente en el mismo bloque** }

$Desact[B]$ { **expresiones desactivadas en *B*** }

12.5.2 Expresiones Disponibles (3)

Solución de las ecuaciones:

Información de entrada: Bloques básicos

$Gen(B_i)$ para $1 \leq i \leq n$

$Desact(B_i)$ para $1 \leq i \leq n$

Información de salida:

$Sal(B_i)$ para $1 \leq i \leq n$

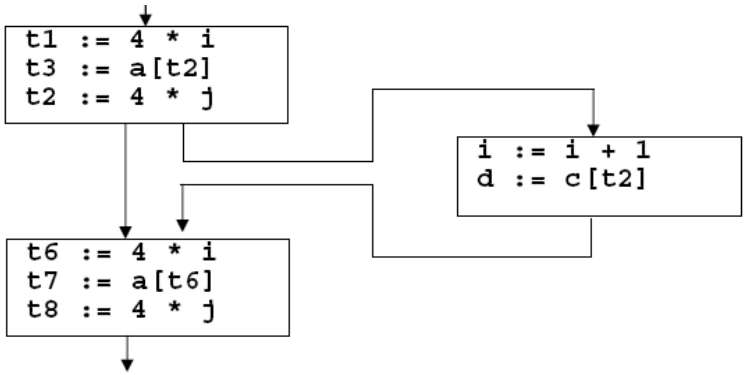
$Ent(B_i)$ para $1 \leq i \leq n$

Algoritmo:

```
Ent(B1)= {}
Sal(B1)= Gen(B1)
Para i=2,n hacer
      Ent(Bi) = {};
      Sal(Bi) = Todas las expresiones - Desact(Bi);
Fin para
Mientras hay cambio en los conjuntos calculados hacer
     Para i=2,n hacer
           Ent(Bi)= Intersección en P de Sal(P);
           Sal(Bi)= (Ent(Bi) - Desact(Bi)) ∪ Gen(Bi)
      Fin para
Fin mientras
```

12.5.2 Expresiones Disponibles (4)

Ejemplo:



Gen (B1) = {4*i, 4*j} *Desact* (B1) = {a[t2]}
Gen (B2) = {4*i, 4*j, a[t6]} *Desact* (B2) = {}
Gen (B3) = {c[t2]} *Desact* (B3) = {4*i, i+i}

Fase inicial

T = {4*i, 4*j, a[t**6**], i+1, c[t**2**]}
ent (B**1**) = {} *sal* (B**1**) = *Gen* (B**1**) = {4*i, 4*j}
ent (B**2**) = {} *sal* (B**2**) = T - *Desact* (B**2**) = {4*i, 4*j, a[t**6**], a[t**2**], i+1, c[t**2**]}
ent (B**3**) = {} *sal* (B**3**) = T - *Desact* (B**2**) = {4*j, a[t**6**], a[t**2**], c[t**2**]}

1ª interacción

ent (B**2**) = *sal* (B**1**) ∩ *sal* (B**3**) = {4*j}
sal (B**2**) = (*ent* (B**2**) - *Desact* (B**2**)) ∪ *Gen* (B**2**) = {4*i, 4*j, a[t**6**]}
ent (B**3**) = *sal* (B**1**) = {4*i, 4*j}
sal (B**3**) = (*ent* (B**3**) - *Desact* (B**3**)) ∪ *Gen* (B**3**) = {4*j, c[t**2**]}

2ª interacción

ent (B**2**) = *sal* (B**1**) ∩ *sal* (B**3**) = {4*j}
sal (B**2**) = (*ent* (B**2**) - *Desact* (B**2**)) ∪ *Gen* (B**2**) = {4*j} ∪ {4*i, 4*j, a[t**6**]}
ent (B**3**) = *sal* (B**1**) = {4*i, 4*j}
sal (B**3**) = (*ent* (B**3**) - *Desact* (B**3**)) ∪ *Gen* (B**3**) = {4*j} ∪ {c[t**2**]}

Resultado

ent (B**1**) = {}	*sal* (B**1**) = {4*i, 4*j}
ent (B**2**) = {4*j}	*sal* (B**2**) = {4*i, 4*j, a[t**6**]}
ent (B**3**) = {4*j,4*i}	*sal* (B**3**) = {4*j, c[t**2**]}

12.5.3 Eliminación de Expresiones Comunes

Grafo de flujo. B_i Bloques básicos $1 \leq i \leq n$.

Dadas todas las expresiones disponibles

```
Para cada bloque Bi hacer
    Para cada sentencia en Bi, de la forma s= x op y,  tal que x op y esté en el conjunto
    de expresiones disponibles al principio de Bi hacer
        Comprobar que dicha expresión no se desactiva en el bloque

        Buscar hacia atrás en el grafo todas las evaluaciones de la expresión x op y y que
          alcanzan a s. Nos situaremos en la última de cada camino que alcanza a Bi.

        Crear una nueva variable u.

        Para cada x op y que alcanza s hacer

          generar: u= x op y | si := u

          Sustitúyase s= x op y por s= u.

        fin para

    fin para
fin para
```

12.5.4 Variables Activas (1)

Variables activas

Dada una variable *x* se dice que está activa en un punto *p* del programa, si *x* se va a utilizar a lo largo de un camino que comienza en *p*, antes de que aparezca una nueva definición.

Sistema de ecuaciones:

Ecuaciones de transferencia:

$$Sal(B_i) = (Ent(B_i) - Desact(B_i)) \cup Gen(B_i)$$

Reglas de confluencia:

$$Sal(B_i) = \bigcap_{P \in Sucesor(B_i)} Ent(P) \qquad \text{donde } 1 \le i \le n.$$

12.5.4 Variables Activas (2)

$Gen[B]$ { variables usadas en *B* antes de una redefinición }

$Desact[B]$ { variables definidas en *B* }

Cálculo de los conjuntos *Gen* y *Desact* para un bloque.

$S \rightarrow$ def1 : a := b op c	$Gen[s] = \{b, c\}$
	$Desact[s] = \{a\}$
$S \rightarrow S_1 ; S_2$	$Gen[S] = Gen[S_2] \cup (Gen[S_1] - Desact[S_2])$
	$Desact[S] = Desact[S_2] \cup (Desact[S_1] - Gen[S_2])$

12.5.4 Variables Activas (3)

Solución de las Ecuaciones:

Información de entrada: Bloques básicos

$Gen\ (B_i)$ para $1 \le i \le n$

$Desact\ (B_i)$ para $1 \le i \le n$

Información de salida:

$Sal\ (B_i)$ para $1 \le i \le n$

$Ent\ (B_i)$ para $1 \le i \le n$

Algoritmo

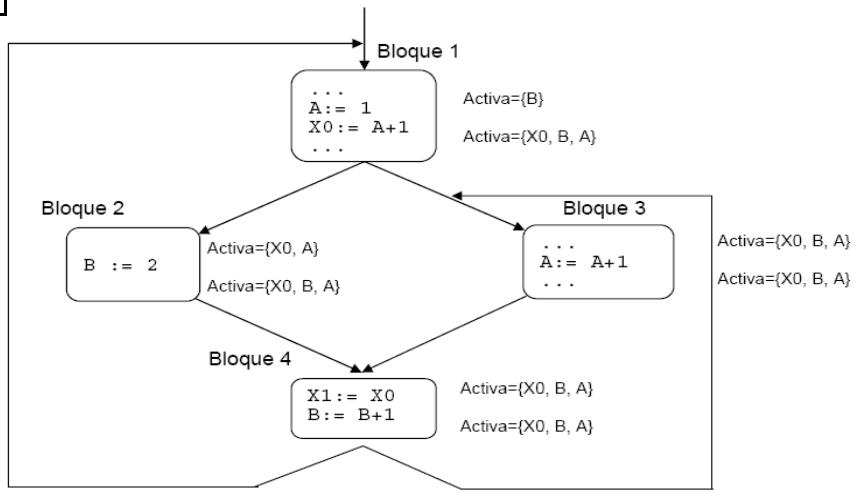
```
Para i=1,n hacer
        Ent(Bi)= Gen(Bi) ;
        Sal(Bi)= {} ;
Fin para
Mientras hay cambio en los conjuntos calculados hacer
        Para i=1,n hacer
                Sal(Bi)= Union en P de Sal(P)
                Ent(Bi)= (Sal(Bi)-Desact(Bi)) ∪ Gen(Bi)
        Fin para
Fin mientras
```

12.5.4 Variables Activas (4)

Ejemplo de variables activas:

	Bloque 1	Bloque 2	Bloque 3	Bloque 4
	Ent={}	Ent={}	Ent={A}	Ent={X0,B}
Iteración 1	Ent={} Sal={A}	Ent={X0} Sal={X0, B}	Ent={X0, B, A} Sal={X0, B}	Ent={X0, B, A} Sal={X0, B, A}
Iteración 2	Ent={B} Sal={X0, B, A}	Ent={X0, A} Sal={ X0, B, A}	Ent={X0, B, A} Sal={X0, B, A}	Ent={X0, B, A} Sal={X0, B, A}
Iteración 3	Ent={B} Sal={X0, B, A}	Ent={X0, A} Sal={X0, B, A}	Ent={X0, B, A} Sal={X0, B, A}	Ent={X0, B, A} Sal={X0, B, A}

Gen(Bloque1) = {} Desact(Bloque1) = {A,X0}
Gen(Bloque2) = {} Desact(Bloque2) = {B}
Gen(Bloque3) = {A} Desact(Bloque3) = {A}
Gen(Bloque4) = {X0,B} Desact(Bloque4) = {X1,B}



12.5.5 Resumen de las Reglas de Confluencia

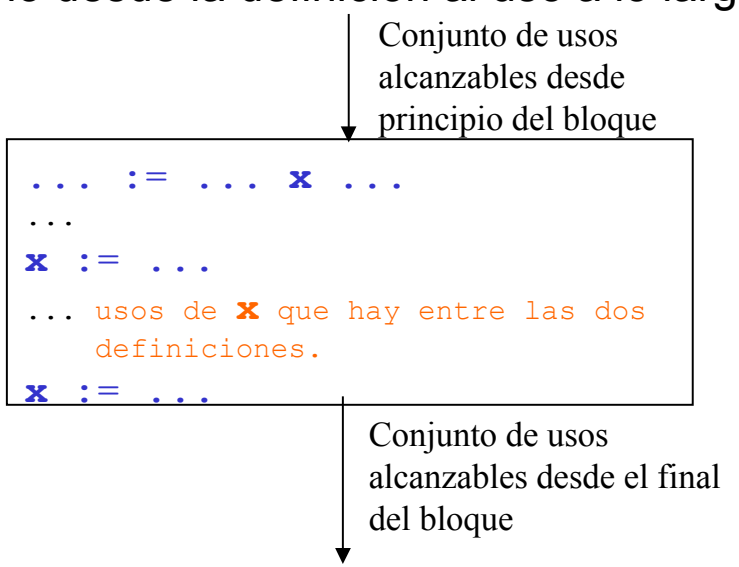
	Operador de confluencia ∪ Cualquier camino	Operador de confluencia ∩ Todos los caminos
Análisis hacia delante (uso de predecesores) →	*** Alcance de una definición** (Cadenas de uso y definición) **Ecuaciones de transferencia** $Sal(B_i) = (Ent(B_i) - Desact(B_i)) \cup Gen(B_i)$ **Reglas de confluencia** $Ent(B_i) = \cup_{P \in Pred(Bi)} Sal(P)$	*** Expresiones disponibles** *** Copias disponibles** **Ecuaciones de transferencia** $Sal(B_i) = (Ent(B_i) - Desact(B_i)) \cup Gen(B_i)$ **Reglas de confluencia:** $Ent(B_i) = \cap_{P \in Pred(Bi)} Sal(P)$
Análisis hacia atrás (uso de sucesores) ←	*** Variables activas** *** Usos alcanzados** *** Cadenas de definición y uso** **Ecuaciones de transferencia** $Ent(B_i) = (Sal(B_i) - Desact(B_i)) \cup Gen(B_i)$ **Reglas de confluencia** $Sal(B_i) = \cup_{P \in Sucesor(Bi)} Ent(P)$	*** Expresiones muy ocupadas.** **Ecuaciones de transferencia:** $Ent(B_i) = (Sal(B_i) - Desact(B_i)) \cup Gen(B_i)$ **Reglas de confluencia** $Sal(B_i) = \cap_{P \in Sucesor(Bi)} Ent(P)$

12.5.6 Cadenas de Definición y Uso (1)

Cadenas de Definición

Se trata de mantener una lista donde, para cada definición, se indica los usos posteriores que se van a hacer de ella, antes de que sea redefinida.

Se dice que un uso de una variable A es alcanzable desde una definición de A si existe un camino desde la definición al uso a lo largo del cual A no se ha redefinido



Realizamos un análisis hacia atrás y en operador de confluencia se encuentra la unión.

12.5.6 Cadenas de Definición y Uso (2)

Sistemas de Ecuaciones

Ecuaciones de transferencia:

$$Ent(B_i) = (Sal(B_i) - Desact(B_i)) \cup Gen(B_i)$$

Reglas de confluencia:

$$Sal(B_i) = \bigcup_{P \in Sucesor(B_i)} Ent(P) \qquad \text{donde } 1 \le i \le n.$$

Gen [*B*] { (*S*,*x*) | *S* es una sentencia del bloque *B* que usa la variable *x* y *x* no ha sido definida en el bloque antes de S}

Desact [*B*] { (S,x) | S es una sentencia que no está en *B* que usa la variable *x* y *x* es una variable definida en *B*}

•Cálculo de los conjuntos *Gen* y *Desact* para un bloque.

S → D1 : a := b op c *Gen* [*s*] = {(D1,c), (D1,b)}

Desact [*s*] = {Ua-S1}

donde *Ua* es el conjunto de todos los pares (*Sa*,*a*) donde *Sa* son todas las definiciones en el programa que usan *a*, y existirá un par para cada variable *x* que se usa en cada sentencia.

•Tanto el cálculo de *Gen* y *Desact* para una secuencia de sentencia como la resolución de las ecuaciones, se realiza como en el caso de variables activas

12.5.7 Copia Disponible y Propagación de Copia (1)

Copia disponible

Se dice que una copia *x= y* está disponible en un punto *p* del grafro, si en todo camino desde el bloque inicial al punto *p*, existe una sentencia *x= y* y además ni *x* ni *y* se han redefinido.

Sistema de ecuaciones:

Ecuaciones de transferencia:

$$Sal(B_i) = (Ent(B_i) - Desact(B_i)) \cup Gen(B_i)$$

Reglas de confluencia:

$$Ent(B_i) = \bigcap_{P \in \mathrm{Pred}(B_i)} Sal(P) \qquad \text{donde } 1 \le i \le \mathrm{n}.$$

$$Ent(B_1) = \{\}$$

12.5.7 Copia Disponible y Propagación de Copia (2)

Gen[*B*] { **Toda copia x= y generada en *B* tal que no hay definiciones posteriores ni de x ni de y** }

Desact[*B*] { **Todas las copias del programa que se ven afectadas por las definiciones realizadas en *B*** }

- Cálculo de los conjuntos *Gen* y *Desact* para un bloque.

$S \rightarrow$ d1 : a := b	*Gen* [*s*] = {a:=b} *Desact* [*s*] = {Ta-a:=b}
$S \rightarrow$ d1 : a := b op c	*Gen* [*s*] = {} *Desact* [*s*] = {Ta}

donde Tx es el conjunto de todas las copias del programa que contienen en uno de los lados x.

• Tanto el cálculo de *Gen* y *Desact* para una secuencia de sentencia como la resolución de las ecuaciones, se realiza como en el caso de las expresiones disponibles.

12.5.7 Copia Disponible y Propagación de Copia (3)

Propagación de copias

Entrada Grafo de flujo.
Copias disponibles a la entrada de cada bloque $ent(B_i)$.
Cadenas de definición y uso.

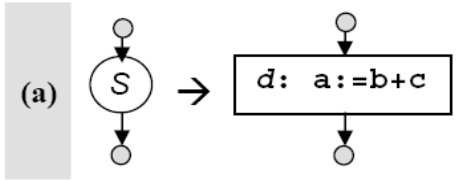
Salida: Grafo de flujo donde se han propagado y eliminado las copias.

```
Para cada copia "s: x := y" hacer
    Determinar, mediante las cadenas de definición y uso, los usos de x alcanzados
    desde la copia.
    Para todo uso de x, calculado en el punto anterior hacer
        comprobar si la copia "s: x := y" está en el conjunto ent(Bi), donde Bi es el bloque
        donde se realiza dicho uso y además no ocurre ninguna definición de x o y antes del
        uso de x dentro de Bi;
        Si se cumplen las condiciones:
          Comprobar los uso de x dentro del bloque donde está definida la copia "s: x :=  y",
          Si se usa x,
             Propáguese la copia para todos los usos internos al bloque.
             Elimínese la copia "s: x := y" y sustitúyanse todos los usos de x por y.
    Fin para
Fin para
```

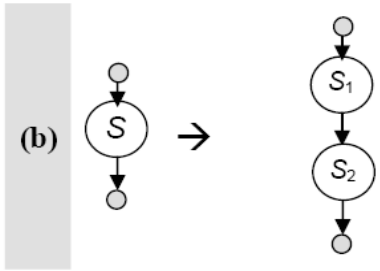
12.5.8 Definición de Variables (1)

Análisis del Flujo de Datos (1)

Ecuaciones para el cálculo de las definiciones de variables



$$gen(S) = \{d\}$$
$$desact(S) = D_a - \{d\}$$
$$sal(S) = gen(S) \cup (ent(S) - desact(S))$$



$$gen(S) = gen(S_2) \cup (gen(S_1) - desact(S_2))$$
$$desact(S) = desact(S_2) \cup (desact(S_1) - gen(S_2))$$
$$ent(S_1) = ent(S)$$
$$ent(S_2) = sal(S_1)$$
$$sal(S) = sal(S_2)$$

12.5.8 Definición de Variables (2)

Análisis del Flujo de Datos (2)

Ecuaciones para el cálculo de las definiciones de variables