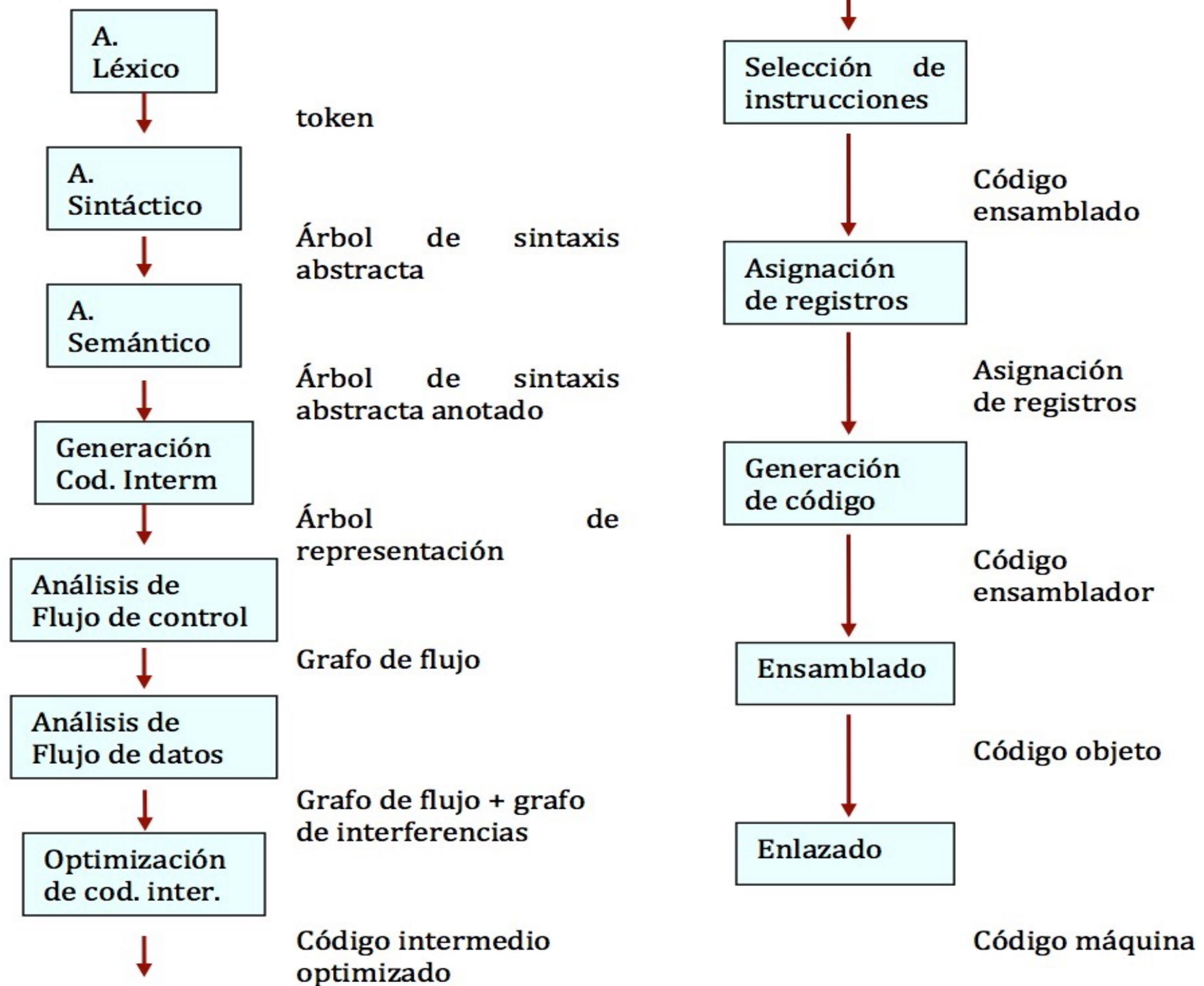


Tema 11: Generación de código

1. Introducción
 - 1.1. Módulos e interfaces de un compilador
 - 1.2. Objetivos de generación de código
2. Selección de instrucciones
 - 2.1. Conceptos básicos
 - 2.2. Algoritmo Maximal Munch
 - 2.3. Revestimiento sintáctico
3. Asignación de registros
 - 3.1. Jerarquía de memoria
 - 3.2. Asignación mediante coloración de grafos

1. Introducción



Salida del generador de código

Código máquina absoluto

Direcciones codificadas de forma fija

Sencillo de generar pero poco flexible (difícil de cargar)

Código máquina relocizable

La asignación definitiva de direcciones de memoria se hace en tiempo de enlace y carga

Permite la compilación separada

Código ensamblador

Simplifica la generación de código al usar macros y nombres simbólicos

Debe traducirse posteriormente (ensamblado + enlazado)

Objetivos de la generación de código

El código generado debe ser correcto y de calidad

Eficiencia de procesadores modernos depende mucho de:

- Mantener el pipeline lleno
- Uso eficiente de registros

Para maximizar el rendimiento del procesador, el compilador debe realizar la **planificación (scheduling) de instrucciones**:
Identificar el mejor orden válido de las instrucciones.

Arquitectura RISC vs CISC

	RISC	CISC
1	32 registros	16, 8,... 32
2	Registros de una sola clase	Registros divididos en clases, con distintas posibilidades de uso
3	Instrucciones de 3 direcciones	Instrucciones de 2 direcciones
4	Operaciones aritméticas solo entre registros	Op. Aritméticas pueden usar direcciones de memoria (distintos modos de direccionamiento)
5	Todas las instrucciones tienen la misma longitud	Instrucciones de distintas longitudes

- 1 y 2 suponen un trabajo extra para la asignación de registros.
- 3 Supone que a veces es necesario copiar instrucciones a los registros antes de operar.
- 4 Supone generar más instrucciones para mover valores a los registros, pero computacionalmente (ciclos de procesador) suelen ser equivalentes.
- 5 No es un problema para el compilador, aunque supone que el programa ensamblador debe tenerlo en cuenta.

2. Selección de instrucciones

Selección de instrucciones mediante reescritura de arboles

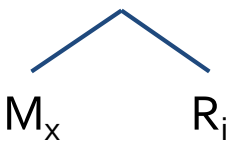
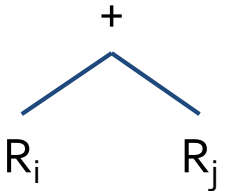
Selección de instrucciones

El generador de código establece la correspondencia entre la representación intermedia del programa y la secuencia de código máquina que será ejecutada en la máquina destino.

Las instrucciones destino pueden representarse mediante árboles:

- **Árbol patrón (plantilla):** Árbol que representa el código intermedio equivalente a cada instrucción de código máquina.
- **Tamaño** de un árbol patrón: Número de nodos que contiene.

Esquemas de traducción de árboles

Árbol patrón	Instrucción	Significado
$R_i \leftarrow C_a$	LD $R_i, \#a$	<i>Carga en registro i la constante a</i>
$R_i \leftarrow M_x$	LD R_i, x	<i>Carga en registro i el contenido de la posición de memoria x</i>
$M_x \leftarrow$ 	ST x, R_i	<i>Almacena en la posición de memoria x el contenido del registro i</i>
$R_i \leftarrow$ 	ADD R_i, R_i, R_j	<i>Guarda en el registro i el resultado de sumar el contenido de los registros i y j</i>

Selección de instrucciones

Consiste en encontrar un **revestimiento** del árbol de código intermedio con el mínimo número de árboles patrón sin que estos se solapen.

Problemas:

- No hay un único revestimiento.
- Una instrucción de código intermedio [máquina] puede representarse por varias secuencias de instrucciones máquina [de código intermedio].
- Generar código para distintas familias de procesadores puede ser muy distinto

- Los árboles patrón de las instrucciones de máquinas CISC son bastante grandes.
- Los árboles patrón de las instrucciones de arquitecturas RISC son pequeños y tienen un coste uniforme.
- *Revestimiento óptimo (optimum tiling)*: La suma de costes de cada árbol patrón es el mínimo,
- *Revestimiento satisfactorio (optimal tiling)*: No habrá dos árboles patrón adyacentes que puedan combinarse en uno solo de menor coste

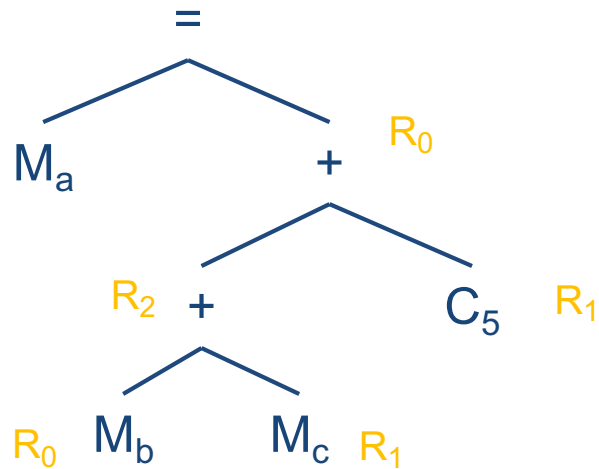
Maximal Munch (A : árbol de código intermedio)¹

1. Encontrar el **árbol patrón (AP)** más grande que puede incluir al nodo raíz de A .
2. **Cubrir el nodo raíz** (y posiblemente otros adyacentes) con AP .
3. **Generar la instrucción** correspondiente a AP .
4. Sean $SAP_1, SAP_2, \dots, SAP_n$ los árboles en que AP ha dividido a A .
5. Mientras quede subárbol SAP_i sin cubrir
Maximal Munch (SAP_i)

¹ Al comenzar por la raíz, el algoritmo genera el código en orden inverso

Ejemplo Algoritmo Maximal Munch

$a = b + c + 5$



LD R_0 , b
 LD R_1 , c
 ADD R_2 , R_0 , R_1
 LD R_1 , #5
 ADD R_0 , R_2 , R_1
 ST a , R_0

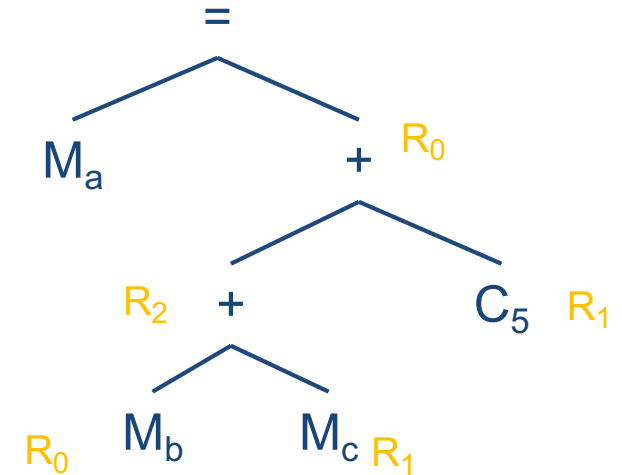
Árbol patrón	Instrucción
$\underline{R_i} <- Ca$	LD $\underline{R_i}$, #a
$\underline{R_i} <- Mx$	LD $\underline{R_i}$, x
$Mx <-$ $\begin{array}{c} = \\ \swarrow \quad \searrow \\ M_x \quad R_i \end{array}$	ST x, $\underline{R_i}$
$\underline{R_i} <-$ $\begin{array}{c} + \\ \swarrow \quad \searrow \\ \underline{R_i} \quad \underline{R_j} \end{array}$	ADD $\underline{R_i}$, $\underline{R_i}$, $\underline{R_j}$

Revestimiento sintáctico

- Convertir un árbol sintáctico en forma linealizada (recorrido en **preorden**): Se ha convertido en una **cadena** de un lenguaje
- Convertir cada **árbol patrón** en una **regla de producción** cuyo lado derecho es la forma prefija del árbol patrón.
- Las **acciones semánticas** de cada regla generan la instrucción de código máquina correspondiente.
- Construir un Analizador Sintáctico LR para la gramática.
- La gramática suele ser muy **ambigua**: Resolución de conflictos dando prioridad a árboles con menor coste.

Ejemplo Revestimiento Sintáctico

$S \rightarrow T \mid S \mid T$
 $T \rightarrow R \mid M \mid \dots$
 $R_i \rightarrow Ca \quad \{ LD R_i, \#a \}$
 $R_i \rightarrow Mx \quad \{ LD R_i, x \}$
 $Mx \rightarrow = Mx \quad R_i \quad \{ ST x, R_i \}$
 $R_i \rightarrow + R_i \quad R_j \quad \{ ADD R_i, R_i, R_j \}$
 $Mx \rightarrow m$



Cadena: “= $M_a + + M_b M_c c_5$ ”

ASA:
 $= M_a + + M_b M_c c_5 \rightarrow$
 $= M_a + + R_0 M_c c_5 \rightarrow$
 $= M_a + + R_0 R_1 c_5 \rightarrow$
 $= M_a + R_2 c_5 \rightarrow$
 $= M_a + R_2 R_1 \rightarrow$
 $= M_a R_0 \rightarrow$
 $= M_a$

$LD R_0, b$
 $LD R_1, c$
 $ADD R_2, R_0, R_1$
 $LD R_1, \#5$
 $ADD R_0, R_2, R_1$
 $ST a, R_0$

Revestimiento sintáctico

Ventajas

- El AS LR es **eficiente** y conocido: Se producen generadores de código eficientes y robustos
- Sencillo **portar** el generador de código a otra arquitectura
- Puede **mejorarse** añadiendo reglas con características específicas de la máquina destino

Revestimiento sintáctico

Inconvenientes

- Orden de evaluación de izquierda a derecha
- Si el juego de instrucciones destino es muy elevado, la gramática será muy grande
- Se necesitan técnicas específicas para codificar y procesar las gramáticas para evitar bloqueo del A.S.:
 - La gramática no maneja algún patrón de operador
 - Se ha resuelto mal algún conflicto sintáctico
 - Bucle infinito de reducciones

3. Asignación de registros

Jerarquía de memoria

	Tiempo de acceso	Capacidad
Registros	0.2 - 0.5 ns	256 - 1024 bytes
Caché primaria (L1)	0.4 -1 ns	32 Kb – 256 Kb
Caché secundaria (L2)	4 – 10 ns	1 – 8 Mb
Caché terciaria (L3)	10 – 50 ns	4 – 64 Mb
Memoria principal	50 – 500 ns	256 – 16 Gb
Memoria auxiliar	5 ms - ...	

Uso de registros

- Almacenar operandos de instrucción
- Almacenar valores temporales.
- Guardar valores. Ej. Variable de inducción calculada en un bloque pero usada en otro.
- Guardar valores relacionados con la gestión de la memoria en tiempo de ejecución. Ej. Stack Pointer.

Asignación de registros

Consiste en decidir en qué registro debe almacenarse cada valor.

Idea básica:

- Asignar el mayor número de variables temporales posibles al menor número de registros posible.
- Adicionalmente, sería deseable conseguir que el origen y destino de las instrucciones MOVE sea el mismo para poder eliminar la instrucción.

Importancia de la asignación de registros

$x := y - z$

```
LD R1, y      // R1 = y
LD R2, z      // R2 = z
SUB R1, R1, R2 // R1 = R1 - R2
ST x, R1      // x = R1
```

Mejoras:

- Si las instrucciones que calcularon los valores de **y** y **z** dejaron sus valores en un registro, podemos evitar las dos instrucciones de carga.
- Si la instrucción que va a usar el valor de **x** lo usa directamente desde el registro 1 (R1) podemos evitarnos la instrucción de almacenaje final.

Grafo de interferencias

Grafo no dirigido donde:

- cada **nodo** representa a un valor (variable).
- habrá un **arco** (t_1, t_2) si ambos valores no pueden asignarse al mismo registro:
 - Porque representan variables activas al mismo tiempo, o
 - Si el procesador no puede producir el resultado de $a := b + c$ en r_i añadir el arco (a, r_i)

Coloreado de grafo de interferencias

k-coloreado:

Asignar un color de entre los k posibles a cada nodo de manera que dos nodos adyacentes no tengan el mismo color.

- Si el procesador tiene k registros, realizaremos un k -coloreado (disponemos de k colores).
- Si no existe un k -coloreado del grafo, algunos valores temporales deberán estar en memoria.
- Es un problema NP-completo: Usar heurística, por ejemplo **coloreado por simplificación**

Alg. Coloreado de grafos por simplificación

1. Construcción del grafo de interferencias

Si $G - \{m\}$ es $k-1$ coloreable, G es k -coloreable

2. Simplificación

- Si hay un nodo m con **menos de k vecinos**, apilar m y eliminarlo del grafo.
- Repetir hasta que el grafo no tenga nodos

3. Volcado

- Si durante la simplificación solo hay nodos de grado mayor o igual que k , la heurística puede fallar (o el grafo no es k -coloreable).
- Entonces elegir un **nodo** para volcar a **memoria** (su valor no estará en registros sino en memoria).
- Eliminar el nodo del grafo y volver a la fase de simplificación.

4. Selección: Se asignan colores a cada nodo del grafo

- Desapilar un nodo y añadirlo al grafo. Asignar color distinto a sus vecinos.
- Al desapilar nodo para ser volcado a memoria:
 - Si sus vecinos usan k colores: volcar a memoria
 - Si 2 o más vecinos tienen el mismo color, se puede colorear: no se vuelca

5. Hasta pila vacía

Ejemplo

Realizar la asignación de los 4 registros disponibles para el bloque básico definido a continuación.

ent[B] = {j, k}
sal[B] = {d, k, j}

g := j+1
h := k-1
f := g*h
e := j+8
m := j + 16
b := f + 12
c := e+8
d := c
k := m+4
j := b

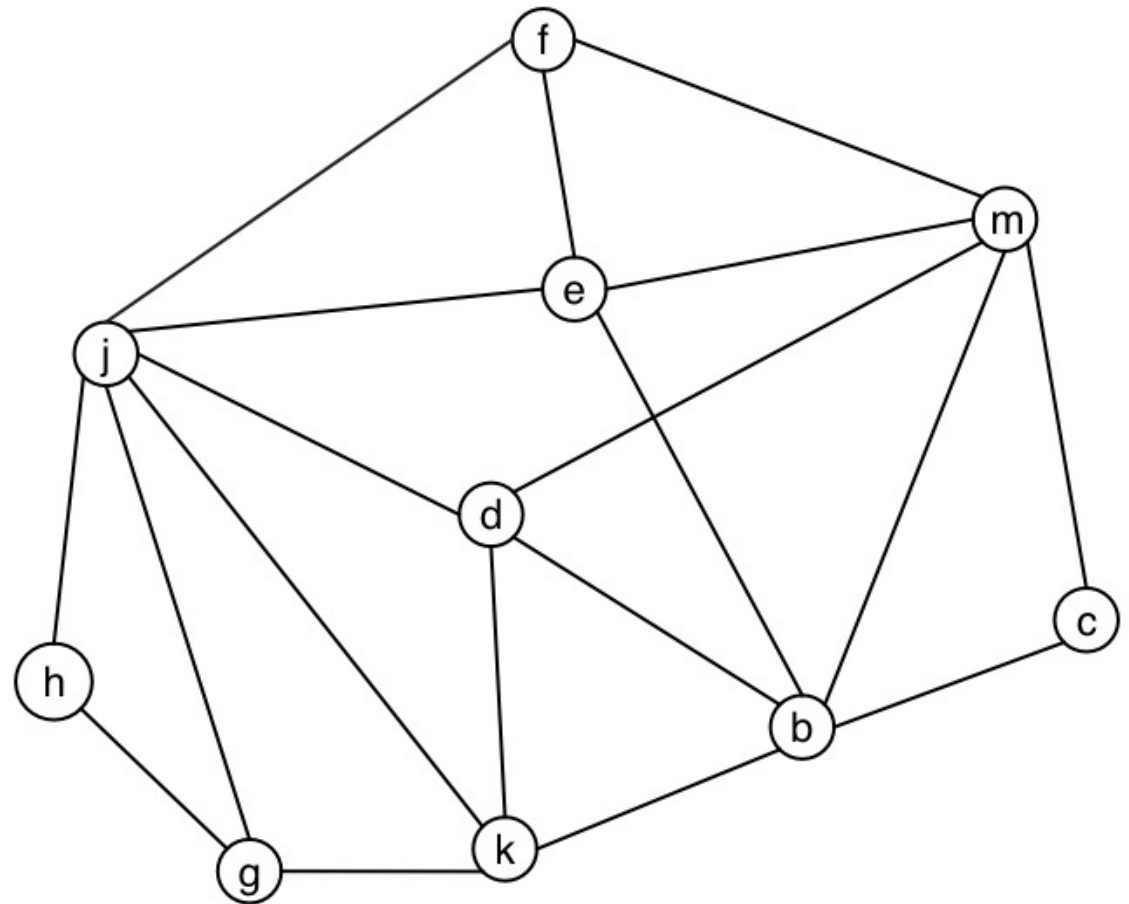
	1ª iteración			
	def	usa	ent[]	sal[]
g := j+1	g	j	j, k	j, g, k
h:=k-1	h	k	j, g, k	j, g, h
f:=g*h	f	g, h	j, g, h	f, j
e:=j+8	e	j	f, j	e, f, j
m := j + 16	m	j	e, f, j	m, e, f
b := f + 12	b	f	m, e, f	b, m, e
c:=e+8	c	e	b, m, e	b, m, c
d:=c	d	c	b, m, c	d, b, m
k:=m+4	k	m	d, b, m	d, k, b
j:=b	j	b	d, k, b	d, k, j

Ejemplo

Número de registros disponibles: 4

ent[]	sal[]
j, k	j, g, k
j, g, k	j, g, h
j, g, h	f, j
f, j	e, f, j
e, f, j	m, e, f
m, e, f	b, m, e
b, m, e	b, m, c
b, m, c	d, b, m
d, b, m	d, k, b
d, k, b	d, k, j

Grafo de interferencias



Ejemplo

Asignación de registros:

Valor (variable)	Registro
c	R1
f	R4
m	R2
e	R1
b	R3
d	R1
k	R2
j	R3
g	R1
h	R2

