

Generación de Código

**Lenguajes de Programación y
Procesadores de Lenguajes**

Tabla de contenido:

Table of Contents

1.	INTRODUCCIÓN.	3
1.1.	MÓDULOS E INTERFACES DE UN COMPILADOR	3
1.2.	RETOS DE LA GENERACIÓN DE CÓDIGO. ARQUITECTURAS RISC Y CISC	4
2.	SELECCIÓN DE INSTRUCCIONES MEDIANTE REESCRITURA DE ÁRBOLES.	6
2.1.	CONCEPTOS BÁSICOS: ESQUEMA DE TRADUCCIÓN DE ÁRBOLES	6
2.2.	SELECCIÓN DE INSTRUCCIONES MEDIANTE REVESTIMIENTO SINTÁCTICO	8
3.	ASIGNACIÓN DE REGISTROS	10
3.1.	JERARQUÍA DE MEMORIA	10
3.2.	ASIGNACIÓN DE REGISTROS MEDIANTE COLORACIÓN DE GRAFOS	10
4.	BIBLIOGRAFÍA.	16

1. Introducción.

1.1. Módulos e interfaces de un compilador

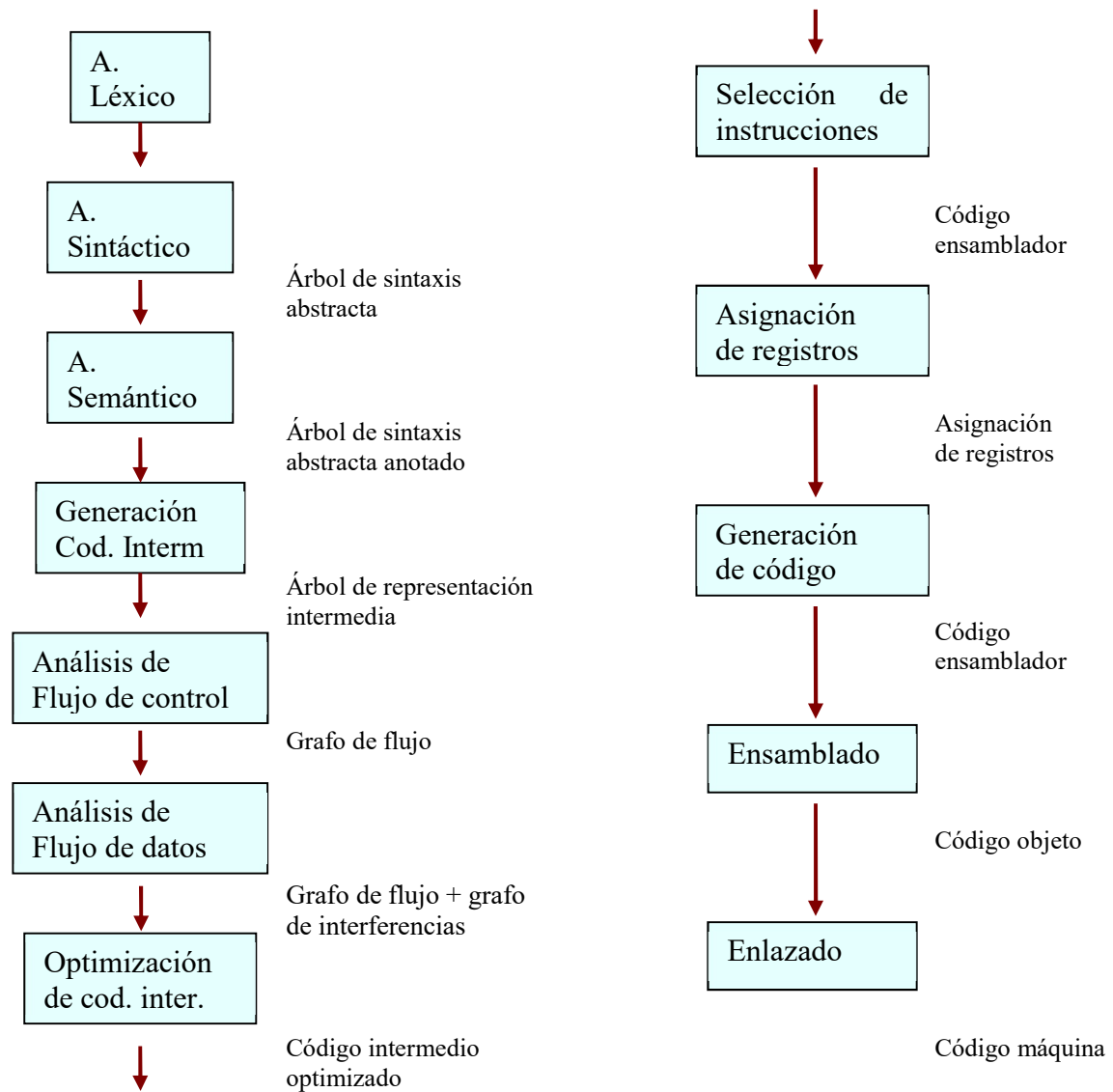


Figura 1: Fuente: [Appel, Ginsburg, 1998]

Generación de código

La entrada al generador de código es la representación intermedia generada por el *front-end* sobre la que se han aplicado optimizaciones, junto con la información contenida en la tabla de símbolos.

Objetivos de la generación de código

- El código generado debe ser correcto y de calidad.
- La eficiencia de los procesadores actuales depende mucho de:

- Mantener el *pipeline* lleno.
- Uso eficiente de registros.

Producto del generador de código

El compilador tiene varias opciones a la hora de generar el código objeto:

- 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 de módulos.
- Código ensamblador
 - Simplifica la generación de código al usar macros y nombres simbólicos.
 - Debe traducirse posteriormente (ensamblado + enlazado).

1.2. Retos de la generación de código. Arquitecturas RISC y CISC

La arquitectura de la máquina destino tiene un impacto muy alto en el nivel de dificultad del proceso de generación de código máquina de calidad.

- Una arquitectura RISC (*Reduced Instruction Set Computer*) típica, por ejemplo ARM, contiene muchos registros, usa instrucciones de 3 direcciones, modos de direccionamiento muy sencillos e instrucciones relativamente simples.
- Una arquitectura CISC (*Complex Instruction Set Computer*) típica, por ejemplo x86, tiene pocos registros, instrucciones de 2 direcciones, gran variedad de modos de direccionamiento, varias clases de registros, e instrucciones de longitud variable.

Una de las dificultades en la compilación para procesadores modernos se encuentra en la necesidad de mantener el *pipeline* lleno y hacer un uso eficiente de los registros.

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

En la Tabla 1 pueden observarse algunas diferencias a tener en cuenta a la hora de diseñar el generador de código de un compilador para una arquitectura CISC o una arquitectura RISC.

- La existencia de registros de distinto uso supone un trabajo extra durante la fase de asignación de registros.
- El empleo de instrucciones de 2 direcciones supone que a veces es necesario copiar instrucciones a los registros antes de operar.
- El hecho de que las instrucciones aritméticas solo operen que valores almacenados en los registros, no directamente con posiciones de memoria, supone generar más instrucciones para mover valores a los registros, pero computacionalmente (ciclos de procesador) suelen ser equivalentes.
- La utilización de un juego de instrucciones que tienen un tamaño variable no es un problema para el compilador, aunque supone que el programa ensamblador debe tenerlo en cuenta.

	RISC	CISC
1	32 registros	16, 8,... 32 registros
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

Tabla 1: Diferencias arquitecturas CISC y RISC

Una máquina RISC suele tener 2 conjuntos de registros, uno para enteros y otro para reales además de otros de propósito especial, como el contador de programa.

En una máquina RISC las instrucciones operan con valores almacenados en registros, por lo tanto, se debe ejecutar una instrucción de carga (LOAD) para cargar los operandos de la instrucción en los registros. Se dice que se trata de arquitecturas carga-almacenaje (*load-store*) o registro-registro. Para las operaciones binarias, las instrucciones de una máquina RISC especifican 3 registros: 2 fuentes y uno destino.

Las máquinas CISC disponen de instrucciones cuyos operandos pueden encontrarse en memoria. Se dice por tanto que se trata de arquitecturas registro-memoria. Algunas máquinas CISC emplean instrucciones de 3 direcciones al estilo RISC (como la VAX) pero la mayoría emplean solo 2 direcciones (como las x86 y 680x0): uno de los operandos siempre se sobrescribe con el resultado de la operación.

Las instrucciones de dos direcciones son más compactas, pero las de 3 direcciones permiten que ambos operandos se reutilicen en las operaciones siguientes. Esta reutilización de valores permite a las máquinas RISC minimizar el número de restricciones en el orden de las instrucciones dando más libertad al compilador para elegir el orden de las instrucciones.

Ejemplo 1:

La instrucción de código intermedio:

$x := y - z$

puede traducirse instrucción a instrucción en la siguiente secuencia:

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

Pero si el objetivo es producir buen código máquina, podemos mejorar la traducción anterior. Si las instrucciones que calcularon los valores de y y z dejaron sus valores en un registro, podemos evitar las dos instrucciones de carga. De igual forma, si la instrucción que va a usar el valor de x lo usa directamente desde el registro 2 ($R2$) podemos evitarnos la instrucción de almacenaje final. Este ejemplo muestra la importancia de realizar una buena asignación de valores a cada registro.

2. Selección de instrucciones mediante reescritura de árboles

2.1. Conceptos básicos: Esquema de traducción de árboles

El generador de código debe establecer la correspondencia entre la representación intermedia del programa y la secuencia de código máquina que será ejecutada en la máquina destino. En las arquitecturas de tipo CISC es una fase bastante ardua debido a la gran cantidad de modos de direccionamiento que permiten.

Una forma bastante habitual de llevar a cabo este proceso es mediante la reescritura de árboles. Con esta técnica, cada instrucción destino se representan mediante un árbol, llamado *árbol patrón o plantilla*. Por ejemplo, el árbol patrón



representa una instrucción que almacena en la posición de memoria x el contenido del registro i .

El *tamaño* de un árbol patrón puede definirse como el número de nodos que contiene. Los árboles patrón de las instrucciones de máquinas CISC son bastante grandes, mientras que los árboles patrón de las instrucciones de una máquina con arquitectura RISC son pequeños y tienen un tamaño uniforme.

A partir de los árboles patrones que representan el juego de instrucciones destino, se pueden definir unas reglas de reescritura como las mostradas en la Tabla 2.


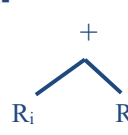
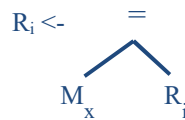
Árbol patrón	Instrucción destino	Significado
$R_i \leftarrow C_a$	LD R_i , #a	Carga en registro i la constante a
$R_i \leftarrow M_x$	LD R_i , x	Carga en registro i el contenido de la posición de memoria x
$M_x \leftarrow$ 	ST x , R_i	Almacena en la posición de memoria x el contenido del registro i
$R_i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

Tabla 2: Reglas de reescritura asociadas a algunas instrucciones de la máquina destino. [Aho, Lam, Sethi, Ullman, 2008]

Dado un árbol representando el código intermedio a traducir, el proceso de *selección de instrucciones* consiste en encontrar un revestimiento del árbol de código intermedio usando el mínimo número de árboles patrón sin que estos se solapen. Para ello se emplean los árboles patrón como reglas de reescritura. Así por ejemplo, la aplicación de la regla de reescritura de la Tabla 2:



consistirá en sustituir el subárbol



por el nodo R_i

No es tan sencillo como pueda parecer:

- No hay un único revestimiento posible.
- 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.

En la búsqueda del revestimiento óptimo, podemos encontrarnos con algoritmos que persiguen el *revestimiento óptimo* (*optimum tiling*), donde la suma de costes de cada árbol patrón es el mínimo, y algoritmos de *revestimiento satisfactorio* (*optimal tiling*), donde no habrá dos árboles patrón adyacentes que puedan combinarse en uno solo de menor coste. Lógicamente, los algoritmos para obtener revestimientos satisfactorios son más sencillos que los que obtienen revestimientos óptimos.

Puesto que los árboles patrón de las instrucciones de máquinas CISC son bastante grandes, la diferencia entre un revestimiento óptimo y satisfactorio es mayor. En cambio, como los árboles patrón de las instrucciones de arquitecturas RISC son pequeños y tienen un coste uniforme, no suele haber apenas diferencias entre un revestimiento óptimo y uno satisfactorio.

Algoritmo de revestimiento satisfactorio *Maximal_Munch*(A)

1. Encontrar el árbol patrón (AP) más grande que puede incluir al nodo raíz de A
2. Cubrir el nodo raíz, y posiblemente otros nodos adyacentes, con este árbol patrón AP.
3. Generar las instrucciones correspondientes al árbol patrón AP
4. Sean $SAP_1, SAP_2, \dots, SAP_n$ los árboles en que AP ha dividido a A
5. Mientras quede un subárbol SAP_i sin cubrir hacer *Maximal_Munch* (SAP_i)

Algoritmo 1: Revestimiento satisfactorio. Fuente: [Appel, Ginsburg, 1998]

Cada vez que se usa un árbol patrón se genera la instrucción máquina correspondiente. Como se comienza por el nodo raíz, las instrucciones se generarán en el *orden inverso* al que tienen

que ejecutarse. Pero no es la única forma de hacerlo, ya que hay otros algoritmos que comienzan el revestimiento por las hojas.

El Algoritmo 1 obtiene un revestimiento satisfactorio, pero no siempre óptimo. Hay otros algoritmos basados en programación dinámica que permiten obtener un revestimiento óptimo (ver [Appel, Ginsburg.1998]).

Ejemplo 2:

Considera el juego de instrucciones y sus correspondientes árboles patrón mostrados en la Tabla 3.

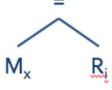
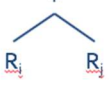
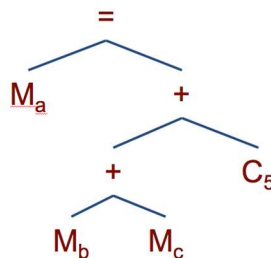
Árbol patrón	Instrucción	Significado
$R_i \leftarrow C_a$	LD R_i , #a	Carga en registro i la constante a
$R_i \leftarrow M_x$	LD R_i , x	Carga en registro i el contenido de la posición de memoria x
$M \leftarrow$ 	ST x , R_i	Almacena en la posición de memoria x el contenido del registro i
$R_i \leftarrow$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

Tabla 3 Juego de instrucciones para el ejemplo 2

La representación intermedia (en forma arbolescente) para la instrucción $a = b + c + 5$ sería:



Un posible revestimiento para el árbol anterior produciría la secuencia de instrucciones

```
LD R0, b
LD R1, c
ADD R0, R0, R1
LD R1, #5
ADD R0, R0, R1
ST a, R0
```

2.2. Selección de instrucciones mediante revestimiento sintáctico

Una representación arbolescente del código intermedio se puede convertir en una cadena usando una notación prefija (recorrido en profundidad - preorden del árbol). Con esto se consigue convertir el árbol en una cadena perteneciente a un lenguaje.

Se puede escribir una *regla sintáctica para cada árbol patrón*, en la que el lado derecho de una regla es una representación prefija del árbol patrón correspondiente. Las acciones semánticas asociadas a cada regla generan las instrucciones de código máquina correspondientes al árbol patrón.

A partir de esta gramática, se construye un *analizador sintáctico LR*.

El proceso de revestimiento consiste en realizar el análisis sintáctico de la cadena en la que se ha convertido el árbol de código intermedio

Estas gramáticas suelen ser ambiguas y hay que tener cuidado en la forma de resolver los conflictos. Se suele dar mayor preferencia a las reglas que representan árboles patrón con menor coste. Si no hay información sobre costes, se puede dar más prioridad a las reglas más largas. En caso de conflictos reducción/desplazamiento se puede dar prioridad al desplazamiento.

Ventajas

- Los analizadores sintácticos LR son eficientes y robustos, con lo que se consigue construir generadores de código eficientes y robustos.
- Es sencillo portar el generador de código a otra máquina.
- Puede mejorarse el generador de código añadiendo reglas con características específicas de la máquina destino.

Inconvenientes

- El orden de evaluación debe ser 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.

Ejemplo 3

Considera el juego de instrucciones del ejemplo 2. Las reglas sintácticas y sus acciones semánticas asociadas serían

```

Ri -> Ca           { LD Ri , #a }
Ri -> Mx           { LD Ri , x }
Mx -> = Mx Ri      { ST x, Ri }
Ri -> + Ri Rj      { ADD Ri, Ri, Rj }
Mx -> m
    
```

El recorrido en preorden para la representación intermedia del ejemplo 2 producirá la cadena
 = M_a + + M_b M_c C₅

El análisis sintáctico ascendente correspondiente será la secuencia de reducciones

```

= Ma + + Mb Mc C5 -> = Ma + + R0 Mc C5 -> = Ma + + R0 R1 C5 ->
= Ma + R0 C5 -> = Ma + R0 R1 -> = Ma R0 -> Ma
    
```

Que al ejecutar las acciones semánticas producirá la secuencia de instrucciones

```
LD R0, b
LD R1, c
ADD R0, R0, R1
LD R1, #5
ADD R0, R0, R1
ST a, R0
```

3. Asignación de registros

3.1. Jerarquía de memoria

La memoria que se encuentra fuera del chip del procesador tiene tiempos de acceso mucho mayores que las que se encuentra en el chip pero es mucho más económica. Por esa razón se establece una jerarquía de memoria (de más rápida/cara a más lenta/económica y abundante).

	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 - ...	

Tabla 4: Jerarquía de memoria. Fuente: [Scott, 2009]

Los compiladores gestionan los registros explícitamente, cargando valores en ellos desde memoria cuando los necesitan y almacenando sus valores en memoria cuando terminan o cuando los registros se necesitan para otras tareas.

Uso de los registros:

- Almacenar los operandos de una instrucción
- Almacenar valores temporales. Se guarda el resultado de una subexpresión mientras se evalúa la expresión.
- Guardar valores globales que se calculan en un bloque básico y se utilizan en otros. 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.

Como el número de registros es limitado, es necesario decidir qué valores se almacenan en cada registro. El proceso de asignación de registros decide en qué registro debe almacenarse cada valor. La idea es asignar registros a las variables que se emplean con más frecuencia y se mantienen consistentes entre los límites de los bloques básicos (de forma global). Por ejemplo, se pueden asignar un número fijo de registros para almacenar los valores más activos dentro de cada bucle, y el resto de los registros pueden usarse para almacenar valores locales

3.2. Asignación de registros mediante coloración de grafos

El objetivo de la asignación de registros es asignar el mayor número posible de variables temporales en el menor número de registros posible. Cuando se necesita un registro para

realizar un cálculo pero todos los registros están en uso, el contenido de uno de los registros debe almacenarse en memoria.

Adicionalmente, sería deseable conseguir que el origen y destino de las instrucciones MOVE sea el mismo para poder eliminar la instrucción.

Llamamos *grafo de interferencias* a un grafo no dirigido donde:

- Cada nodo representa un valor (variable).
- Hay una arista entre los nodos t_1 y nodo t_2 (t_1, t_2) si los valores que representan ambos nodos no pueden asignarse al mismo registro porque representan variables activas al mismo tiempo.
- Si el procesador no puede producir el resultado de $a := b + c$ en el registro r_i añadir el arco (a, r_i) .

Coloreado del grafo de interferencias

- Si se dispone de k registros, se realiza un k -coloreado del grafo de interferencias, que consiste en asignar un color de entre los k posibles a cada nodo, de manera que dos nodos adyacentes no tengan el mismo color. Se intentará usar el mínimo número de colores posibles.
- Si no es posible encontrar un k -coloreado del grafo, algunos valores temporales deberán almacenarse en memoria.

El problema del k -coloreado de un grafo es un problema NP-completo, pero se suelen usar aproximaciones como el Coloreado por simplificación.

Algoritmo de coloreado por simplificación

1. Construcción

Construir el grafo de interferencias usando análisis del flujo de datos.

Añadir un arco entre todas las variables temporales activas en cada punto del programa.

2. Simplificación

Para colorear el grafo usaremos una heurística sencilla: Si hay un nodo m con menos de k vecinos, eliminarlo del grafo.

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

Para este proceso se usará una pila. La simplificación consiste en ir eliminando repetidamente del grafo de interferencias nodos de grado menor que k y apilarlos.

3. Volcado

Si en algún momento durante la simplificación solo hay nodos de grado mayor o igual que k , la heurística puede estar fallando (o el grafo no es k -coloreable). Entonces se elige un nodo (preferiblemente de entre las variables temporales) y se convierte en candidato para ser volcado a memoria (su valor no estará en registros sino en memoria durante la ejecución).

Eliminar el nodo del grafo y volver a la fase de simplificación.

4. Selección

Se asignan colores a cada nodo del grafo

Comenzar con el grafo vacío y reconstruir el grafo desapilando nodos. Al desapilar un nodo se le asigna un color de entre los disponibles.

Al desapilar un nodo marcado como candidato a ser volcado a memoria:

- Si sus vecinos usan los k colores, se convierte en volcado real, no se le asigna color y se continua en la fase de selección.
- Si 2 o más de sus vecinos tienen el mismo color, se puede colorear y por lo tanto deja de ser un nodo “volcado”

5. Finalización

Si en la fase de selección no se pudo encontrar color para algún nodo, reescribir el programa para poner los valores volcados en memoria y reaplicar el algoritmo.

Ejemplo 4

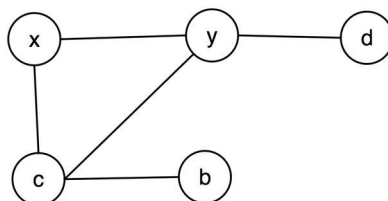
Construye el grafo de interferencias para el siguiente bloque de instrucciones, sabiendo que las variables activas a la entrada son $\{y,d\}$ y a la salida $\{x,y\}$.

```
c=y*d
x=y-2
b=x+y
x=5*b
y=c+x
```

Calculamos las variables activas en cada instrucción:

	Def	Usa	Ent[]	Sal[]
c:=y*d	c	y,d	y,d	c,y
x:=y-2	x	y	c,y	c,x,y
b:=x+y	b	x,y	c,x,y	c,b
x:=5*b	x	b	c,b	c,x
y:=c+x	y	c,x	c,x	x,y

El grafo de interferencias quedará:



Ejemplo 5

Realiza la asignación de registros para el siguiente bloque básico de código. Solo se dispone de 4 registros y las variables activas a la entrada y salida del bloque son las siguientes:

$Ent[B] = \{j,k\}$ $Sal[B] = \{d,k,j\}$

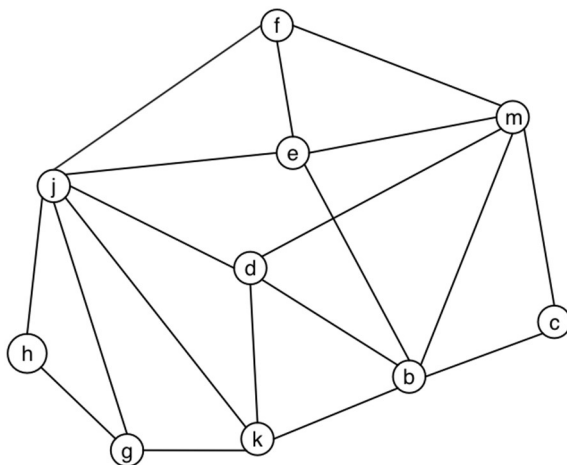
```

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
    
```

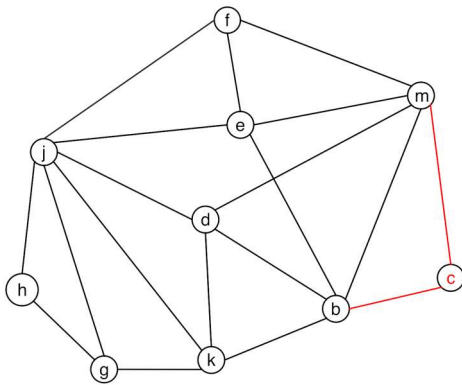
Calculamos las variables activas antes y después de cada instrucción para conocer las variables que están activas simultáneamente y poder construir el grafo de interferencias.

	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

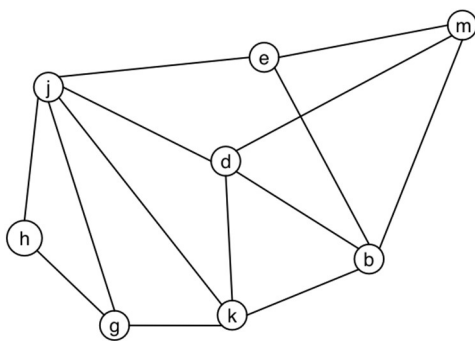
El grafo de interferencias resultante es:



En la fase de simplificación elegimos algún nodo con menos de k (4) nodos adyacentes, por ejemplo **c**, y lo eliminamos del grafo. Apilamos este nodo en la pila: PILA=[c].



Seguidamente podemos eliminar, por ejemplo, el nodo f (que actualmente tiene un grado menor de 4) y lo apilamos: PILA=[f, c].

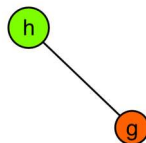


Y así sucesivamente, hasta eliminar todos los nodos. Es posible seguir varios órdenes en este proceso de simplificación, por ejemplo PILA=[h, g, j, k, d, b, e, m, f, c].

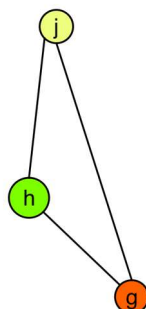
Seguidamente, en la fase de selección, se va desapilando nodo a nodo y se le asigna a cada uno un color (registro) que no esté siendo usado por ningún nodo adyacente. Comenzamos con el nodo h que se encuentra en la cima de la pila y le asignamos uno de los 4 colores disponibles.



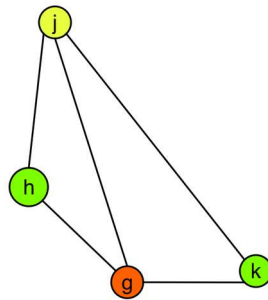
Seguidamente desapilamos el siguiente nodo (g) y le asignamos cualquiera de los colores que no esté siendo usado por el nodo h (que es adyacente).



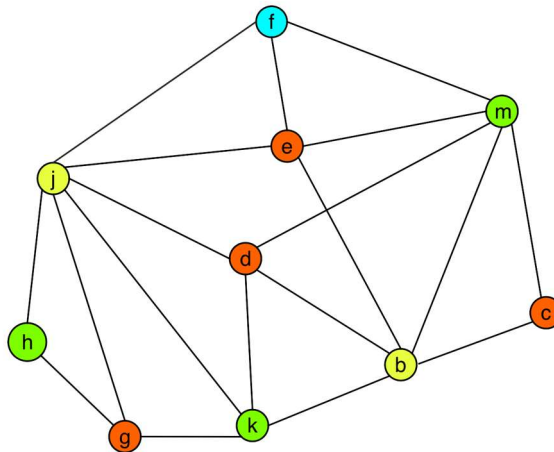
El siguiente nodo es el j, al que se le asigna un nuevo color.



De igual forma se asigna un color al nodo k.



Siguiendo este mismo proceso, se acaba asignando un color distinto a cada nodo.



Por lo tanto, los registros asignados a cada valor son los siguientes:

	Registro
c	R1
f	R4
m	R2
e	R1
b	R3
d	R1
k	R2
j	R3
g	R1
h	R2

4. Bibliografía.

- [Aho, Lam, Sethi, Ullman 2008]. Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman. *Compiladores. Principios, técnicas y herramientas*. 2ª edición. Pearson. 2008
- [Appel, 1998]. Andrew Appel. *Modern Compiler Implementation in C*. Cambridge University Press. 1998.
- [Scott, 2009] Michael L. Scott. *Programming Language Pragmatics*. 3rd edition. Elsevier, 2009