Tema 10. Generación de Código

Francisco José Ribadas Pena

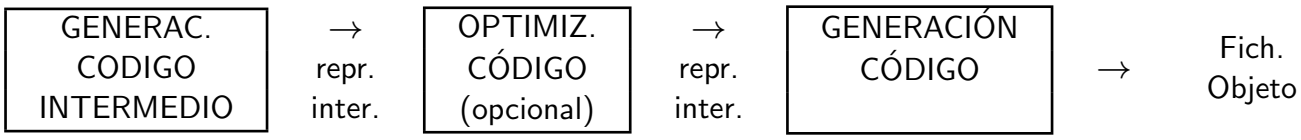
PROCESADORES DE LENGUAJES
4º Informática
ribadas@uvigo.es

11 de abril de 2012

10.1 Introducción

Fase final de la compilación.

Objetivo: generar el mejor código objeto posible para la R.I.



Entrada: código intermedio (RPN, cod. 3 direcciones, árboles o GDAs,...)
Salida: código objeto (3 opciones)

1. No generar código objeto
   - Compilación termina al generar R.I.
   - Una máquina abstracta interpretará la R.I.
   - Común en intérpretes (Prolog, Basic, ...) y compiladores incrementales (agiliza ciclo edición-recompilación)
2. Generar código ensamblador máquina obejeto
   - Facilita traducción
   - Necesita un ensamblador para generar verdadero cod. objeto
3. Generar código máquina directamnete
   - Alta dependencia del Sist. Operativo y de la arquitectura CPU
   - Opciones:
     a) Código absoluto: programa se localizará en una posición fija de memoria
     $\longrightarrow$ Ejecución directa, pero esquema muy restrictivo
     $\longrightarrow$ Ejemplo: ejecutables **.com** en MS/DOS
     b) Código relocalizable: código del programa puede situarse en cualquier posición
     $\longrightarrow$ Permite *compilación separada* de distintos módulos
     $\longrightarrow$ Necesario incluir información de localización en fich. objeto
     $\longrightarrow$ Necesidad de procesamiento adicional (enlazadores/*linkers* y cargadores)

10.1 (cont)

Tareas:

- Administración de memoria
  - Determinar direcciones definitivas para nombres del programa (variables, funciones)
  - Convertir etiquetas en las direcciones de instrucciones reales correspondientes
- Selección de instrucciones
  - Elegir instrucciones, modos de direccionamiento, etc,...
  - Asignación de registros eficiente
  - Maximizar: velocidad de ejecución
  - Minimizar: tamaño ejecutable + necesidades de memoria

Máquina objetivo: Arquitecturas:

- CISC (*complex instruction set computer*)
  - Gran número de instrucciones máquina
  - Instrucciones potentes (realizan tareas complejas)
  - Múltiples modos direccionamiento + pocos registros
- RISC (*reduced instruction set computer*)
  - Número limitado de instrucciones máquina
  - Instrucciones realizan tareas simples
  - Modos direccionamiento limitado + muchos registros
- En general, RISC facilita compilación
  - Conj. instrucciones más uniforme y completo
  - Facilita seleccionar instrucciones y optimizar registros
  - Dificil usar directamente instrucciones CISC complejas

Ejemplo máquina objetivo

- Microprocesador con n registros: R0, R1, R2, ... Rn-1
- Modos direccionamiento

Modo	**Notación**	**Dir. accedida**	**Coste adicional**
absoluto	M	M	1
registro	R	R	0
indexado	c(R)	contenido(R) + c	1
registro indexado	*R	contenido(R)	0
indexado indirecto	*c(R)	contenido(contenido(R)+c)	1
literal (constante)	#n	-	-

- Instrucciones de 2 direcciones de la forma

OP	**fuente**	,	**destino**
↓	↓		↓
codigo operación	dir. oper 1		dir. oper 2 y resultado

- Instrucciones básicas en ensamblador

`MOV a,b`	mueve contenido de la dir. **a** a la dir. **b**
`ADD a,b`	suma contenido de **a** y de **b** guarda resultado en **b**
	IDEM para **SUB a,b**, **MUL a,b**, **DIV a,b**
`GOTO etq`	salta al direccion **etq**
`CALL proc`	llamada a un trozo de código (subrutina) que comienza en la dir. **proc**
`RETURN`	retorno de un **CALL**
`HALT`	parada de la ejecución
`BGZ etq`	salto a la dir. **etq** si resultado de última operación fue > 0
`BLZ etq`	salto a la dir. **etq** si resultado de última operación fue < 0
`BEZ etq`	salto a la dir. **etq** si el resultado de última operación fue 0
...	...

- Ejemplos:

`ADD *7(R1), R2` guarda en R2 el resultado de:
`contenido(contenido(R1)+7) + R2`

`MOV #17,3(R1)` guarda el valor 17 en la posición de memoria:
`contenido(R1)+3`

10.2 Generación en Modo Interpretación

Idea Base: Asociar a cada tupla de código intermedio una secuencia de instrucciones de código objeto fija (*"caja de código"*).

Ejemplo: generación simple para cuartetos.

```
(operacion, op1, op2, resultado)   ⇝   MOV op1, Ri
                                       operacion op2, Ri
                                       MOV Ri, resultado
```

$r := (a - b) * c + d * f$

```
(1)  (-,a,b,tmp1)
(2)  (*,tmp1,c,tmp2)
(3)  (*,d,f,temp3)
(4)  (+,tmp2,tmp3,r)
```

```
(1)  MOV a, R0          (3)  MOV d, R0
     SUB b, R0               MUL f, R0
     MOV R0, tmp1            MOV R0, tmp3
(2)  MOV tmp1, R0       (4)  MOV tmp2, R0
     MUL c, R0               ADD tmp3, R0
     MOV R0, tmp2            MOV R0,r
```

Problema: Código de muy mala calidad (aunque correcto)

- Redundante (repetición operaciones)
- Mal aprovechamiento de registros

  - no reutilización
  - cargas/descargas innecesarias

  $\Rightarrow$ aumento tráfico memoria

Necesidad de usar info. contextual

- Mantener traza del contexto (estado) del programa mientras se genera código
- Info. contextual mejora selección instrucciones y asignación registros
  - $\longrightarrow$ aprovechar valores ya calculados
  - $\longrightarrow$ gestionar eficientemente temporales CI (uso de registros)
  - $\longrightarrow$ reutilizar registros cuyo contenido no vuelva a utilizarse
  - $\longrightarrow$ aprovechar modos direccionamiento CPU (evita cálculos de direcciones)

Nos centraremos en:

- Evitar cálculos redundantes
  - identificar subexpresiones comunes
  - si ya han sido calculadas y su valor aún es válido $\Rightarrow$ reutilizarlas
    $\longrightarrow$ un valor sigue siendo válido si no se han asignado nuevos valores a los operandos
- Rastrear uso de registros
  - mantener info. sobre el contenido de los registros
  - mantener info. sobre el uso de las variables
  - OBJETIVO: mejorar asignación/libreración/reutilización registros
    $\longrightarrow$ evitar accesos innecesarios a memoria
    $\longrightarrow$ mantener valores usados frecuentemente en registros

Nos limitaremos a un estudio local, dentro de *bloques básicos*

Información a recoger: actividad y uso de las variables

Variables Activas

Una instrucción "`X := Y op Z`":

- define la variable **X**
- usa las vars. **Y** y **Z**, si **Y** y **Z** no fueron modificadas desde su definición

Una variable está **Activa** en punto dado del programa si despues de haber sido definida será usada en algún otro punto del programa.

- **X** está Activa desde que se define hasta el último uso que se hace de ella
- **X** está Inactiva desde su último uso hasta que se define nuevamente

Idea Base: Intentar mantener en registros CPU las variables activas

10.2.1 Bloques Básicos y Diagramas de Flujo

Bloque Básico (B.B.): Secuencia consecutiva de instrucciones de C.I. donde el flujo de control es lineal (no hay saltos intermedios)

Grafo de Flujo: Herramienta para estructurar y recoger info. de un programa

- Nodos : Bloques Básicos
- Arcos : Representan flujo de control entre B.B.

Identificación Bloques Básicos

1. Identificación de líderes
   - Lider: primera instrucción de un B.B.
     a) La primera instrucción del programa es LÍDER
     b) Cualquier instrucción destino de un salto es LÍDER
     c) Cualquier instrucción inmediatamente despues de un salto es LÍDER
2. Para cada LÍDER: su B.B. comprende las instrucciones que van desde ese líder hasta el siguiente líder (sin incluirlo) o hasta el final de programa

Construcción Grafos de Flujo

1. Si el B.B. termina en salto incondicional:
   - incluir un arco hasta el B.B. destino del salto
2. Si el B.B. termina en salto condicional
   - incluir un arco hasta el B.B. destino del salto
   - incluir otro arco al siguiente B.B.
3. En otro caso: incluir arco al siguiente B.B. (si lo hay)

Ejemplo

Código Fuente

```
int[100] a;
...
void quicksort(int m,n) {
    int i,j;
    int v,x;

    if (n <= m) return;
    i = m - 1; j = n; v = a[n];
    while(1) {
        do i = i + 1
        while (a[i] < v);
        do j = j - 1
        while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] := x;
    }
    x = a[i]; a[i] = a[n]; a[n] := x;
    quicksort(m, j);
    quicksort(i + 1, n);
}
```

Bloques Básicos

B_1
- (101) i = m - 1 ← (a)
- (102) j = n
- (103) t1 = n*4
- (104) v = a[t1]

B_2
- (105) i = i + 1 ← (b)
- (106) t2 = 4*i
- (107) t3 = a[t2]
- (108) if t3 < v goto (105)

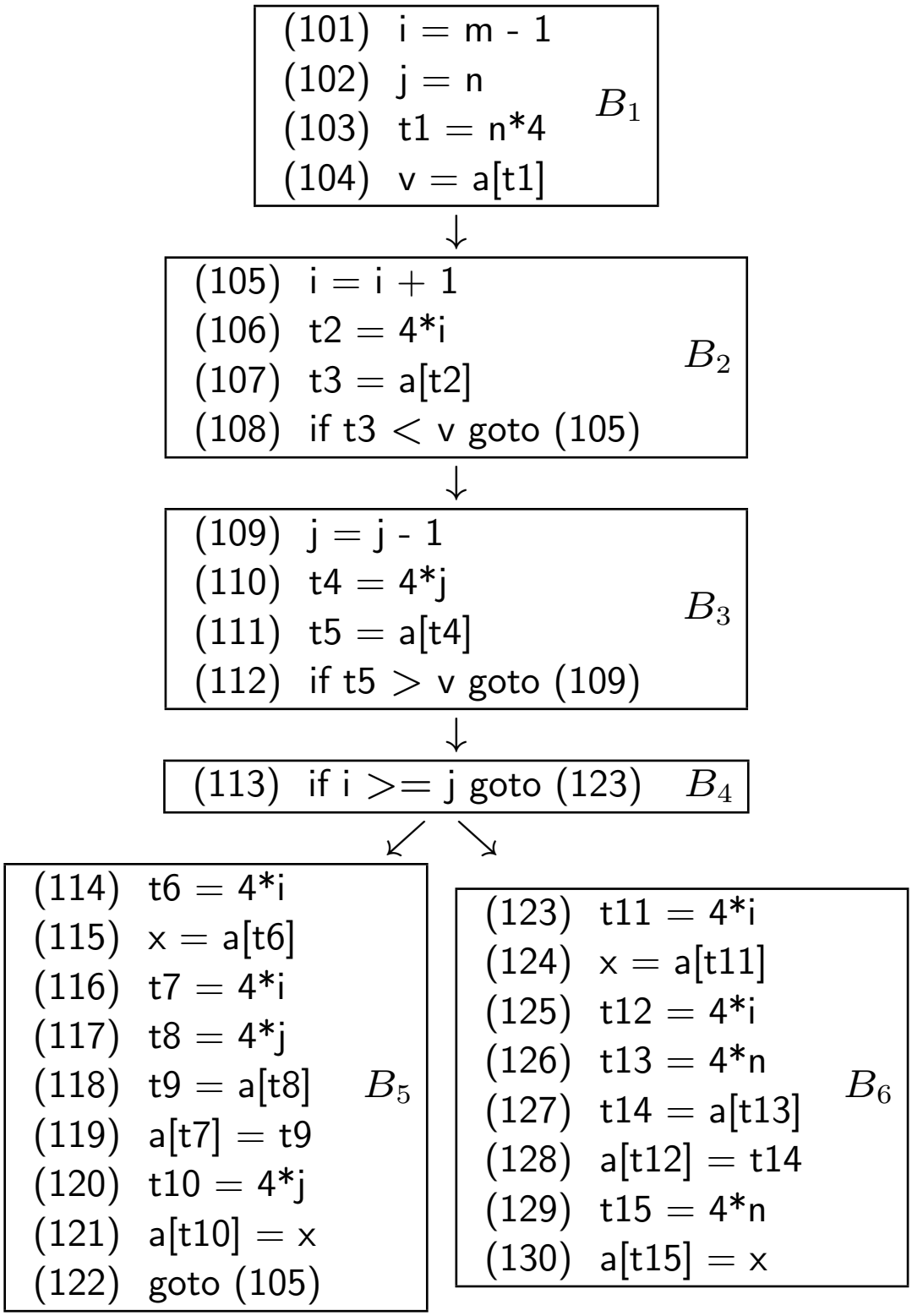
B_3
- (109) j = j - 1 ← (c) (b)
- (110) t4 = 4*j
- (111) t5 = a[t4]
- (112) if t5 > v goto (109)

B_4
- (113) if i >= j goto (123) ← (c)

B_5
- (114) t6 = 4*i ← (c)
- (115) x = a[t6]
- (116) t7 = 4*i
- (117) t8 = 4*j
- (118) t9 = a[t8]
- (119) a[t7] = t9
- (120) t10 = 4*j
- (121) a[t10] = x
- (122) goto (105)

B_6
- (123) t11 = 4*i ← (b) (c)
- (124) x = a[t11]
- (125) t12 = 4*i
- (126) t13 = 4*n
- (127) t14 = a[t13]
- (128) a[t12] = t14
- (129) t15 = 4*n
- (130) a[t15] = x

Grafo Flujo


(101) i = m - 1
(102) j = n
(103) t1 = n*4
(104) v = a[t1]
B_1
(105) i = i + 1
(106) t2 = 4*i
(107) t3 = a[t2]
(108) if t3 < v goto (105)
B_2
(109) j = j - 1
(110) t4 = 4*j
(111) t5 = a[t4]
(112) if t5 > v goto (109)
B_3
(113) if i >= j goto (123)
B_4
(114) t6 = 4*i
(115) x = a[t6]
(116) t7 = 4*i
(117) t8 = 4*j
(118) t9 = a[t8]
(119) a[t7] = t9
(120) t10 = 4*j
(121) a[t10] = x
(122) goto (105)
B_5
(123) t11 = 4*i
(124) x = a[t11]
(125) t12 = 4*i
(126) t13 = 4*n
(127) t14 = a[t13]
(128) a[t12] = t14
(129) t15 = 4*n
(130) a[t15] = x
B_6

10.2.2 Generación de Código para B.B.

Se generará código para cada B.B. de forma independiente.

- Todos los valores se guardarán en registros (si es posible)
- Sólo se pasarán valores a memoria:
  - si el registro es necesario para otro cálculo
  - antes de llamadas a procedimiento
  - antes de salir del B.B.

    $\longrightarrow$ todas las vars. estarán en memoria al pasar al siguiente B.B.

Generación en dos pasadas:

1. Recoger info sobre Activación + Uso de Variables
   - Recorrido hacia atrás de las instrucciones del B.B.
   - OBJETIVO: determinar para cada instrucción `[X = Y op Z]` los siguientes usos de `X`, `Y`, `Z` y si están activas en ese punto o no
   - Info. recogida se guarda en TDS
2. Generar código asignando registros
   - Recorrido descendente de las instrucciones del B.B., decidiendo utilización de registros en cada instrucción
   - Usa información recopilada en fase anterior
   - Info. necesaria (estructuras de datos)
     - Descriptor de Registros: info. sobre el contenido de cada registro

       $\longrightarrow$ Para cada registro: lista de variables cuyos valores almacena, o marca de libre

       $\longrightarrow$ Inicialmente todos los registros están vacíos
     - Descriptor de Variables: info. sobre la ubicación del valor actual de una variable durante la ejecución

       $\longrightarrow$ Para cada variable/temporal: lista de posiciones (dirs. de mem. o registro) donde está su valor actual

       $\longrightarrow$ Esta información puede almacenarse en la tabla de símbolos

ALGORITMO: Informacion sobre actividad y uso de las variables en un BB
ENTRADA: Bloque básico de instrucciones de tres direcciones
SALIDA: Para cada instrucción i, la info sobre actividad y uso de `X`, `Y`, y i: `[X = Y op Z]`

- Inicilizamos info de la TDS con todas las variables no temporales activas (al final del BB)
- Recorremos el BB del final al principio.
  Para cada instrucción i: `[X = Y op Z]`
  1. Se adjunta a i, la info sobre actividad y uso de `X`, `Y`, y `Z`, almacenada en la TDS
  2. `X` se establece como no activa y sin uso en la TDS
  3. `Y` y `Z` se establecen como activas y con siguiente uso en i en la TDS

  Para cada instrucción i: `[X = op Y]` o i: `[X = Y]`, se realiza el mismo procedimiento, sin `Z`

ALGORITMO: Asignación de registros para generación de código objeto
ENTRADA: Bloque básico de instrucciones de tres direcciones e Información sobre activación de variables
SALIDA: Código objeto

- Para cada instrucción i: `[X = Y op Z]`
  1. Se determina la posición libre óptima `P` para el resultado `X`, a ser posible un registro
  2. Se consulta descriptor de variables de `Y`, para determinar su posición actual `Y'`
     Si `Y` en memoria y registro $\Rightarrow$ se escoge registro como `Y'`
  3. ídem para `Z` y su posición actual `Z'`
  4. Si `Y'` distinto de `P` $\Rightarrow$ se copia `Y` a `P`: `[MOV Y', P]`
  5. Se genera `OP Z', P`
     `P` se añade a descriptor de variables de `X`
     Si `P` es un registro $\Rightarrow$ se añade `X` a su descriptor de registro, y `X` se elimina del resto de descriptores de registros
  6. Si `Y` y `Z` no están activos tras i y están en registros, se mueven a memoria, y se eliminan `Y` y `Z` de los descriptores de registros en los que se aparezcan
- Para instrucciones i: `[X = op Y]` se emplea el mismo procedimiento
  Caso especial: Asignación `[X = Y]`
  1. Si `Y` está en registro `Y'` $\Rightarrow$ cambio de descriptores de registros y variables para reflejar que `X` está en `Y'`
  2. Si `Y` en memoria $\Rightarrow$ igual, pero antes `Y` se carga en registro `Y'`
  3. Si `Y` no activa en este punto `Y'` $\Rightarrow$ `Y` se mueve a memoria, y se elimina del descriptor de registro de `Y'`

ALGORITMO: Determinación de posición óptima `P`
ENTRADA: Instrucción `[X = Y op Z]` o `[X = op Y]`
SALIDA: Posición `P` en la que se almacena `X`

1. Si `Y` se encuentra en un registro `Y'` no compartido con otras variables, e `Y` no está activa $\Rightarrow$ Se devuelve `Y'`
2. Si lo anterior falla y hay un registro vacío $\Rightarrow$ se devuelve dicho registro
3. Si falla lo anterior, `X` está activa después de la instrucción y `op` requiere un registro como resultado:
   - Se escoge registro ocupado `R` $\Rightarrow$ aquel que se referencie más tarde o cuyo valor ya esté en memoria
   - El valor de `R` se almacena en pos. de memoria de las variables referenciadas por `R`: `[MOV R, M1], [MOV R, M2], ...`
   - Se actualiza descriptor/es de variable/s apropiado/s para indicar que el contenido de `R` se ha copiado a `M1, M2, ...`
   - Se devuelve `R`
4. Si `X` no activa tras la instrucción o no se puede encontrar un registro adecuado $\Rightarrow$ se devuelve la pos. de memoria de `X`

EJEMPLO: `d = (a-b) + (a-c) + (a-c)` $\Rightarrow$

```
t1 = a-b
t2 = a-c
t3 = t1+t2
d = t3+t2
```

		Descr. registros	Descr. variables
`t1 = a-b`	MOV a, R0 SUB b, R0	t1 en R0	R0 en t1
`t2 = a-c`	MOV a, R1 SUB c, R1	t1 en R0 t2 en R1	R0 en t1 R1 en t2
`t3 = t1+t2`	ADD R1, R0	t3 en R0 t2 en R1	R1 en t2 R0 en t3
`d = t3+t2`	ADD R1, R0 MOV R0, d	t2 en R1 d en R0	R1 en t2 R0 y pos. mem en d

ACCESO A VALORES INDEXADOS

Instrucciones de acceso a arrays $\Rightarrow$ se selecciona un registro `R`

	`i` en registro `Ri`	`i` en pos. mem. `Mi`
`a = b[i]`	`MOV b(Ri), R`	`MOV Mi, R` `MOV b(R), R`
`a[i] = b`	`MOV b, a(Ri)`	`MOV Mi, R` `MOV b, a(R)`

Instrucciones de acceso a punteros $\Rightarrow$ se selecciona un registro `R`

	`p` en registro `Rp`	`p` en pos. mem. `Mp`
`a = *p`	`MOV *Rp, R`	`MOV Mp, R` `MOV *R, R`
`*p = a`	`MOV a, *Rp`	`MOV Mp, R` `MOV a, *R`

10.2.3 Generación de Código para B.B. basada en GDAs

Idea: Representar las expresiones del B.B en un GDA (grafo dirigido acíclico)

HOJAS = constantes o vars, iniciales que recibe el bloque

NODOS = operadores, se les asocian las variables ó temporales donde se guarda el resultado

- Facilita identificar expresiones comunes

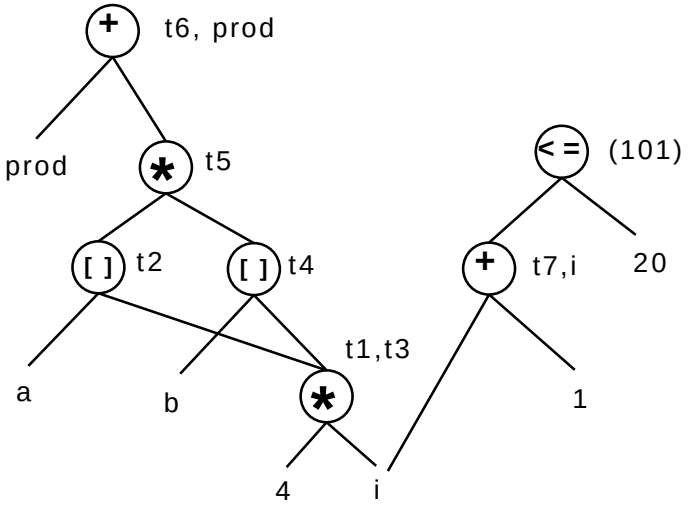
  $\longrightarrow$ expresiones comunes representadas forma implícita (tienen 2 o más padres)
- Facilita transformaciones sobre las instrucciones para mejorar el código
- Existen algoritmos para recorrer el GDA realizando la generación óptima de código

Ejemplo:

CODIGO INICIAL

(101) t1 = 4*i
(102) t2 = a[t1]
(103) t3 = 4*i
(104) t4 = b[t3]
(105) t5 = t2*t4
(106) t6 = prod * t5
(107) prod = t6
(108) t7 = i+1
(109) i = t7
(110) if i <= 20 goto (101)

G.D.A.



CODIGO RESULTANTE

(101) t3 = 4*i
(102) t4 = b[t3]
(103) t2 = a[t3]
(104) t5 = t2*t4
(105) prod = prod * t5
(106) i = i+1
(110) if i <= 20 goto (101)

Ahorro 3 temprales: t6, t1 y t7

ALGORITMO: Construcción de un GDA
ENTRADA: Un bloque básico
SALIDA: Un GDA para el bloque básico que contiene:

1. Una etiqueta para cada nodo
   - Hoja: identificador o constante
   - Nodo interno: un símbolo de operador
2. Para cada nodo, una lista de identificadores asociados

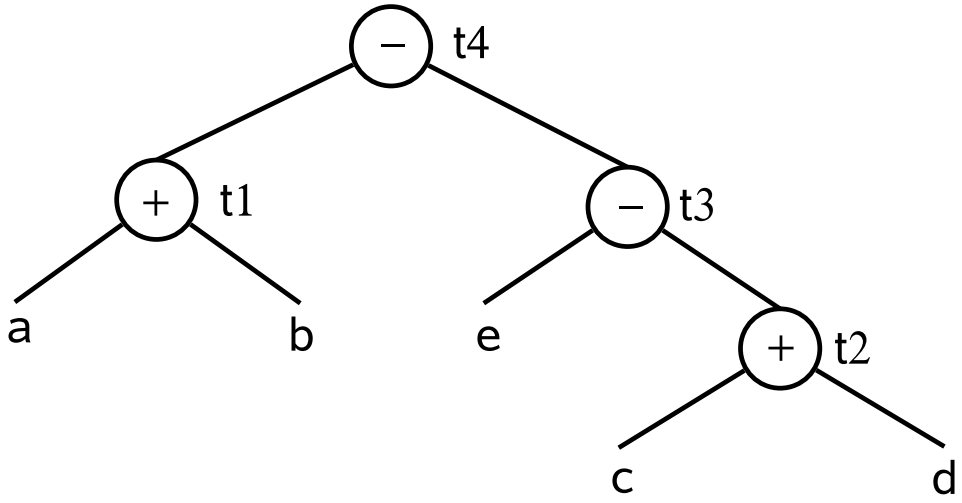
Para cada instrucción de tipo:
a) `[X = Y op Z]`
b) `[X = op Y]`
c) `[X = Y]`

`nodo(Y)` $\equiv$ nodo más recientemente creado con `Y` como identificador asociado

1. Si `nodo(Y)` no está definido, se crea. Lo mismo para `Z`, si estamos en el caso a)
2. Caso a): Se obtiene `n` = nodo etiquetado con `op`, con nodo(Y) y nodo(Z) como hijos izquierdo y derecho, respectivamente. Si no existe se crea.
   Caso b): Se obtiene `n` = nodo etiquetado con `op`, con nodo(Y) como único hijo. Si no existe se crea.
   Caso c): Se obtiene `n = nodo(y)`
3. `X` se borra de la lista de identificadores asociados a `nodo(X)`, y se añade a la lista de identificadores del nodo `n`. Se iguala `n` a `nodo(X)`

Nota: Los operadores relacionales `Y rel Z` se tratan como instrucciones de tipo a), con `X` indefinida

REORGANIZACIÓN DEL ORDEN



Reordenando instrucciones se puede ahorrar código.

Nota: suponemos sólo dos registros disponibles, `R0` y `R1`.

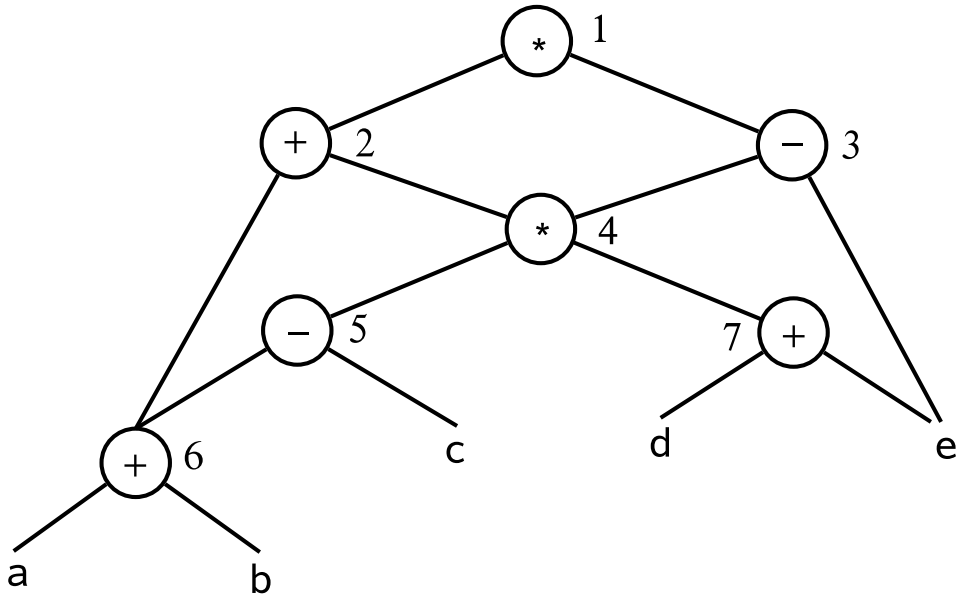
Postorden		Reordenando	
`t1 = a+b`	`MOV a, R0`	`t2 = c+d`	`MOV c, R0`
	`ADD b, R0`		`ADD d, R0`
`t2 = c+d`	`MOV c, R1`	`t3 = e-t2`	`MOV e, R1`
	`ADD d, R1`		`SUB R0, R1`
	`MOV R0, t1`	`t1 = a+b`	`MOV a, R0`
`t3 = e-t2`	`MOV e, R0`		`ADD b, R0`
	`SUB R1, R0`	`t4 = t1-t3`	`SUB R1, R0`
`t4 = t1-t3`	`MOV t1, R1`		`MOV R0, t4`
	`SUB R0, R1`		
	`MOV R1, t4`		

`t4` se calcula justo después de su hijo izquierdo `t1` $\Rightarrow$ `t1` permanece en registro para ser operado con `t4`

HEURÍSTICA: se genera código recorriendo los nodos del grafo de modo que la instrucción asociada a un nodo se genere justo después de la de su hijo izquierdo

ALGORITMO: Orden de recorrido de los nodos de un GDA
ENTRADA: GDA
SALIDA: lista de nodos a partil de la cual se generará código (una instrucción por nodo)

```
while (quedan nodos interiores sin listar) do begin
  seleccionar un nodo no listado n, con padre/s listados
  meter n en la lista
  while (el hijo más izquierdo m de n no sea hoja AND
         m no tiene padres sin listar) do begin
    meter m en la lista
    n = m
  end
end
```



Comenzando por la raíz, se añadirían nodos a la lista en el orden: 1234567, con lo que el orden de evaluación sería 7654321

```
(101) t7 = d+e
(102) t6 = a+b
(103) t5 = t6-c
(104) t4 = t5*t7
(105) t3 = t4-e
(106) t2 = t6-t4
(107) t1 = t2*t3
```

ORDENAMIENTO ÓPTIMO PARA ÁRBOLES

Objetivo: obtener el orden de recorrido de los nodos del GDA, cuando éste es un árbol, de modo que dicho recorrido produzca la secuencia de instrucciones más corta.

Dos partes:

1. Se etiqueta cada nodo del árbol de forma ascendente, con un entero que indica el número de mínimo de registros exigido para evaluar el árbol sin almacenamientos en memoria de resultados intermedios.
2. Recorrido del árbol en el orden determinado por las etiquetas de los nodos. Dados dos operandos (hijos de un mismo nodo) se evalúa primero aquel que precise más registros.

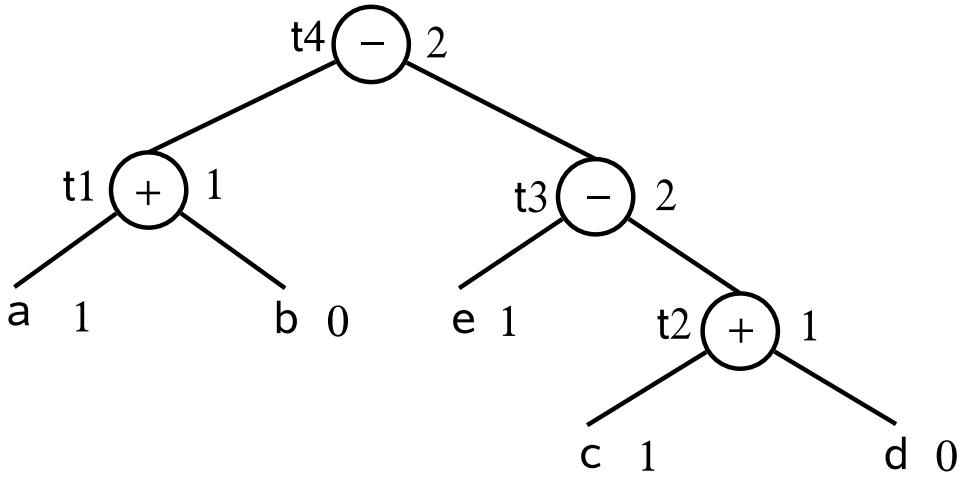
ALGORITMO: Etiquetado de nodos
ENTRADA: árbol
SALIDA: árbol etiquetado con mínimo número de registros necesarios para calcular cada nodo

```
Se recorre el árbol de forma ascendente. Para cada nodo n:
  if (n es una hoja) then
    if (n es el hijo más izquierdo)
      then etiqueta(n) = 1
      else etiqueta(n) = 0
  else begin
    sean n1, n2, ... nk hijos de n ordenados por
      etiqueta(n1) >= etiqueta(n2) >= ... >= etiqueta(nk)
    etiqueta(n) = max (etiqueta(ni) + i - 1)
  end
```

Si n es un nodo binario la fórmula de cálculo de su etiqueta (penúltima línea) a partir de las de sus hijos, se simplifica:

$$etiqueta(n) = \left\{ \begin{array}{ll} max(etiqueta(n_1), etiqueta(n_2)) & \textbf{si } n_1 \neq n_2 \\ etiqueta(n_1) + 1 & \textbf{si } n_1 = n_2 \end{array} \right\}$$

Ejemplo:



ALGORITMO: Generación de Código a partir de un GDA con forma de árbol
ENTRADA: árbol etiquetado
SALIDA: Código objeto generado a partir del árbol

Se llama a la función recursiva `generarCodigo(n)`, con el nodo raíz como argumento, que devuelve los cálculos necesarios para evaluar el árbol, almacenando el resultado de cálculo resultante en el registro R0

Se usan las estructuras auxiliares:

- `pilaReg`: pila de registros disponibles
- `pilaTemp`: pila de variables temporales disponibles

Y las funciones:

- `swap`: intercambia los dos elementos de la cima de la pila
- `push`: introduce un elemento de la cima de la pila
- `pop`: extrae un elemento de la cima de la pila
- `top`: obtiene el elemento de la cima de la pila

Nota: `r` es el número de registros del procesador

```
procedure generarCodigo(n)
begin
//CASO 0
  if (n hoja AND n hijo más a la izda de su padre)
    print 'MOV' . operando(n) . ',' . top(pilaReg)
  else if (n nodo interno) begin
    sea op = operador(n), n1 = hijo_izdo(n), n2 = hijo_dcho(n)
//CASO 1
    if (etiqueta(n2) = 0) then begin
      generarCodigo(n1)
      print op . operando(n2) . ',' . top(pilaReg)
    end
//CASO 2
    else if (1 <= etiqueta(n1) <= etiqueta(n2) AND
             etiqueta(n1) < r) then begin
      swap(pilaReg)
      generarCodigo(n2)
      R = pop(pilaReg)
      generarCodigo(n1)
      print op . R . ',' . top(pilaReg)
      push(pilaReg, R)
      swap(pilaReg)
    end
//CASO 3
    else if (1 <= etiqueta(n2) < etiqueta(n1) AND
             etiqueta(n2) < r) then begin
      generarCodigo(n1)
      R = pop(pilaReg)
      generarCodigo(n2)
      print op . R . ',' . top(pilaReg)
    end
//CASO 4
    else begin
      generarCodigo(n2)
      T = pop(pilaTemp)
      print 'MOV' . top(pilaReg) . ',' . T
      generarCodigo(n1)
      push(pilaTemp, T)
      print op . T . ',' . top(pilaReg)
    end
  end
```