Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI4722 - Lenguajes de Programación III
Trimestre Septiembre - Diciembre 2021
Prof. Ricardo Monascal
Amin Arriaga 16-10072

Examen I

(20 puntos)

El repositorio de Github usado para este examen se encuentrá en este link.

1. *(2 puntos)* Considere el siguiente fragmento de código, escrito en TAC:

```
a := b + c
b := a / 2
b := d - c
a := a + d
c := b * a
```

Al final de esta instrucción, las variables vivas son: `{b, d}`. Para cada instrucción, establezca el conjunto de variables vivas y la información de uso futuro para cada cada una de éstas.

Respuesta:

	Instrucción	**Vivas**	**Usos**
i		{b, c, d}	nextuse(a) = undef nextuse(b) = `i+1` nextuse(c) = `i+1` nextuse(d) = `i+3`
i+1	`a := b + c`	{a, c, d}	nextuse(a) = `i+2` nextuse(b) = undef nextuse(c) = `i+3` nextuse(d) = `i+3`
i+2	`b := a / 2`	{a, c, d}	nextuse(a) = `i+4` nextuse(b) = undef nextuse(c) = `i+3` nextuse(d) = `i+3`
i+3	`b := d - c`	{a, b, d}	nextuse(a) = `i+4` nextuse(b) = `i+5` nextuse(c) = undef nextuse(d) = `i+4`
i+4	`a := a + d`	{a, b, d}	nextuse(a) = `i+5` nextuse(b) = `i+5` nextuse(c) = undef nextuse(d) = undef
i+5	`c := b * a`	{b, d}	nextuse(a) = undef nextuse(b) = undef nextuse(c) = undef nextuse(d) = undef

2. *(5 puntos)* En clase, vimos como calcular el número de *Ershov* para expresiones de uno ($\otimes a$) o dos operandos ($a \otimes b$). Considere ahora expresiones de la siguiente forma: $\otimes(a, b, c)$ y código de cuatro direcciones FAC (*Four Address Code*), con instrucciones de la forma $\otimes\ Rr\ R0\ R1\ R2$ que corresponde a $Rr := \otimes(R0, R1, R2)$.

   (a) *(1 puntos)* ¿Cómo se calcula $label(n)$, cuando n tiene tres hijos: c_{left}, c_{middle} y c_{right}?

   Respuesta:
   Sean P, Q, R las etiquetas del nodo n tal que $P \geq Q \geq R$, entonces:

$$label(n) \longleftarrow \begin{cases} P & \text{si} \quad P > Q > R \\ P + 1 & \text{si} \quad P = Q > R \\ max(P, Q + 2) & \text{si} \quad P \geq Q = R \end{cases}$$

   (b) *(1 puntos)* Describa el proceso de generación de código usando este etiquetado extendido.

   Respuesta:
   Partimos desde la raíz del árbol etiquetado de *Ershov* hacia las hojas. Definiremos a $b \geq 1$ como el registro "base" para la generación por nodo. La raíz del árbol comienza con $b = 1$. Al procesar un nodo, la etiqueta k indica que se necesitan k registros para el código, los cuales serán los registros $R_b, ..., R_{b+k-1}$. El resultado debe quedar en R_{b+k-1}.

   - **Caso base:** Una hoja.
     Representa el operando `x`. Siendo b la base del nodo, se genera el código:

     `LD` R_b `x`

   - **Caso recursivo 1:** Nodo con dos hijos.
     El proceso de generación de código es igual al visto en clase para este caso.

   - **Caso recursivo 2:** Nodo con hijos c_p, c_q y c_r de claves diferentes.
     Sean $P > Q > R$ las etiquetas de los nodos c_p, c_q y c_r respectivamente.
     i. Generamos código para c_p usando base b. Se almacena el resultado en el registro R_{b+P-1}.
     ii. Generamos código para c_q usando base b. Se almacena el resultado en el registro R_{b+Q-1}.
     iii. Generamos código para c_r usando base b. Se almacena el resultado en el registro R_{b+R-1}.
     iv. Sean R_l, R_m y R_r los registros donde se almacenó el resultado de los hijos izquierdo, medio y derecho respectivamente. Entonces, se genera el código:

     `OP` $R_{b+P-1}\ R_l\ R_m\ R_r$

   - **Caso recursivo 3:** Nodo con hijos c_p, c_q y c_r tal que 2 tienen claves iguales y otra menor.
     Sean $P = Q > R$ las etiquetas de los nodos c_p, c_q y c_r respectivamente. El nodo padre tienen etiqueta $P + 1$.
     i. Generamos código para c_p usando base $b + 1$. Se almacena el resultado en el registro R_{b+P}.
     ii. Generamos código para c_q usando base b. Se almacena el resultado en el registro R_{b+P-1}.
     iii. Generamos código para c_r usando base b. Se almacena el resultado en el registro R_{b+R-1}.
     iv. Sean R_l, R_m y R_r los registros donde se almacenó el resultado de los hijos izquierdo, medio y derecho respectivamente. Entonces, se genera el código:

     `OP` $R_{b+P}\ R_l\ R_m\ R_r$

- **Caso recursivo 4:** Nodo con hijos c_p, c_q y c_r tal que 2 tienen claves iguales y otra mayor o igual.
  Sean $P \geq Q = R$ las etiquetas de los nodos c_p, c_q y c_r respectivamente. El nodo padre tienen clave $max(P, Q+2)$.
  i. Generamos código para c_p usando base $b+t$, donde $t = max(Q+2-P, 0)$ (notemos que t es a lo sumo 2 cuando $P = Q$, y se vuelve 0 cuando $P > Q+1$). Se almacena el resultado en el registro $R_{b+P+t-1}$ (notemos que $b+P+t-1$ es al menos $b+Q+1$ cuando $Q \leq P \leq Q+2$).
  ii. Generamos código para c_q usando base $b+1$. Se almacena el resultado en el registro R_{b+Q}.
  iii. Generamos código para c_r usando base b. Se almacena el resultado en el registro R_{b+Q-1}.
  iv. Sean R_l, R_m y R_r los registros donde se almacenó el resultado de los hijos izquierdo, medio y derecho respectivamente. Entonces, se genera el código:

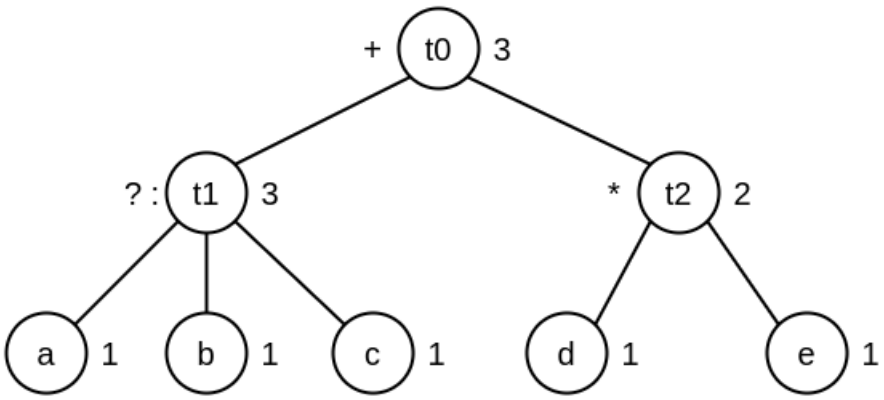
  `OP` $R_{b+P+t-1}$ R_l R_m R_r

(c) *(1 punto)* Use el proceso diseñado en la pregunta anterior para generar código TAC de la siguiente expresión de alto nivel:

```
(a ? b : c) + d * e
```

Puede suponer que tiene registros ilimitados.

Respuesta:
Creamos el arbol etiquetado de *Ershov*:



i. Raíz (t0) con $k = 3$ y $b = 1$. Usa los registros R_1, R_2 y R_3.
ii. Generar nodo izquierdo (t1) con $k = 3$ y $b = 1$. Usa los registros R_1, R_2 y R_3.
iii. Generar hoja derecha (c) con $k = 1$ y $b = 3$. Usa el registro R_3. Genera el código:
  `LD` R_3 `c`
iv. Generar hoja del medio (b) con $k = 1$ y $b = 2$. Usa el registro R_2. Genera el código:
  `LD` R_2 `b`
v. Generar hoja izquierda (a) con $k = 1$ y $b = 1$. Usa el registro R_1. Genera el código:
  `LD` R_1 `a`
vi. Volviendo al nodo (t1). Genera el código:
  `CONDVAL` R_3 R_1 R_2 R_3
vii. Volviendo al nodo (t0).
viii. Generar nodo derecho (t2) con $k = 2$ y $b = 1$. Usa los registros R_1 y R_2.
ix. Generar hoja derecha (e) con $k = 1$ y $b = 2$. Usa el registro R_2. Genera el código:
  `LD` R_2 `e`

x. Generar hoja izquierda (d) con $k = 1$ y $b = 1$. Usa el registro R_1. Genera el código:
`LD` R_1 `d`

xi. Volviendo al nodo (t2). Genera el código:
`MULT` R_2 R_1 R_2

xii. Volviendo al nodo (t0). Genera el código:
`ADD` R_3 R_3 R_2

Resultado:

`LD` R_3 `c`
`LD` R_2 `b`
`LD` R_1 `a`
`CONDVAL` R_3 R_1 R_2 R_3
`LD` R_2 `e`
`LD` R_1 `d`
`MULT` R_2 R_1 R_2
`ADD` R_3 R_3 R_2

(d) *(2 puntos)* Para la misma expresión de la pregunta anterior, calcule los costos de evaluación contigua usando el algoritmo de programación dinámica. Para esto, suponga que tiene a los sumo tres (3) registros disponibles.

Respuesta:
El costo de las hojas es siempre $(0, 1, 1, 1)$.

El costo del nodo $(t2)$ es básicamente igual al visto en clase para los nodos cuyos 2 hijos son hojas, pues con 3 registros el costo sigue siendo 2. Es decir, el costo del nodo $(t2)$ es $(3, 2, 2, 2)$.

Calculamos los costos del nodo $(t1)$:
Tenemos dos (2) plantillas disponibles:

- $OP\ R_i\ R_i\ R_j\ R_k$
- $OP\ R_i\ R_i\ R_j\ M_k$

Entonces:

- $C_{t1}[1] = \infty$
- $C_{t1}[2] = C_a[1] + C_b[1] + C_c[0] + OP_{R,M} = 1 + 1 + 0 + 1 = 3$
- $C_{t1}[3] = min(\{$
  $C_a[1] + C_b[1] + C_c[0] + OP_{R,M},$
  $C_a[1] + C_b[1] + C_c[1] + OP_{R,M},$
  $C_a[1] + C_b[1] + C_c[2] + OP_{R,M},$
  $C_a[1] + C_b[1] + C_c[3] + OP_{R,M},$
  $C_a[1] + C_b[2] + C_c[0] + OP_{R,M},$
  $C_a[1] + C_b[2] + C_c[1] + OP_{R,M},$
  $C_a[1] + C_b[2] + C_c[2] + OP_{R,M},$
  $C_a[1] + C_b[2] + C_c[3] + OP_{R,M},$
  $C_a[1] + C_b[3] + C_c[0] + OP_{R,M},$
  $C_a[1] + C_b[3] + C_c[2] + OP_{R,M},$
  $C_a[2] + C_b[1] + C_c[0] + OP_{R,M},$
  $C_a[2] + C_b[1] + C_c[1] + OP_{R,M},$
  $C_a[2] + C_b[1] + C_c[2] + OP_{R,M},$
  $C_a[2] + C_b[1] + C_c[3] + OP_{R,M},$
  $C_a[2] + C_b[2] + C_c[0] + OP_{R,M},$
  $C_a[2] + C_b[2] + C_c[1] + OP_{R,M},$
  $C_a[2] + C_b[3] + C_c[0] + OP_{R,M},$
  $C_a[2] + C_b[3] + C_c[1] + OP_{R,M},$
  $C_a[3] + C_b[1] + C_c[0] + OP_{R,M},$

$C_a[3] + C_b[1] + C_c[1] + OP_{R,M}$,
$C_a[3] + C_b[1] + C_c[2] + OP_{R,M}$,
$C_a[3] + C_b[2] + C_c[0] + OP_{R,M}$,
$C_a[3] + C_b[2] + C_c[1] + OP_{R,M}$,
})
$= min(\{3, 4, 4, 4, 3, 4, 4, 4, 3, 4, 3, 4, 4, 4, 3, 4, 3, 4, 3, 4, 4, 3, 4\}) = 3$

- $C_{t1}[0] = C_{t1}[3] + ST_{M,R} = 3 + 1 = 4$

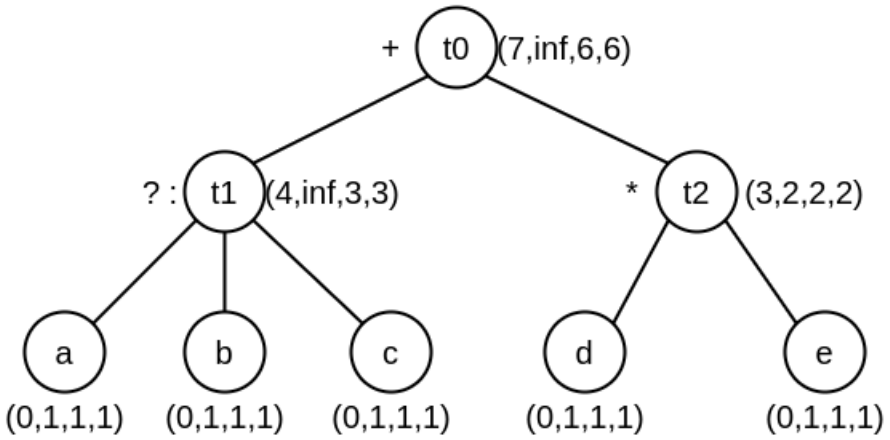
Calculamos los costos del nodo $(t0)$:
Tenemos dos (2) plantillas disponibles:

- $OP\ R_i\ R_i\ R_j$
- $OP\ R_i\ R_i\ M_j$

Entonces:

- $C_{t0}[1] = C_{t1}[1] + C_{t2}[0] + OP_{R,M} = \infty + 3 + 1 = \infty$
- $C_{t0}[2] = min(\{$
  $C_{t1}[2] + C_{t2}[0] + OP_{R,M}$,
  $C_{t1}[2] + C_{t2}[1] + OP_{R,M}$
  })
  $= min(\{7, 6\}) = 6$
- $C_{t0}[3] = min(\{$
  $C_{t1}[2] + C_{t2}[0] + OP_{R,M}$,
  $C_{t1}[2] + C_{t2}[1] + OP_{R,M}$,
  $C_{t1}[2] + C_{t2}[2] + OP_{R,M}$,
  $C_{t1}[3] + C_{t2}[0] + OP_{R,M}$,
  $C_{t1}[3] + C_{t2}[1] + OP_{R,M}$,
  $C_{t1}[3] + C_{t2}[2] + OP_{R,M}$,
  })
  $= min(\{7, 6, 6, 7, 6, 6\}) = 6$
- $C_{t0}[0] = C_{t0}[3] + ST_{M,R} = 6 + 1 = 7$

Resultado:



3. *(2 puntos)* Considere la siguiente instrucción, escrita en en el lenguaje de alto nivel a compilar:

```
we[have] = to[go[deep] + er]
```

(a) (0.5 puntos) Traduzca a TAC, suponiendo que los símbolos `we`, `to` y `go` corresponden a la dirección del primer elemento de arreglos de enteros.

Respuesta:
Supondremos que los enteros tienen tamaño de ocho (8) bytes.

```
t0 := have * 8
t1 := deep * 8
t2 := go [ t1 ]
t3 := t2 + er
t4 := t3 * 8
t5 := to [ t4 ]
we [ t0 ] := t5
```

(b) *(1.5 puntos)* Genere código de máquina para el fragmento empleando el algoritmo basado en Asignación de Registros según Descriptores de Uso y Asignación. Suponga que dispone de tres (3) registros para el cómputo de las expresiones. Muestre los descriptores en cada paso de generación.
Respuesta:

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
			we	have	to	go	deep	er						

`t0 := have * 8` Genera
```
LD R1 have
MULT R2 R1 8
```

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
have	t0		we	have, R1	to	go	deep	er	R2					

`t1 := deep * 8` Genera
```
LD R3 deep
MULT R1 R3 8
```

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
t1	t0	deep	we	have	to	go	deep, R3	er	R2	R1				

`t2 := go [ t1 ]` Genera
```
LD R3 go(R1)
```

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
t1	t0	t2	we	have	to	go	deep	er	R2	R1	R3			

`t3 := t2 + er` Genera
```
LD R1 er
ADD R3 R3 R1
```

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
er	t0	t3	we	have	to	go	deep	er, R1	R2			R3		

`t4 := t3 * 8` Genera
```
MULT R3 R3 8
```

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
er	t0	t4	we	have	to	go	deep	er, R1	R2				R3	

`t5 := to [ t4 ]` Genera `LD R3 to(R3)`

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
er	t0	t5	we	have	to	go	deep	er, R1	R2					R3

`we [ t0 ] := t5` Genera `ST we(R2) R3`

R1	R2	R3	we	have	to	go	deep	er	t0	t1	t2	t3	t4	t5
er	t0	t5	we	have	to	go	deep	er, R1	R2					R3

Resultado:

```
LD R1 have
MULT R2 R1 8
LD R3 deep
MULT R1 R3 8
LD R3 go(R1)
LD R1 er
ADD R3 R3 R1
MULT R3 R3 8
LD R3 to(R3)
ST we(R2) R3
```

4. *(3 puntos)* Considere el siguiente fragmento de código en pseudo-código que realiza la operación merge:

```
i := 0;
j := 0;
k := 0;
while (i < n && j < m) {
    if (a[i] < b[j]) {
        c[k] := a[i];
        i := i + 1;
    } else {
        c[k] := b[j];
        j := j + 1;
    }
    k := k + 1;
}
while (i < n) {
    c[k] := a[i];
    i := i + 1;
    k := k + 1;
}
while (j < m) {
    c[k] := b[j];
    j := j + 1;
    k := k + 1;
}
```

Puede suponer que `n` y `m` son enteros positivos. También puede suponer que `a` es un arreglo de enteros de tamaño `n`, `b` es un arreglo de enteros de tamaño `m` y `c` es un arreglo de enteros de tamaño `n + m`.

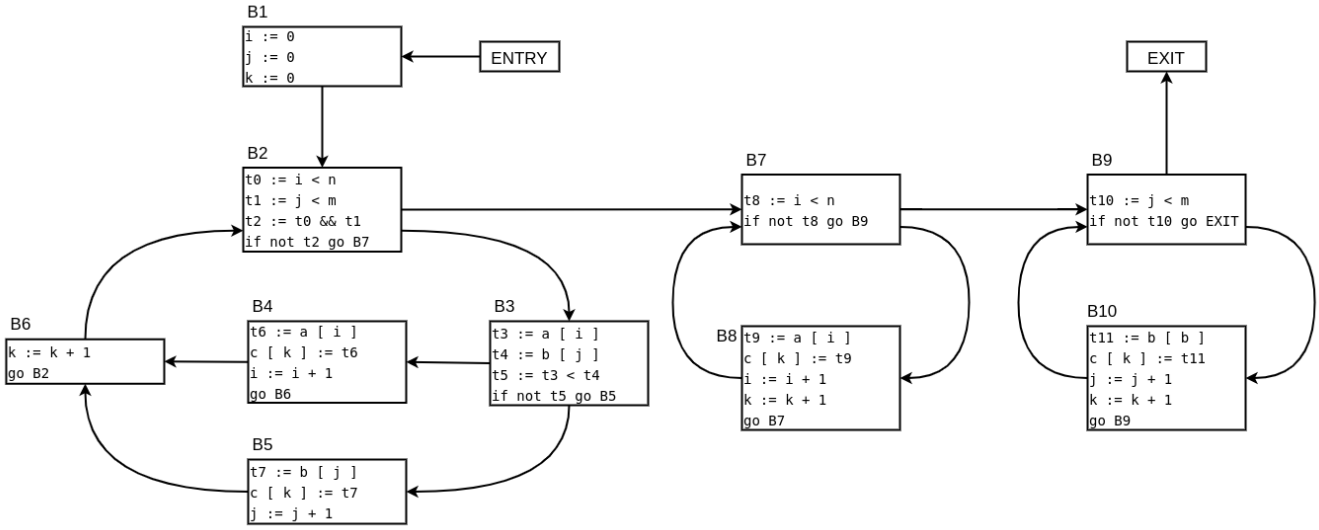
(a) *(1 punto)* Construya el TAC que se genera a partir del fragmento de código anterior.

Respuesta:

```
 0|  i := 0
 1|  j := 0
 2|  k := 0
 3|  t0 := i < n
 4|  t1 := j < m
 5|  t2 := t0 && t1
 6|  if not t2 go (20)
 7|  t3 := a [ i ]
 8|  t4 := b [ j ]
 9|  t5 := t3 < t4
10|  if not t5 go (15)
11|  t6 := a [ i ]
12|  c [ k ] := t6
13|  i := i + 1
14|  go (18)
15|  t7 := b [ j ]
16|  c [ k ] := t7
17|  j := j + 1
18|  k := k + 1
19|  go (3)
20|  t8 := i < n
21|  if not t8 go (27)
22|  t9 := a [ i ]
23|  c [ k ] := t9
24|  i := i + 1
25|  k := k + 1
26|  go (20)
27|  t10 := j < m
28|  if not t10 go (34)
29|  t11 := b [ b ]
30|  c [ k ] := t11
31|  j := j + 1
32|  k := k + 1
33|  go (27)
34|
```

(b) *(0.5 puntos)* Del TAC obtenido, construya el grafo de flujo asociado.

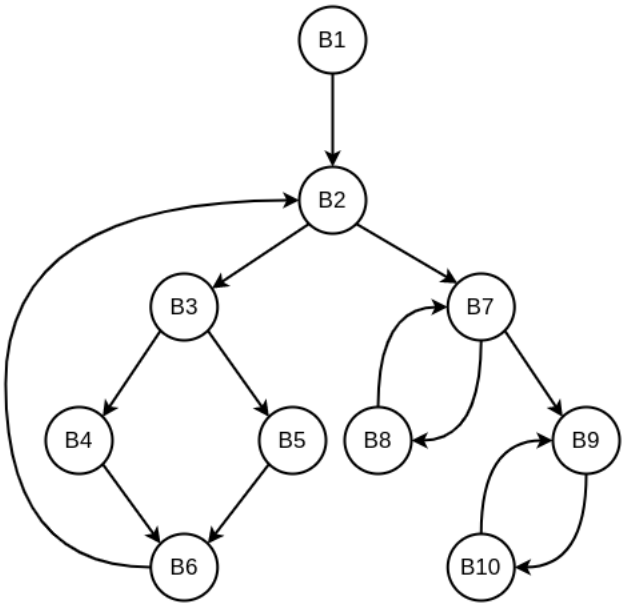
Respuesta:



(c) *(1.5 puntos*) Del TAC obtenido, construya el árbol de dominadores, mostrando los diferentes tipos de aristas resultantes (avance, retroceso, retorno y cruce). Con esta información identifique los ciclos naturales y diga si el grafo es reducible o no.

Respuesta:

Dado el grafo de flujo que representa el TAC:



Calculamos el *arbol de dominadores* usando el algoritmo de análisis de flujo asociado visto en clase:

0	E	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$	$B8$	$B9$	$B10$
IN	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
OUT	$\{E\}$	N	N	N	N	N	N	N	N	N	N

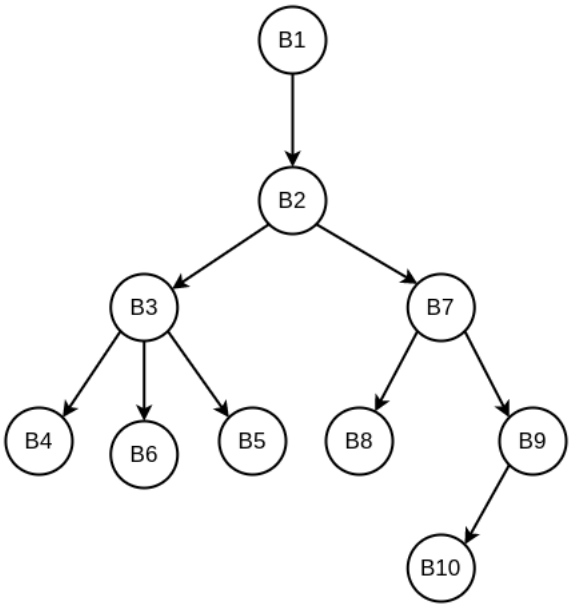
1	E	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$
IN	$\emptyset$	$\{E\}$	$\{E,1\}$	$\{E,1,2\}$	$\{E,1,2,3\}$	$\{E,1,2,3\}$	$\{E,1,2,3\}$	$\{E,1,2\}$
OUT	$\{E\}$	$\{E,1\}$	$\{E,1,2\}$	$\{E,1,2,3\}$	$\{E,1,2,3,4\}$	$\{E,1,2,3,5\}$	$\{E,1,2,3,6\}$	$\{E,1,2,7\}$

1	$B8$	$B9$	$B10$
IN	$\{E,1,2,7\}$	$\{E,1,2,7\}$	$\{E,1,2,7,9\}$
OUT	$\{E,1,2,7,8\}$	$\{E,1,2,7,9\}$	$\{E,1,2,7,9,10\}$

2	E	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$
IN	$\emptyset$	$\{E\}$	$\{E,1\}$	$\{E,1,2\}$	$\{E,1,2,3\}$	$\{E,1,2,3\}$	$\{E,1,2,3\}$	$\{E,1,2\}$
OUT	$\{E\}$	$\{E,1\}$	$\{E,1,2\}$	$\{E,1,2,3\}$	$\{E,1,2,3,4\}$	$\{E,1,2,3,5\}$	$\{E,1,2,3,6\}$	$\{E,1,2,7\}$

2	$B8$	$B9$	$B10$
IN	$\{E,1,2,7\}$	$\{E,1,2,7\}$	$\{E,1,2,7,9\}$
OUT	$\{E,1,2,7,8\}$	$\{E,1,2,7,9\}$	$\{E,1,2,7,9,10\}$

Por lo tanto, el *arbol de dominadores* queda como:

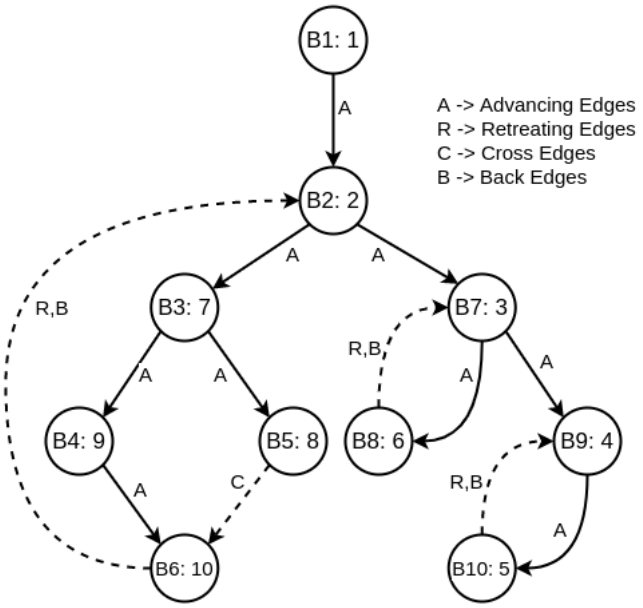


Ahora calculamos el *grafo de expansión* usando el algoritmo visto en clase:

i. search($B1$). Escoger $B2$. Agregar $B1 \to B2$.
ii. search($B2$). Escoger $B3$. Agregar $B2 \to B3$.
iii. search($B3$). Escoger $B4$. Agregar $B3 \to B4$.
iv. search($B4$). Escoger $B6$. Agregar $B4 \to B6$.
v. search($B6$). $B2$ ya fue visitado. No quedan sucesores. $dfn[B6] \leftarrow 10$.
vi. Regresar a search($B4$). No quedan sucesores. $dfn[B4] \leftarrow 9$.
vii. Regresar a search($B3$). Escoger $B5$. Agregar $B3 \to B5$.
viii. search($B5$). $B6$ ya fue visitado. No quedan sucesores. $dfn[B5] \leftarrow 8$.
ix. Regresar a search($B3$). No quedan sucesores. $dfn[B3] \leftarrow 7$.

x. Regresar a search($B2$). Escoger $B7$. Agregar $B2 \rightarrow B7$.
xi. search($B7$). Escoger $B8$. Agregar $B7 \rightarrow B8$.
xii. search($B8$). $B7$ ya fue visitado. No quedan sucesores. $dfn[B8] \leftarrow 6$.
xiii. Regresar a search($B7$). Escoger $B9$. Agregar $B9 \rightarrow B9$.
xiv. search($B9$). Escoger $B10$. Agregar $B9 \rightarrow B10$.
xv. search($B10$). $B9$ ya fue visitado. No quedan sucesores. $dfn[B10] \leftarrow 5$.
xvi. Regresar a search($B9$). No quedan sucesores. $dfn[B9] \leftarrow 4$.
xvii. Regresar a search($B7$). No quedan sucesores. $dfn[B7] \leftarrow 3$.
xviii. Regresar a search($B2$). No quedan sucesores. $dfn[B2] \leftarrow 2$.
xix. Regresar a search($B1$). No quedan sucesores. $dfn[B1] \leftarrow 1$.

Resultado (las aristas sólidas conforman el DFST), junto a la clasificación de cada arista:



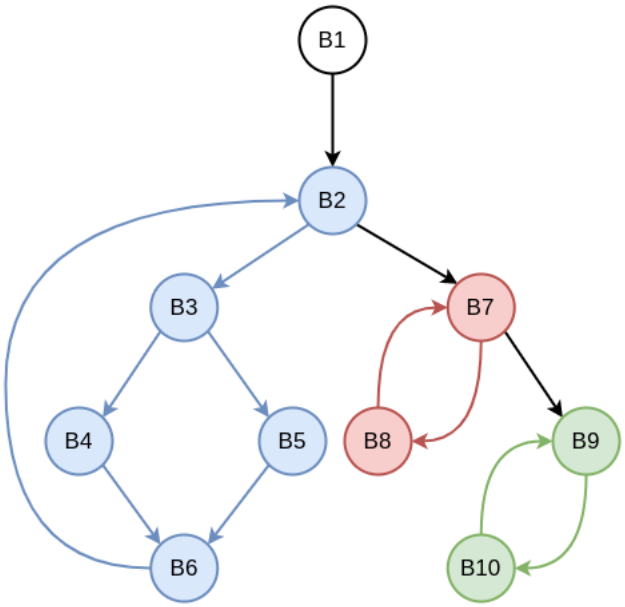
Como todas las aristas de retroceso también son de retorno, entonces el grafo es reducible. Para identificar los ciclos naturales ejecutamos DFS sobre el grafo de flujo inverso, partiendo desde el nodo origen de cada arista de retorno.

- Arista de retorno: $B6 \rightarrow B2$. Header: $B2$. Nodo inicial: $B6$.
  i. DFS($B6$). Agregamos $B5$.
  ii. DFS($B5$). Agregamos $B3$.
  iii. DFS($B3$). $B2$ ya fue visitado. No quedan sucesores.
  iv. Regresamos a DFS($B5$). No quedan sucesores.
  v. Regresamos a DFS($B6$). Agregamos $B4$.
  vi. DFS($B4$). $B3$ ya fue visitados. No quedan sucesores.
  vii. Regresamos a DFS($B6$). No quedan sucesores.
  Ciclo natural: $\{B2, B3, B4, B5, B6\}$

- Arista de retorno: $B8 \rightarrow B7$. Header: $B7$. Nodo inicial: $B8$
  i. DFS($B8$). $B7$ ya fue visitado. No quedan sucesores.
  Ciclo natural: $\{B7, B8\}$

- Arista de retorno: $B10 \rightarrow B9$. Header: $B9$. Nodo inicial: $B10$
  i. DFS($B10$). $B9$ ya fue visitado. No quedan sucesores.
  Ciclo natural: $\{B9, B10\}$

Así, los ciclos naturales (coloreado cada uno con un color distinto) son:



5. *(3 puntos)* Sea un número entero positivo n. Se construye un nuevo número n' de la siguiente forma:

   - Si n es par, entonces $n' = \frac{n}{2}$
   - Si n es impar, entonces $n' = 3 \times n + 1$

   La *conjetura de Collatz* establece que si se vuelve a aplicar este proceso de forma repetida, tomando n' como entrada al mismo proceso, eventualmente se alcanza el número 1.

   (a) *(0.5 puntos)* Escriba un programa en pseudo-código que, dado un n, devuelva la cantidad de pasos necesarios para alcanzar 1 siguiendo el procedimiento descrito anteriormente.

   Respuesta:

```
read(n)
steps := 0;
while (n != 1) {
    steps := steps + 1;
    if (n % 2 == 0) {
        n := n / 2;
    } else {
        n := 3 * n + 1;
    }
}
print(steps)
```

   (b) *(0.5 puntos)* Traduzca su programa a TAC.

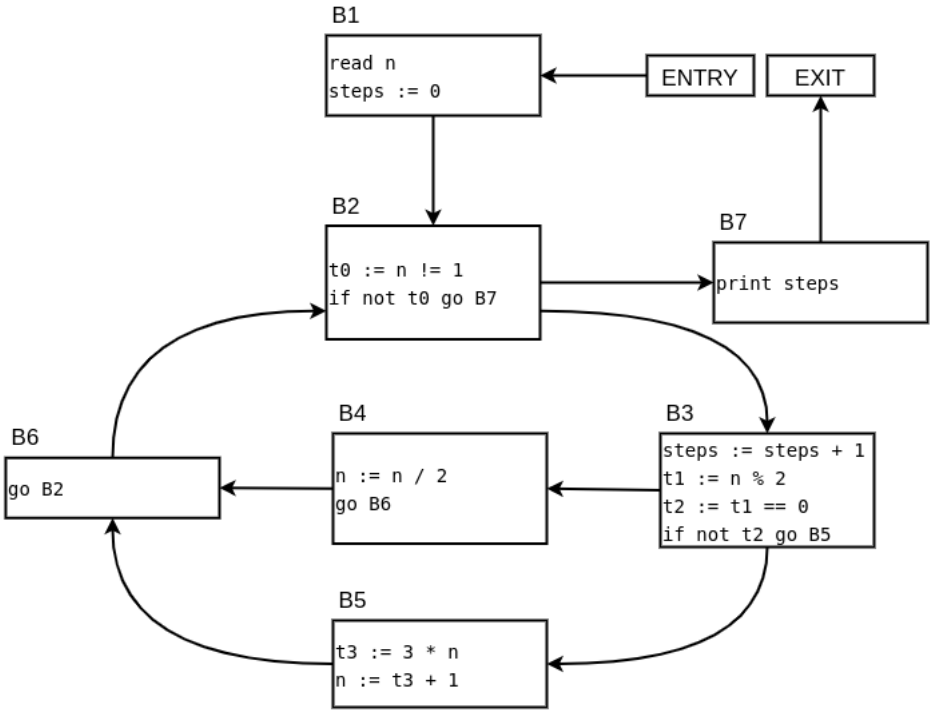
   Respuesta:

```
 0| read n
 1| steps := 0
 2| t0 := n != 1
 3| if not t0 go (13)
 4| steps := steps + 1
```

```
 5| t1 := n % 2
 6| t2 := t1 == 0
 7| if not t2 go (10)
 8| n := n / 2
 9| go (12)
10| t3 := 3 * n
11| n := t3 + 1
12| go (2)
13| print steps
```

(c) *(0.5 puntos)* Construya el grafo de flujo para el programa en TAC generado.

Respuesta:



(d) *(1.5 puntos)* Construya el grafo de interferencia y genere código con Asignación de Registros por Coloración de Grafos, suponiendo que se dispone de dos (2) registros.

Respuesta:

Usamos análisis de flujo para conocer el tiempo de vida de las variables:

	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$
def_B	$\{n, steps\}$	$\{t0\}$	$\{t1, t2\}$	$\emptyset$	$\{t3\}$	$\emptyset$	$\emptyset$
use_B	$\emptyset$	$\{n\}$	$\{steps, n\}$	$\{n\}$	$\{n\}$	$\emptyset$	$\{steps\}$

0	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$	EX
IN	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
OUT	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

1	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$	EX
IN	$\emptyset$	$\{n, steps\}$	$\{n, steps\}$	$\{n\}$	$\{n\}$	$\emptyset$	$\{steps\}$	$\emptyset$
OUT	$\{n, steps\}$	$\{n, steps\}$	$\{n\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

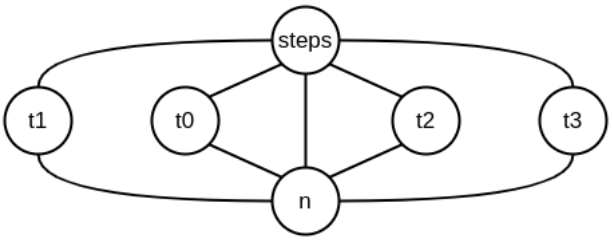
2	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$	EX
IN	$\emptyset$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{steps\}$	$\emptyset$
OUT	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\emptyset$	$\emptyset$

3	$B1$	$B2$	$B3$	$B4$	$B5$	$B6$	$B7$	EX
IN	$\emptyset$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{steps\}$	$\emptyset$
OUT	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\{n, steps\}$	$\emptyset$	$\emptyset$

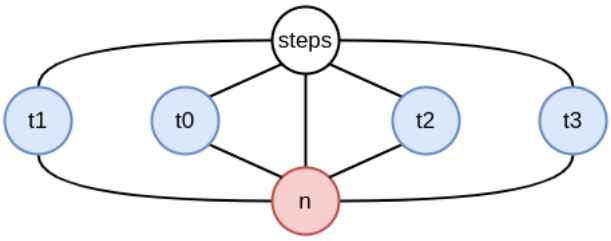
Entonces, los tiempos de vida de cada variable son:

- $n \Rightarrow [0..12]$
- $steps \Rightarrow [1..13]$
- $t0 \Rightarrow [2..3]$
- $t1 \Rightarrow [5..6]$
- $t2 \Rightarrow [6..7]$
- $t3 \Rightarrow [10..11]$

Así, el grafo de interferencia es:



No hay ningún nodo con menos de $k = 2$ vecinos. Se decide aplicar *spill* sobre la variable $steps$. Por lo tanto, sin colorear el nodo $steps$, la coloración del grafo queda como:



Finalmente, el código generado usando esta coloración es:

```
    READ  R1
    ST    steps 0
B2:
    NEQ   R2 R1 1
    BEZ   R2 B7
B3:
    LD    R2 steps
    ADD   R2 R2 1
    ST    steps R2
    MOD   R2 R1 2
    EQ    R2 R2 0
    BEZ   R2 B5
B4:
    DIV   R1 R1 2
```

```
        B     B6
    B5:
        MULT  R2 R1 3
        ADD   R1 R2 1
    B6:
        B     B2
    B7:
        LD    R2 steps
        PRINT R2
```

6. *(5 puntos)* Considere un TAC cuyas expresiones son todas números naturales (esto es, no permiten números negativos) y todas las variables son inicializadas en un número natural arbitrario. La sintaxis y operaciones disponibles para este TAC se entienden con la siguiente gramática:

```
TAC    ->  TAC \n LINE
        |  LINE
LINE   ->  id : INSTR
        |  INSTR
INSTR  ->  id := ARIT
        |  advance id
        |  goto id
        |  goif CMP id
ARIT   ->  VAL + VAL
        |  VAL - VAL
        |  VAL * VAL
        |  VAL / VAL
CMP    ->  VAL < VAL
        |  VAL > VAL
        |  VAL == VAL
        |  VAL != VAL
VAL    ->  id
        |  num
```

La mayoría de las operaciones aritméticas y relacionales tienen la semántica convencional. Además, existirá una instrucción especial advance, de forma tal que `advance x` es equivalente a `x := x + 1`. Las operaciones de resta y división, sin embargo, tienen semánticas especiales en este contexto:

- La resta es siempre mayor o igual a cero: $a -_{\mathbb{N}} b = max(a -_{\mathbb{Z}} b, 0)$.
- La división es truncada al natural inmediatamente inferior: $a /_{\mathbb{N}} b = \lfloor a /_{\mathbb{R}} b \rfloor$.

Considerando este TAC que trabaja sobre números naturales:

(a) *(1 punto)* Queremos saber si una variable en nuestro TAC puede llegar a ser igual a cero (por algún camino posible de ejecución). Plantee el problema de encontrar las variables que son potencialmente cero para cada instrucción como un problema de flujo de datos, proponiendo la construcción de IN y OUT (incluyendo el estado inicial) y la función de transferencia asociada. Diga también si dicho razonamiento será hacia adelante o hacia atrás.

Respuesta:
Sea $\mathcal{N}$ el conjunto de variables usadas en el TAC, entonces:

- Dominio: $\mathcal{N}$
- Análisis: Hacia adelante, es decir, $OUT[s] = f_s(IN[s])$ y $OUT[B] = F_B(IN[B])$
- Estado inicial:

$$OUT[ENTRY] = \mathcal{N}$$
$$OUT[B] = \mathcal{N} \qquad \forall B \neq ENTRY$$

- Función de transferencia:

$$IN[B] = \bigcup_{P \text{ es un predecesor de } B} OUT[P]$$

$$f_{advance\ u}(X) = X - \{u\}$$

$$f_{u:=v+w} = \begin{cases} X \cup \{u\} & \textbf{si} \quad (v \in X \vee v = 0) \wedge (w \in X \vee w = 0) \\ X - \{u\} & \text{en caso contrario} \end{cases}$$

$$f_{u:=v*w} = \begin{cases} X \cup \{u\} & \textbf{si} \quad v \in X \vee v = 0 \vee w \in X \vee w = 0 \\ X - \{u\} & \text{en caso contrario} \end{cases}$$

$$f_{u:=v-w} = \begin{cases} X - \{u\} & \textbf{si} \quad v \notin X \wedge v \neq 0 \wedge w = 0 \\ X \cup \{u\} & \text{en caso contrario} \end{cases}$$

$$f_{u:=v/w} = \begin{cases} X - \{u\} & \textbf{si} \quad w = 0 \\ X \cup \{u\} & \text{en caso contrario} \end{cases}$$

$$f_{id:\ s}(X) = f_s(X)$$
$$f_{goto\ id}(X) = X$$
$$f_{goif\ s\ id}(X) = X$$

(b) *(1 punto)* Usando la información de qué variables son potencialmente cero, establezca cuáles expresiones de división tienen riesgo de estar indefinidas (por tener denominador igual a cero). Considerando el mismo ejemplo de la parte anterior, la asignación `c := b / c` es de riesgo, pues es posible que `c` haya sido igual a cero antes de evaluar `b / c`.

Respuesta:

Sea s una instrucción del TAC de la forma $u := v/w$ ó $id : u := v/w$. Entonces, s tiene riesgo de estar indefinida si y solo si $w \in IN[s] \vee w = 0$.

(c) (3 puntos) Implemente un parser recursivo descendente para el TAC utilizado en esta pregunta. Luego, construya el grafo de flujo para el mismo, aplique el análisis de flujo propuesto por usted en las dos partes anteriores y reporte como warnings las expresiones de división que potencialmente están indefinidas. Deberá reportar también errores léxicos, sintácticos y de etiquetas indefinidas. Recuerde que todas las variables están definidas con algún número natural arbitrario al momento de iniciar la ejecución de un programa.

Respuesta:

Una vez dentro del repositorio, para correr el parser recursivo descendente ejecute:

```
python3 NaturalNumbersTAC.py [--verbose|-v] FILE
```

donde el flag `--verbose|-v` permite ver una descripción detallada del grafo de flujo generado y `FILE` es el archivo con el TAC a procesar.

¡Gracias por otro gran examen!