Generación de Código

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Mejorando lo presente

La Generación Perfecta de Ershov tiene propiedades interesantes:

- Código óptimo en tiempo $O(n)$ – n es el tamaño del árbol.
- Aplicable a *cualquier* arquitectura cuyos cálculos se hacen en registros.
- Aplicable a *cualquier* arquitectura con operadores binarios.
  - Sobre dos registros.
  - Sobre un registro y memoria.
  - Con resultado siempre en registro.

UNIVERSIDAD SIMÓN BOLÍVAR

Mejorando lo presente

La Generación Perfecta de Ershov tiene propiedades interesantes:

- Código óptimo en tiempo $O(n)$ – n es el tamaño del árbol.
- Aplicable a *cualquier* arquitectura cuyos cálculos se hacen en registros.
- Aplicable a *cualquier* arquitectura con operadores binarios.
  - Sobre dos registros.
  - Sobre un registro y memoria.
  - Con resultado siempre en registro.

¿Podemos hacerlo mejor?

UNIVERSIDAD SIMÓN BOLÍVAR

Cubrir más arquitecturas

- Queremos extender la clase de máquinas para las cuales se pueda generar código perfecto para las expresiones
  - Siempre máquinas de registros – no ayuda en máquinas de pila.
  - Los registros son intercambiables – si la máquina tiene registros "especiales", se excluyen.
- Queremos incorporar modelos de costo arbitrariamente complejos – incluso costo diferente para cada instrucción.
- Que siga siendo lineal en el tamaño del árbol.

UNIVERSIDAD SIMÓN BOLÍVAR

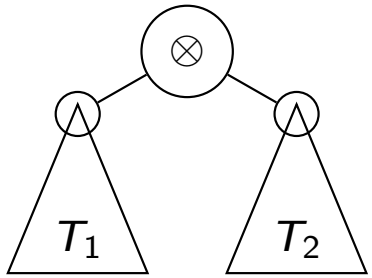
Particionar el problema

- Problema – generar código óptimo para $E_1 \otimes E_2$
- Partirlo en tres problemas.
  - Generar código óptimo para E_1
  - Generar código óptimo para E_2
  - Generar código para la operación $\otimes$
- Siempre es posible generar el código de E_2 antes que el de E_1 – ambos son óptimos, intercambiarlos no afecta el resultado.

Lo más importante – la generación es "contigua"

UNIVERSIDAD SIMÓN BOLÍVAR

Contigüidad



- Un programa P evalúa un árbol T de manera contigua si
  1. Evalúa en **memoria** los subárboles de T_1 necesarios.
  2. Evalúa en **memoria** los subárboles de T_2 necesarios.
  3. Evalúa el operador en la raíz.
- Es indiferente si el final se hace $T_1 \otimes T_2$ o $T_2 \otimes T_1$
- **No** es evaluación contigua
  - Evaluar parte de T_1 a un registro.
  - Evaluar T_2 a memoria.
  - Evaluar lo que resta de T_1.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cuál es el beneficio?

Optimalidad asegurada

- En máquinas de registros similares a la que usamos, para cualquier programa P *siempre* se puede hallar un P' que cumple
  - P' es igual o menos costoso que P
  - P' usa los mismos o menos registros que P
  - P' evalúa su árbol de manera contigua
- Arquitecturas con "pares de registros" no gozan de ese beneficio – hay operaciones "por partes" ¡hola Intel!

Evaluación contigua lleva a programa óptimo – particionar por subárbol lleva a Programación Dinámica

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo vía Programación Dinámica

Tres fases

1. Calcular un arreglo de costos C para cada nodo n.
   - De las hojas hasta la raíz.
   - $C[i]$ – mínimo costo posible para el cálculo del subárbol con raíz n usando i registros.
   - $1 \leq i \leq r$ – donde r es el número de registros.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo vía Programación Dinámica

Tres fases

1. Calcular un arreglo de costos C para cada nodo n.
   - De las hojas hasta la raíz.
   - $C[i]$ – mínimo costo posible para el cálculo del subárbol con raíz n usando i registros.
   - $1 \leq i \leq r$ – donde r es el número de registros.
2. Determinar los subárboles que se calculan a memoria.
   - Recorrer desde la raíz hasta las hojas.
   - Usar los arreglos de costos C.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo vía Programación Dinámica

Tres fases

1. Calcular un arreglo de costos C para cada nodo n.
   - De las hojas hasta la raíz.
   - $C[i]$ – mínimo costo posible para el cálculo del subárbol con raíz n usando i registros.
   - $1 \leq i \leq r$ – donde r es el número de registros.
2. Determinar los subárboles que se calculan a memoria.
   - Recorrer desde la raíz hasta las hojas.
   - Usar los arreglos de costos C.
3. Recorrer el árbol generando código.
   - Primero generar para los subárboles que deben calcularse en memoria.
   - Luego generar para el resto.

Las tres fases tienen complejidad $O(n)$ proporcional al tamaño del árbol.

Fase 1 – Calcular costos

¿Cuánto cuesta evaluar T ocupando i registros?

- El costo incluye la cantidad de cargas y almacenamientos necesarios para evaluar T con i registros.
- Incluye el costo de calcular $\otimes$ en la raíz.
- $C[0]$ – costo de calcular *todo* en memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

Fase 1 – Calcular costos

¿Cómo calcular C para cada n?

- Aplicamos la técnica de reescritura de árboles.
- Sea E una plantilla que coincide – usamos los costos de los hijos para determinar el costo del nodo actual.
  - Considerar todos los órdenes posibles para evaluar los operandos en las hojas de E – ¡contiguos!
  - Para cada combinación el primero se evalúa con i registros y el segundo con $i - 1$ registros.
  - Agregar el costo de la operación en E
- Repetir para todas las plantillas E que coincidan con el nodo – $C[i]$ es el mínimo entre todos los costos.

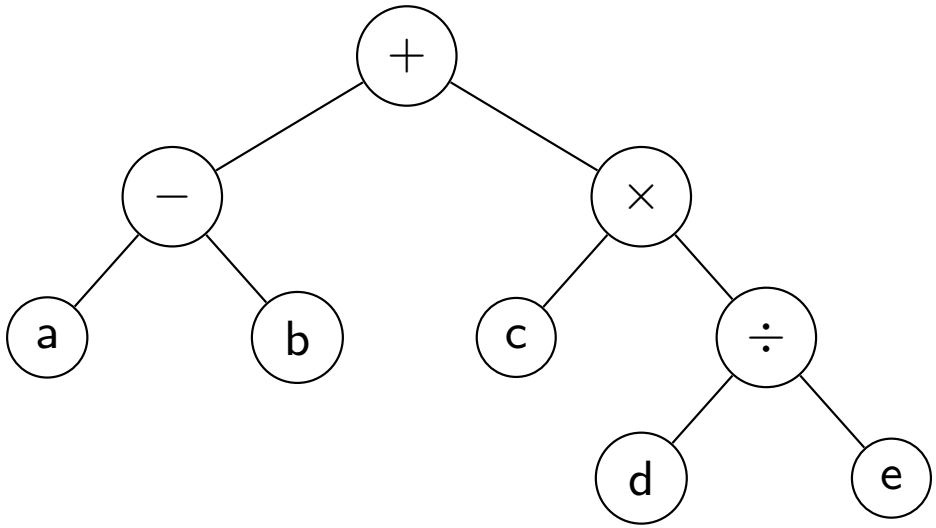
UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Suposiciones sobre la máquina

Instrucciones

```
LD Ri, Mj
OP Ri, Ri, Rj
OP Ri, Ri, Mj
LD Ri, Rj
ST Mi, Rj
```



- Todas costo unidad.
- $\mathtt{R}_i$ – sólo R1 o R2.
- $\mathtt{M}_j$ – memoria.

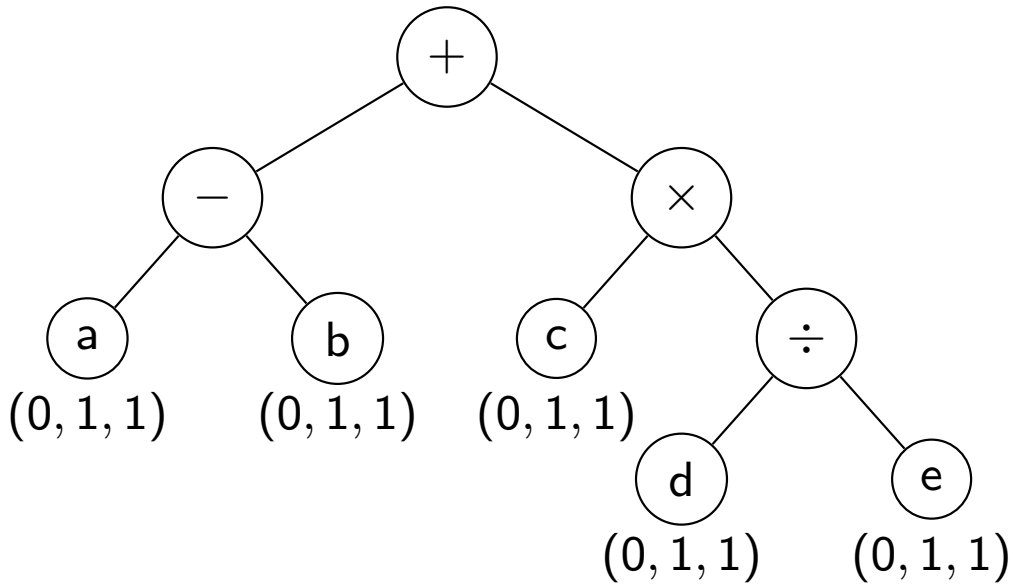
UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Costos en las hojas – igual para todas en este modelo

```
LD  Ri , Mj
OP  Ri , Ri , Rj
OP  Ri , Ri , Mj
LD  Ri , Rj
ST  Mi , Rj
```

- $C[0] = 0$ –
  ya están en memoria.
- $C[1] = 1$ –
  cargar con `LD`.
- $C[2] = 1$ –
  se carga *una* vez.

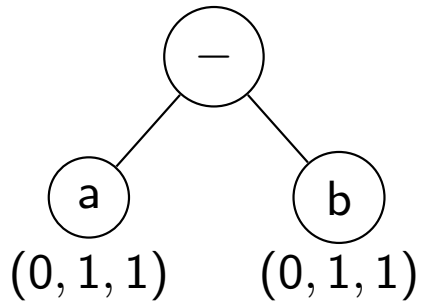


UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Costo de nodos intermedios – la resta y la división

- Dos plantillas aplicables:
  `OP` $\mathtt{R}_i$`,` $\mathtt{R}_i$`,` $\mathtt{M}_j$
  `OP` $\mathtt{R}_i$`,` $\mathtt{R}_i$`,` $\mathtt{R}_j$
- $C_{sub}[1] = 1 + 0 + 1 = 2 -$
  $C_a[1] + C_b[0] + \mathrm{OP}_{R,M}$
- $C_{sub}[2] = min\{$
  $C_a[2] + C_b[1] + \mathrm{OP}_{R,R}$,
  $C_a[2] + C_b[0] + \mathrm{OP}_{R,M}$,
  $C_a[1] + C_b[2] + \mathrm{OP}_{R,R}$,
  $C_a[1] + C_b[1] + \mathrm{OP}_{R,R}$,
  $C_a[1] + C_b[0] + \mathrm{OP}_{R,M}$
  $\} = 3, 2, 3, 3, 2 = 2$
- $C_{sub}[0] = 3$
  $C_{sub}[2] + \mathrm{ST}_{M,R}$



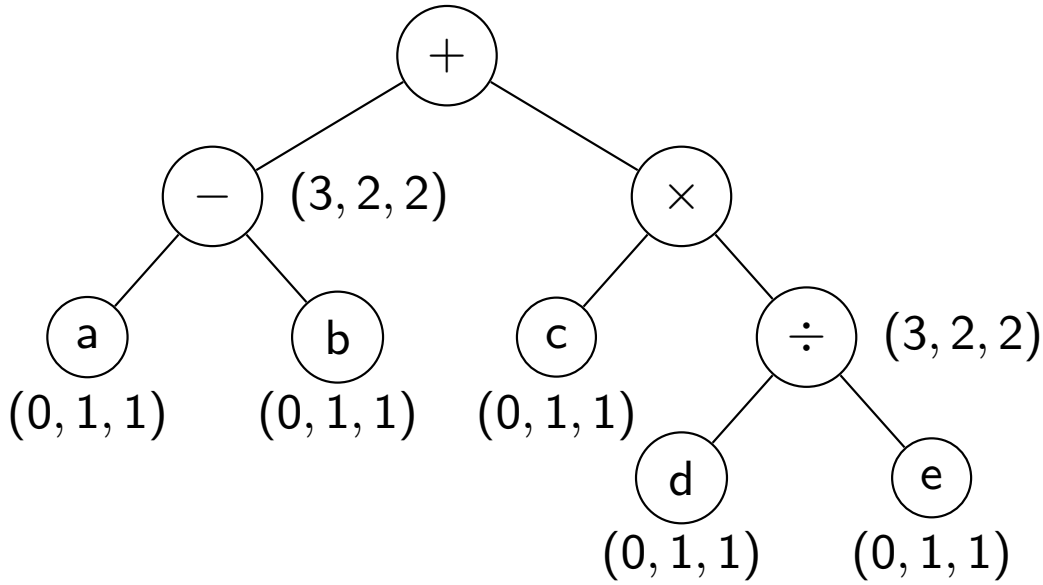
UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Costo de nodos intermedios – la resta y la división

```
LD Ri , Mj
OP Ri , Ri , Rj
OP Ri , Ri , Mj
LD Ri , Rj
ST Mi , Rj
```

- $C_{sub} = (3, 2, 2)$
- C_{div} idéntico.

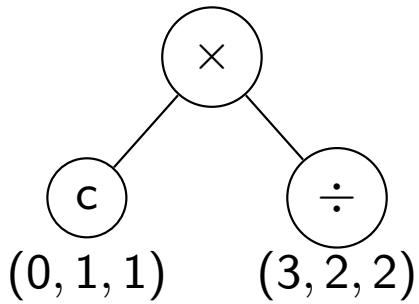


UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Costo de nodos intermedios – el producto

- $C_{mul}[2] = min\{$
  $C_c[2] + C_{div}[0] + \mathrm{OP}_{R,M}$,
  $C_c[2] + C_{div}[1] + \mathrm{OP}_{R,R}$,
  $C_c[1] + C_{div}[2] + \mathrm{OP}_{R,R}$,
  $C_c[1] + C_{div}[1] + \mathrm{OP}_{R,R}$,
  $C_c[1] + C_{div}[0] + \mathrm{OP}_{R,M}$
  $\} = 5, 4, 5, 4, 5 = 4$
- $C_{mul}[1] = 5$
  $C_c[1] + C_{div}[0] + \mathrm{OP}_{R,M}$
- $C_{mul}[0] = 5$
  $C_{mul}[2] + \mathrm{ST}_{M,R}$



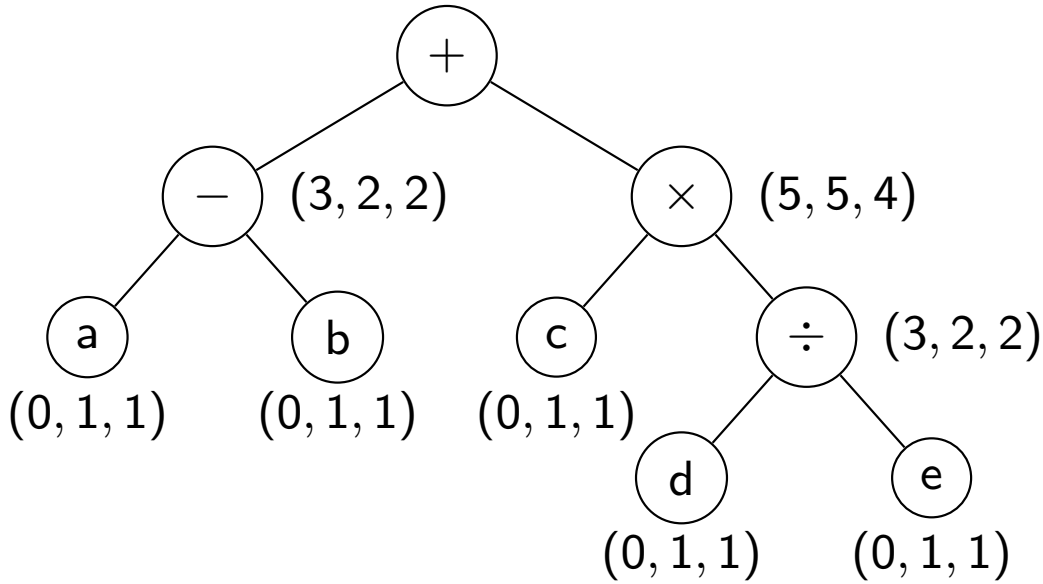
UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Costo de nodos intermedios – el producto

```
LD  Ri , Mj
OP  Ri , Ri , Rj
OP  Ri , Ri , Mj
LD  Ri , Rj
ST  Mi , Rj
```

- $C_sub = (5, 5, 4)$

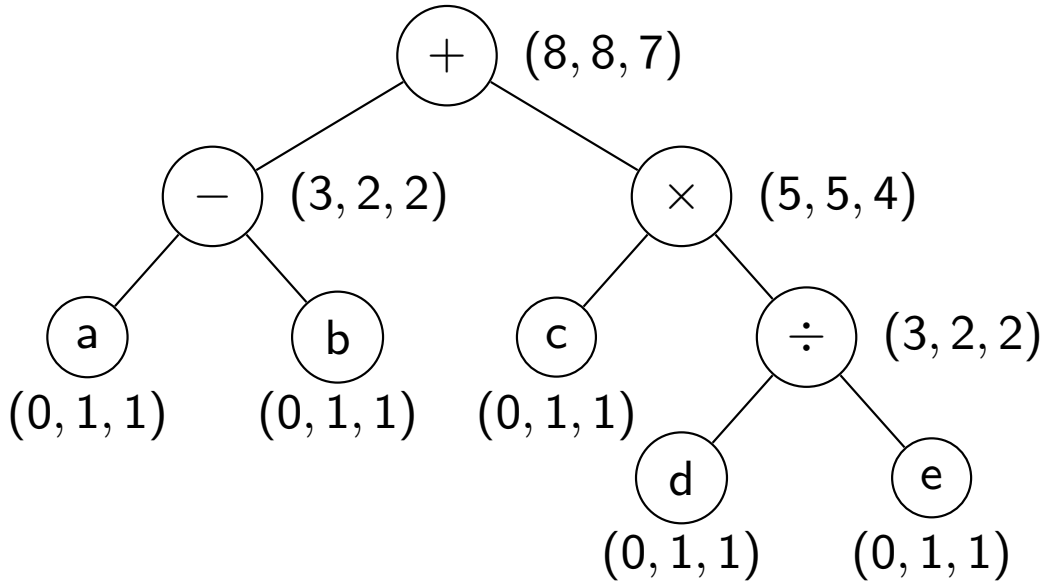


UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de cálculo de costos

Costo de nodos intermedios – la suma

```
LD Ri, Mj
OP Ri, Ri, Rj
OP Ri, Ri, Mj
LD Ri, Rj
ST Mi, Rj
```



UNIVERSIDAD SIMÓN BOLÍVAR

El código generado

- Durante el ascenso, cada vez que se escoge una plantilla como la de mínimo costo, se asocia al nodo.
- Una vez calculados los costos, se desciende por el árbol aplicando la plantilla seleccionada en cada caso.
- En nuestro caso

```
LD   R0, c
LD   R1, d
DIV  R1, R1, e
MUL  R0, R0, R1
LD   R1, a
SUB  R1, R1, b
ADD  R1, R1, R0
```

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Código para Saltos

- Los saltos siempre aparecen al final de los bloques básicos
  - El generador inocente ya generó los `LD` para almacenar todas las variables en memoria.
  - El generador con información de uso no necesariamente.
- Los saltos incondicionales se generan directamente – no afectan los contenidos de los registros de cálculo.
- Los saltos condicionales pueden afectar a los registros – invalidar la información de disponibilidad en registros.

UNIVERSIDAD SIMÓN BOLÍVAR

Saltos Condicionales

Arquitecturas que condicionan en registros individuales

¿Cómo traducir `if x == y goto L`?

- Disponen de saltos según un registro cumpla una condición.
- Típicamente ser cero, positivo, negativo o las negaciones.
- Basta generar la aritmética de comparación y verificar

```
LD   R0, x
SUB  R0, R0, y
JZ   R0, L
```

  donde `JZ` corresponde a *jump if zero*.
- Para un generador con descriptores
  - `x` e `y` mantienen su disponibilidad.
  - `R0` no estaría asignado al efectuar el salto.

UNIVERSIDAD SIMÓN BOLÍVAR

Saltos Condicionales

Arquitecturas que condicionan en resultados

¿Cómo traducir `if x == y goto L`?

- Saltos según el último resultado calculado o cargado en un registro – *bits de condición*.
- Típicamente cero, positivo, negativo o las negaciones.
- Se puede usar aritmética y verificar la condición.
- Algunos tienen una instrucción `CMP` que altera los bits de condición pero no los operandos – basta verificar la condición de interés.

```
CMP  x, y
JZ   L
```

donde `JZ` corresponde a *jump if condition zero*.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*]
  - Secciones 8.11
  - Ejercicios 8.11.1 y 8.11.2

UNIVERSIDAD SIMÓN BOLÍVAR