Generación de Código

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo trabajar con el contexto?

- La información relativa al contexto ayuda a generar buen código
  - Seleccionar la instrucción adecuada es más fácil si se conoce la secuencia de instrucciones en la cual participa.
  - Asignar registros es más fácil si se conoce cómo y cuándo se definen valores, además de sus usos.

¿Cómo trabajar con el contexto?

- La información relativa al contexto ayuda a generar buen código
  - Seleccionar la instrucción adecuada es más fácil si se conoce la secuencia de instrucciones en la cual participa.
  - Asignar registros es más fácil si se conoce cómo y cuándo se definen valores, además de sus usos.
- La solución es particionar el código
  - **Bloques Básicos** como unidad mínima.
  - Serán nodos de un **Grafo de Flujo**.
- Técnica aplicable al código intermedio y definitivo – generalizar análisis y mejoramiento en ambas etapas.

UNIVERSIDAD SIMÓN BOLÍVAR

Bloques Básicos

- Los **Bloques Básicos** *particionan* el código (intermedio o final) definiendo espacios de contexto.
- Cada bloque es el conjunto *más grande* de instrucciones *en secuencia* tal que el flujo de control:
  - Entra *exclusivamente* a través de la primera instrucción – no hay saltos hacia puntos intermedios del bloque.
  - Sale *exclusivamente* a través de la última instrucción – no hay saltos desde puntos intermedios del bloque.

UNIVERSIDAD SIMÓN BOLÍVAR

Bloques Básicos

- Los **Bloques Básicos** *particionan* el código (intermedio o final) definiendo espacios de contexto.
- Cada bloque es el conjunto *más grande* de instrucciones *en secuencia* tal que el flujo de control:
  - Entra *exclusivamente* a través de la primera instrucción – no hay saltos hacia puntos intermedios del bloque.
  - Sale *exclusivamente* a través de la última instrucción – no hay saltos desde puntos intermedios del bloque.
- Un Bloque Básico está en **Forma Normal** si cada instrucción que define una variable temporal siempre crea una variable nueva – no hay dos asignaciones a la misma temporal.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

Los saltos indican las fronteras

- Opera sobre la secuencia de instrucciones.
- Recorre la secuencia construyendo el conjunto de *líderes de bloque* – son líderes de bloque:
  - La primera instrucción de la secuencia.
  - Toda instrucción *destino* de un salto.
  - Toda instrucción *que sigue* a un salto.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

Los saltos indican las fronteras

- Opera sobre la secuencia de instrucciones.
- Recorre la secuencia construyendo el conjunto de *líderes de bloque* – son líderes de bloque:
  - La primera instrucción de la secuencia.
  - Toda instrucción *destino* de un salto.
  - Toda instrucción *que sigue* a un salto.
- Se construye un bloque básico:
  - Comenzando con cada instrucción líder.
  - Agregando las instrucciones hasta, pero sin incluir, al siguiente líder.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

```
 1: i      := 1
 2: j      := 1
 3: t1     := 10 * i
 4: t2     := t1 + j
 5: t3     := 8 * t2
 6: t4     := t3 - 88
 7: a[t4]  := 0.0
 8: j      := j + 1
 9: if j <= 10 goto (3)
10: i      := i + 1
11: if i <= 10 goto (2)
12: i      := 1
13: t5     := i - 1
14: t6     := 88 * t5
15: a[t6]  := 1.0
16: i      := i + 1
17: if i <= 10 goto (13)
```

Conjunto de Líderes

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

```
 1: i      := 1
 2: j      := 1
 3: t1     := 10 * i
 4: t2     := t1 + j
 5: t3     := 8 * t2
 6: t4     := t3 - 88
 7: a[t4]  := 0.0
 8: j      := j + 1
 9: if j <= 10 goto (3)
10: i      := i + 1
11: if i <= 10 goto (2)
12: i      := 1
13: t5     := i - 1
14: t6     := 88 * t5
15: a[t6]  := 1.0
16: i      := i + 1
17: if i <= 10 goto (13)
```

Conjunto de Líderes

- (1) – primera instrucción

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

```
 1:  i       := 1
 2:  j       := 1
 3:  t1      := 10 * i
 4:  t2      := t1 + j
 5:  t3      := 8 * t2
 6:  t4      := t3 - 88
 7:  a[t4]   := 0.0
 8:  j       := j + 1
 9:  if j <= 10 goto (3)
10:  i       := i + 1
11:  if i <= 10 goto (2)
12:  i       := 1
13:  t5      := i - 1
14:  t6      := 88 * t5
15:  a[t6]   := 1.0
16:  i       := i + 1
17:  if i <= 10 goto (13)
```

Conjunto de Líderes

- (1) – primera instrucción
- (2) – destino del goto en (11)
- (3) – destino del goto en (9)
- (13) – destino del goto en (17)

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

```
 1: i      := 1
 2: j      := 1
 3: t1     := 10 * i
 4: t2     := t1 + j
 5: t3     := 8 * t2
 6: t4     := t3 - 88
 7: a[t4]  := 0.0
 8: j      := j + 1
 9: if j <= 10 goto (3)
10: i      := i + 1
11: if i <= 10 goto (2)
12: i      := 1
13: t5     := i - 1
14: t6     := 88 * t5
15: a[t6]  := 1.0
16: i      := i + 1
17: if i <= 10 goto (13)
```

Conjunto de Líderes

- (1) – primera instrucción
- (2) – destino del goto en (11)
- (3) – destino del goto en (9)
- (13) – destino del goto en (17)
- (10) – sigue al goto en (10)
- (12) – sigue al goto en (11)
- Si hubiera una instrucción en (18), sería líder.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo de Particionamiento

Identificación de los bloques básicos

B_1
```
i := 1
```

B_2
```
j := 1
```

B_3
```
t1    := 10 * i
t2    := t1 + j
t3    := 8 * t2
t4    := t3 - 88
a[t4] := 0.0
j     := j + 1
if j <= 10 goto B3
```

B_4
```
i     := i + 1
if i <= 10 goto B2
```

B_5
```
i := 1
```

B_6
```
t5    := i - 1
t6    := 88 * t5
a[t6] := 1.0
i     := i + 1
if i <= 10 goto B6
```

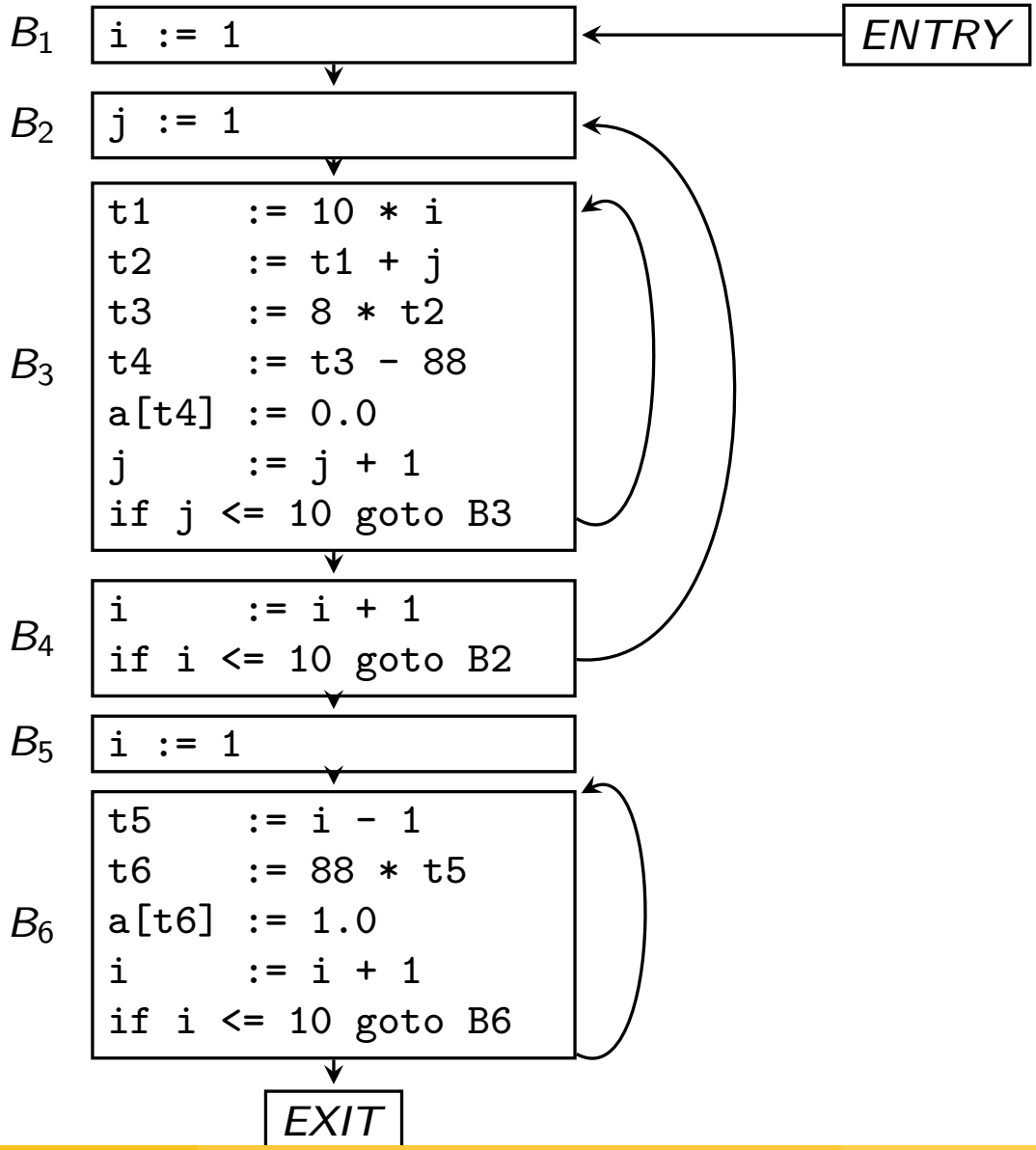
UNIVERSIDAD SIMÓN BOLÍVAR

Grafos de Flujo

- Representan el flujo de control entre los Bloques Básicos.
- Cada bloque básico es un nodo del Grafo de Flujo.
- Sean B_s y B_d Bloques Básicos para la secuencia – habrá una arista desde B_s hacia B_d siempre y cuando
  - Hay un salto desde el final de B_s hacia el comienzo de B_d.
  - B_s es seguido inmediatamente por B_d en el orden original del código y B_s no termina con un salto incondicional.
- B_s es **predecesor** de B_d – B_d es **sucesor** de B_s.
- Se agregan dos nodos *ENTRY* y *EXIT*
  - No corresponden a instrucciones de código.
  - *ENTRY* es el predecesor del primer bloque.
  - *EXIT* es sucesor de cualquier bloque que contenga una instrucción que pueda ser la última del programa.

UNIVERSIDAD SIMÓN BOLÍVAR

Grafo de Flujo



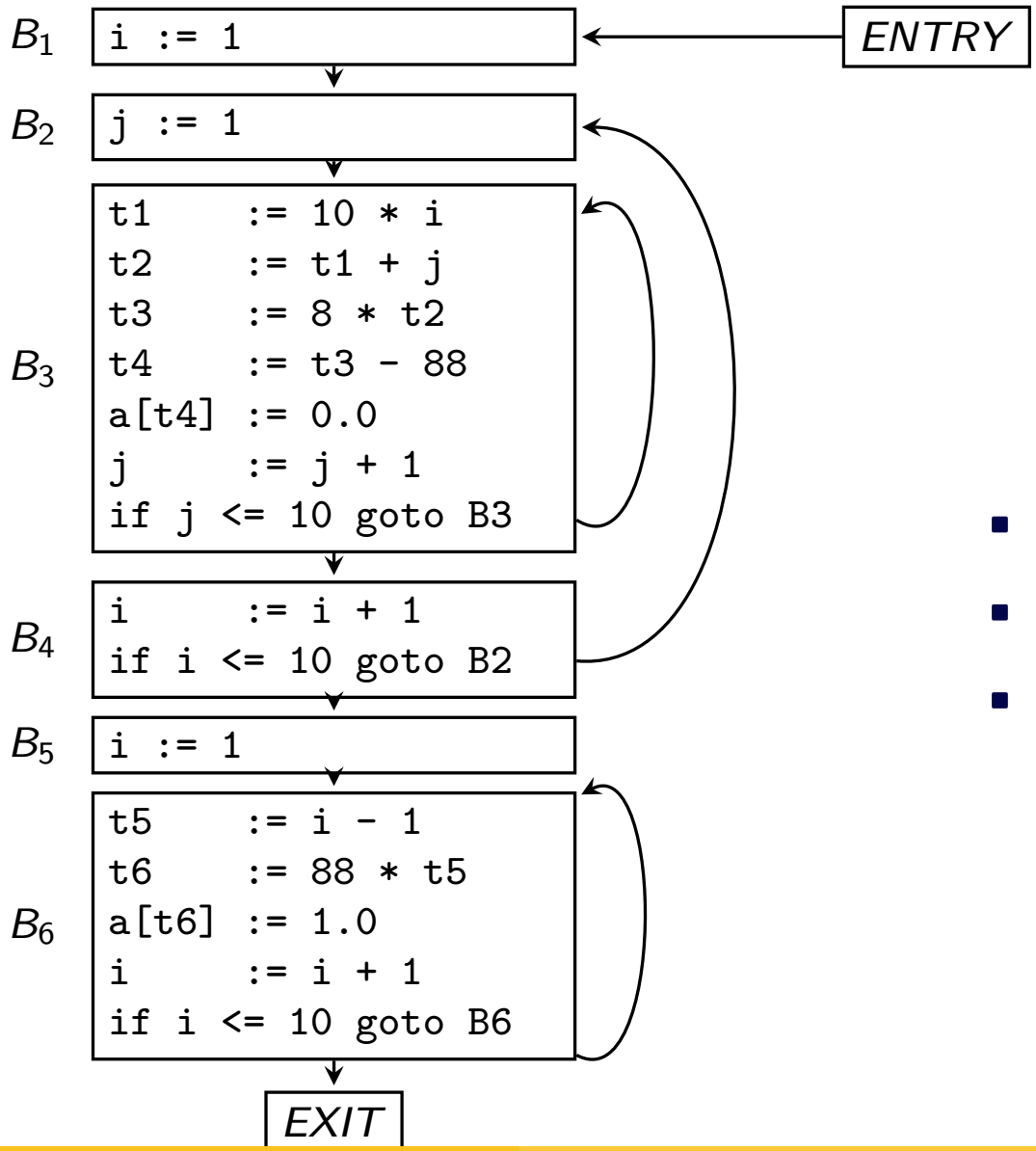
UNIVERSIDAD SIMÓN BOLÍVAR

Ciclos en el Grafo de Flujo

- Generar buen código para los ciclos requiere identificarlos.
- Un conjunto de Bloques Básicos L forma un ciclo si $\exists e \in L$ conocido como **nodo de entrada** (*loop entry node*) tal que:
  - $e \neq ENTRY$
  - $\forall n \in L : n \neq e \wedge precede(p, n) \Rightarrow p \in L$
    – todo camino desde $ENTRY$ a cualquier nodo en L pasa por e.
  - $\forall n \in L$ hay un camino no vacío $n, n_1, n_2, \ldots, e$ con $n_i \in L$
    – el conjunto es fuertemente conexo.

UNIVERSIDAD SIMÓN BOLÍVAR

Ciclos en el Grafo de Flujo



- $\{B_3\}$ es un ciclo.
- $\{B_6\}$ es un ciclo.
- $\{B_2, B_3, B_4\}$ es un ciclo.

UNIVERSIDAD SIMÓN BOLÍVAR

Información sobre Próximo Uso

Sólo en el Bloque Básico actual

- Determinar el próximo uso de un valor es útil para reasignar el registro en el cual se calculó.
- Podemos recorrer un bloque básico desde su *final* hasta su *comienzo*, y estudiar cada instrucción para refinar la información local:
  - Supongamos que la i-ésima instrucción del bloque asigna un valor para x – decimos que x esta **viva** (*live*) en i.
  - Si la j-ésima instrucción ($j > i$) tiene a x como operando y el control puede llegar desde i hasta j sin ninguna asignación para x, entonces decimos que j **usa** el valor de x calculado en i.
- Nos interesa calcular para *todas* las instrucciones de un bloque, cuál es el próximo uso de los nombres que participan en ella.

UNIVERSIDAD SIMÓN BOLÍVAR

Información sobre Próximo Uso

Algoritmo de cálculo sobre un bloque B

- Mantendremos la información de vida y uso en la Tabla de Símbolos – suponemos que todos los símbolos **no** temporales son marcados como vivos al salir de B .
- Recorrer el bloque B desde la última instrucción hasta la primera.
- Al procesar la $i-$ésima instrucción `x := y + z`:
  1. Los próximos usos para x, y y z según la Tabla de Símbolos se asocian a la instrucción i.
  2. Marcar x como no viva y sin próximo uso.
  3. Marcar y y z como vivas y con próximo uso en i.

UNIVERSIDAD SIMÓN BOLÍVAR

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`		
j	`a := b + c`		
k	`t := a + b`		

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`		
j	`a := b + c`		
k	`t := a + b`	$\{a, b, t\}$	

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`		
j	`a := b + c`		
k	`t := a + b`	$\{a, b, t\}$	nextuse(a) = undef nextuse(b) = undef nextuse(c) = undef

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`		
j	`a := b + c`	$\{a, b\}$	
k	`t := a + b`	$\{a, b, t\}$	nextuse(a) = undef nextuse(b) = undef nextuse(c) = undef

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`		
j	`a := b + c`	$\{a, b\}$	nextuse(a) = k nextuse(b) = k nextuse(c) = undef
k	`t := a + b`	$\{a, b, t\}$	nextuse(a) = undef nextuse(b) = undef nextuse(c) = undef

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`	$\{b, c\}$	
j	`a := b + c`	$\{a, b\}$	nextuse(a) = k nextuse(b) = k nextuse(c) = undef
k	`t := a + b`	$\{a, b, t\}$	nextuse(a) = undef nextuse(b) = undef nextuse(c) = undef

UNIVERSIDAD SIMÓN BOLÍVAR

Información sobre próximo uso

Una corrida simple

	Instrucción	Vivas	Usos
i	`b := b + 1`	$\{b, c\}$	nextuse(a) = undef nextuse(b) = j nextuse(c) = j
j	`a := b + c`	$\{a, b\}$	nextuse(a) = k nextuse(b) = k nextuse(c) = undef
k	`t := a + b`	$\{a, b, t\}$	nextuse(a) = undef nextuse(b) = undef nextuse(c) = undef

UNIVERSIDAD SIMÓN BOLÍVAR

Transformaciones en Bloques Básicos

- Una *transformación de mejora* sobre un fragmento de código:
  - Aumenta su velocidad.
  - Reduce su tamaño.

Transformaciones en Bloques Básicos

- Una *transformación de mejora* sobre un fragmento de código:
  - Aumenta su velocidad.
  - Reduce su tamaño.
- La transformación debe ser
  - *Segura* – el bloque resultante debe ser equivalente.
  - *Preservar la semántica* – no agrega ni elimina comportamientos.

UNIVERSIDAD SIMÓN BOLÍVAR

Transformaciones en Bloques Básicos

- Una *transformación de mejora* sobre un fragmento de código:
  - Aumenta su velocidad.
  - Reduce su tamaño.
- La transformación debe ser
  - *Segura* – el bloque resultante debe ser equivalente.
  - *Preservar la semántica* – no agrega ni elimina comportamientos.
- La transformación puede ser
  - *Local* – afecta un sólo Bloque Básico.
  - *Global* – afecta varios Bloques Básicos.

Las Transformaciones Locales son prácticas
de incorporar en la generación intermedia y final

UNIVERSIDAD SIMÓN BOLÍVAR

Generación local mejorada

Acción local, intención global

- La información de contexto accesible para el Bloque Básico local permite refinar la generación de código – **mejoras locales**.
- Construiremos un DAG para las instrucciones del Bloque Básico.
  - Un nodo por cada valor inicial para las variables.
  - Un nodo N por cada instrucción s en el bloque – serán hijos de N los nodos con instrucciones que definen por *última vez* los operandos de s.
  - Cada nodo N se etiqueta con el operador de s y la lista de variables para las cuales es última definición.
  - Si un nodo tiene variables vivas al salir del bloque, se designa como **nodo de salida**.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación local mejorada

Acción local, intención global

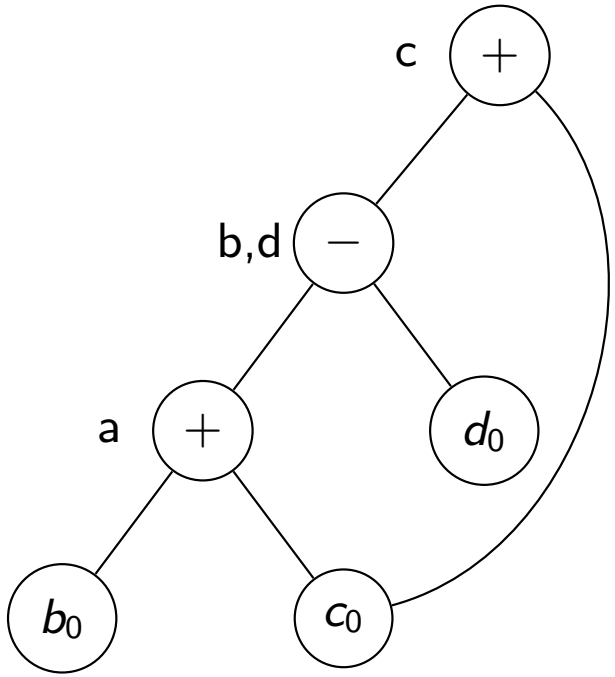
- La información de contexto accesible para el Bloque Básico local permite refinar la generación de código – **mejoras locales**.
- Construiremos un DAG para las instrucciones del Bloque Básico.
  - Un nodo por cada valor inicial para las variables.
  - Un nodo N por cada instrucción s en el bloque – serán hijos de N los nodos con instrucciones que definen por *última vez* los operandos de s.
  - Cada nodo N se etiqueta con el operador de s y la lista de variables para las cuales es última definición.
  - Si un nodo tiene variables vivas al salir del bloque, se designa como **nodo de salida**.

From code to DAG and back again...

UNIVERSIDAD SIMÓN BOLÍVAR

Paso del código a DAG

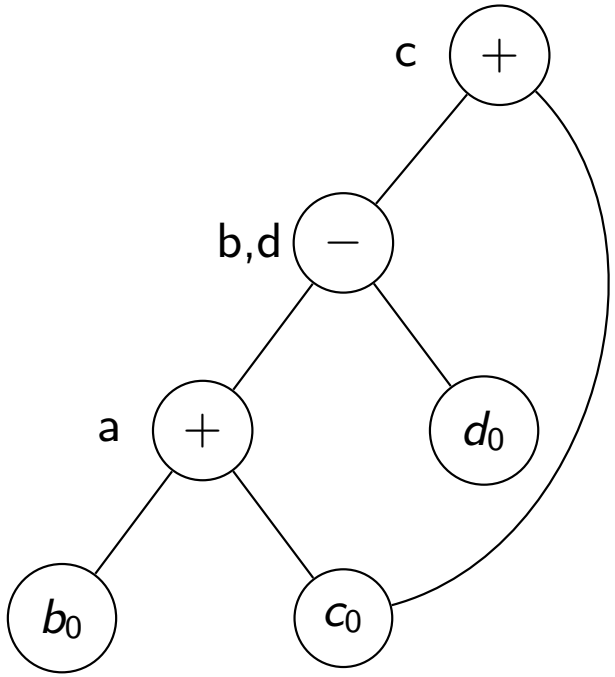
```
a := b + c
b := a - d
c := b + c
d := a - d
```



- Antes de agregar, buscar si existe uno con hijos iguales y en orden.
- Si existe, sólo agregar la variable definida a la lista existente.

UNIVERSIDAD SIMÓN BOLÍVAR

Regresando al código desde el DAG

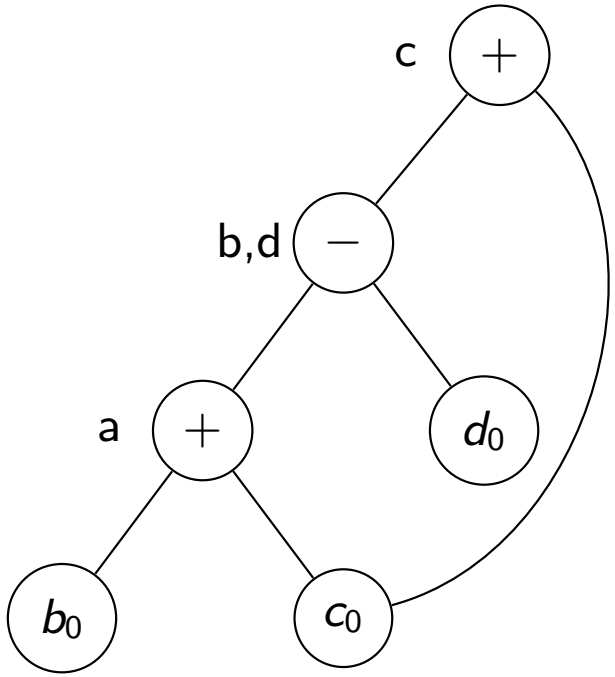



```
a := b + c
d := a - d
c := d + c
```

- Si b no está viva después del bloque se puede omitir su cálculo.
- Se reutiliza d para calcular b necesario para la última expresión.

UNIVERSIDAD SIMÓN BOLÍVAR

Regresando al código desde el DAG




```
a := b + c
d := a - d
c := d + c
```

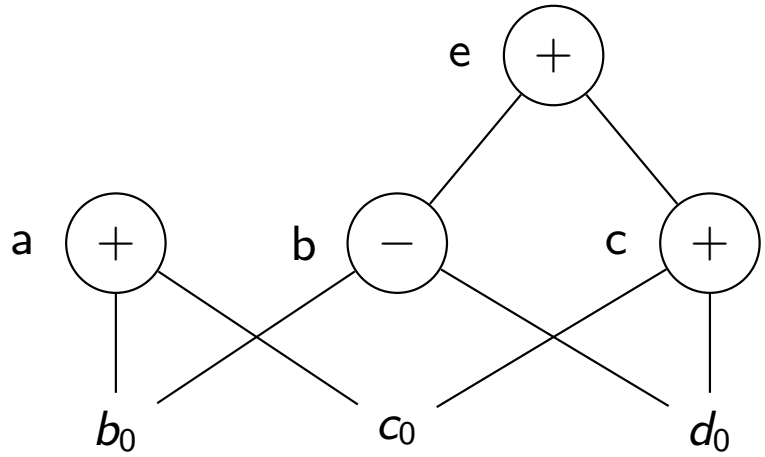
- Si `b` no está viva después del bloque se puede omitir su cálculo.
- Se reutiliza `d` para calcular `b` necesario para la última expresión.

Simplificamos la expresión común local.

UNIVERSIDAD SIMÓN BOLÍVAR

Explotando la información de vida

```
a := b + c
b := b - d
c := c + d
e := b + c
```

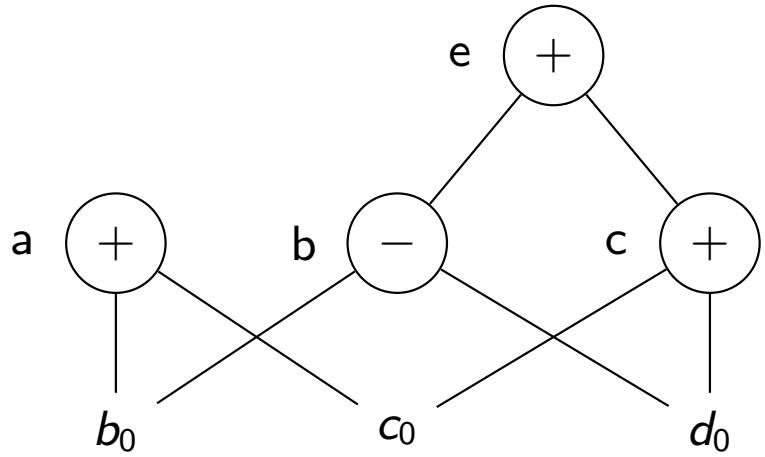


- Cualquier nodo raíz que no tiene variables vivas asociadas puede eliminarse – calcular su valor es inútil.

UNIVERSIDAD SIMÓN BOLÍVAR

Explotando la información de vida

```
a := b + c
b := b - d
c := c + d
e := b + c
```



- Cualquier nodo raíz que no tiene variables vivas asociadas puede eliminarse – calcular su valor es inútil.

Así eliminamos código inútil.

Reordenar instrucciones

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

$\Rightarrow$

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

- Instrucciones independientes adyacentes pueden intercambiarse.
- Si el bloque está en forma normal, se maximiza la posibilidad de intercambio de instrucciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Reducir la complejidad aritmética

Remove all neutrality!

Antes	Después
`x + 0`	`x`
`x - 0`	`x`
`x * 1`	`x`
`x / 1`	`x`
`x - x`	0

Puede conducir a eliminar instrucciones
o a *constant folding*.

UNIVERSIDAD SIMÓN BOLÍVAR

Reducir la complejidad aritmética

Evitar operaciones costosas

- Reemplazar operadores costosos por otros más económicos
  - Multiplicaciones repetidas en lugar de potencias – suponiendo que el TAC tiene el operador potencia arbitraria.
  - Sustituir `2 * x` con `x + x`.
  - Sustituir multiplicaciones y divisiones con potencias de dos por desplazamientos a izquierda y derecha, respectivamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Reducir la complejidad aritmética

Evitar operaciones costosas

- Reemplazar operadores costosos por otros más económicos
  - Multiplicaciones repetidas en lugar de potencias – suponiendo que el TAC tiene el operador potencia arbitraria.
  - Sustituir `2 * x` con `x + x`.
  - Sustituir multiplicaciones y divisiones con potencias de dos por desplazamientos a izquierda y derecha, respectivamente.
- Modelo de costo ajustado con precisión – ¿qué es más barato?

```
x := 22 * y
```

```
t1 := y << 2
t2 := t1 + y
t3 := t2 << 1
t4 := t3 + y
x  := t4 << 1
```

Se reduce la fuerza (*Strength Reduction*) del cómputo.

¡Cuidado con los arreglos!

No son operadores convencionales

```
x    := a[i]
a[j] := y
z    := a[i]
```

- Si suponemos que `[]` es un operador convencional . . .
  - . . . entonces `a[i]` es una expresión común.
  - . . . y podríamos usar `z := x` en la última instrucción

UNIVERSIDAD SIMÓN BOLÍVAR

¡Cuidado con los arreglos!

No son operadores convencionales

```
x    := a[i]
a[j] := y
z    := a[i]
```

- Si suponemos que `[]` es un operador convencional . . .
  - . . . entonces `a[i]` es una expresión común.
  - . . . y podríamos usar `z := x` en la última instrucción
- ¿Y si `j` es igual a `i`? – el cambio sería incorrecto.

Necesitamos una representación más detallada.

UNIVERSIDAD SIMÓN BOLÍVAR

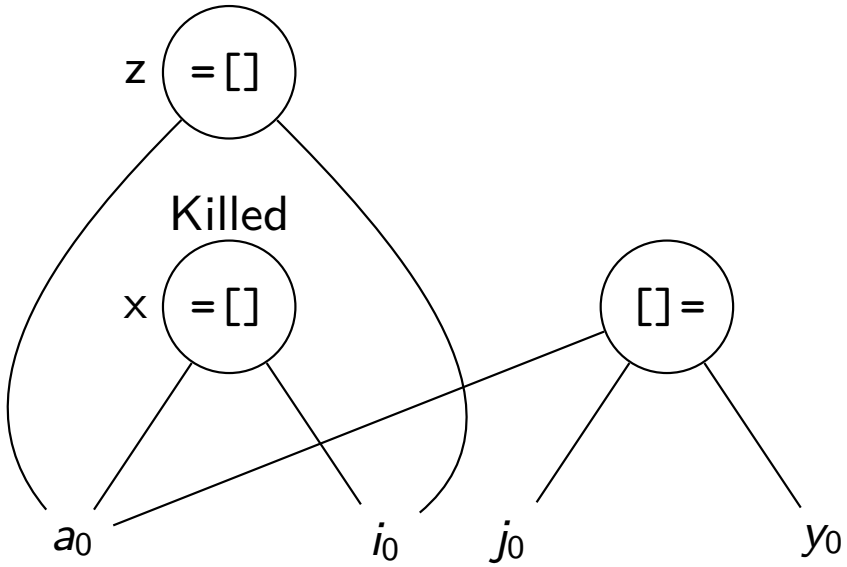
DAG para acceso a arreglos

- La asignación *desde* un arreglo – `x := a[i]`
  - Nodo `=[]` etiquetado con `x`.
  - Dos hijos `a` e `i` representando el arreglo y el índice respectivamente.
- La asignación *hacia* un arreglo – `a[j] := y`
  - Nodo `[]=` **sin** etiqueta.
  - Tres hijos `a`, `j` e `y`
- Cuando se construye un nodo `[]=` este **mata** a cualquier nodo existente que dependa del arreglo.
- Un nodo muerto no puede etiquetarse más – no puede convertirse en expresión común.

UNIVERSIDAD SIMÓN BOLÍVAR

DAG para acceso a arreglos

```
x      := a[i]
a[j]   := y
z      := a[i]
```



UNIVERSIDAD SIMÓN BOLÍVAR

El efecto del *aliasing*

```
x := *p
```

```
*q := y
```

- `p` y `q` apuntan a "cualquier cosa"
  - `x := *p` es un uso de *cualquier* variable.
  - `*q := y` es una asignación a *cualquier* variable.
- El DAG para =* tiene como hijos a **todos** los nodos que tengan identificadores asociados – porque no sabemos cuál en particular.
- El DAG para *= debe matar a **todos** los nodos que formen parte del DAG hasta ese punto – porque no sabemos cuál en particular.
- Llamadas a procedimientos tienen el mismo efecto – asumir que usan y cambian **todo** lo que alcanzan.

UNIVERSIDAD SIMÓN BOLÍVAR

El efecto del *aliasing*

```
x := *p
```

```
*q := y
```

- `p` y `q` apuntan a "cualquier cosa"
  - `x := *p` es un uso de *cualquier* variable.
  - `*q := y` es una asignación a *cualquier* variable.
- El DAG para =* tiene como hijos a **todos** los nodos que tengan identificadores asociados – porque no sabemos cuál en particular.
- El DAG para *= debe matar a **todos** los nodos que formen parte del DAG hasta ese punto – porque no sabemos cuál en particular.
- Llamadas a procedimientos tienen el mismo efecto – asumir que usan y cambian **todo** lo que alcanzan.

Problema No Decidible – Análisis de Flujo ayudará

UNIVERSIDAD SIMÓN BOLÍVAR

Regenerando código a partir del DAG

Considerar la información de vida y uso

- Una vez manipulado el DAG para aplicar mejoras, se genera nuevamente el código a partir de cada nodo y sus hijos.
- Si no se cuenta con información global, puede suponerse que todas las variables están vivas al salir del bloque – para las temporales se decide en base a la información de uso local.
- Si el nodo tiene más de una variable viva asociada, es necesario generar instrucciones de copia adicionales – sigue siendo mejor que tener expresiones comunes.

UNIVERSIDAD SIMÓN BOLÍVAR

Regenerando código a partir del DAG

Reglas para la reconstrucción

- Proceder desde las hojas hacia la raíz.
- Las asignaciones a un arreglo deben suceder a todas las asignaciones o evaluaciones al mismo arreglo, según el original.
- Las evaluaciones de un arreglo deben suceder a todas las asignaciones al mismo arreglo, según el orden original.
- El uso de una variable debe suceder a todas las llamadas a procedimientos o asignaciones indirectas, según el orden original.
- Toda llamada a procedimiento o asignación indirecta debe suceder a las evaluaciones de variables, según el orden original.

Eliminar, simplificar o intercambiar preservando el cómputo.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*]
  - Secciones 8.4 y 8.5.
  - Ejercicios 8.4.1 y 8.4.2, 8.5.1 a 8.5.8.

UNIVERSIDAD SIMÓN BOLÍVAR