Generación de Código

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Un Generador de Código

Un Bloque Básico a la vez

- Reduciremos el problema un Bloque Básico a la vez.
- Consideraremos cada instrucción en orden
  - Generaremos código para la instrucción usando el modelo de plantillas.
  - Mantendremos una relación de valores calculados y registros empleados – queremos minimizar las instrucciones de copia.
  - Aplicaremos mejoras locales simples.
- Estoy es *muy* inocente pero *muy* efectivo.

Luego refinaremos el modelo para incorporar métodos cada vez más complejos.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo usar los registros?

Con mucho cuidado...

- Casi cualquier arquitectura requiere que algunos o todos los operandos estén en registros para poder operar con ellos.
- Son convenientes para almacenar resultados intermedios o reutilizables en un mismo bloque.
- Son convenientes para almacenar resultados globales que afectan varios bloques – piensen "índice de iteración".
- Son necesarios para el manejo del ambiente de ejecución – tope de pila, registro de activación, base del *heap*.

Como hay pocos registros,
esos usos entran en conflicto.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo usar los registros?

Suposiciones iniciales

- Disponemos de un conjunto no vacío de registros para utilizar en el Bloque Básico para el cual generamos código.
- Es un subconjunto propio del total de registros y **no** incluye registros dedicados a información global o de ambiente de ejecución.
- El Bloque Básico ya ha sido mejorado y tiene la estructura definitiva.
- Por cada operador en el TAC
  - Hay *exactamente* un operador equivalente en la máquina real.
  - El operador recibe operandos y produce el resultado en *registros*.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo usar los registros?

Suposiciones iniciales

- Disponemos de un conjunto no vacío de registros para utilizar en el Bloque Básico para el cual generamos código.
- Es un subconjunto propio del total de registros y **no** incluye registros dedicados a información global o de ambiente de ejecución.
- El Bloque Básico ya ha sido mejorado y tiene la estructura definitiva.
- Por cada operador en el TAC
  - Hay *exactamente* un operador equivalente en la máquina real.
  - El operador recibe operandos y produce el resultado en *registros*.

Si les parece un modelo muy optimista,
están en lo correcto.

UNIVERSIDAD SIMÓN BOLÍVAR

Conjunto de Instrucciones

Reducción para generación de código inicial

Con nuestras suposiciones, podemos tener un conjunto reducido de instrucciones con las cuales trabajar

- `LD` *reg*, *mem* – cargar desde memoria
  reg $\leftarrow$ *contents*(*mem*)

UNIVERSIDAD SIMÓN BOLÍVAR

Conjunto de Instrucciones

Reducción para generación de código inicial

Con nuestras suposiciones, podemos tener un conjunto reducido de instrucciones con las cuales trabajar

- `LD` *reg*, *mem* – cargar desde memoria
  $reg \leftarrow contents(mem)$
- `ST` *mem*, *reg* – almacenar en memoria
  $mem \leftarrow contents(reg)$

UNIVERSIDAD SIMÓN BOLÍVAR

Conjunto de Instrucciones

Reducción para generación de código inicial

Con nuestras suposiciones, podemos tener un conjunto reducido de instrucciones con las cuales trabajar

- `LD` *reg*, *mem* – cargar desde memoria
  $reg \leftarrow contents(mem)$
- `ST` *mem*, *reg* – almacenar en memoria
  $mem \leftarrow contents(reg)$
- *OP* reg_d, reg_1, reg_2 – aplicar la operación binaria
  $reg_d \leftarrow reg_1$ op reg_2

UNIVERSIDAD SIMÓN BOLÍVAR

Conjunto de Instrucciones

Reducción para generación de código inicial

Con nuestras suposiciones, podemos tener un conjunto reducido de instrucciones con las cuales trabajar

- `LD` *reg*, *mem* – cargar desde memoria
  $reg \leftarrow contents(mem)$
- `ST` *mem*, *reg* – almacenar en memoria
  $mem \leftarrow contents(reg)$
- *OP* reg_d, reg_1, reg_2 – aplicar la operación binaria
  $reg_d \leftarrow reg_1$ op reg_2
- *OP* reg_d, reg_1 – aplicar la operación unaria
  $reg_d \leftarrow$ op reg_1

UNIVERSIDAD SIMÓN BOLÍVAR

Conjunto de Instrucciones

Reducción para generación de código inicial

Con nuestras suposiciones, podemos tener un conjunto reducido de instrucciones con las cuales trabajar

- `LD` *reg*, *mem* – cargar desde memoria
  $reg \leftarrow contents(mem)$
- `ST` *mem*, *reg* – almacenar en memoria
  $mem \leftarrow contents(reg)$
- *OP* reg_d, reg_1, reg_2 – aplicar la operación binaria
  $reg_d \leftarrow reg_1$ op reg_2
- *OP* reg_d, reg_1 – aplicar la operación unaria
  $reg_d \leftarrow$ op reg_1

Lidiaremos con los saltos después.

UNIVERSIDAD SIMÓN BOLÍVAR

Generando un camino de código

¿Qué hará el generador?

- Para cada instrucción TAC
  - Determina las cargas necesarias para obtener sus operandos.
  - Emite la instrucción de máquina para el operador.
  - De ser necesario, emite las instrucciones de almacenamiento.

UNIVERSIDAD SIMÓN BOLÍVAR

Generando un camino de código

¿Qué hará el generador?

- Para cada instrucción TAC
  - Determina las cargas necesarias para obtener sus operandos.
  - Emite la instrucción de máquina para el operador.
  - De ser necesario, emite las instrucciones de almacenamiento.
- ¿Cómo generar código razonablemente bueno a lo largo del camino?

UNIVERSIDAD SIMÓN BOLÍVAR

Generando un camino de código

¿Qué hará el generador?

- Para cada instrucción TAC
  - Determina las cargas necesarias para obtener sus operandos.
  - Emite la instrucción de máquina para el operador.
  - De ser necesario, emite las instrucciones de almacenamiento.
- ¿Cómo generar código razonablemente bueno a lo largo del camino?
  - Ahorrar cargas – ¿cuáles valores han sido cargados previamente? ¿cuáles valores han sido resultado de cálculos intermedios?

UNIVERSIDAD SIMÓN BOLÍVAR

Generando un camino de código

¿Qué hará el generador?

- Para cada instrucción TAC
  - Determina las cargas necesarias para obtener sus operandos.
  - Emite la instrucción de máquina para el operador.
  - De ser necesario, emite las instrucciones de almacenamiento.
- ¿Cómo generar código razonablemente bueno a lo largo del camino?
  - Ahorrar cargas – ¿cuáles valores han sido cargados previamente? ¿cuáles valores han sido resultado de cálculos intermedios?
  - Evitar cargas – un valor ha sido calculado pero todavía no ha sido almacenado en memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

Generando un camino de código

¿Qué hará el generador?

- Para cada instrucción TAC
  - Determina las cargas necesarias para obtener sus operandos.
  - Emite la instrucción de máquina para el operador.
  - De ser necesario, emite las instrucciones de almacenamiento.
- ¿Cómo generar código razonablemente bueno a lo largo del camino?
  - Ahorrar cargas – ¿cuáles valores han sido cargados previamente? ¿cuáles valores han sido resultado de cálculos intermedios?
  - Evitar cargas – un valor ha sido calculado pero todavía no ha sido almacenado en memoria.

¿Cómo describir el estado de ocupación de registros y ubicación de nombres?

UNIVERSIDAD SIMÓN BOLÍVAR

Descriptores de Asignación

¿Qué hay en el registro R_i?

Descriptor de Registros

- Uno por cada registro disponible para generar.
- Al iniciar la generación, se suponen todos vacíos.
- Cuando el valor de una variable sea calculado en un registro, se agrega la referencia al descriptor del registro.
- Cada registro tiene cero o más variables asociadas.

UNIVERSIDAD SIMÓN BOLÍVAR

Descriptores de Asignación

¿Qué hay en el registro R_i?

Descriptor de Registros

- Uno por cada registro disponible para generar.
- Al iniciar la generación, se suponen todos vacíos.
- Cuando el valor de una variable sea calculado en un registro, se agrega la referencia al descriptor del registro.
- Cada registro tiene cero o más variables asociadas.

```
LD R0, x
```

"R0 contiene x"

UNIVERSIDAD SIMÓN BOLÍVAR

Descriptores de Disponibilidad

¿Dónde puedo encontrar x?

Descriptor de Variables

- Uno por cada variable del programa.
- Cada vez que la variable es cargada, copiada o calculada, se agregan sus ubicaciones más recientes.
- Ubicaciones de memoria, de pila o registros.
- Cada nombre tiene una o más ubicaciones asociadas.

UNIVERSIDAD SIMÓN BOLÍVAR

Descriptores de Disponibilidad

¿Dónde puedo encontrar x?

Descriptor de Variables

- Uno por cada variable del programa.
- Cada vez que la variable es cargada, copiada o calculada, se agregan sus ubicaciones más recientes.
- Ubicaciones de memoria, de pila o registros.
- Cada nombre tiene una o más ubicaciones asociadas.

```
LD  R0, x
LD  R1, R0
```

"x está en R0 y R1"

UNIVERSIDAD SIMÓN BOLÍVAR

El Algoritmo de Generación

Los siete problemas

- Selección de Registros para cada instrucción.
- Generación de las instrucciones para operaciones.
- Generación de las instrucciones de copia.
- Cerrar la generación para el Bloque Básico.
- Mantener los descriptores consistentes.
- Generación de las instrucciones para arreglos.
- Generación de las instrucciones para apuntadores.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de Registros

Abstracción del problema

- `getReg(I)` – ¿cuáles registros usaremos en el código de `I`?

Selección de Registros

Abstracción del problema

- `getReg(I)` – ¿cuáles registros usaremos en el código de `I`?
- Aprovecha los descriptores para determinar cuál valor está en cuál registro y la disponibilidad de variables.

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de Registros

Abstracción del problema

- `getReg(I)` – ¿cuáles registros usaremos en el código de `I`?
- Aprovecha los descriptores para determinar cuál valor está en cuál registro y la disponibilidad de variables.
- Tiene acceso a toda la información adicional relevante para el bloque (típicamente uso y vida, posiblemente flujo de datos).

UNIVERSIDAD SIMÓN BOLÍVAR

Selección de Registros

Abstracción del problema

- `getReg(I)` – ¿cuáles registros usaremos en el código de `I`?
- Aprovecha los descriptores para determinar cuál valor está en cuál registro y la disponibilidad de variables.
- Tiene acceso a toda la información adicional relevante para el bloque (típicamente uso y vida, posiblemente flujo de datos).
- Vamos a suponer, sin pérdida de generalidad, que siempre hay suficientes registros disponibles para completar cualquier operación.
  - ¡Pero eso puede requerir almacenar cosas de nuevo en memoria! – ese es un problema diferente.
  - No importa *cuántos* registros disponibles hay en total, lo que suponemos es que siempre podemos *liberar* suficientes para cualquier operación que debamos generar.

Selección de Registros

Abstracción del problema

- `getReg(I)` – ¿cuáles registros usaremos en el código de `I`?
- Aprovecha los descriptores para determinar cuál valor está en cuál registro y la disponibilidad de variables.
- Tiene acceso a toda la información adicional relevante para el bloque (típicamente uso y vida, posiblemente flujo de datos).
- Vamos a suponer, sin pérdida de generalidad, que siempre hay suficientes registros disponibles para completar cualquier operación.
  - ¡Pero eso puede requerir almacenar cosas de nuevo en memoria! – ese es un problema diferente.
  - No importa *cuántos* registros disponibles hay en total, lo que suponemos es que siempre podemos *liberar* suficientes para cualquier operación que debamos generar.

Supongamos que `getReg(I)` está disponible
y concentrémonos en el algoritmo de generación.

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.
- ¿Cómo generar código?

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.
- ¿Cómo generar código?
  1. $(R_x, R_y, R_z) \leftarrow getReg($ `x := y + z` $)$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.
- ¿Cómo generar código?
  1. $(R_x, R_y, R_z) \leftarrow getReg($ `x := y + z` $)$
  2. Si y no está en R_y, entonces emitir `LD` R_y, y'
     - El descriptor de R_y asiste en la condición.
     - El descriptor de y asiste en determinar el y' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.
- ¿Cómo generar código?
  1. $(R_x, R_y, R_z) \leftarrow getReg($ `x := y + z` $)$
  2. Si y no está en R_y, entonces emitir `LD` R_y, y'
     - El descriptor de R_y asiste en la condición.
     - El descriptor de y asiste en determinar el y' más económico.
  3. Si z no está en R_z, entonces emitir `LD` R_z, z'

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.
- ¿Cómo generar código?
  1. $(R_x, R_y, R_z) \leftarrow getReg($ `x := y + z` $)$
  2. Si y no está en R_y, entonces emitir `LD` R_y, y'
     - El descriptor de R_y asiste en la condición.
     - El descriptor de y asiste en determinar el y' más económico.
  3. Si z no está en R_z, entonces emitir `LD` R_z, z'
  4. Emitir `ADD` R_x, R_y, R_z

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir una operación?

- Sea el TAC `x := y + z`
  - Establecimos que + se traducirá como `ADD`.
  - No vamos a suponer nada sobre la conmutatividad, i.e. `y` es el primer operando, `z` el segundo.
- ¿Cómo generar código?
  1. $(R_x, R_y, R_z) \leftarrow getReg($ `x := y + z` $)$
  2. Si y no está en R_y, entonces emitir `LD` R_y, y'
     - El descriptor de R_y asiste en la condición.
     - El descriptor de y asiste en determinar el y' más económico.
  3. Si z no está en R_z, entonces emitir `LD` R_z, z'
  4. Emitir `ADD` R_x, R_y, R_z

¿Quieres conmutatividad?
Genera ambos casos y escoje el más barato.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si la instrucción es una copia?

- Seguramente habrá TAC de la forma `x := y`
- ¿Cómo generar código?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si la instrucción es una copia?

- Seguramente habrá TAC de la forma `x := y`
- ¿Cómo generar código?
  1. $(R_x, R_y) \leftarrow getReg($ `x := y` $)$
     - Supondremos que `getReg` produce $R_x \equiv R_y$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Y si la instrucción es una copia?

- Seguramente habrá TAC de la forma `x := y`
- ¿Cómo generar código?
  1. $(R_x, R_y) \leftarrow getReg($ `x := y` $)$
     - Supondremos que `getReg` produce $R_x \equiv R_y$
  2. Si y no está en R_y, entonces emitir `LD` R_y, y'
     - El descriptor de R_y asiste en la condición.
     - El descriptor de y asiste en determinar el y' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

Terminando el Bloque Básico

- La generación dejará variables que terminaron en uno o más registros.

Terminando el Bloque Básico

- La generación dejará variables que terminaron en uno o más registros.
- Para las variables temporales
  - Si sólo existe en ese bloque, la información de asignación y disponibilidad simplemente se descarta.
  - Si sabemos (o debemos suponer) que la variable está viva al salir del bloque, mantenemos el descriptor.

Terminando el Bloque Básico

- La generación dejará variables que terminaron en uno o más registros.
- Para las variables temporales
  - Si sólo existe en ese bloque, la información de asignación y disponibilidad simplemente se descarta.
  - Si sabemos (o debemos suponer) que la variable está viva al salir del bloque, mantenemos el descriptor.
- Para toda variables de programa `x` se examina el descriptor de disponibilidad y si **no** incluye a su *propia* dirección de memoria, se emite `ST x,` R_x

UNIVERSIDAD SIMÓN BOLÍVAR

Manteniendo los Descriptores

Se ajustan según el tipo de instrucción emitida

Manteniendo los Descriptores

Se ajustan según el tipo de instrucción emitida

1. `LD R,x`
   - $Assignment(R) \leftarrow \{x\}$
   - $Availability(x) \leftarrow Availability(x) \cup \{R\}$

UNIVERSIDAD SIMÓN BOLÍVAR

Manteniendo los Descriptores

Se ajustan según el tipo de instrucción emitida

1. `LD R,x`
   - $Assignment(R) \leftarrow \{x\}$
   - $Availability(x) \leftarrow Availability(x) \cup \{R\}$
2. `ST x,R`
   - $Availability(x) \leftarrow Availability(x) \cup \{x\}$

UNIVERSIDAD SIMÓN BOLÍVAR

Manteniendo los Descriptores

Se ajustan según el tipo de instrucción emitida

1. `LD R,x`
   - $Assignment(R) \leftarrow \{x\}$
   - $Availability(x) \leftarrow Availability(x) \cup \{R\}$
2. `ST x,R`
   - $Availability(x) \leftarrow Availability(x) \cup \{x\}$
3. `OP` R_x, R_y, R_z para `x := y OP z`
   - $Assignment(R_x) \leftarrow \{x\}$
   - $Availability(x) \leftarrow \{R_x\}$
     – ¡valor más reciente no está en memoria!
   - $\forall v \neq x : Availability(v) \leftarrow Availability(v) - \{R_x\}$

UNIVERSIDAD SIMÓN BOLÍVAR

Manteniendo los Descriptores

Se ajustan según el tipo de instrucción emitida

1. `LD R,x`
   - $Assignment(R) \leftarrow \{x\}$
   - $Availability(x) \leftarrow Availability(x) \cup \{R\}$
2. `ST x,R`
   - $Availability(x) \leftarrow Availability(x) \cup \{x\}$
3. `OP` R_x, R_y, R_z para `x := y OP z`
   - $Assignment(R_x) \leftarrow \{x\}$
   - $Availability(x) \leftarrow \{R_x\}$
     - ¡valor más reciente no está en memoria!
   - $\forall v \neq x : Availability(v) \leftarrow Availability(v) - \{R_x\}$
4. `LD R,y` – para copias (`x := y`), después de cumplir la Regla 1
   - $Assignment(R) \leftarrow Assignment(R) \cup \{x\}$
   - $Availability(x) \leftarrow \{R\}$
     - ¡valor más reciente no está en memoria!

UNIVERSIDAD SIMÓN BOLÍVAR

Un ejemplo

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

- `t1`, `t2` y `t3` son temporales locales al bloque.
- El resto de las variables debe mantenerse viva al final.
- Supondremos tres registros disponibles.

UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de Generación

Descriptores Previos

R1	R2	R3	a	b	c	d	t1	t2	t3
			a	b	c	d			

```
t1 := a - b
```

generamos

```
LD R1, a
LD R2, b
SUB R2, R1, R2
```

Descriptores Ajustados

R1	R2	R3	a	b	c	d	t1	t2	t3
a	t1		a,R1	b	c	d	R2		

UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de Generación

Descriptores Previos

R1	R2	R3	a	b	c	d	t1	t2	t3
a	t1		a,R1	b	c	d	R2		

`t2 := a - c` generamos

```
LD R3, c
SUB R1, R1, R3
```

Descriptores Ajustados

R1	R2	R3	a	b	c	d	t1	t2	t3
t2	t1	c	a	b	c,R3	d	R2	R1	

UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de Generación

Descriptores Previos

R1	R2	R3	a	b	c	d	t1	t2	t3
t2	t1	c	a	b	c,R3	d	R2	R1	

`t3 := t1 + t2` generamos `ADD R3, R2, R1`

Descriptores Ajustados

R1	R2	R3	a	b	c	d	t1	t2	t3
t2	t1	t3	a	b	c	d	R2	R1	R3

UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de Generación

Descriptores Previos

R1	R2	R3	a	b	c	d	t1	t2	t3
t2	t1	t3	a	b	c	d	R2	R1	R3

`a := d` generamos `LD R2, d`

Descriptores Ajustados

R1	R2	R3	a	b	c	d	t1	t2	t3
t2	a,d	t3	R2	b	c	d,R2		R1	R3

UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de Generación

Descriptores Previos

R1	R2	R3	a	b	c	d	t1	t2	t3
t2	a,d	t3	R2	b	c	d,R2		R1	R3

`d := t3 + t2` generamos `ADD R1, R3, R1`

Descriptores Ajustados

R1	R2	R3	a	b	c	d	t1	t2	t3
d	a	t3	R2	b	c	R1			R3

UNIVERSIDAD SIMÓN BOLÍVAR

Ejemplo de Generación

Descriptores Previos

R1	R2	R3	a	b	c	d	t1	t2	t3
d	a	t3	R2	b	c	R1			R3

Final del Bloque generamos

```
ST a, R2
ST d, R1
```

Descriptores Finales

R1	R2	R3	a	b	c	d	t1	t2	t3
d	a		a,R2	b	c	d,R1			

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para cada operando – primero `y` y luego `z`, el método es el mismo.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para cada operando – primero `y` y luego `z`, el método es el mismo.
- Los casos simples ocurren cuando el operando. . .
  - Ya está en un registro – aprovecharlo.
  - No está en ningún registro, pero hay alguno disponible – usarlo.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para cada operando – primero `y` y luego `z`, el método es el mismo.
- Los casos simples ocurren cuando el operando. . .
  - Ya está en un registro – aprovecharlo.
  - No está en ningún registro, pero hay alguno disponible – usarlo.
- Para el caso complejo, el operando no está en ningún registro.
  - No hay registros disponibles – todos asignados con al menos un valor.
  - Es necesario seleccionar algún registro R para reciclarlo.
  - Ese reciclado debe *asegurar el valor* en R – *safe reuse*.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

¿Cómo reciclar de manera segura?

```
x := y + z
```

- Sea `R` un registro candidato para reciclaje.
- Su descriptor contiene uno o más valores `v` asignados – para cada uno de ellos, consideramos:

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

¿Cómo reciclar de manera segura?

```
x := y + z
```

- Sea `R` un registro candidato para reciclaje.
- Su descriptor contiene uno o más valores `v` asignados – para cada uno de ellos, consideramos:
  1. Si `v` está disponible en otra ubicación, `R` es seguro.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

¿Cómo reciclar de manera segura?

```
x := y + z
```

- Sea `R` un registro candidato para reciclaje.
- Su descriptor contiene uno o más valores `v` asignados – para cada uno de ellos, consideramos:
  1. Si `v` está disponible en otra ubicación, `R` es seguro.
  2. Si `v = x` y `x` **no** es uno de los operandos, `R` es seguro.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

¿Cómo reciclar de manera segura?

```
x := y + z
```

- Sea `R` un registro candidato para reciclaje.
- Su descriptor contiene uno o más valores `v` asignados – para cada uno de ellos, consideramos:
  1. Si `v` está disponible en otra ubicación, `R` es seguro.
  2. Si `v = x` y `x` **no** es uno de los operandos, `R` es seguro.
  3. Si `v` no tiene usos posteriores, `R` es seguro.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

¿Cómo reciclar de manera segura?

```
x := y + z
```

- Sea `R` un registro candidato para reciclaje.
- Su descriptor contiene uno o más valores `v` asignados – para cada uno de ellos, consideramos:
  1. Si `v` está disponible en otra ubicación, `R` es seguro.
  2. Si `v = x` y `x` **no** es uno de los operandos, `R` es seguro.
  3. Si `v` no tiene usos posteriores, `R` es seguro.
  4. Si después de los pasos 1 a 3 R aún no es seguro, se emite `ST v, R` – se cuenta como un *spill*.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

¿Cómo reciclar de manera segura?

```
x := y + z
```

- Sea `R` un registro candidato para reciclaje.
- Su descriptor contiene uno o más valores `v` asignados – para cada uno de ellos, consideramos:
  1. Si `v` está disponible en otra ubicación, `R` es seguro.
  2. Si `v = x` y `x` **no** es uno de los operandos, `R` es seguro.
  3. Si `v` no tiene usos posteriores, `R` es seguro.
  4. Si después de los pasos 1 a 3 R aún no es seguro, se emite `ST v, R` – se cuenta como un *spill*.
- Aplicar el criterio para *todos* los candidatos – seleccionar cualquiera con mínimo *spill*.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para el resultado.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para el resultado.
- Si algún registro *sólo* contiene a `x`, es la mejor opción – incluso si `x` es operando.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para el resultado.
- Si algún registro *sólo* contiene a `x`, es la mejor opción – incluso si `x` es operando.
- Si `y` no tiene usos posteriores, la mejor opción es algún registro que *sólo* contenga a `y` – el mismo argumento aplica para `z`

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones Generales

```
x := y + z
```

- Determinar el registro para el resultado.
- Si algún registro *sólo* contiene a `x`, es la mejor opción – incluso si `x` es operando.
- Si `y` no tiene usos posteriores, la mejor opción es algún registro que *sólo* contenga a `y` – el mismo argumento aplica para `z`
- Si no pueden aprovecharse esos casos, se usa el mismo método de selección aplicado para los operandos.

UNIVERSIDAD SIMÓN BOLÍVAR

Primera implantación de `getReg`

Caso: Operaciones de Copia

```
x := y
```

- Determinar el registro para el operando con el método descrito.
- El registro para el resultado será el mismo que para el operando.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir acceso a arreglos?

- Traducir TAC de la forma `x := a[i]` – operador binario `x := a =[] i` pero con `a` constante.
- ¿Cómo generar código?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir acceso a arreglos?

- Traducir TAC de la forma `x := a[i]` – operador binario `x := a =[] i` pero con `a` constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow$ *getReg*(`x := a =[] i`)

¿Cómo traducir acceso a arreglos?

- Traducir TAC de la forma `x := a[i]` – operador binario `x := a =[] i` pero con `a` constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow getReg($ `x := a =[] i` $)$
  2. Si i no está en R_i, entonces emitir `LD` R_i, i'
     - El descriptor de R_i asiste en la condición.
     - El descriptor de i asiste en determinar el i' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir acceso a arreglos?

- Traducir TAC de la forma `x := a[i]` – operador binario `x := a =[] i` pero con `a` constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow getReg($ `x := a =[] i` $)$
  2. Si i no está en R_i, entonces emitir `LD` R_i, i'
     - El descriptor de R_i asiste en la condición.
     - El descriptor de i asiste en determinar el i' más económico.
  3. Emitir `LD` $R_x, a(R_i)$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación a arreglos?

- Traducir TAC de la forma `a[i] := x` – operador binario `a := i []= x` pero con `a` constante.
- ¿Cómo generar código?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación a arreglos?

- Traducir TAC de la forma `a[i] := x` – operador binario `a := i []= x` pero con a constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow$ *getReg*(`a := i []= x`)

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación a arreglos?

- Traducir TAC de la forma `a[i] := x` – operador binario `a := i []= x` pero con a constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow$ *getReg*(`a := i []= x`)
  2. Si x no está en R_x, entonces emitir `LD` R_x, x'
     - El descriptor de R_x asiste en la condición.
     - El descriptor de x asiste en determinar el x' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación a arreglos?

- Traducir TAC de la forma `a[i] := x` – operador binario `a := i []= x` pero con a constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow getReg($ `a := i []= x` $)$
  2. Si x no está en R_x, entonces emitir `LD` R_x, x'
     - El descriptor de R_x asiste en la condición.
     - El descriptor de x asiste en determinar el x' más económico.
  3. Si i no está en R_i, entonces emitir `LD` R_i, i'
     - El descriptor de R_i asiste en la condición.
     - El descriptor de i asiste en determinar el i' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación a arreglos?

- Traducir TAC de la forma `a[i] := x` – operador binario `a := i []= x` pero con a constante.
- ¿Cómo generar código?
  1. $(R_x, R_i) \leftarrow getReg($ `a := i []= x` $)$
  2. Si x no está en R_x, entonces emitir `LD` R_x, x'
     - El descriptor de R_x asiste en la condición.
     - El descriptor de x asiste en determinar el x' más económico.
  3. Si i no está en R_i, entonces emitir `LD` R_i, i'
     - El descriptor de R_i asiste en la condición.
     - El descriptor de i asiste en determinar el i' más económico.
  4. Emitir `ST` $a(R_i), R_x$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir acceso con apuntadores?

- Traducir TAC de la forma `x := *p`
- ¿Cómo generar código?

¿Cómo traducir acceso con apuntadores?

- Traducir TAC de la forma `x := *p`
- ¿Cómo generar código?
  1. $(R_x, R_p) \leftarrow getReg($ `x := *p` $)$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir acceso con apuntadores?

- Traducir TAC de la forma `x := *p`
- ¿Cómo generar código?
  1. $(R_x, R_p) \leftarrow getReg($ `x := *p` $)$
  2. Si p no está en R_p emitir `LD` R_p, p'
     - El descriptor de R_p asiste en la condición.
     - El descriptor de p asiste en determinar el p' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir acceso con apuntadores?

- Traducir TAC de la forma `x := *p`
- ¿Cómo generar código?
  1. $(R_x, R_p) \leftarrow getReg($ `x := *p` $)$
  2. Si p no está en R_p emitir `LD` R_p, p'
     - El descriptor de R_p asiste en la condición.
     - El descriptor de p asiste en determinar el p' más económico.
  3. Emitir `LD` $R_x, *R_p$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación vía apuntadores?

- Traducir TAC de la forma `*p := x`
- ¿Cómo generar código?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación vía apuntadores?

- Traducir TAC de la forma `*p := x`
- ¿Cómo generar código?
  1. $(R_p, R_x) \leftarrow getReg($ `*p := x` $)$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación vía apuntadores?

- Traducir TAC de la forma `*p := x`
- ¿Cómo generar código?
  1. $(R_p, R_x) \leftarrow getReg($ `*p := x` $)$
  2. Si p no está en R_p emitir `LD` R_p, p'
     - El descriptor de R_p asiste en la condición.
     - El descriptor de p asiste en determinar el p' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación vía apuntadores?

- Traducir TAC de la forma `*p := x`
- ¿Cómo generar código?
  1. $(R_p, R_x) \leftarrow getReg($ `*p := x` $)$
  2. Si p no está en R_p emitir `LD` R_p, p'
     - El descriptor de R_p asiste en la condición.
     - El descriptor de p asiste en determinar el p' más económico.
  3. Si x no está en R_x emitir `LD` R_x, x'
     - El descriptor de R_x asiste en la condición.
     - El descriptor de x asiste en determinar el x' más económico.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo traducir asignación vía apuntadores?

- Traducir TAC de la forma `*p := x`
- ¿Cómo generar código?
  1. $(R_p, R_x) \leftarrow getReg($ `*p := x` $)$
  2. Si p no está en R_p emitir `LD` R_p, p'
     - El descriptor de R_p asiste en la condición.
     - El descriptor de p asiste en determinar el p' más económico.
  3. Si x no está en R_x emitir `LD` R_x, x'
     - El descriptor de R_x asiste en la condición.
     - El descriptor de x asiste en determinar el x' más económico.
  4. Emitir `ST` $*R_p, R_x$

UNIVERSIDAD SIMÓN BOLÍVAR

Peephole Optimization

Fisgonear el código

- Una vez emitido el código final se recorre desde el principio a través de una "ventana de observación" (*peephole*).

Peephole Optimization

Fisgonear el código

- Una vez emitido el código final se recorre desde el principio a través de una "ventana de observación" (*peephole*).
- Se identifican secuencias de instrucciones que puedan ser reemplazadas por otras más cortas o más rápidas.
  - La cantidad de instrucciones examinadas en la ventana es decisión del implantador.
  - Las secuencias a identificar dependen del lenguaje destino.

UNIVERSIDAD SIMÓN BOLÍVAR

Peephole Optimization

Fisgonear el código

- Una vez emitido el código final se recorre desde el principio a través de una "ventana de observación" (*peephole*).
- Se identifican secuencias de instrucciones que puedan ser reemplazadas por otras más cortas o más rápidas.
  - La cantidad de instrucciones examinadas en la ventana es decisión del implantador.
  - Las secuencias a identificar dependen del lenguaje destino.
- Aplicar una mejora puede crear la oportunidad de aplicar más mejoras sobre el resultado – recorrer varias veces.

Algunos fisgonean más que otros
y no hay nada de malo en eso.

UNIVERSIDAD SIMÓN BOLÍVAR

Peephole Optimization

Eliminar cargas y almacenamientos redundantes

```
LD R0, a
ST a, R0
```

- Podemos eliminar la instrucción de almacenamiento, siempre y cuando *ambas* estén en el mismo bloque básico.
- El método de generación discutido hoy **nunca** genera código como este, pero el método inocente si.

UNIVERSIDAD SIMÓN BOLÍVAR

Peephole Optimization

Simplificación algebráica

- `INC` y `DEC` en lugar de sumar y restar uno.
- *Secuencias* de `INC` y `DEC` cuando el costo total sea menor que sumas o restas con valores pequeños.
- Multiplicación repetida en lugar de potencias.
- *Shifts* en lugar de multiplicaciones y divisiones por potencias de dos, teniendo cuidado con las *representaciones*.
- División entre una constante punto flotante podría reemplazarse por la multiplicación con el inverso.

Importante conocer la *jerga* de la máquina destino.

UNIVERSIDAD SIMÓN BOLÍVAR

Peephole Optimization

Optimización del Flujo de Control

```
    goto L1
    ...
L1: goto L2
```

$\Rightarrow$

```
    goto L2
    ...
L1: goto L2
```

- Eliminar saltos a saltos.
- Si no quedan saltos a `L1`, se podría eliminar la instrucción

  `L1: goto L2`

  siempre que esté precedida por un salto incondicional.

UNIVERSIDAD SIMÓN BOLÍVAR

Peephole Optimization

Optimización del Flujo de Control

```
    goto L1
    ...
L1: if a < b goto L2
L3:
```

$\Rightarrow$

```
    if a < b goto L2
    goto L3
    ...
L3:
```

- Funciona si `goto L1` es el *único* salto a `L1` en todo el programa.
- La cantidad de instrucciones es la misma.
- La primera forma *siempre* usa el salto condicional,
  la segunda forma *a veces* le pasa por encima – más eficiente.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*]
  - Secciones 8.6 y 8.7
  - Ejercicios 8.6.1 a 8.6.5, 8.7.1 a 8.7.3.

UNIVERSIDAD SIMÓN BOLÍVAR