Representaciones Intermedias

CI4721 – Lenguajes de Programación II

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright ©2012-2016

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching

- Generar *Jumping Code* requiere establecer la correspondencia entre salto y destino – cuando se emite el salto, se desconoce el destino.
- El esquema estudiado hasta ahora resuelve el problema empleando atributos heredados.
  - Código para booleanos transfiere control según las etiquetas heredadas.
  - Código para instrucciones establece destinos "de afuera hacia adentro".
  - Esto requiere dos pasadas.

Backpatching

- Generar *Jumping Code* requiere establecer la correspondencia entre salto y destino – cuando se emite el salto, se desconoce el destino.
- El esquema estudiado hasta ahora resuelve el problema empleando atributos heredados.
  - Código para booleanos transfiere control según las etiquetas heredadas.
  - Código para instrucciones establece destinos "de afuera hacia adentro".
  - Esto requiere dos pasadas.

Construiremos un esquema S-atribuido
que permita hacerlo en *una* sola pasada.

Backpatching

Please fill the blanks to make it happen

- *Backpatching* – generar código y “parcharlo” al final.
- La técnica asigna etiquetas para *todas* las instrucciones.
    - Al final no se usarán todas, pero simplifica el proceso.
    - Posible refinarla para estructuras dinámicas o archivos con marcas.

Backpatching

Please fill the blanks to make it happen

- *Backpatching* – generar código y "parcharlo" al final.
- La técnica asigna etiquetas para *todas* las instrucciones.
  - Al final no se usarán todas, pero simplifica el proceso.
  - Posible refinarla para estructuras dinámicas o archivos con marcas.
- Generar *jumping code* para expresiones booleanas, pero:
  - Dejar en blanco las etiquetas – desconocidas cuando se genera el salto.
  - Recordar *cuáles* etiquetas están en blanco – atributo heredado simple se convierte en atributo sintetizado complejo.
  - Tan pronto se determina un destino de salto, se "parchan" las instrucciones desde las cuales se salta completando sus etiquetas.

Técnica diseñada específicamente para *LR*

Backpatching para expresiones booleanas

Atributos e infraestructura

- Atributos sintetizados *B.truelist* y *B.falselist*
  - Listas de apuntadores a las instrucciones que requieren ser parchadas – se van llenando con instrucciones pendientes por completar.
  - `makelist()` – crea una lista con el primer apuntador.
  - `merge()` – combina dos listas de apuntadores en una sola.

Backpatching para expresiones booleanas

Atributos e infraestructura

- Atributos sintetizados *B.truelist* y *B.falselist*
  - Listas de apuntadores a las instrucciones que requieren ser parchadas – se van llenando con instrucciones pendientes por completar.
  - `makelist()` – crea una lista con el primer apuntador.
  - `merge()` – combina dos listas de apuntadores en una sola.
- `nextinstr`
  - Variable global con apuntador a *siguiente* instrucción a generar.
  - Cada invocación a `gen()` la actualiza apropiadamente.
  - Supondremos que son índices enteros en la secuencia – posición en arreglo, *offset* en archivo temporal, ...

Backpatching para expresiones booleanas

Atributos e infraestructura

- Atributos sintetizados *B.truelist* y *B.falselist*
  - Listas de apuntadores a las instrucciones que requieren ser parchadas – se van llenando con instrucciones pendientes por completar.
  - `makelist()` – crea una lista con el primer apuntador.
  - `merge()` – combina dos listas de apuntadores en una sola.
- `nextinstr`
  - Variable global con apuntador a *siguiente* instrucción a generar.
  - Cada invocación a `gen()` la actualiza apropiadamente.
  - Supondremos que son índices enteros en la secuencia – posición en arreglo, *offset* en archivo temporal, ...
- `backpatch()` – aplica los parches
  - Recorre una lista de apuntadores a instrucciones pendientes.
  - Rellena la etiqueta pendiente de cada instrucción indicada en la lista, usando la dirección destino de salto suministrada como argumento.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching Expresiones Booleanas

Jumping Code – casos base

$B \rightarrow$ **true**

$B \rightarrow$ **false**

$B \rightarrow$ **not** B_1

$B \rightarrow E_1 < E_2$

Backpatching Expresiones Booleanas

Jumping Code – casos base

$B \rightarrow$ **true** $\quad \left\{ \begin{array}{l} B.truelist \leftarrow makelist(nextinstr) \\ gen('\text{goto } _')\} \end{array} \right\}$

$B \rightarrow$ **false** $\quad \left\{ \begin{array}{l} B.falselist \leftarrow makelist(nextinstr) \\ gen('\text{goto } _')\} \end{array} \right\}$

$B \rightarrow$ **not** B_1

$B \rightarrow E_1 < E_2$

- Generar las instrucciones incompletas y recordar su posición.

Backpatching Expresiones Booleanas

Jumping Code – casos base

$B \rightarrow$ **true** $\quad \left\{ \begin{array}{l} B.truelist \leftarrow makelist(nextinstr) \\ gen('\text{goto }_')\} \end{array} \right\}$

$B \rightarrow$ **false** $\quad \left\{ \begin{array}{l} B.falselist \leftarrow makelist(nextinstr) \\ gen('\text{goto }_')\} \end{array} \right\}$

$B \rightarrow$ **not** B_1 $\quad \left\{ \begin{array}{l} B.truelist \leftarrow B_1.falselist \\ B.falselist \leftarrow B_1.truelist \end{array} \right\}$

$B \rightarrow E_1 < E_2$

- Generar las instrucciones incompletas y recordar su posición.
- La negación mantiene el código, pero invierte el sentido de saltos.

Backpatching Expresiones Booleanas

Jumping Code – casos base

$$B \rightarrow \textbf{true} \quad \left\{ \begin{array}{l} B.truelist \leftarrow makelist(nextinstr) \\ gen('\text{goto _}')\} \end{array} \right\}$$

$$B \rightarrow \textbf{false} \quad \left\{ \begin{array}{l} B.falselist \leftarrow makelist(nextinstr) \\ gen('\text{goto _}')\} \end{array} \right\}$$

$$B \rightarrow \textbf{not}\ B_1 \quad \left\{ \begin{array}{l} B.truelist \leftarrow B_1.falselist \\ B.falselist \leftarrow B_1.truelist \end{array} \right\}$$

$$B \rightarrow E_1 < E_2 \quad \left\{ \begin{array}{l} B.truelist \leftarrow makelist(nextinstr) \\ B.falselist \leftarrow makelist(nextinstr+1) \\ gen('\text{if}'\ E_1.addr\ '<'\ E_2.addr\ '\text{goto _}') \\ gen('\text{goto _}') \end{array} \right\}$$

- Generar las instrucciones incompletas y recordar su posición.
- La negación mantiene el código, pero invierte el sentido de saltos.
- Cada operador relacional genera un par de saltos a etiquetas desconocidas consecutivas que deben recordarse.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching Expresiones Booleanas

Jumping Code – disyunción y conjunción

$B \rightarrow B_1$ **or** $M\ B_2$

$B \rightarrow B_1$ **and** $M\ B_2$

$M \rightarrow \lambda$

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching Expresiones Booleanas

Jumping Code – disyunción y conjunción

$B \rightarrow B_1$ **or** $M\ B_2$ $\left\{ backpatch(B_1.falselist, M.instr) \right\}$

$B \rightarrow B_1$ **and** $M\ B_2$ $\left\{ backpatch(B_1.truelist, M.instr) \right\}$

$M \rightarrow \lambda$ $\{M.instr = nextinstr\}$

- Parchar el brazo cuya dirección destino puede calcularse.
- Marcador para acceder a la dirección a usar para parchar.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching Expresiones Booleanas

Jumping Code – disyunción y conjunción

$$B \rightarrow B_1 \textbf{ or } M\ B_2 \qquad \left\{ \begin{array}{l} backpatch(B_1.falselist, M.instr) \\ B.truelist \leftarrow merge(B_1.truelist, B_2.truelist) \\ \end{array} \right\}$$

$$B \rightarrow B_1 \textbf{ and } M\ B_2 \qquad \left\{ \begin{array}{l} backpatch(B_1.truelist, M.instr) \\ B.falselist \leftarrow merge(B_1.falselist, B_2.falselist) \\ \end{array} \right\}$$

$$M \rightarrow \lambda \qquad \{M.instr = nextinstr\}$$

- Parchar el brazo cuya dirección destino puede calcularse.
- Marcador para acceder a la dirección a usar para parchar.
- Combinar la información pendiente para brazos de corto circuito – aún no tienen valor pues dependen de código por generar.

Backpatching Expresiones Booleanas

Jumping Code – disyunción y conjunción

$$B \rightarrow B_1 \textbf{ or } M\ B_2 \quad \left\{ \begin{array}{l} backpatch(B_1.falselist, M.instr) \\ B.truelist \leftarrow merge(B_1.truelist, B_2.truelist) \\ B.falselist \leftarrow B_2.falselist \end{array} \right\}$$

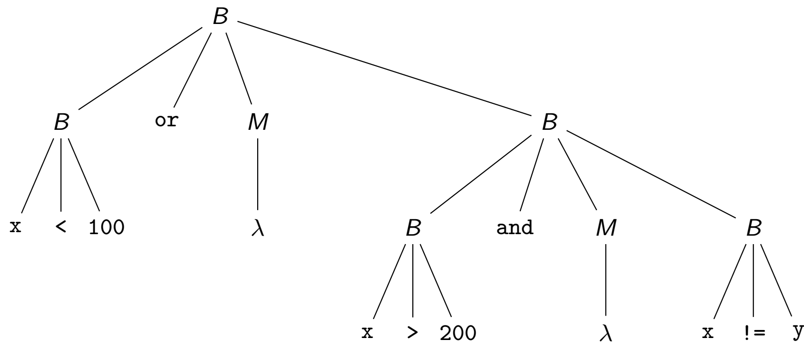
$$B \rightarrow B_1 \textbf{ and } M\ B_2 \quad \left\{ \begin{array}{l} backpatch(B_1.truelist, M.instr) \\ B.falselist \leftarrow merge(B_1.falselist, B_2.falselist) \\ B.truelist \leftarrow B_2.truelist \end{array} \right\}$$

$$M \rightarrow \lambda \quad \{M.instr = nextinstr\}$$

- Parchar el brazo cuya dirección destino puede calcularse.
- Marcador para acceder a la dirección a usar para parchar.
- Combinar la información pendiente para brazos de corto circuito – aún no tienen valor pues dependen de código por generar.
- Mantener los pendientes que no aumentaron.

UNIVERSIDAD SIMÓN BOLÍVAR

El mismo ejemplo, con *backpatching*

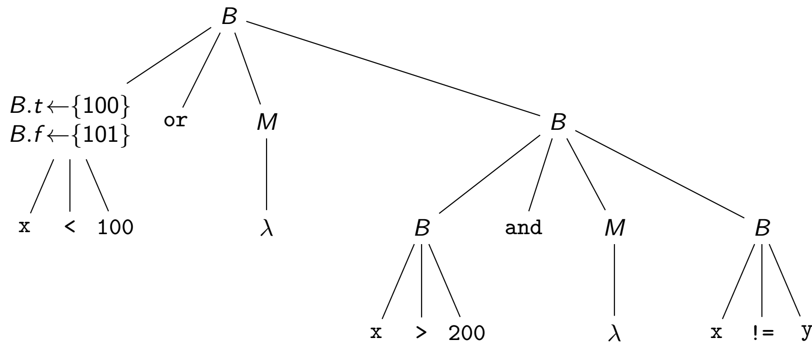



```
->100:
  101:
  102:
  103:
  104:
  105:
```

- Arbol construido de abajo arriba.
- Generaremos código sobre un arreglo de estructuras *Quads* – eso permite "llenar los blancos".

UNIVERSIDAD SIMÓN BOLÍVAR

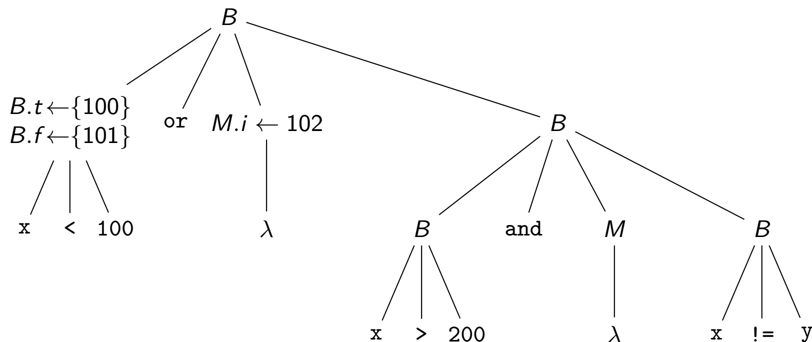
El mismo ejemplo, con *backpatching*




```
  100: if x < 100 goto _
  101: goto _
->102:
  103:
  104:
  105:
```

- Reducir `x < 100` genera dos instrucciones con destino pendiente
- $B.truelist = \{100\}$
  $B.falselist = \{101\}$.

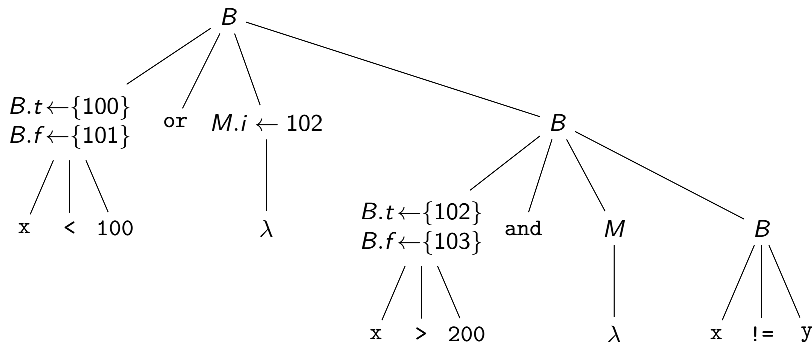
El mismo ejemplo, con *backpatching*




```
  100: if x < 100 goto _
  101: goto _
->102:
  103:
  104:
  105:
```

- Reducir M registra siguiente dirección.
- Permitirá saltar hacia el destino real en el lado derecho cuando se genere.

El mismo ejemplo, con *backpatching*




```
  100: if x < 100 goto _
  101: goto _
  102: if x > 200 goto _
  103: goto _
->104:
  105:
```

- Reducir `x > 200` genera dos instrucciones con destino pendiente
- $B.truelist = \{102\}$
  $B.falselist = \{103\}$.

El mismo ejemplo, con *backpatching*

B

$B.t \leftarrow \{100\}$ $B.f \leftarrow \{101\}$ or $M.i \leftarrow 102$ B

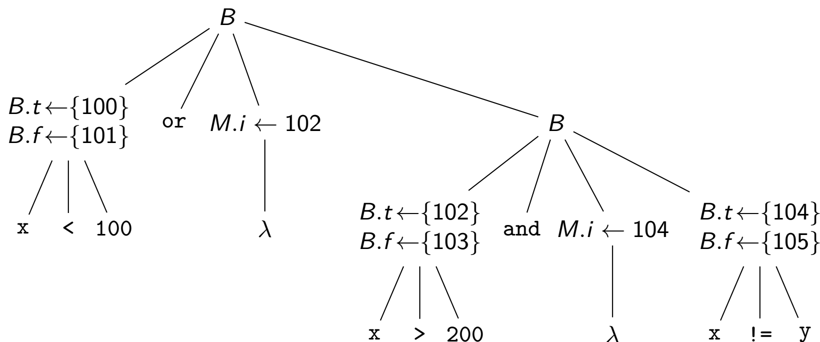
x   <   100 λ $B.t \leftarrow \{102\}$ $B.f \leftarrow \{103\}$ and $M.i \leftarrow 104$ B

x   >   200 λ x   !=   y

```
  100: if x < 100 goto _
  101: goto _
  102: if x > 200 goto _
  103: goto _
->104:
  105:
```

- Reducir M registra siguiente dirección.
- Permitirá saltar hacia el destino real en el lado derecho cuando se genere.

El mismo ejemplo, con *backpatching*




```
100: if x < 100 goto _
101: goto _
102: if x > 200 goto _
103: goto _
104: if x != y goto _
105: goto _
```

- Reducir `x != y` genera dos instrucciones con destino pendiente
- $B.truelist = \{102\}$
  $B.falselist = \{103\}$.

UNIVERSIDAD SIMÓN BOLÍVAR

El mismo ejemplo, con *backpatching*

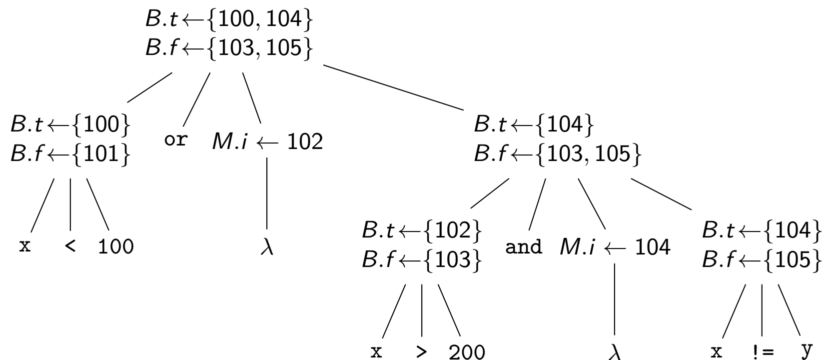
$$
\begin{array}{c}
B \\
\swarrow\ \ \downarrow\ \ \downarrow\ \ \searrow
\end{array}
$$

- $B.t \leftarrow \{100\}$, $B.f \leftarrow \{101\}$ → `x` `<` `100`
- `or`
- $M.i \leftarrow 102$ → λ
- $B.t \leftarrow \{104\}$, $B.f \leftarrow \{103, 105\}$
  - $B.t \leftarrow \{102\}$, $B.f \leftarrow \{103\}$ → `x` `>` `200`
  - `and`
  - $M.i \leftarrow 104$ → λ
  - $B.t \leftarrow \{104\}$, $B.f \leftarrow \{105\}$ → `x` `!=` `y`

```
100: if x < 100 goto _
101: goto _
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

- Reducir $B \rightarrow B$ **and** $M\ B$
  $backpatch(B.t, M.i)$
- $B.truelist = \{104\}$ – copiada,
  $B.falselist = \{103, 105\}$ – combinada.

El mismo ejemplo, con *backpatching*




```
100: if x < 100 goto _
101: goto 102
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

- Reducir $B \to B$ `or` $M\ B$
  $backpatch(B.f, M.i)$
- $B.truelist = \{100, 104\}$ – combinada,
  $B.falselist = \{103, 105\}$ – copiada.

El mismo ejemplo, con *backpatching*

```
100: if x < 100 goto _
101: goto 102
102: if x > 200 goto 104
103: goto _
104: if x != y goto _
105: goto _
```

- La expresión booleana será cierta si se efectúan los `goto` de las instrucciones 100 y 104 – por eso $B.truelist = \{100, 104\}$.
- La expresión booleana será falsa si se efectúan los `goto` de las instrucciones 103 y 105 – por eso $B.falselist = \{100, 104\}$.

Completar esas etiquetas depende
de la estructura de control que la utiliza.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de Instrucciones

Ahora parchamos la siguiente instrucción

- Usaremos dos no terminales para las instrucciones
  - S para las instrucciones individuales – selector, iterador y asignación.
  - L para la secuenciación dentro de un bloque.
- El atributo *nextlist* contendrá una lista de todas las instrucciones que necesitan saltar hacia la instrucción siguiente.
  - Se inicializa en las asignaciones – una asignación no necesita saltar.
  - Se copia directamente en los bloques – la instrucción envolvente probablemente querrá saltar al final.
  - Se manipula y actualiza con las *truelist* y *falselist* de las expresiones booleanas de control.

Backpatching de instrucciones

Esquema de generación para casos base

$S \rightarrow \textbf{id} := E$

$S \rightarrow \{\ L\ \}$

$L \rightarrow S$

$L \rightarrow L_1 \quad M\ S$

$M \rightarrow \lambda$

Backpatching de instrucciones

Esquema de generación para casos base

$$S \rightarrow \textbf{id} := E \qquad \{\ S.nextlist = \{\}\ \}$$
$$S \rightarrow \{\ L\ \}$$
$$L \rightarrow S$$

$$L \rightarrow L_1 \quad M\ S$$

$$M \rightarrow \lambda$$

- Asignaciones no tienen saltos pendientes.

Backpatching de instrucciones

Esquema de generación para casos base

$$S \rightarrow \textbf{id} := E \qquad \{\ S.nextlist = \{\}\ \}$$
$$S \rightarrow \{\ L\ \} \qquad \{\ S.nextlist = L.nextlist\ \}$$
$$L \rightarrow S$$

$$L \rightarrow L_1 \ \ M\ S$$

$$M \rightarrow \lambda$$

- Asignaciones no tienen saltos pendientes.
- Saltos pendientes en el cuerpo serán pendientes del bloque.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación para casos base

$$
\begin{array}{ll}
S \rightarrow \textbf{id} := E & \{\ S.nextlist = \{\}\ \} \\
S \rightarrow \{\ L\ \} & \{\ S.nextlist = L.nextlist\ \} \\
L \rightarrow S & \{\ L.nextlist = S.nextlist\ \}
\end{array}
$$

$$L \rightarrow L_1 \quad M\ S$$

$$M \rightarrow \lambda$$

- Asignaciones no tienen saltos pendientes.
- Saltos pendientes en el cuerpo serán pendientes del bloque.
- Primera instrucción de lista inicializa los saltos pendientes. . .

Backpatching de instrucciones

Esquema de generación para casos base

$$S \to \textbf{id} := E \quad \{\ S.nextlist = \{\}\ \}$$

$$S \to \{\ L\ \} \quad \{\ S.nextlist = L.nextlist\ \}$$

$$L \to S \quad \{\ L.nextlist = S.nextlist\ \}$$

$$L \to L_1 \ \ M\ S \quad \left\{ \begin{array}{l} backpatch(L_1.nextlist, M.instr) \\ L.nextlist \leftarrow S.nextlist \end{array} \right\}$$

$$M \to \lambda \quad \{\ M.instr = nextinstr\ \}$$

- Asignaciones no tienen saltos pendientes.
- Saltos pendientes en el cuerpo serán pendientes del bloque.
- Primera instrucción de lista inicializa los saltos pendientes. . .
- . . . y al procesar la $n-$ésima instrucción
  - Se parchan las anteriores de ser posible.
  - Se incorporan los saltos pendientes de la nueva instrucción.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `if-then`

$$S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1 \quad \left\{ \begin{array}{l} backpatch(B.truelist, M.instr) \\ S.nextlist \leftarrow merge(B.falselist, S_1.nextlist) \end{array} \right\}$$

- El marcador M obtiene la dirección inicial del cuerpo del condicional.
- Sirve para parchar $B.truelist$ – ingresar al cuerpo si B resulta **true**.
- Si B resulta **false** es necesario saltar a la siguiente instrucción – combinar los pendientes de B con los siguientes de S_1.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `if-then-else`

$S \rightarrow \textbf{if}(\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$

$N \rightarrow \lambda$

Backpatching de instrucciones

Esquema de generación – caso `if-then-else`

$S \rightarrow$ **if**(B) M_1 S_1 N **else** M_2 S_2 $\left\{ \begin{array}{l} backpatch(B.truelist, M_1.instr) \\ \\ \\ \end{array} \right\}$

$N \rightarrow \lambda$

- M_1 obtiene la dirección inicial del brazo cuando B resulta **true**.
- Sirve para parchar $B.truelist$ – ingresar a ese brazo si B resulta **true**.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `if-then-else`

$$S \rightarrow \textbf{if}(\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2 \qquad \left\{ \begin{array}{l} backpatch(B.truelist, M_1.instr) \\ backpatch(B.falselist, M_2.instr) \\ \\ \end{array} \right\}$$

$$N \rightarrow \lambda$$

- M_1 obtiene la dirección inicial del brazo cuando B resulta **true**.
- Sirve para parchar $B.truelist$ – ingresar a ese brazo si B resulta **true**.
- M_2 obtiene la dirección inicial del brazo cuando B resulta **false**.
- Sirve para parchar $B.falselist$ – ingresar a ese brazo si B resulta **false**.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `if-then-else`

$$S \rightarrow \textbf{if}(\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2 \qquad \left\{ \begin{array}{l} backpatch(B.truelist, M_1.instr) \\ backpatch(B.falselist, M_2.instr) \\ \\ \end{array} \right\}$$

$$N \rightarrow \lambda \qquad \left\{ \begin{array}{l} N.nextlist \leftarrow makelist(nextinstr) \\ gen('\text{goto _}') \end{array} \right\}$$

- M_1 obtiene la dirección inicial del brazo cuando B resulta **true**.
- Sirve para parchar $B.truelist$ – ingresar a ese brazo si B resulta **true**.
- M_2 obtiene la dirección inicial del brazo cuando B resulta **false**.
- Sirve para parchar $B.falselist$ – ingresar a ese brazo si B resulta **false**.
- N obtiene la última dirección del primer brazo e inserta el salto final.

Backpatching de instrucciones

Esquema de generación – caso `if-then-else`

$$S \rightarrow \textbf{if}(\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2 \qquad \left\{\begin{array}{l} backpatch(B.truelist, M_1.instr) \\ backpatch(B.falselist, M_2.instr) \\ t \leftarrow merge(S_1.nextlist, N.nextlist) \\ S.nextlist \leftarrow merge(t, S_2.nextlist) \end{array}\right\}$$

$$N \rightarrow \lambda \qquad \left\{\begin{array}{l} N.nextlist \leftarrow makelist(nextinstr) \\ gen('\text{goto }_') \end{array}\right\}$$

- M_1 obtiene la dirección inicial del brazo cuando B resulta **true**.
- Sirve para parchar $B.truelist$ – ingresar a ese brazo si B resulta **true**.
- M_2 obtiene la dirección inicial del brazo cuando B resulta **false**.
- Sirve para parchar $B.falselist$ – ingresar a ese brazo si B resulta **false**.
- N obtiene la última dirección del primer brazo e inserta el salto final.
- Combinar todos los saltos pendientes.

Backpatching de instrucciones

Esquema de generación – caso `while`

$S \rightarrow$ **while** M_1 (B) M_2 S_1

Backpatching de instrucciones

Esquema de generación – caso `while`

$$S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1 \quad \left\{ \begin{array}{l} backpatch(S_1.nextlist, M_1.instr) \\ \\ \\ \end{array} \right\}$$

- M_1 obtiene la dirección inicial de evaluación del condicional.
- Sirve para parchar $S_1.nextlist$ – si termina S_1, evaluar de nuevo.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `while`

$$S \rightarrow \textbf{while } M_1 \text{ (} B \text{) } M_2 \; S_1 \qquad \left\{ \begin{array}{l} backpatch(S_1.nextlist, M_1.instr) \\ backpatch(B.truelist, M_2.instr) \\ \\ \\ \end{array} \right\}$$

- M_1 obtiene la dirección inicial de evaluación del condicional.
- Sirve para parchar $S_1.nextlist$ – si termina S_1, evaluar de nuevo.
- M_2 obtiene la dirección inicial del cuerpo del ciclo.
- Sirve para parchar $B.truelist$ – ingresar al ciclo si B resulta **true**.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `while`

$$S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1 \quad \left\{ \begin{array}{l} backpatch(S_1.nextlist, M_1.instr) \\ backpatch(B.truelist, M_2.instr) \\ S.nextlist \leftarrow B.falselist \\ \end{array} \right\}$$

- M_1 obtiene la dirección inicial de evaluación del condicional.
- Sirve para parchar $S_1.nextlist$ – si termina S_1, evaluar de nuevo.
- M_2 obtiene la dirección inicial del cuerpo del ciclo.
- Sirve para parchar $B.truelist$ – ingresar al ciclo si B resulta **true**.
- Los saltos pendientes por parchar en S son tomados de $B.falselist$.

UNIVERSIDAD SIMÓN BOLÍVAR

Backpatching de instrucciones

Esquema de generación – caso `while`

$$S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1 \qquad \left\{ \begin{array}{l} backpatch(S_1.nextlist, M_1.instr) \\ backpatch(B.truelist, M_2.instr) \\ S.nextlist \leftarrow B.falselist \\ gen('\text{goto}'\ M_1.instr) \end{array} \right\}$$

- M_1 obtiene la dirección inicial de evaluación del condicional.
- Sirve para parchar $S_1.nextlist$ – si termina S_1, evaluar de nuevo.
- M_2 obtiene la dirección inicial del cuerpo del ciclo.
- Sirve para parchar $B.truelist$ – ingresar al ciclo si B resulta **true**.
- Los saltos pendientes por parchar en S son tomados de $B.falselist$.
- Generar el salto incondicional para evaluar el condicional nuevamente.

Multiselectores – `switch` o `case`

¿Cómo traducir?

$$S \rightarrow \textbf{switch} \ (\ E \ ) \ \{ \ C \ \}$$
$$C \rightarrow \textbf{case} \ E \ : \ S$$
$$C \rightarrow \textbf{default} : \ S$$
$$C \rightarrow C \ \textbf{case} \ E \ : \ S$$

- Omitiremos las verificaciones estáticas obvias
  - Tipo de E equivalente al tipo de las todas las etiquetas.
  - A lo sumo *una* etiqueta `default`.
  - No puede haber expresiones selectoras duplicadas.
- Etiquetas constantes o variables – depende del lenguaje.

UNIVERSIDAD SIMÓN BOLÍVAR

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.
- El segundo paso es un conjunto de n saltos – tantos como etiquetas.

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.
- El segundo paso es un conjunto de n saltos – tantos como etiquetas.
  - Secuencia de saltos condicionales – si son pocas (menos de 10).

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.
- El segundo paso es un conjunto de n saltos – tantos como etiquetas.
  - Secuencia de saltos condicionales – si son pocas (menos de 10).
  - Tabla de despacho – ciclo que recorre una tabla de saltos.
    - Parejas (*value*, *address*) – si la expresión selectora coincide con *value*, saltar hasta *address* para ejecutar el cuerpo.
    - Ultima pareja tiene expresión selectora y dirección del bloque `default`.

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.
- El segundo paso es un conjunto de n saltos – tantos como etiquetas.
  - Secuencia de saltos condicionales – si son pocas (menos de 10).
  - Tabla de despacho – ciclo que recorre una tabla de saltos.
    - Parejas (*value*, *address*) – si la expresión selectora coincide con *value*, saltar hasta *address* para ejecutar el cuerpo.
    - Ultima pareja tiene expresión selectora y dirección del bloque `default`.
  - Arreglo de despacho – si las etiquetas cubren casi todo el rango.

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.
- El segundo paso es un conjunto de n saltos – tantos como etiquetas.
  - Secuencia de saltos condicionales – si son pocas (menos de 10).
  - Tabla de despacho – ciclo que recorre una tabla de saltos.
    - Parejas (*value*, *address*) – si la expresión selectora coincide con *value*, saltar hasta *address* para ejecutar el cuerpo.
    - Ultima pareja tiene expresión selectora y dirección del bloque `default`.
  - Arreglo de despacho – si las etiquetas cubren casi todo el rango.
  - Tabla de hash, búsqueda binaria, . . .

Multiselectores – `switch` o `case`

¿Cómo traducir?

- La semántica de la instrucción requiere una traducción tal que:
  1. Evalúe el valor de la expresión inicial E una sola vez.
  2. Encuentre alguna etiqueta `case` cuya etiqueta coincida con este valor, y ejecute la instrucción asociada.
  3. Si ninguna etiqueta coincide con el valor, ejecutar la instrucción asociada a la etiqueta `default`.
- El segundo paso es un conjunto de n saltos – tantos como etiquetas.
  - Secuencia de saltos condicionales – si son pocas (menos de 10).
  - Tabla de despacho – ciclo que recorre una tabla de saltos.
    - Parejas (*value*, *address*) – si la expresión selectora coincide con *value*, saltar hasta *address* para ejecutar el cuerpo.
    - Ultima pareja tiene expresión selectora y dirección del bloque `default`.
  - Arreglo de despacho – si las etiquetas cubren casi todo el rango.
  - Tabla de hash, búsqueda binaria, ...

Cualquier técnica es un compromiso demasiado fuerte.

Una representación neutral

"Cascada de selectores"

```
        evaluar E en t
        if t != V1 goto L1
        código para S1
        goto next
L1:     if t != V2 goto L2
        código para S2
        goto next
L2:
        ...
Ln:     if t != Vn goto Ld
        código para Sn
        goto next
Ld:     código para default (si hubiera)
next:
```

- Saltos hacia adelante – atributos heredados o *backpatching*.
- Mezcla de saltos y código dificulta la optimización.

UNIVERSIDAD SIMÓN BOLÍVAR

Una representación neutral

Pero esta vez más astuta

```
        evaluar E en t
        goto test
L1:     código para S1
        goto next
        ...
Ln:     código para Sn
        goto next
Ld:     código para default (si hubiera)
        goto next
test:   if t = V1 goto L1
        ...
        if t = Vn goto Ln
        goto Ld
next:
```

¡Comparaciones en un sólo bloque!

UNIVERSIDAD SIMÓN BOLÍVAR

Facilitando la optimización

These are the `ifs` you're looking for

- Extenderemos el código de tres direcciones con la instrucción

```
case t v l
```

  como *sinónimo* para

```
if t = v goto l
```

- Simplifica la tarea del optimizador de código
  - Trivial identificar el bloque final de comparación.
  - Facilita determinar los valores posibles.
  - Cualquiera de las técnicas particulares puede derivarse con ésta información.

Llamadas a Funciones y Procedimientos

- Generar código intermedio para llamadas es *intencionalmente* simple.
- Detalles de implantación se dejan para la generación de código final:
  - Pila real o simulada.
  - Parámetros pasados por pila, registros o ambos.
  - Pasaje por valor, referencia o copia.
  - Valores de retorno pasados por pila, registros o ambos.
- Basta generar el cálculo de los argumentos y representar la llamada.
  - Si el lenguaje determina un orden particular, se establece de una vez.
  - Si el lenguaje permite cualquier orden, el optimizador hará las reorganizaciones más adelante.

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Llamadas

Mezclando cálculo y pasaje de argumentos

$$S \rightarrow \textbf{id}\ (\ ) \qquad \left\{ gen('\text{call}'\ \textbf{id}\ ','\ 0) \right\}$$

$$S \rightarrow \textbf{id}\ (\ A\ ) \qquad \left\{ gen('\text{call}'\ \textbf{id}\ ','\ A.length) \right\}$$

$$E \rightarrow \textbf{id}\ (\ ) \qquad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr\ ':= \text{call}'\ \textbf{id}\ ','\ 0) \end{array} \right\}$$

$$E \rightarrow \textbf{id}\ (\ A\ ) \qquad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr\ ':= \text{call}'\ \textbf{id}\ ','\ A.length) \end{array} \right\}$$

$$A \rightarrow E \qquad \left\{ \begin{array}{l} gen('\text{param}'\ E.addr) \\ A.length \leftarrow 1 \end{array} \right\}$$

$$A \rightarrow A_1\ ,\ E \qquad \left\{ \begin{array}{l} gen('\text{param}'\ E.addr) \\ A.length \leftarrow A_1.length + 1 \end{array} \right\}$$

- Sin argumentos vs. cantidad arbitraria.
- Expresión (función) vs. instrucción (procedimiento).

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Llamadas

Mezclando cálculo y pasaje de argumento

Así, para la llamada

```
qux = foo ( bar + 17, 42, baz * 69 )
```

se genera el código intermedio

```
t1 := bar + 17
param t1
t2 := 42
param t2
t3 := baz * 69
param t3
qux := call foo,3
```

Para separar cálculo de pasaje,
es necesaria una estructura de datos auxiliar

UNIVERSIDAD SIMÓN BOLÍVAR

Generación de Llamadas

Separando cálculo de pasaje de argumentos – izquierda a derecha

$$S \rightarrow \textbf{id} \; (\; A \;) \quad \left\{ \begin{array}{l} \textbf{foreach } a \textbf{ in } A.queue \textbf{ do} \\ \; gen('\text{param }' \; a) \\ E.addr \leftarrow newtemp() \\ gen('\text{call}' \; \textbf{id} \; ',' \; A.length) \end{array} \right\}$$

$$E \rightarrow \textbf{id} \; (\; A \;) \quad \left\{ \begin{array}{l} \textbf{foreach } a \textbf{ in } A.queue \textbf{ do} \\ \; gen('\text{param }' \; a) \\ E.addr \leftarrow newtemp() \\ gen(E.addr \; ':= \text{call}' \; \textbf{id} \; ',' \; A.length) \end{array} \right\}$$

$$A \rightarrow E \quad \left\{ \begin{array}{l} A.queue \leftarrow new \;\; Queue \\ enqueue(A.queue, E.addr) \\ A.length \leftarrow 1 \end{array} \right\}$$

$$A \rightarrow A_1 \; , \; E \quad \left\{ \begin{array}{l} enqueue(A_1.queue, E.addr) \\ A.queue \leftarrow A_1.queue \\ A.length \leftarrow A.length + 1 \end{array} \right\}$$

Generación de Llamadas

Separando cálculo y pasaje de argumento – izquierda a derecha

Esto generaría el código intermedio

```
t1 := bar + 17
t2 := 42
t3 := baz * 69
param t1
param t2
param t3
qux := call foo,3
```

- Se preserva el orden izquierda a derecha – como en Java.
- El código de la función no necesita alteración adicional.

Generación de Llamadas

Separando cálculo de pasaje de argumentos – derecha a izquierda

$$S \rightarrow \textbf{id} \; (\; A \;) \qquad \left\{ \begin{array}{l} \textbf{while } a \leftarrow pop(A.stack) \textbf{ do} \\ \;\; gen('\text{param }' \; a) \\ E.addr \leftarrow newtemp() \\ gen('\text{call}' \; \textbf{id} \; ',' \; A.length) \end{array} \right\}$$

$$E \rightarrow \textbf{id} \; (\; A \;) \qquad \left\{ \begin{array}{l} \textbf{while } a \leftarrow pop(A.stack) \textbf{ do} \\ \;\; gen('\text{param }' \; a) \\ E.addr \leftarrow newtemp() \\ gen(E.addr \; ':= \text{call}' \; \textbf{id} \; ',' \; A.length) \end{array} \right\}$$

$$A \rightarrow E \qquad \left\{ \begin{array}{l} A.stack \leftarrow new \;\; Stack \\ push(A.stack, E.addr) \\ A.length \leftarrow 1 \end{array} \right\}$$

$$A \rightarrow A_1 \; , \; E \qquad \left\{ \begin{array}{l} push(A_1.stack, E.addr) \\ A.stack \leftarrow A_1.stack \\ A.length \leftarrow A.length + 1 \end{array} \right\}$$

Generación de Llamadas

Separando cálculo y pasaje de argumento – derecha a izquierda

Esto generaría el código intermedio

```
t1 := bar + 17
t2 := 42
t3 := baz * 69
param t3
param t2
param t1
qux := call foo,3
```

- Se convierte el orden de derecha a izquierda – como en C.
- El código de la función necesita alterarse para tomarlo en cuenta.

Bibliografía

- [*Aho*] (Primera Edición)
  - Secciones 8.5 a 8.7
  - Ejercicios 8.11 a 8.15
- [*Aho*] (Segunda Edición)
  - Secciones 6.7 a 6.9
  - Ejercicios 6.7.1 a 6.7.3 y 6.8.3
- Escriba sendos esquemas de traducción dirigidos por sintaxis para las dos estrategias de generación para el `switch`.