Representaciones Intermedias

CI4721 – Lenguajes de Programación II

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright ©2012-2016

UNIVERSIDAD SIMÓN BOLÍVAR

Acceso a elementos en arreglos

- Haremos las suposiciones modernas de almacenamiento para los elementos de un arreglo
  - Elementos almacenados en posiciones contiguas.
  - Disposición por filas (*row major*).
  - La anchura de cada elemento es conocido e incluye el *padding* sufijo cuando aplique.
- El acceso a cualquier elemento requiere
  - Conocer la base de almacenamiento del arreglo.
  - Calcular el desplazamiento hasta la posición deseada.
  - Determinar si se quiere el *r-value* o el *l-value* según el contexto gramatical.

. . . con la mínima cantidad de operaciones

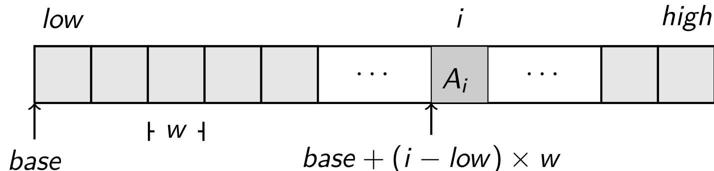
Acceso a Arreglos

Caso general de una dimensión

Sea un arreglo A

- Unidimensional.
- Elementos de anchura w.
- Indices en el rango *low* y *high*.
- Dispuesto a partir de la dirección *base*.

Entonces el $i-$ésimo elemento está en la dirección



Acceso a Arreglos

Cálculo más eficiente

Como *base*, *low* y w son conocidas a tiempo de compilación, podemos expandir y reordenar la fórmula para obtener

$$i \times w + (base - low \times w)$$

- ¡La expresión $c = base - low \times w$ es constante!
  - Se evalúa *una* vez a tiempo de compilación al procesar la declaración.
  - Se almacena como atributo para el arreglo en la tabla de símbolos.
- Para calcular la dirección de $A[i]$ solamente es necesario generar código para la expresión $i \times w + c$.

UNIVERSIDAD SIMÓN BOLÍVAR

Acceso a Arreglos

Agreguemos una dimensión

Si A tiene dos dimensiones y está dispuesto en *row major* ...

- $A[i,j]$ es equivalente a $A[i][j]$ – arreglo de arreglos.
- $A[i_1, i_2]$ – "desplazarse i_1 filas y agregar i_2 elementos".

Así que podemos deducir la fórmula

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

donde n_2 es la cardinalidad de i_2.
Como $base$, low_1, low_2 y w son conocidas a tiempo de compilación, nuevamente reescribimos

$$((i_1 \times n_2) \times i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

Acceso a Arreglos

El caso general

Si A es un arreglo $k-$dimensional con la $j-$ésima dimensión variando entre low_j y $high_j$, la posición

$$A[i_1, i_2, \ldots, i_k]$$

tiene dirección

$$((\cdots((i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots)n_k + i_k) \times w$$
$$+base - (\cdots((low_1 \times n_2 + low_2) \times n_3 + low_3)\cdots)n_k + low_k) \times w$$

- $\forall j : n_j = high_j - low_j + 1$ – el segundo sumando puede calcularse a tiempo de compilación.
- Si las dimensiones tienen base cero, se reduce a *base*.

UNIVERSIDAD SIMÓN BOLÍVAR

Acceso a Arreglos

Esquema directo con base cero – El uso de los arreglos

$$S \rightarrow L := E$$

$$E \rightarrow L$$

- L genera el nombre del arreglo y la secuencia de expresiones índice.
- Asignaciones y expresiones extendidas permitiendo acceso a arreglos.

UNIVERSIDAD SIMÓN BOLÍVAR

Acceso a Arreglos

Esquema directo con base cero – El uso de los arreglos

$$S \rightarrow L := E \quad \{gen(L.array.base\ '['\ L.addr\ ']'\ ':='\ E.addr)\}$$

$$E \rightarrow L \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp(); \\ gen(E.addr\ ':='\ L.array.base\ '['\ L.addr\ ']') \end{array} \right\}$$

- L genera el nombre del arreglo y la secuencia de expresiones índice.
- Asignaciones y expresiones extendidas permitiendo acceso a arreglos.
- El acceso a cualquier posición de un arreglo sólo depende de tener la base y el *offset* calculados previamente.
  - $L.array.base$ debe contener la base del arreglo.
  - $L.addr$ debe contener el *offset*.

Acceso a Arreglos

Esquema directo con base cero – Los cálculos

$L \rightarrow \textbf{id} \; [\; E \;]$

$L \rightarrow L_1 \; [\; E \;]$

Acceso a Arreglos

Esquema directo con base cero – Los cálculos

$$L \rightarrow \textbf{id} \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow lookup(top, \textbf{id}.lexema) \\ L.type \leftarrow contents(L.array.type) \\ L.addr \leftarrow newtemp() \\ gen(L.addr \; ':=' \; E.addr \; '*' \; L.type.width) \end{array} \right\}$$

$$L \rightarrow L_1 \; [\; E \;]$$

- $L.array$ apunta a la tabla de símbolos – la base está en $L.array.base$.
- `contents()` – aplicado sobre $array(I, T)$ retorna T

Acceso a Arreglos

Esquema directo con base cero – Los cálculos

$$L \rightarrow \textbf{id} \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow lookup(top, \textbf{id}.lexema) \\ L.type \leftarrow contents(L.array.type) \\ L.addr \leftarrow newtemp() \\ gen(L.addr \; ':=' \; E.addr \; '*' \; L.type.width) \end{array} \right\}$$

$$L \rightarrow L_1 \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow L_1.array \\ L.type \leftarrow contents(L_1.type) \\ t \leftarrow newtemp() \\ L.addr \leftarrow newtemp() \\ gen(t \; ':=' \; E.addr \; '*' \; L.type.width) \\ gen(L.addr \; ':=' \; L_1.addr \; '+' \; t) \end{array} \right\}$$

- $L.array$ apunta a la tabla de símbolos – la base está en $L.array.base$.
- `contents()` – aplicado sobre $array(I, T)$ retorna T

¿Cómo funciona?

Accediendo a un arreglo

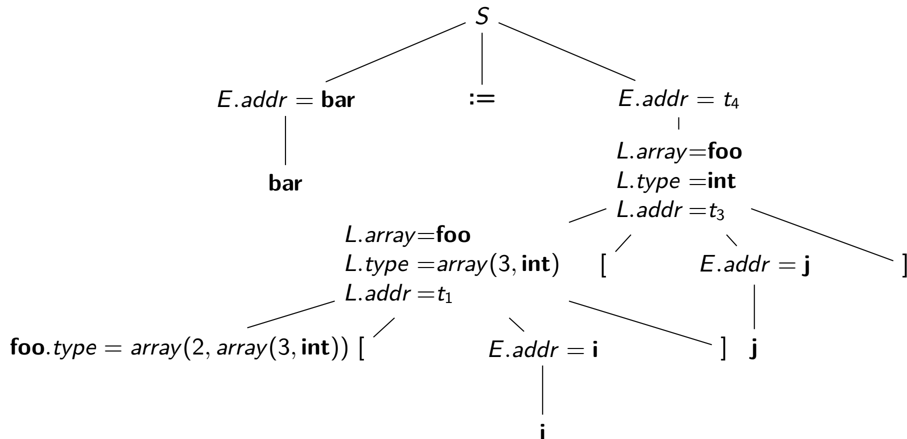
```
foo     : array [2] of array [3] of int;
bar,i,j : int;

bar := foo[i][j]
```

- Supongamos que la anchura de `int` es 4 bytes.
- El tipo de `foo` es $array(2, array(3, integer))$ – anchura es 24 bytes.
- El tipo de `foo[i]` es $array(3, integer)$ – anchura es 12 bytes.

¿Cómo funciona?

El árbol decorado – `bar := foo[i][j]`



¿Cómo funciona?

El código generado – `bar := foo[i][j]`

$L \rightarrow$ **id** $[\ E\ ]$

$L \rightarrow L_1\ [\ E\ ]$

$E \rightarrow L$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo funciona?

El código generado – `bar := foo[i][j]`

$$L \rightarrow \textbf{id} \; [\; E \;] \qquad \left\{ \begin{array}{l} L.array \leftarrow lookup(top, \textbf{id}.lexema) \\ L.type \leftarrow contents(L.array.type) \\ L.addr \leftarrow newtemp() \\ gen(L.addr \; ':=' \; E.addr \; '*' \; L.type.width) \end{array} \right\}$$

$$L \rightarrow L_1 \; [\; E \;]$$

$$E \rightarrow L$$

- $L.type = array(3, \textbf{int})$ con anchura 12
  $L.addr = t_1$, $E.addr = \textbf{i}$

```
t1  := i * 12
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo funciona?

El código generado – `bar := foo[i][j]`

$$L \rightarrow \textbf{id} \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow lookup(top, \textbf{id}.lexema) \\ L.type \leftarrow contents(L.array.type) \\ L.addr \leftarrow newtemp() \\ gen(L.addr \; ':=' \; E.addr \; '*' \; L.type.width) \end{array} \right\}$$

$$L \rightarrow L_1 \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow L_1.array \\ L.type \leftarrow contents(L_1.type) \\ t \leftarrow newtemp() \\ L.addr \leftarrow newtemp() \\ gen(t \; ':=' \; E.addr \; '*' \; L.type.width) \\ gen(L.addr \; ':=' \; L_1.addr \; '+' \; t) \end{array} \right\}$$

$$E \rightarrow L$$

- $L.type = array(3, \textbf{int})$ con anchura 12
  $L.addr = t_1$, $E.addr = \textbf{i}$
- $L.type = \textbf{int}$ con anchura 4
  $L.addr = t_3$, $E.addr = \textbf{j}$

```
t1  := i * 12
t2  := j * 4
t3  := t1 + t2
```

¿Cómo funciona?

El código generado – `bar := foo[i][j]`

$$L \rightarrow \textbf{id} \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow lookup(top, \textbf{id}.lexema) \\ L.type \leftarrow contents(L.array.type) \\ L.addr \leftarrow newtemp() \\ gen(L.addr \; ':=' \; E.addr \; '*' \; L.type.width) \end{array} \right\}$$

$$L \rightarrow L_1 \; [\; E \;] \quad \left\{ \begin{array}{l} L.array \leftarrow L_1.array \\ L.type \leftarrow contents(L_1.type) \\ t \leftarrow newtemp() \\ L.addr \leftarrow newtemp() \\ gen(t \; ':=' \; E.addr \; '*' \; L.type.width) \\ gen(L.addr \; ':=' \; L_1.addr \; '+' \; t) \end{array} \right\}$$

$$E \rightarrow L \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr \; ':=' \; L.array.base \; '[' \; L.addr \; ']') \end{array} \right\}$$

- $L.type = array(3, \textbf{int})$ con anchura 12
  $L.addr = t_1$, $E.addr = \textbf{i}$
- $L.type = \textbf{int}$ con anchura 4
  $L.addr = t_3$, $E.addr = \textbf{j}$
- $L.array.base = \textbf{foo}$, $L.addr = t_3$, $E.addr = t_4$

```
t1  := i * 12
t2  := j * 4
t3  := t1 + t2
t4  := foo [ t3 ]
```

¿Cómo funciona?

El código generado – `bar := foo[i][j]`

$$L \rightarrow \textbf{id}\ [\ E\ ] \quad \left\{ \begin{array}{l} L.array \leftarrow lookup(top, \textbf{id}.lexema) \\ L.type \leftarrow contents(L.array.type) \\ L.addr \leftarrow newtemp() \\ gen(L.addr\ ':='\ E.addr\ '*'\ L.type.width) \end{array} \right\}$$

$$L \rightarrow L_1\ [\ E\ ] \quad \left\{ \begin{array}{l} L.array \leftarrow L_1.array \\ L.type \leftarrow contents(L_1.type) \\ t \leftarrow newtemp() \\ L.addr \leftarrow newtemp() \\ gen(t\ ':='\ E.addr\ '*'\ L.type.width) \\ gen(L.addr\ ':='\ L_1.addr\ '+'\ t) \end{array} \right\}$$

$$E \rightarrow L \quad \left\{ \begin{array}{l} E.addr \leftarrow newtemp() \\ gen(E.addr\ ':='\ L.array.base\ '['\ L.addr\ ']') \end{array} \right\}$$

- $L.type = array(3, \textbf{int})$ con anchura 12
  $L.addr = t_1,\ E.addr = \textbf{i}$
- $L.type = \textbf{int}$ con anchura 4
  $L.addr = t_3,\ E.addr = \textbf{j}$
- $L.array.base = \textbf{foo},\ L.addr = t_3,\ E.addr = t_4$
- $L.addr = t_4,\ E.addr = \textbf{bar}$

```
t1  := i * 12
t2  := j * 4
t3  := t1 + t2
t4  := foo [ t3 ]
bar := t4
```

Estructuras de control

- La generación de código para estructuras de control está influenciada por la generación de código para expresiones booleanas.
- Vamos a suponer que el problema de generación de código para una expresión boolean B está resuelto y cumple
  - Los atributos $B.true$ y $B.false$ son etiquetas.
  - El código de B salta a $B.true$ o $B.false$ según el resultado de la expresión booleana.
- El atributo heredado $S.next$ corresponde a la etiqueta de la *siguiente* instrucción después del código de S.
  - Esto va a generar muchas etiquetas – pero son baratas.
  - Permite evitar saltos a saltos.
- `newlabel()` – cada llamada retorna un nombre de etiqueta único.
- `label()` – genera código para incorporar la etiqueta.

Instrucciones Simples

Esquema de generación

$$P \rightarrow S$$

$$S \rightarrow S_1 \; S_2$$

Instrucciones Simples

Esquema de generación

$$P \rightarrow S \qquad \left\{ \begin{array}{l} S.next \leftarrow newlabel() \\ P.code \leftarrow S.code \mathbin{+\!+} \\ \qquad\qquad label(S.next) \end{array} \right\}$$

$$S \rightarrow S_1 \ S_2$$

- El programa principal es el caso base para $S.next$.

UNIVERSIDAD SIMÓN BOLÍVAR

Instrucciones Simples

Esquema de generación

$$P \rightarrow S \qquad \left\{ \begin{array}{l} S.next \leftarrow newlabel() \\ P.code \leftarrow S.code \; ++ \\ \qquad\qquad label(S.next) \end{array} \right\}$$

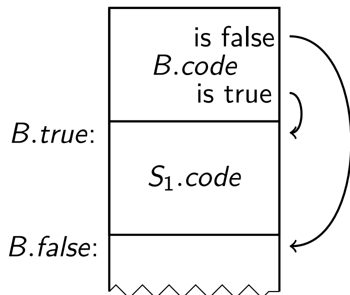
$$S \rightarrow S_1 \; S_2 \qquad \left\{ \begin{array}{l} S_1.next \leftarrow newlabel() \\ S_2.next \leftarrow S.next \\ S.code \leftarrow S_1.code \; ++ \\ \qquad\qquad label(S_1.next) \\ \qquad\qquad S_2.code \end{array} \right\}$$

- El programa principal es el caso base para $S.next$.
- Al secuenciar instrucciones:
  - La siguiente a la secuencia es natural – piensen “FOLLOW”.
  - Hace falta definir la siguiente para el punto medio.

UNIVERSIDAD SIMÓN BOLÍVAR

Selectores

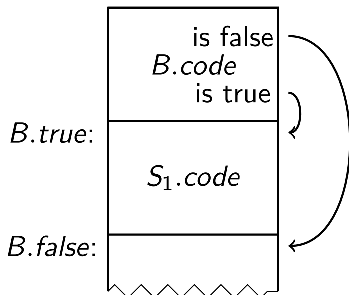
$S \rightarrow$ **if** B **then** S_1



UNIVERSIDAD SIMÓN BOLÍVAR

Selectores

$S \rightarrow$ **if** B **then** S_1



¡$B.false$ corresponde a $S.next$!

Selectores

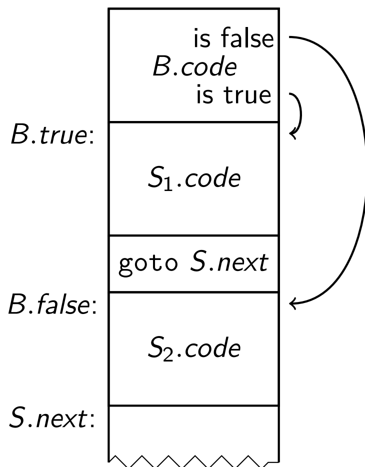
Esquema para el `if-then`

$$S \rightarrow \textbf{if } B \textbf{ then } S_1 \quad \left\{ \begin{array}{l} B.true \leftarrow newlabel() \\ B.false \leftarrow S.next \\ S_1.next \leftarrow S.next \\ S.code \leftarrow B.code \mathbin{++} \\ \qquad\qquad label(B.true) \mathbin{++} \\ \qquad\qquad S_1.code \end{array} \right\}$$

- Necesitamos una nueva etiqueta para el cuerpo.
- Encadenamos $S.next$ dos veces
  - Para el salto de salida cuando B es falso.
  - Como siguiente de la instrucción como un todo.

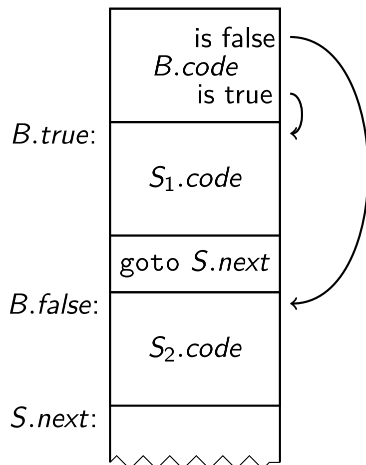
Selectores

$S \to$ **if** B **then** S_1 **else** S_2



Selectores

$S \rightarrow$ **if** B **then** S_1 **else** S_2



¡$S_1.next$ y $S_2.next$ deben coincidir con $S.next$!

Selectores

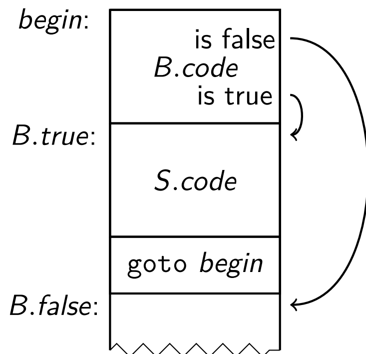
Esquema para el `if-then-else`

$$S \rightarrow \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \quad \left\{ \begin{array}{l} B.true \leftarrow newlabel() \\ B.false \leftarrow newlabel() \\ S_1.next \leftarrow S.next \\ S_2.next \leftarrow S.next \\ S.code \leftarrow B.code \mathbin{++} \\ \qquad label(B.true) \mathbin{++} \\ \qquad S_1.code \mathbin{++} \\ \qquad gen('\text{goto}'\ S.next) \mathbin{++} \\ \qquad label(B.false) \mathbin{++} \\ \qquad S_2.code \end{array} \right\}$$

- Etiquetas dedicadas para los brazos del `if-then-else`.
- Salidas de los brazos coinciden con salida del bloque – instrucciones anidadas en los brazos saltan directo a la salida.

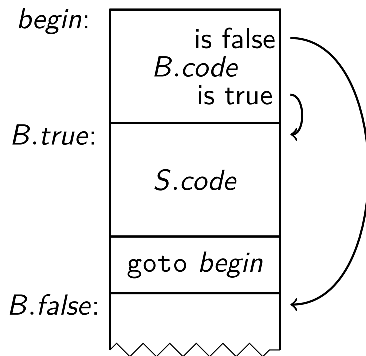
Iteración

$S \rightarrow$ `while` S_1



Iteración

$S \rightarrow$ `while` S_1



¡$B.false$ corresponde a $S.next$!

Iteración

Esquema para el `while`

$$S \rightarrow \textbf{while } B \textbf{ do } S_1 \qquad \left\{ \begin{array}{ll} begin & \leftarrow newlabel() \\ B.true & \leftarrow newlabel() \\ B.false & \leftarrow S.next \\ S_1.next & \leftarrow S.begin \\ S.code & \leftarrow label(begin) \mathbin{++} \\ & \quad B.code \mathbin{++} \\ & \quad label(B.true) \mathbin{++} \\ & \quad S_1.code \mathbin{++} \\ & \quad gen('\text{goto}'\ begin) \end{array} \right\}$$

- Necesitamos una nueva etiqueta para el cuerpo.
- Al ejecutar el cuerpo se regresa a evaluar la condición.
  - Se ejecutó todo hasta alcanzar el `goto` explícito.
  - Hay un salto en el bloque anidado que aprovecha $S_1.next$.

UNIVERSIDAD SIMÓN BOLÍVAR

Expresiones Booleanas para Control de Flujo

Jumping Code

- Las expresiones booleanas se interpretan según su contexto
  - Para alterar el flujo de control.
  - Como valor de una expresión booleana.

Expresiones Booleanas para Control de Flujo

Jumping Code

- Las expresiones booleanas se interpretan según su contexto
  - Para alterar el flujo de control.
  - Como valor de una expresión booleana.
- Si la definición del lenguaje obliga a evaluar todas las partes de una expresión booleana, la generación de código es similar a la de expresiones.
  - Codificar `true` y `false` como 1 y 0, respectivamente.
  - Generar código para calcular el resultado en un temporal.

Expresiones Booleanas para Control de Flujo

Jumping Code

- Las expresiones booleanas se interpretan según su contexto
  - Para alterar el flujo de control.
  - Como valor de una expresión booleana.
- Si la definición del lenguaje obliga a evaluar todas las partes de una expresión booleana, la generación de código es similar a la de expresiones.
  - Codificar `true` y `false` como `1` y `0`, respectivamente.
  - Generar código para calcular el resultado en un temporal.
- Si la definición del lenguaje permite evaluar parcialmente la expresión, se genera código de **corto circuito** o *Jumping Code*.
  - Los operadores booleanos se traducen a saltos.
  - La posición en el código representa el valor parcial o definitivo de la expresión booleana.

Expresiones Booleanas para Control de Flujo

Jumping Code – casos base

$B \rightarrow$ **true**
$B \rightarrow$ **false**

$B \rightarrow$ **not** B_1

$B \rightarrow E_1 < E_2$

Expresiones Booleanas para Control de Flujo

Jumping Code – casos base

$B \rightarrow$ **true** $\quad \{B.code \leftarrow gen('\text{goto}'\ B.true)\}$
$B \rightarrow$ **false** $\quad \{B.code \leftarrow gen('\text{goto}'\ B.false)\}$

$B \rightarrow$ **not** B_1

$B \rightarrow E_1 < E_2$

- $B.true$ y $B.false$ contienen la etiqueta destino – Just do it!

Expresiones Booleanas para Control de Flujo

Jumping Code – casos base

$$
\begin{array}{ll}
B \rightarrow \textbf{true} & \{B.code \leftarrow gen('\text{goto}'\ B.true)\} \\
B \rightarrow \textbf{false} & \{B.code \leftarrow gen('\text{goto}'\ B.false)\} \\
B \rightarrow \textbf{not}\ B_1 & \left\{\begin{array}{l} B_1.true \leftarrow B.false \\ B_1.false \leftarrow B.true \\ B.code \leftarrow B_1.code \end{array}\right\}
\end{array}
$$

$$B \rightarrow E_1 < E_2$$

- $B.true$ y $B.false$ contienen la etiqueta destino – Just do it!
- La negación mantiene el código, pero invierte el sentido de saltos.

Expresiones Booleanas para Control de Flujo

Jumping Code – casos base

$$
\begin{array}{ll}
B \rightarrow \textbf{true} & \{B.code \leftarrow gen('\text{goto}'\ B.true)\} \\
B \rightarrow \textbf{false} & \{B.code \leftarrow gen('\text{goto}'\ B.false)\} \\
B \rightarrow \textbf{not}\ B_1 & \left\{\begin{array}{l} B_1.true \leftarrow B.false \\ B_1.false \leftarrow B.true \\ B.code \leftarrow B_1.code \end{array}\right\} \\
B \rightarrow E_1 < E_2 & \left\{\begin{array}{ll} B.code \leftarrow & E_1.code \mathbin{++} \\ & E_2.code \mathbin{++} \\ & gen('\text{if}'\ E_1.addr\ '<'\ E_2.addr\ '\text{goto}'\ B.true) \mathbin{++} \\ & gen('\text{goto}'\ B.false) \end{array}\right\}
\end{array}
$$

- $B.true$ y $B.false$ contienen la etiqueta destino – Just do it!
- La negación mantiene el código, pero invierte el sentido de saltos.
- Cada operador relacional genera los saltos a las etiquetas correspondientes.

Expresiones Booleanas para Control de Flujo

Jumping Code – disyunción

$$B \rightarrow B_1 \textbf{ or } B_2 \quad \left\{ \begin{array}{l} B_1.true \leftarrow B.true \\ B_1.false \leftarrow newlabel() \\ B_2.true \leftarrow B.true \\ B_2.false \leftarrow B.false \\ B.code \leftarrow B_1.code +\!+ \\ \qquad\qquad label(B_1.false) +\!+ \\ \qquad\qquad B_2.code \end{array} \right\}$$

- Si B_1 es cierta, seguimos directo a $B.true$ – cortocircuito.
- Si B_1 es falsa, evaluamos B_2 saltando a la nueva etiqueta.
- El resto de salidas para B_1 y B_2 se mantienen a partir de B.

Expresiones Booleanas para Control de Flujo

Jumping Code – conjunción

$$B \rightarrow B_1 \textbf{ and } B_2 \quad \left\{ \begin{array}{ll} B_1.true & \leftarrow newlabel() \\ B_1.false & \leftarrow B.false \\ B_2.true & \leftarrow B.true \\ B_2.false & \leftarrow B.false \\ B.code & \leftarrow B_1.code \mathbin{++} \\ & \quad label(B_1.true) \mathbin{++} \\ & \quad B_2.code \end{array} \right\}$$

- Si B_1 es falsa, seguimos directo a $B.false$ – cortocircuito.
- Si B_1 es cierta, evaluamos B_2 saltando a la nueva etiqueta.
- El resto de salidas para B_1 y B_2 se mantienen a partir de B.

Un ejemplo

...que conduce a una mejora

```
if (x < 100 or x > 200 and x != y ) x = 0;
```

El esquema generaría código similar a

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x := 0
L1:
```

- `goto L3` es redundante
- Ambos `goto L1` podrían eliminarse si se contara con `ifnot` en el TAC.

UNIVERSIDAD SIMÓN BOLÍVAR

Mejorando el código para el `if`

Fall Through Optimization

$$S \rightarrow \textbf{if } B \textbf{ then } S_1 \quad \left\{ \begin{array}{l} B.true \leftarrow fall \\ B.false \leftarrow S.next \\ S_1.next \leftarrow S.next \\ S.code \leftarrow B.code \mathbin{++} \\ \qquad\qquad S_1.code \end{array} \right\}$$

- El código de S_1 siempre está después del código de B.
  - No generar una etiqueta en medio.
  - La etiqueta `fall` indica "dejarse caer" a la siguiente instrucción.
- Se aplica la misma técnica para el **while**.

Mejorando el código para expresiones booleanas

Fall Through Optimization para operadores relacionales

$$B \rightarrow E_1 < E_2 \quad \left\{ \begin{array}{l} test \leftarrow E_1.addr \mathbin{++} '<' \mathbin{++} E_2.addr \\ \textbf{if } B.true \neq fall \textbf{ and } B.false \neq fall \textbf{ then} \\ \ s \leftarrow gen('\text{if}'\ test'\text{goto}'\ B.true) \mathbin{++} gen('\text{goto}'\ B.false) \\ \textbf{else if } B.true \neq fall \textbf{ then} \\ \ s \leftarrow gen('\text{if}'\ test\ '\text{goto}'\ B.true) \\ \textbf{else if } B.false \neq fall \textbf{ then} \\ \ s \leftarrow gen('\text{ifnot}'\ test\ '\text{goto}'\ B.false) \\ \textbf{else} \\ \ s \leftarrow '' \\ B.code \leftarrow E_1.code \mathbin{++} E_2.code \mathbin{++} s \end{array} \right\}$$

Ahora genera la cantidad mínima de saltos.

UNIVERSIDAD SIMÓN BOLÍVAR

Mejorando el código para expresiones booleanas

Fall Through Optimization para disyunción

$$B \rightarrow B_1 \textbf{ or } B_2 \quad \left\{ \begin{array}{l} B_1.\mathit{true} \leftarrow \textbf{if } B.\mathit{true} \neq \mathit{fall} \textbf{ then } B.\mathit{true} \textbf{ else } \mathit{newlabel}() \\ B_1.\mathit{false} \leftarrow \mathit{fall} \\ B_2.\mathit{true} \leftarrow B.\mathit{true} \\ B_2.\mathit{false} \leftarrow B.\mathit{false} \\ B.\mathit{code} \leftarrow \textbf{if } B.\mathit{true} \neq \mathit{fall} \textbf{ then} \\ \qquad\qquad B_1.\mathit{code} \mathbin{++} B_2.\mathit{code} \\ \qquad\quad \textbf{else} \\ \qquad\qquad B_1.\mathit{code} \mathbin{++} B_2.\mathit{code} \mathbin{++} \mathit{label}(B_1.\mathit{true}) \end{array} \right\}$$

Ahora genera la cantidad mínima de saltos.

El ejemplo, después de la optimización

```
if (x < 100 or x > 200 and x != y ) x = 0;
```

El esquema optimizado generaría código similar a

```
    if x < 100 goto L2
    ifnot x > 200 goto L1
    ifnot x != y goto L1
L2: x := 0
L1:
```

- No hay etiquetas redundantes
- Menos saltos.

Evaluando Expresiones Booleanas

Jumping Code hasta el resultado

- *Jumping Code* sigue aplicando para evaluar expresiones booleanas para obtener su valor.
  - Se salta por cortocircuito hasta determinar el valor final.
  - La instrucción final produce el valor booleano correspondiente.
  - El valor booleano puede ser natural o codificado según convenga al lenguaje y al TAC.
- Este método es el más eficiente si la semántica del lenguaje permite evaluar con corto circuito.

Expresiones Booleanas por su valor

Cálculo del valor final

$$S \rightarrow \textbf{id} := B \quad \left\{ \begin{array}{l} B.true \leftarrow newlabel() \\ B.false \leftarrow newlabel() \\ S.code \leftarrow B.code \mathbin{++} \\ \qquad label(B.true) \mathbin{++} \\ \qquad gen(\textbf{id}.addr \; ':=' \; \textbf{true}) \mathbin{++} \\ \qquad gen('\mathrm{goto}' \; S.next) \mathbin{++} \\ \qquad label(B.false) \mathbin{++} \\ \qquad gen(\textbf{id}.addr \; ':=' \; \textbf{false}) \end{array} \right\}$$

Consideraciones

- La generación para instrucciones utiliza la técnica de concatenación de cadenas. Reescríbala para hacer generación progresiva.
- La generación para instrucciones utiliza atributos heredados.
  - Intente escribirla para utilizar atributos sintetizados.
  - Aplique la técnica de marcadores y posible reescritura de reglas para convertir la herencia en copia a través de la pila de un reconocedor LR.
- Extienda el esquema de generación por corto circuito para incluir implicación, disyunción exclusiva y equivalencia.
- Complete el esquema de generación por corto circuito aplicando la técnica de *fall through* para la conjunción y los casos agregados anteriormente.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*] (Primera Edición)
  - Secciones 8.3 y 8.4
  - Ejercicios 8.7 a 8.10
- [*Aho*] (Segunda Edición)
  - Secciones 6.4.3, 6.4.4
  - Ejercicios 6.4.1 a 6.4.9

UNIVERSIDAD SIMÓN BOLÍVAR