Compiladores: Generación de Código Intermedio

Pontificia Universidad Javeriana Cali
Ingenieria de Sistemas y Computación
Prof. Gloria Inés Alvarez V.

Expresiones Booleanas

- Son usadas para:
  - Calcular valores lógicos
  - Como expresiones condicionales en instrucciones que alteran el control de flujo.
- Representación del Valor de una Expresión Booleana:
  - Codificar verdadero y falso numéricamente (V =y F=; o V<>0 y F=0; o V>=0 y F<0) , y evaluar las expresiones booleanas de manera similar a las aritméticas.
  - Implementar las expresiones booleanas por control de flujo: permite optimizar la evaluación de expresiones booleanas, calculando solamente lo necesario para determinar su valor. Esto depende de la semántica del lenguaje (ej. Que sucede si una parte de la expresión ejecuta una función que cambia el valor de una variable global?)

Representación Numérica

- El código de tres direcciones soporta los operadores and, or, not.
- La expresion x > y and z < f se traduce:

```
        If x > y goto lab1
        t1 := 0
        goto lab2
lab1: t1 := 1
lab2: if z < f goto lab3
        t2 := 0
        goto lab4
lab3: t2 := 1
lab4: t3 := t1 and t2
```

Short-circuit code

- No usa los operadores and, or, not en el codigo de tres direcciones
- Util en el contexto de instrucciones de control de flujo
- La expresion if x > y and z < f then <s1> se traduce:

```
        If x > y goto lab1
        goto lab2
lab1: if z < f goto lab3
       goto lab2
lab3: <s1>
lab2:
```

Short-circuit code

- Para la traducción dirigida por sintaxis:
  - Las expresiones booleanas heredan dos etiquetas como atributo: E.true y E.false, que corresponden a las etiquetas a las que deben saltar cuando la expresión es verdadera o falsa respectivamente.
  - Las listas de instrucciones heredan el atributo S.next, que es la etiqueta que corresponde a la siguiente instrucción.

Reglas Semánticas [Aho p.402]

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} (B) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow \textbf{if} (B) S_1 \textbf{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow \textbf{while} (B) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Exp. booleanas [Aho p. 404]

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow\, !\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto' $B.true)$ $\mid\mid gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'goto' $B.false)$

Ejemplo short-circuit [Aho p.405]

Example 6.22: Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;        (6.13)
```

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
      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:
```

Backpatching

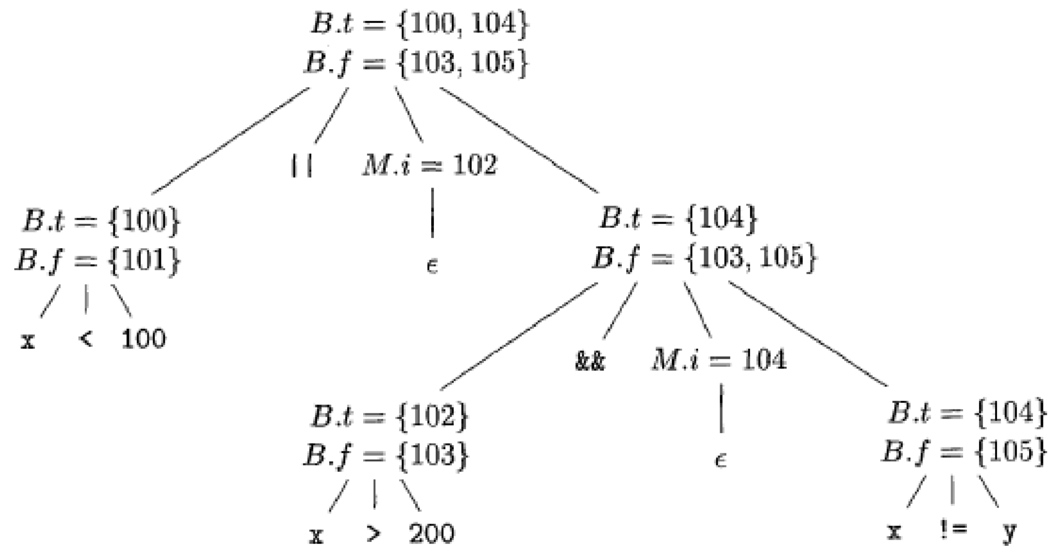
- Se usan dos fases para implementar las traducciones dirigidas por sintaxis que generan código para expresiones booleanas e instrucciones de control de flujo:
  - 1a. Fase: Calcula las traducciones de la expresion dada. En esta fase no se conocen las etiquetas que controlan los saltos que son generados, entonces, se generan las instrucciones de salto dejando las etiquetas sin especificar, y se coloca la instrucción e una lista de instrucciones de salto por especificar.
  - 2a. Fase: Backpatching: cuando se determinan las etiquetas, se llenan las instrucciones de salto
- Las dos fases se pueden ejecutar en una sola pasada.

Acciones semánticas backpatching [Aho p. 411]

The translation scheme is in Fig. 6.43.

1)	$B \to B_1$ || $M\ B_2$	$\{$ $backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;$ $\}$
2)	$B \to B_1$ && $M\ B_2$	$\{$ $backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);$ $\}$
3)	$B \to\ !\ B_1$	$\{$ $B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;$ $\}$
4)	$B \to (\ B_1\ )$	$\{$ $B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist;$ $\}$
5)	$B \to E_1$ **rel** E_2	$\{$ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $emit($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto _'$);$ $emit($'goto _'$);$ $\}$
6)	$B \to$ **true**	$\{$ $B.truelist = makelist(nextinstr);$ $emit($'goto _'$);$ $\}$
7)	$B \to$ **false**	$\{$ $B.falselist = makelist(nextinstr);$ $emit($'goto _'$);$ $\}$
8)	$M \to \epsilon$	$\{$ $M.instr = nextinstr;$ $\}$

Ejemplo backpatching [Aho p.412]



Backpatching [Aho p. 415]

1) $S \to \textbf{if}\,(\,B\,)\,M\,S_1$ { $backpatch(B.truelist,\ M.instr);$
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist);$ }

2) $S \to \textbf{if}\,(\,B\,)\,M_1\,S_1\,N\,\textbf{else}\,M_2\,S_2$
{ $backpatch(B.truelist,\ M_1.instr);$
$backpatch(B.falselist,\ M_2.instr);$
$temp\ =\ merge(S_1.nextlist,\ N.nextlist);$
$S.nextlist\ =\ merge(temp,\ S_2.nextlist);$ }

3) $S \to \textbf{while}\,M_1\,(\,B\,)\,M_2\,S_1$
{ $backpatch(S_1.nextlist,\ M_1.instr);$
$backpatch(B.truelist,\ M_2.instr);$
$S.nextlist\ =\ B.falselist;$
$emit($'goto' $M_1.instr);$ }

4) $S \to \{\,L\,\}$ { $S.nextlist\ =\ L.nextlist;$ }

5) $S \to A\ ;$ { $S.nextlist\ =\ \textbf{null};$ }

6) $M \to \epsilon$ { $M.instr\ =\ nextinstr;$ }

7) $N \to \epsilon$ { $N.nextlist\ =\ makelist(nextinstr);$
$emit($'goto _'$);$ }

8) $L \to L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr);$
$L.nextlist\ =\ S.nextlist;$ }

9) $L \to S$ { $L.nextlist\ =\ S.nextlist;$ }

Reusar Variables Temporales

- La generación de variables temporales que contienen valores intermedios requiere que se localice espacio de memoria (en tiempo de ejecución) para mantener estos valores, es entonces importante reusar las variables

- Una forma es crear una funcion que administre las temporales usando una pila

Direccionamiento de elementos de arreglos

- Si los elementos del arreglo son almacenados en posiciones de memoria consecutivas, pueden ser accesados rapidamente.
- Si cada elemento del arreglo ocupa w bytes, entonces, el i-esimo elemento del arreglo A inicia en la posición:

  Base + (i – low) * w

  Donde low es el limite inferior del arreglo

  Base es la dirección relativa asignada al arreglo (es la dirección de A[low])
- Si se expresa como i*w + (Base – low*w), parte de la expresión puede ser evaluada en tiempo de compilación

Arreglos Multidimensionales

- Generalizando la expresión para un arreglo A de k dimensiones:
  - La posición del elemento A[i1, i2, i3, ...,ik] esta dada por:

    $((...((i_1 * n_2 + i_2) * n_3 + i_3)...) * n_k + i_k) * w$

    $+ base - ((... ((low_1 * n_2 + low_2) * n_3 + low_3)...) * n_k + low_k) * w$

    Donde, $n_j = high_j - low_j$

Acciones para arreglos [Aho p.383]

Producción	Acción
$S \rightarrow \textbf{id} = E\ ;$	$\{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \}$
$\quad \mid\ L = E\ ;$	$\{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$
$E \rightarrow E_1 + E_2$	$\{\ E.addr = \textbf{new}\ Temp\,();$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$
$\quad \mid\ \textbf{id}$	$\{\ E.addr = top.get(\textbf{id}.lexeme);\ \}$
$\quad \mid\ L$	$\{\ E.addr = \textbf{new}\ Temp\,();$ $gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}$
$L \rightarrow \textbf{id}\ [\ E\ ]$	$\{\ L.array = top.get(\textbf{id}.lexeme);$ $L.type = L.array.type.elem;$ $L.addr = \textbf{new}\ Temp\,();$ $gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \}$
$\quad \mid\ L_1\ [\ E\ ]$	$\{\ L.array = L_1.array;$ $L.type = L_1.type.elem;$ $t = \textbf{new}\ Temp\,();$ $L.addr = \textbf{new}\ Temp\,();$ $gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \}$ $gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}$

Ejemplo arreglos [Aho p. 385]

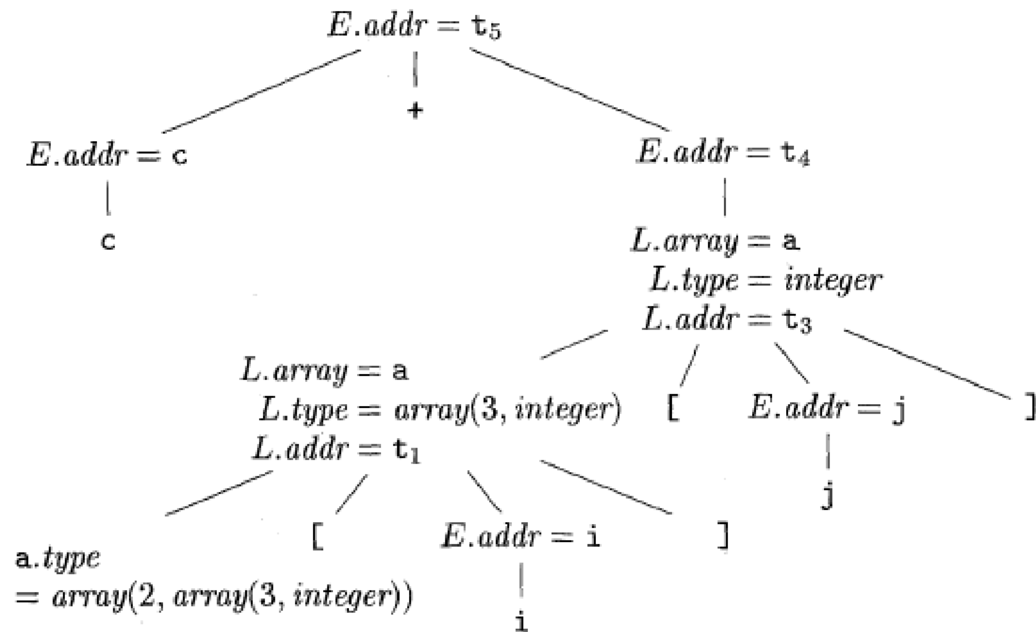


Figure 6.23: Annotated parse tree for c + a[i][j]

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```