Ejercicio 6.5.2: Como en Ada, suponga que cada expresión debe tener un tipo único, pero que a partir de esa expresión por sí sola, todo lo que podemos deducir es un conjunto de posibles tipos. Es decir, la aplicación de la función E_1 al argumento E_2, representada por $E \rightarrow E_1 (E_2)$, tiene la siguiente regla asociada:

$$E.tipo = \{\ t \mid \text{para alguna } s \text{ en } E_2.tipo,\ s \rightarrow t \text{ está en } E_1.tipo\ \}$$

Describa una definiciones dirigidas por la sintaxis que determine un tipo único para cada subexpresión, usando un atributo *tipo* para sintetizar un conjunto de posibles tipos de abajo hacia arriba y, una vez que se determine el tipo único de la expresión en general, proceda de arriba hacia abajo para determinar el atributo *único* para el tipo de cada subexpresión.

6.6 Flujo de control

La traducción de instrucciones como if-else y while está enlazada a la traducción de expresiones booleanas. En los lenguajes de programación, las expresiones booleanas se utilizan con frecuencia para:

1. *Alterar el flujo de control.* Las expresiones booleanas se utilizan como expresiones condicionales en las instrucciones que alteran el flujo de control. El valor de dichas expresiones booleanas es implícito en una posición a la que se llega en un programa. Por ejemplo, en **if** (E) S, la expresión E debe ser verdadera si se llega a la instrucción S.

2. *Calcular los valores lógicos.* Una expresión booleana puede representar a *true* o *false* como valores. Dichas expresiones booleanas pueden evaluarse en analogía con las expresiones aritméticas, mediante las instrucciones de tres direcciones con los operadores lógicos.

El uso que se pretende de las expresiones booleanas se determina mediante su contexto sintáctico. Por ejemplo, una expresión que va después de la palabra clave **if** se utiliza para alterar el flujo de control, mientras que una expresión del lado derecho de una asignación se utiliza para denotar un valor lógico. Dichos contextos sintácticos pueden especificarse en una variedad de formas: podemos usar dos no terminales distintos, usar los atributos heredados o establecer una bandera durante el análisis sintáctico. De manera alternativa, podemos construir un árbol sintáctico e invocar distintos procedimientos para los dos usos distintos de las expresiones booleanas.

Esta sección se concentra en el uso de las expresiones booleanas para alterar el flujo de control. Por claridad, presentaremos una nueva no terminal B para este fin. En la sección 6.6.6 consideraremos la forma en que un compilador puede permitir que las expresiones booleanas representen valores lógicos.

6.6.1 Expresiones booleanas

Las expresiones booleanas están compuestas de los operadores booleanos (que denotamos como &&, || y !, usando la convención de C para los operadores AND, OR y NOT, respectivamente) que se aplican a elementos que son variables booleanas o expresiones relacionales. Estas expresiones relacionales son de la forma E_1 **rel** E_2, en donde E_1 y E_2 son expresiones

aritméticas. En esta sección, consideraremos las expresiones booleanas generadas por la siguiente gramática:

$$B \rightarrow B \texttt{ || } B \mid B \texttt{ \&\& } B \mid \texttt{!}B \mid \texttt{(}\ B\ \texttt{)} \mid E\ \textbf{rel}\ E \mid \textbf{true} \mid \textbf{false}$$

Utilizamos el atributo **rel**.*op* para indicar cuál de los seis operadores de comparación $<$, $<=$, $=$, $!=$, $>$ o $>=$ se representa mediante **rel**. Como es costumbre, suponemos que `||` y && son asociativos por la izquierda, y que `||` tiene la menor precedencia, después && y después !.

Dada la expresión B_1 `||` B_2, si determinamos que B_1 es verdadera, entonces podemos concluir que toda la expresión es verdadera sin tener que evaluar B_2. De manera similar, dada la expresión B_1&&B_2, si B_1 es falsa, entonces toda la expresión es falsa.

La definición semántica del lenguaje de programación determina si deben evaluarse todas las partes de una expresión booleana. Si la definición de lenguaje permite (o requiere) que se queden sin evaluar partes de una expresión booleana, entonces el compilador puede optimizar la evaluación de expresiones booleanas calculando sólo lo suficiente de una expresión como para poder determinar su valor. Por ende, en una expresión como B_1 `||` B_2, no necesariamente se evalúan por completo B_1 o B_2. Si B_1 o B_2 es una expresión con efectos adicionales (por ejemplo, si contiene una función que modifique a una variable global), entonces puede obtenerse una respuesta inesperada.

6.6.2 Código de corto circuito

En el código de *corto circuito* (o de *salto*), los operadores booleanos &&, `||` y ! se traducen en saltos. Los mismos operadores no aparecen en el código; en vez de ello, el valor de una expresión booleana se representa mediante una posición en la secuencia de código.

Ejemplo 6.21: La instrucción

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

podría traducirse en el código de la figura 6.34. En esta traducción, la expresión booleana es verdadera si el control llega a la etiqueta L_2. Si la expresión es falsa, el control pasa de inmediato a L_1, ignorando a L_2 y la asignación `x = 0`. □

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

Figura 6.34: Código de salto

6.6.3 Instrucciones de flujo de control

Ahora consideraremos la traducción de expresiones booleanas en código de tres direcciones, en el contexto de instrucciones como las que se generan mediante la siguiente gramática:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if} \; (\; B \;) \; S_1 \\ S & \rightarrow & \textbf{if} \; (\; B \;) \; S_1 \; \textbf{else} \; S_2 \\ S & \rightarrow & \textbf{while} \; (\; B \;) \; S_1 \end{array}$$

En estas producciones, el no terminal B representa a una expresión booleana y en la no terminal S representa a una instrucción.

Esta gramática generaliza el ejemplo abierto de las expresiones while que presentamos en el ejemplo 5.19. Como en ese ejemplo, tanto B como S tienen un atributo sintetizado llamado *codigo*, el cual proporciona la traducción en instrucciones de tres direcciones. Por simplicidad, generamos las traducciones *B.codigo* y *S.codigo* como cadenas, usando definiciones dirigidas por la sintaxis. Las reglas semánticas que definen los atributos *codigo* podrían implementarse en vez de generar árboles sintácticos y después emitir código durante el recorrido de un árbol, o mediante cualquiera de los métodos descritos en la sección 5.5.

La traducción de **if** (B) S_1 consiste en *B.codigo* seguida de S_1*.codigo*, como se muestra en la figura 6.35(a). Dentro de *B.codigo* hay saltos con base en el valor de B. Si B es verdadera, el control fluye hacia la primera instrucción de S_1*.codigo*, y si B es falsa, el control fluye a la instrucción que sigue justo después de S_1*.codigo*.

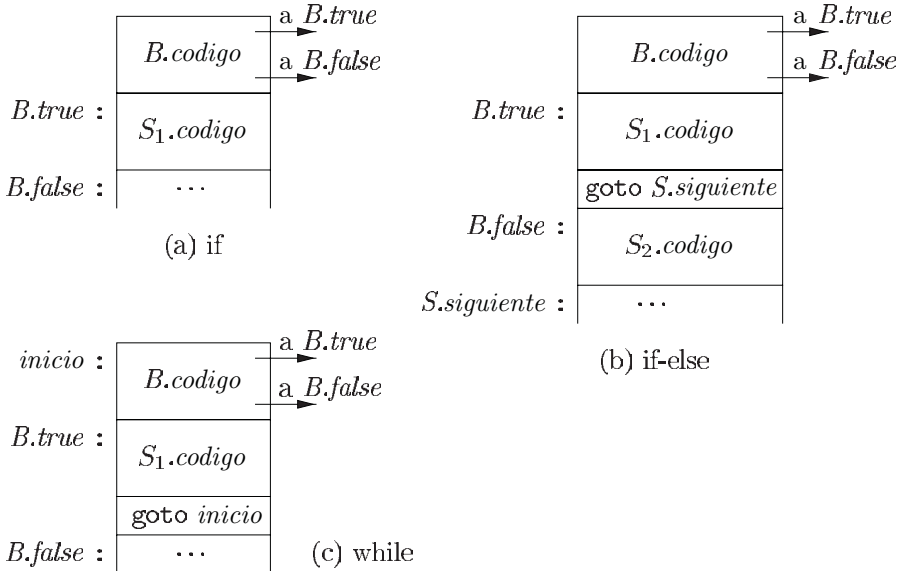

Figura 6.35: Código para las instrucciones if, if-else y while

Las etiquetas para los saltos en *B.codigo* y *S.codigo* se administran usando atributos heredados. Con una expresión booleana B, asociamos dos etiquetas: *B.true*, la etiqueta hacia la

cual fluye el control si B es verdadera, y $B.false$, la etiqueta hacia la cual fluye el control si B es falsa. Con una instrucción S, asociamos un atributo heredado $S.siguiente$ que denota a una etiqueta para la instrucción que sigue justo después del código para S. En algunos casos, la instrucción que va justo después de $S.codigo$ es un salto hacia alguna etiqueta L. Un salto hacia un salto a L desde el interior de $S.codigo$ se evita mediante el uso de $S.siguiente$.

La definición dirigida por la sintaxis en la figura 6.36-6.37 produce código de tres direcciones para las expresiones booleanas en el contexto de instrucciones if, if-else y while.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$P \rightarrow S$	$S.siguiente = nuevaetiqueta()$ $P.codigo = S.codigo \mid\mid etiqueta(S.siguiente)$
$S \rightarrow$ **assign**	$S.codigo =$ **assign**$.codigo$
$S \rightarrow$ **if** (B) S_1	$B.true = nuevaetiqueta()$ $B.false = S_1.siguiente = S.siguiente$ $S.codigo = B.codigo \mid\mid etiqueta(B.true) \mid\mid S_1.codigo$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = nuevaetiqueta()$ $B.false = nuevaetiqueta()$ $S_1.siguiente = S_2.siguiente = S.siguiente$ $S.codigo = B.codigo$ $\mid\mid etiqueta(B.true) \mid\mid S_1.codigo$ $\mid\mid gen('\texttt{goto}'\ S.siguiente)$ $\mid\mid etiqueta(B.false) \mid\mid S_2.codigo$
$S \rightarrow$ **while** (B) S_1	$inicio = nuevaetiqueta()$ $B.true = nuevaetiqueta()$ $B.false = S.siguiente$ $S_1.siguiente = inicio$ $S.codigo = etiqueta(inicio) \mid\mid B.codigo$ $\mid\mid etiqueta(B.true) \mid\mid S_1.codigo$ $\mid\mid gen('\texttt{goto}'\ inicio)$
$S \rightarrow S_1\ S_2$	$S_1.siguiente = nuevaetiqueta()$ $S_2.siguiente = S.siguiente$ $S.codigo = S_1.codigo \mid\mid etiqueta(S_1.siguiente) \mid\mid S_2.codigo$

Figura 6.36: Definición dirigida por la sintaxis para las instrucciones de flujo de control

Suponemos que *nuevaEtiqueta*() crea una nueva etiqueta cada vez que se llama, y que *etiqueta*(L) adjunta la etiqueta L a la siguiente instrucción de tres direcciones que se va a generar.[8]

[8] Si se implementan en sentido literal, las reglas semánticas generarán muchas etiquetas y pueden adjuntar más de una etiqueta a una instrucción de tres direcciones. El método de "parcheo de retroceso" de la sección 6.7 crea etiquetas

Un programa consiste en una instrucción generada por $P \rightarrow S$. Las reglas semánticas asociadas con esta producción inicializan *S.siguiente* con una nueva etiqueta. *P.codigo* consiste en *S.codigo* seguido de la nueva etiqueta *S.siguiente*. El token **asigna** en la producción $S \rightarrow$ ***asigna*** es un receptor para las instrucciones de asignación. La traducción de las asignaciones es como se describió en la sección 6.4; para esta explicación del flujo de control, *S.codigo* es simplemente **asigna**.*codigo*.

Al traducir $S \rightarrow$ **if** (B) S_1, las reglas semánticas en la figura 6.36 crean una nueva etiqueta *B.true* y la adjuntan a la primera instrucción de tres direcciones generada para la instrucción S_1, como se muestra en la figura 6.35(a). Por ende, los saltos a *B.true* dentro del código para B irán al código para S_1. Además, al establecer *B.false* a *S.siguiente*, aseguramos que el control ignore el código para S_1, si B se evalúa como falsa.

Al traducir la instrucción if-else $S \rightarrow$ **if** (B) S_1 **else** S_2, el código para la expresión booleana B tiene saltos hacia fuera de ésta, que van a la primera instrucción del código para S_1 si B es verdadera, y a la primera instrucción del código para S_2 si B es falsa, como se ilustra en la figura 6.35(b). Además, el control fluye desde S_1 y S_2 hacia la instrucción de tres direcciones que va justo después del código para S; su etiqueta se proporciona mediante el atributo heredado *S.siguiente*. Una instrucción `goto` *S.siguiente* explícita aparece después del código de S_1 para ignorar el código de S_2. No se necesita un goto después de S_2, ya que S_2.*siguiente* es igual que *S.siguiente*.

El código para $S \rightarrow$ **while** (B) S_1 se forma a partir de *B.codigo* y S_1.*codigo* como se muestra en la figura 6.35(c). Utilizamos una variable local *inicio* para guardar una nueva etiqueta que se adjunta a la primera instrucción para esta instrucción while, que también es la primera instrucción para B. Usamos una variable en vez de un atributo, ya que *inicio* es local para las reglas semánticas de esta producción. La etiqueta heredada *S.siguiente* marca la instrucción hacia la cual debe fluir el control si B es falsa; por ende, *B.falsa* se establece para ser *S.siguiente*. Una nueva etiqueta *B.true* se adjunta a la primera instrucción para S_1; el código para B genera un salto hacia esta etiqueta si B es verdadera. Después del código para S_1, colocamos la instrucción `goto` *inicio*, la cual produce un salto de regreso al principio del código para la expresión booleana. Observe que S_1.*siguiente* se establece a esta etiqueta *inicio*, por lo que los saltos desde el interior de S_1.*codigo* pueden ir directo hacia *inicio*.

El código para $S \rightarrow S_1\ S_2$ consiste en el código para S_1 seguido del código para S_2. Las reglas semánticas administran las etiquetas; la primera instrucción después del código para S_1 es el principio del código para S_2; y la instrucción después del código para S_2 es también la instrucción después del código para S.

En la sección 6.7 hablaremos sobre la traducción de las instrucciones de flujo de control. Ahí veremos un método alternativo, conocido como "parcheo de retroceso", el cual emite código para las instrucciones en una sola pasada.

6.6.4 Traducción del flujo de control de las expresiones booleanas

Las reglas semánticas para las expresiones booleanas en la figura 6.37 complementan a las reglas semánticas para las instrucciones en la figura 6.36. Como en la distribución de código de la figura 6.35, una expresión booleana B se traduce en instrucciones de tres direcciones que

sólo cuando se necesitan. De manera alternativa, las etiquetas innecesarias pueden eliminarse durante una fase de organización siguiente.

evalúan a B mediante saltos condicionales e incondicionales hacia una de dos etiquetas: $B.true$ si B es verdadera, y $B.false$ si B es falsa.

PRODUCCIÓN	REGLAS SEMÁNTICAS
$B \rightarrow B_1$ `||` B_2	$B_1.true = B.true$ $B_1.false = nuevaetiqueta()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.codigo = B_1.codigo \,\|\|\, etiqueta(B_1.false) \,\|\|\, B_2.codigo$
$B \rightarrow B_1$ && B_2	$B_1.true = nuevaetiqueta()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.codigo = B_1.codigo \,\|\|\, etiqueta(B_1.true) \,\|\|\, B_2.codigo$
$B \rightarrow$! B_1	$B_1.true = B.false$ $B_1.false = B.true$ $B.codigo = B_1.codigo$
$B \rightarrow E_1$ **rel** E_2	$B.codigo = E_1.codigo \,\|\|\, E_2.codigo$ $\quad \|\| \; gen('\texttt{if}' \; E_1.dir \; \textbf{rel}.op \; E_2.dir \; '\texttt{goto}' \; B.true)$ $\quad \|\| \; gen('\texttt{goto}' \; B.false)$
$B \rightarrow$ **true**	$B.codigo = gen('\texttt{goto}' \; B.true)$
$B \rightarrow$ **false**	$B.codigo = gen('\texttt{goto}' \; B.false)$

Figura 6.37: Generación del código de tres direcciones para valores booleanos

La cuarta producción en la figura 6.37, $B \rightarrow E_1$ **rel** E_2, se traduce directamente en una instrucción de tres direcciones de comparación, con saltos hacia los lugares apropiados. Por ejemplo, B de la forma $a < b$ se traduce en:

```
if a < b goto B.true
goto B.false
```

El resto de las producciones para B se traducen de la siguiente manera:

1. Suponga que B es de la forma B_1 `||` B_2. Si B_1 es verdadera, entonces sabemos de inmediato que B en sí es verdadera, por lo que $B_1.true$ es igual que $B.true$. Si B_1 es falsa, entonces hay que evaluar B_2, para hacer que $B_1.false$ sea la etiqueta de la primera instrucción en el código para B_2. Las salidas verdadera y falsa de B_2 son iguales que las salidas verdadera y falsa de B, respectivamente.

2. La traducción de $B_1 \&\& B_2$ es similar.
3. No se necesita código para una expresión B de la forma $!B_1$: sólo se intercambian las salidas verdadera y falsa de B para obtener las salidas verdadera y falsa de B_1.
4. Las constantes **true** y **false** se traducen en saltos hacia *B.true* y *B.false*, respectivamente.

Ejemplo 6.22: Considere de nuevo la siguiente instrucción del ejemplo 6.21:

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

Si utilizamos las definiciones dirigidas por la sintaxis en las figuras 6.36 y 6.37, obtendremos el código de la figura 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:
```

Figura 6.38: Traducción del flujo de control de una instrucción if simple

La instrucción (6.13) constituye un programa generado por $P \rightarrow S$ de la figura 6.36. Las reglas semánticas para la producción generan una nueva etiqueta L_1 para la instrucción después del código para S. La instrucción S tiene la forma **if** (B) S_1, en donde S_1 es `x = 0;`, por lo que las reglas en la figura 6.36 generan una nueva etiqueta L_2 y la adjuntan a la primera instrucción (y sólo a ésta, en este caso) en S_1.*codigo*, que es `x = 0`.

Como `||` tiene menor precedencia que &&, la expresión booleana en (6.13) tiene la forma B_1 `||` B_2, en donde B_1 es $x < 100$. Siguiendo las reglas en la figura 6.37, B_1.*true* es L_2, la etiqueta de la asignación `x = 0;`. B_1.*false* es una nueva etiqueta L_3, que se adjunta a la primera instrucción en el código para B_2.

Observe que el código generado no es óptimo, en cuanto a que la traducción tiene tres instrucciones más (goto) que el código en el ejemplo 6.21. La instrucción `goto` L_3 es redundante, ya que L_3 es la etiqueta de la siguiente instrucción. Las dos instrucciones `goto` L_1 pueden eliminarse mediante el uso de `ifFalse` en vez de instrucciones `if`, como en el ejemplo 6.21. □

6.6.5 Evitar gotos redundantes

En el ejemplo 6.22, la comparación $x > 200$ se traduce en el siguiente fragmento de código:

```
        if x > 200 goto L4
        goto L1
L4:     ···
```

En vez de ello, considere la siguiente instrucción:

```
        ifFalse x > 200 goto L1
L4:     ···
```

Esta instrucción `ifFalse` aprovecha el flujo natural de una instrucción a la siguiente en secuencia, por lo que el control simplemente "pasa" hacia la etiqueta L_4 si $x > 200$ es falsa, con lo cual se evita un salto.

En los esquemas de código para las instrucciones if y while de la figura 6.35, el código para la instrucción S_1 sigue justo después del código para la expresión booleana B. Mediante el uso de una etiqueta especial llamada *pasar* (es decir, "no generar ningún salto"), podemos adaptar las reglas semánticas de las figuras 6.36 y 6.37 para permitir que el control pase del código para B al código para S_1. Las nuevas reglas para $S \rightarrow \textbf{if}\ (B)\ S_1$ en la figura 6.36 establecen $B.true$ a *pasar*:

$$\begin{aligned} B.true &= pasar \\ B.false &= S_1.siguiente = S.siguiente \\ S.codigo &= B.codigo \ ||\ S_1.codigo \end{aligned}$$

De manera similar, las reglas para las instrucciones if-else y while también establecen $B.true$ a *pasar*.

Ahora vamos a adaptar las reglas semánticas a las expresiones booleanas, para permitir que el control pase a través de un código al otro, siempre que sea posible. Las nuevas reglas para $B \rightarrow E_1\ \textbf{rel}\ E_2$ en la figura 6.39 generan dos instrucciones, como en la figura 6.37, si $B.true$ y $B.false$ son etiquetas explícitas; es decir, que ninguna sea igual a *pasar*. En caso contrario, si $B.true$ es una etiqueta explícita, entonces $B.false$ debe ser *pasar*, para que generen una instrucción `if` que permita que el control pase de un código a otro si la condición es falsa. En cambio, si $B.true$ es una etiqueta explícita, entonces generan una instrucción `ifFalse`. En el caso restante, tanto $B.true$ como $B.false$ son *pasar*, por lo que no se genera ningún salto.[9]

En las nuevas reglas para $B \rightarrow B_1\ ||\ B_2$ en la figura 6.40, observe que el significado de la etiqueta *pasar* para B es distinto del significado para B_1. Suponga que $B.true$ es *pasar*; es decir, el control pasa a través de B, si B se evalúa como verdadera. Aunque B se evalúa como verdadera si B_1 lo hace, $B_1.true$ debe asegurar que el control salte sobre el código para B_2, para llegar a la siguiente instrucción después de B.

Por otro lado, si B_1 se evalúa como falsa, el valor verdadero de B se determina mediante el valor de B_2, por lo que las reglas en la figura 6.40 aseguran que $B_1.false$ correspondan al control que pasa desde B_1 hasta el código para B_2.

Las reglas semánticas para $B \rightarrow B_1\ \&\&\ B_2$ son similares a las de la figura 6.40. Las dejaremos como un ejercicio.

Ejemplo 6.23: Con las nuevas reglas que utilizan la etiqueta especial *pasar*, el programa (6.13) del ejemplo 6.21

[9] En C y Java, las expresiones pueden contener asignaciones en su interior, por lo que debe generarse código para las subexpresiones E_1 y E_2, incluso si tanto $B.true$ como $B.false$ son *pasar*. Si se desea, el código muerto puede eliminarse durante una fase de optimización.

$$
\begin{aligned}
\mathit{prueba} &= E_1.\mathit{dir}\ \mathbf{rel}.\mathit{op}\ E_2.\mathit{dir}\\
s &= \mathbf{if}\ B.\mathit{true} \neq \mathit{pasar}\ \mathbf{and}\ B.\mathit{false} \neq \mathit{pasar}\ \mathbf{then}\\
&\qquad \mathit{gen}('\texttt{if}'\ \mathit{prueba}\ '\texttt{goto}'\ B.\mathit{true}) \mid\mid \mathit{gen}('\texttt{goto}'\ B.\mathit{false})\\
&\quad \mathbf{else\ if}\ B.\mathit{true} \neq \mathit{pasar}\ \mathbf{then}\ \mathit{gen}('\texttt{if}'\ \mathit{prueba}\ '\texttt{goto}'\ B.\mathit{true})\\
&\quad \mathbf{else\ if}\ B.\mathit{false} \neq \mathit{pasar}\ \mathbf{then}\ \mathit{gen}('\texttt{ifFalse}'\ \mathit{prueba}\ '\texttt{goto}'\ B.\mathit{false})\\
&\quad \mathbf{else}\ ''\\
B.\mathit{codigo} &= E_1.\mathit{codigo} \mid\mid E_2.\mathit{codigo} \mid\mid s
\end{aligned}
$$

Figura 6.39: Reglas semánticas para $B \rightarrow E_1\ \mathbf{rel}\ E_2$

$$
\begin{aligned}
B_1.\mathit{true} &= \mathbf{if}\ B.\mathit{true} \neq \mathit{pasar}\ \mathbf{then}\ B.\mathit{true}\ \mathbf{else}\ \mathit{nuevaetiqueta}()\\
B_1.\mathit{false} &= \mathit{pasar}\\
B_2.\mathit{true} &= B.\mathit{true}\\
B_2.\mathit{false} &= B.\mathit{false}\\
B.\mathit{codigo} &= \mathbf{if}\ B.\mathit{true} \neq \mathit{pasar}\ \mathbf{then}\ B_1.\mathit{codigo} \mid\mid B_2.\mathit{codigo}\\
&\qquad \mathbf{else}\ B_1.\mathit{codigo} \mid\mid B_2.\mathit{codigo} \mid\mid \mathit{etiqueta}(B_1.\mathit{true})
\end{aligned}
$$

Figura 6.40: Reglas semánticas para $B \rightarrow B_1$ `||` B_2

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

se traduce en el código de la figura 6.41.

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

Figura 6.41: Instrucción if traducida mediante el uso de la técnica de pasar de un código a otro

Como en el ejemplo 6.22, las reglas para $P \rightarrow S$ crean la etiqueta L_1. La diferencia del ejemplo 6.22 es que el atributo heredado $B.\mathit{true}$ es *pasar* cuando se aplican las reglas semánticas para $B \rightarrow B_1$ `||` B_2 ($B.\mathit{false}$ es L_1). Las reglas en la figura 6.40 crean una nueva etiqueta L_2 para permitir un salto sobre el código para B_2, si B_1 se evalúa como verdadera. Por ende, $B_1.\mathit{true}$ es L_2. y $B_1.\mathit{false}$ es *pasar*, ya que B_2 debe evaluarse si B_1 es falsa.

Por lo tanto, llegamos a la producción $B \rightarrow E_1\ \mathbf{rel}\ E_2$ que genera a $x < 100$ con $B.\mathit{true} = L_2$ y $B.\mathit{false} = \mathit{pasar}$. Con estas etiquetas heredadas, las reglas en la figura 6.39 generan, por lo tanto, una sola instrucción `if x < 100 goto` $\mathtt{L}_2$. □

6.6.6 Valores boleanos y código de salto

Esta sección se ha enfocado en el uso de expresiones booleanas para alterar el flujo de control en las instrucciones. Una expresión booleana también puede evaluarse en base a su valor, como en las instrucciones de asignación como `x = true;` o `x = a<b;`.

Una forma limpia de manejar ambas funciones de las expresiones booleanas es primero construir un árbol sintáctico para las expresiones, usando cualquiera de los siguientes métodos:

1. *Usar dos pasadas.* Construya un árbol sintáctico completo para la entrada, y después recorra el árbol en orden por profundidad, calculando las transacciones especificadas por las reglas semánticas.
2. *Usar una pasada para las instrucciones, pero dos pasadas para las expresiones.* Con este método, traduciríamos E en **while** (E) S_1 antes de examinar S_1. Sin embargo, la traducción de E se realizaría construyendo su árbol sintáctico y después recorriendo ese árbol.

La siguiente gramática tiene un solo no terminal E para las expresiones:

$$S \rightarrow \textbf{id} = E \textbf{ ; } \mid \textbf{if} (E) S \mid \textbf{while} (E) S \mid S\, S$$
$$E \rightarrow E \textbf{ || } E \mid E \&\& E \mid E \textbf{ rel } E \mid E + E \mid (E) \mid \textbf{id} \mid \textbf{true} \mid \textbf{false}$$

El no terminal E gobierna el flujo de control en $S \rightarrow$ **while** (E) S_1. El mismo no terminal E denota un valor en $S \rightarrow \textbf{id} = E$; y $E \rightarrow E + E$.

Podemos manejar estas dos funciones de las expresiones mediante el uso de funciones separadas para generación de código. Suponga que el atributo $E.n$ denota el nodo del árbol sintáctico para una expresión E y que los nodos son objetos. Suponga que el método *saltar* genera código de salto en un nodo de una expresión, y que el método *r-value* genera código para calcular el valor del nodo en un nombre temporal.

Cuando E aparece en $S \rightarrow$ **while** (E) S_1, se hace una llamada al método *saltar* en el nodo $E.n$. La implementación de *saltar* se basa en las reglas para las expresiones booleanas en la figura 6.37. En específico, se genera el código de salto mediante la llamada a $E.n.saltar(t, f)$, en donde t es una nueva etiqueta para la primera instrucción de $S_1.codigo$ y f es la etiqueta $S.siguiente$.

Cuando E aparece en $S \rightarrow \textbf{id} = E$;, se hace una llamada al método *r-value* en el nodo $E.n$. Si E tiene la forma $E_1 + E_2$, la llamada al método $E.n.$ *r-value*() genera código como se vio en la sección 6.4. Si E tiene la forma $E_1 \&\& E_2$, primero generamos código de salto para E y después asignamos verdadero o falso a un nuevo nombre temporal `t` en las salidas de verdadero y falso, respectivamente, desde el código de salto.

Por ejemplo, la asignación `x = a<b && c<d` puede implementarse mediante el código en la figura 6.42.

6.6.7 Ejercicios para la sección 6.6

Ejercicio 6.6.1: Agregue reglas a la definición dirigida por la sintaxis de la figura 6.36 para las siguientes construcciones de flujo de control:

a) Una instrucción repeat: **repeat** S **while** B.