# Capítulo 8

# Generación de código



En este capítulo volveremos a la tarea final de un compilador, la de generar código ejecutable para una máquina objetivo que sea una fiel representación de la semántica del código fuente. La generación de código es la fase más compleja de un compilador, puesto que no sólo depende de las características del lenguaje fuente sino también de contar con información detallada acerca de la arquitectura objetivo, la estructura del ambiente de ejecución y el sistema operativo que esté corriendo en la máquina objetivo. La generación de código por lo regular implica también algún intento por **optimizar**, o mejorar, la velocidad y/o el tamaño del código objetivo recolectando más información acerca del programa fuente y adecuando el código generado para sacar ventaja de las características especiales de la máquina objetivo, tales como registros, modos de direccionamiento, distribución y memoria caché.

Debido a la complejidad de la generación del código, un compilador por lo regular divide esta fase en varios pasos, los cuales involucran varias estructuras de datos intermedias, y a menudo incluyen alguna forma de código abstracto denominada **código intermedio**. Un compilador también puede detener en breve la generación de código ejecutable real pero, en vez de esto genera alguna forma de código ensamblador que debe ser procesado adicionalmente por un ensamblador, un ligador y un cargador, los cuales pueden ser proporcionados por el sistema operativo o compactados con el compilador. En este capítulo nos concentraremos únicamente en los fundamentos de la generación del código intermedio y el código ensamblador, los cuales tienen muchas características en común. Ignoraremos el problema del procesamiento adicional del código ensamblador

en código ejecutable, el cual puede ser controlado más adecuadamente mediante un lenguaje ensamblador o sistemas de texto de programación.

En la primera sección de este capítulo analizaremos dos formas populares de código intermedio, el código de tres direcciones y el código P, y comentaremos algunas de sus propiedades. En la segunda sección describiremos los algoritmos básicos que permiten generar código intermedio o ensamblador. En secciones posteriores se analizarán técnicas de generación de código para varias características del lenguaje, incluyendo expresiones, sentencias de asignación y sentencias de flujo de control, tales como sentencias if y sentencias while y llamadas a procedimiento/función. A estas secciones les seguirán estudios de caso del código producido para estas características mediante dos compiladores comerciales: el compilador C de Borland para la arquitectura 80×86 y el compilador C de Sun para la arquitectura RISC de Sparc.

En una sección posterior aplicaremos las técnicas que permiten desarrollar un generador de código ensamblador para el lenguaje TINY estudiadas hasta aquí. Como la generación de código en este nivel de detalle requiere una máquina objetivo real, primero analizaremos una arquitectura objetivo simple y un simulador de máquina denominado TM, para el cual se proporciona un listado fuente en el apéndice C. A continuación describiremos el generador de código complejo para TINY. Finalmente, daremos una visión general de las técnicas para el mejoramiento u optimización del código estándar, y describiremos cómo algunas de las técnicas recién adquiridas pueden incorporarse en el generador de código de TINY.

## 8.1 CÓDIGO INTERMEDIO Y ESTRUCTURAS DE DATOS PARA GENERACIÓN DE CÓDIGO

Una estructura de datos que representa el programa fuente durante la traducción se denomina **representación intermedia**, o **IR** (por las siglas del término en inglés) para abreviar. En este texto hasta ahora hemos usado un árbol sintáctico abstracto como el IR principal. Además del IR, la principal estructura de datos utilizada durante la traducción es la tabla de símbolos, la cual se estudió en el capítulo 6.

Aunque un árbol sintáctico abstracto es una representación adecuada del código fuente, incluso para la generación de código (como veremos en una sección posterior), no se parece ni remotamente al código objetivo, en particular en su representación de construcciones de flujo de control, donde el código objetivo, como el código de máquina o código ensamblador, emplean saltos más que construcciones de alto nivel, como las sentencias if y while. Por lo tanto, un escritor de compiladores puede desear generar una nueva forma de representación intermedia del árbol sintáctico que se parezca más al código objetivo o reemplace del todo al árbol sintáctico mediante una representación intermedia de esa clase, y entonces genere código objetivo de esta nueva representación. Una representación intermedia de esta naturaleza que se parece al código objetivo se denomina **código intermedio**.

El código intermedio puede tomar muchas formas: existen casi tantos estilos de código intermedio como compiladores. Sin embargo, todos representan alguna forma de **linealización** del árbol sintáctico, es decir, una representación del árbol sintáctico en forma secuencial. El código intermedio puede ser de muy alto nivel, representar todas las operaciones de manera casi tan abstracta como un árbol sintáctico, o parecerse mucho al código objetivo. Puede o no utilizar información detallada acerca de la máquina objetivo y el ambiente de ejecución, como los tamaños de los tipos de datos, las ubicaciones de las variables y la disponibilidad de los registros. Puede o no incorporar toda la información contenida en la tabla

de símbolos, tal como los ámbitos, los niveles de anidación y los desplazamientos de las variables. Si lo hace, entonces la generación de código objetivo puede basarse sólo en el código intermedio; si no, el compilador debe retener la tabla de símbolos para la generación del código objetivo.

El código intermedio es particularmente útil cuando el objetivo del compilador es producir código muy eficiente, ya que para hacerlo así se requiere una cantidad importante del análisis de las propiedades del código objetivo, y esto se facilita mediante el uso del código intermedio. En particular, las estructuras de datos adicionales que incorporan información de un detallado análisis posterior al análisis sintáctico se pueden generar fácilmente a partir del código intermedio, aunque no es imposible hacerlo directamente desde el árbol sintáctico.

El código intermedio también puede ser útil al hacer que un compilador sea más fácilmente redirigible: si el código intermedio es hasta cierto punto independiente de la máquina objetivo, entonces generar código para una máquina objetivo diferente sólo requiere volver a escribir el traductor de código intermedio a código objetivo, y por lo regular esto es más fácil que volver a escribir todo un generador de código.

En esta sección estudiaremos dos formas populares de código intermedio: el **código de tres direcciones** y el **código P**. Ambos se presentan en muchas formas diferentes, y nuestro estudio aquí se enfocará sólo en las características generales, en lugar de presentar una descripción detallada de una versión de cada uno. Tales descripciones se pueden encontrar en la literatura que se describe en la sección de notas y referencias al final del capítulo.

## 8.1.1 Código de tres direcciones

La instrucción básica del código de tres direcciones está diseñada para representar la evaluación de expresiones aritméticas y tiene la siguiente forma general:

```
x = y op z
```

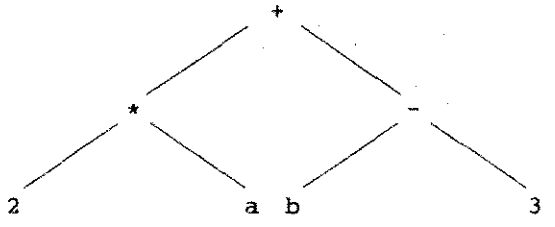
Esta instrucción expresa la aplicación del operador *op* a los valores de `y` y `z`, y la asignación de este valor para que sea el nuevo valor de `x`. Aquí *op* puede ser un operador aritmético como + o − o algún otro operador que pueda actuar sobre los valores de `y` y `z`.

El nombre "código de tres direcciones" viene de esta forma de instrucción, ya que por lo general cada uno de los nombres `x`, `y` y `z` representan una dirección de la memoria. Sin embargo, observe que el uso de la dirección de `x` difiere del uso de las direcciones de `y` y `z`, y que tanto `y` como `z` (pero no `x`) pueden representar constantes o valores de literales sin direcciones de ejecución.

Para ver cómo las secuencias de código de tres direcciones de esta forma pueden representar el cálculo de una expresión, considere la expresión aritmética

```
2*a+(b-3)
```

con árbol sintáctico



El código de tres direcciones correspondiente es

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

El código de tres direcciones requiere que el compilador genere nombres para elementos temporales, a los que en este ejemplo hemos llamado `t1`, `t2` y `t3`. Estos elementos temporales corresponden a los nodos interiores del árbol sintáctico y representan sus valores calculados, con el último elemento temporal (`t3`, en este ejemplo) representando el valor de la raíz.[1] La manera en que estos elementos temporales finalmente son asignados en la memoria no es especificada por este código; por lo regular serán asignados a registros, pero también se pueden conservar en registros de activación (véase el análisis de la pila temporal en el capítulo anterior).

El código de tres direcciones que se acaba de dar representa una linealización de izquierda a derecha del árbol sintáctico, debido a que el código correspondiente a la evaluación del subárbol izquierdo de la raíz se lista primero. Es posible que un compilador desee utilizar un orden diferente en ciertas circunstancias. Aquí simplemente advertimos que puede haber otro orden para este código de tres direcciones, a saber (con un significado diferente para los elementos temporales),

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

Evidentemente, la forma del código de tres direcciones que hemos mostrado no basta para representar todas las características ni siquiera del lenguaje de programación más pequeño. Por ejemplo, los operadores unitarios, como la negación, requieren de una variación del código de tres direcciones que contenga sólo dos direcciones, tal como

```
t2 = - t1
```

Si se desea tener capacidad para todas las construcciones de un lenguaje de programación estándar, será necesario variar la forma del código de tres direcciones en cada construcción. Si un lenguaje contiene características poco habituales, puede ser necesario incluso inventar nuevas formas del código de tres direcciones para expresarlas. Ésta es una de las razones por las que no existe una forma estándar para el código de tres direcciones (del mismo modo que no existe una forma estándar para árboles sintácticos).

En las siguientes secciones de este capítulo trataremos algunas construcciones comunes de lenguaje de programación de manera individual y mostraremos cómo estas construcciones son traducidas por lo común como código de tres direcciones. Sin embargo, para acostumbrarnos a lo que nos espera presentamos aquí un ejemplo completo en el que se utiliza el lenguaje TINY presentado anteriormente.

Considere el programa de muestra TINY de la sección 1.7 (capítulo 1) que calcula el máximo común divisor de dos enteros, el cual repetimos en la figura 8.1. El código de tres direcciones de muestra para este programa se ofrece en la figura 8.2. Este código contiene

---

1. Los nombres `t1`, `t2`, y así sucesivamente están sólo destinados a ser representativos del estilo general de tal código. De hecho, los nombres temporales en código de tres direcciones deben ser distintos de cualquier nombre que pudiera ser utilizado en el código fuente real, si se van a mezclar los nombres de código fuente, como ocurre aquí.

Figura 8.1
Programa de muestra TINY

```
{ Programa de muestra
  en lenguaje TINY--
  calcula el factorial
}
read x; { introducir un entero }
if 0 < x then { no calcular si x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { salida del factorial de x }
end
```

Figura 8.2
Código de tres direcciones para el programa TINY de la figura 8.1

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

varias formas diferentes de código de tres direcciones. En primer lugar, las operaciones integradas de entrada y salida `read` y `write` se tradujeron directamente en instrucciones de una dirección. En segundo lugar, existe una instrucción de salto condicional `if_false` que se utiliza para traducir tanto sentencias if como sentencias repeat y que contiene dos direcciones: el valor condicional que se probará y la dirección de código a la que se saltará. Las posiciones de las direcciones de salto también se indican mediante instrucciones (de una dirección) `label`. Estas instrucciones `label` pueden ser innecesarias, dependiendo de las estructuras de datos empleadas para implementar el código de tres direcciones. En tercer lugar, una instrucción `halt` (sin direcciones) sirve para marcar el final del código.

Finalmente, observamos que las asignaciones en el código fuente producen la generación de **instrucciones de copia** de la forma

```
x = y
```

Por ejemplo, la sentencia de programa de muestra

```
fact := fact * x;
```

se traduce en las dos instrucciones del código de tres direcciones

```
t2 = fact * x
fact = t2
```

aunque sería suficiente una instrucción de tres direcciones. Esto sucede por razones técnicas que se explicarán en la sección 8.2.

## 8.1.2 Estructuras de datos para la implementación del código de tres direcciones

El código de tres direcciones por lo regular no está implementado en forma textual como lo hemos escrito (aunque podría estarlo). En vez de eso, cada instrucción de tres direcciones está implementada como una estructura de registro que contiene varios campos, y la secuencia completa de instrucciones de tres direcciones está implementada como un arreglo o lista ligada, la cual se puede conservar en la memoria o escrita para (y leída desde) archivos temporales cuando sea necesario.

La implementación más común consiste en implementar el código de tres direcciones fundamentalmente como se muestra, lo que significa que son necesarios cuatro campos: uno para la operación y tres para las direcciones. Para las instrucciones que necesitan menos de tres direcciones, uno o más de los campos de dirección proporcionan un valor nulo o "vacío"; la selección de los campos depende de la implementación. Como son necesarios cuatro campos, una representación de código de tres direcciones de esta naturaleza se denomina **cuádruple**. Una posible implementación cuádruple del código de tres direcciones de la figura 8.2 se proporciona en la figura 8.3, donde se escribieron los cuádruples en notación matemática de "tuplas".

Los `typedef` posibles de C para implementar los cuádruples mostrados en la figura 8.3 se ofrecen en la figura 8.4. En estas definiciones permitimos que una dirección sea sólo una constante entera o una cadena (que representa el nombre de un elemento temporal o una variable). También, puesto que se utilizan nombres, éstos deben introducirse en una tabla de símbolos, y se necesitará realizar búsquedas durante procesamiento adicional. Una alternativa para conservar los nombres en los cuádruples es mantener apuntadores hacia entradas de la tabla de símbolos. Esto evita tener que realizar búsquedas adicionales y es particularmente ventajoso en un lenguaje con ámbitos anidados, donde se necesita más información de ámbito que sólo el nombre para realizar una búsqueda. Si también se introducen constantes en la tabla de símbolos, entonces no es necesaria una unión en el tipo de datos `Address`.

Una implementación diferente del código de tres direcciones consiste en utilizar las instrucciones mismas para representar los elementos temporales. Esto reduce la necesidad de campos de dirección de tres a dos, puesto que en una instrucción de tres direcciones que contiene la totalidad de las tres direcciones, la dirección objetivo es siempre un elemento temporal.[2] Una implementación del código de tres direcciones de esta clase se denomina **triple** y requiere que cada instrucción de tres direcciones sea referenciable, ya sea como

2. Esto no es una verdad inherente acerca del código de tres direcciones, pero puede asegurarse mediante la implementación. Por ejemplo, es verdadero en el código de la figura 8.2 (véase también la figura 8.3).

Figura 8.3
Implementación cuádruple para el código de tres direcciones de la figura 8.2

```
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)
```

Figura 8.4
Código C que define estructuras de datos posibles para los cuádruples de la figura 8.3

```
typedef enum {rd,gt,if_f,asn,lab,mul,
                    sub,eq,wri,halt,. . .} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
            { AddrKind kind;
              union
              { int val;
                char * name;
              } contents;
            } Address;
typedef struct
            { OpKind op;
              Address addr1,addr2,addr3;
            } Quad;
```

un índice en un arreglo o como un apuntador en una lista ligada. Por ejemplo, una representación abstracta de una implementación del código de tres direcciones de la figura 8.2 como triples se proporciona en la figura 8.5 (página 404). En esa figura utilizamos un sistema de numeración que correspondería a índices de arreglo para representar los triples. Las referencias de triple también se distinguen de las constantes poniéndolas entre paréntesis en los mismos triples. Adicionalmente, en la figura 8.5 eliminamos las instrucciones `label` y las reemplazamos con referencias a los índices de triple mismos.

Los triples son una manera eficiente para representar el código de tres direcciones, ya que la cantidad de espacio se reduce y el compilador no necesita generar nombres para elementos temporales. Sin embargo, los triples tienen una importante desventaja, y ésta consiste en que, si se los representa mediante índices de arreglo, entonces cualquier movimiento de sus posiciones se vuelve difícil. Por otra parte, una representación de lista ligada no sufre de esta deficiencia. Problemas adicionales que involucran triples, y el código C apropiado para la definición de los mismos, se dejan como ejercicios.

Figura 8.5
Una representación del código de tres direcciones de la figura 8.2 como triples

```
(0)     (rd,x,_)
(1)     (gt,x,0)
(2)     (if_f,(1),(11))
(3)     (asn,1,fact)
(4)     (mul,fact,x)
(5)     (asn,(4),fact)
(6)     (sub,x,1)
(7)     (asn,(6),x)
(8)     (eq,x,0)
(9)     (if_f,(8),(4))
(10)    (wri,fact,_)
(11)    (halt,_,_)
```

## 8.1.3 Código P

El código P comenzó como un código ensamblador objetivo estándar producido por varios compiladores Pascal en la década de 1970 y principios de la de 1980. Fue diseñado para ser el código real de una máquina de pila hipotética, denominada **máquina P**, para la que fue escrito un intérprete en varias máquinas reales. La idea era hacer que los compiladores de Pascal se transportaran fácilmente requiriendo sólo que se volviera a escribir el intérprete de la máquina P para una nueva plataforma. El código P también ha probado ser útil como código intermedio, y se han utilizado varias extensiones y modificaciones del mismo en diversos compiladores de código nativo, la mayor parte para lenguajes tipo Pascal.

Como el código P fue diseñado para ser directamente ejecutable, contiene una descripción implícita de un ambiente de ejecución particular que incluye tamaños de datos, además de mucha información específica para la máquina P, que se debe conocer si se desea que un programa de código P sea comprensible. Para evitar este detalle describiremos aquí una versión simplificada y abstracta de código P apropiada para la exposición. Las descripciones de diversas versiones de código P real podrán encontrarse en varias referencias enumeradas al final del capítulo.

Para nuestros propósitos la máquina P está compuesta por una memoria de código, una memoria de datos no especificada para variables nombradas y una pila para datos temporales, junto con cualquier registro que sea necesario para mantener la pila y apoyar la ejecución.

Como un primer ejemplo de código P, considere la expresión

```
2*a+(b-3)
```

empleada en la sección 8.1.1, cuyo árbol sintáctico aparece en la página 399. Nuestra versión de código P para esta expresión es la que se muestra en seguida:

```
ldc 2       ; carga la constante 2
lod a       ; carga el valor de la variable a
mpi         ; multiplicación entera
lod b       ; carga el valor de la variable b
ldc 3       ; carga la constante 3
sbi         ; sustracción o resta entera
adi         ; adición de enteros
```

Estas instrucciones se ven como si representaran las siguientes operaciones en una máquina P. En primer lugar, `ldc 2` inserta el valor 2 en la pila temporal. Luego, `lod a` inserta el valor de la variable `a` en la pila. La instrucción `mpi` extrae estos dos valores de la pila, los multiplica (en orden inverso) e inserta el resultado en la pila. Las siguientes dos instrucciones (`lod b` y `ldc 3`) insertan el valor de `b` y la constante 3 en la pila (ahora tenemos tres valores en la pila). Posteriormente, la instrucción `sbi` extrae los dos valores superiores de la pila, resta el primero del segundo, e inserta el resultado. Finalmente, la instrucción `adi` extrae los dos valores restantes de la pila, los suma e inserta el resultado. El código finaliza con un solo valor en la pila, que representa el resultado del cálculo.

Como un segundo ejemplo introductorio, considere la sentencia de asignación

```
x := y + 1
```

Esto corresponde a las siguientes instrucciones en código P:

```
lda x       ; carga dirección de x
lod y       ; carga valor de y
ldc 1       ; carga constante 1
adi         ; suma
sto         ; almacena tope a dirección
            ; debajo del tope y extrae ambas
```

Advierta cómo este código calcula primero la dirección de `x`, luego el valor de la expresión que será asignada a `x`, y finalmente ejecuta un comando `sto`, el cual requiere que se encuentren dos valores en la parte superior de la pila temporal: el valor que será almacenado y, debajo de éste, la dirección en la memoria variable en la cual se almacenará. La instrucción `sto` también extrae estos dos valores (dejando la pila vacía en este ejemplo). De este modo el código P hace una distinción entre direcciones de carga (`lda`) y valores ordinarios (`lod`), que corresponden a la diferencia entre el uso de `x` en el lado izquierdo y el uso de `y` en el lado derecho de la asignación `x:=y+1`.

Como último ejemplo de código P en esta sección damos una traducción de código P en la figura 8.6 para el programa TINY de la figura 8.1, junto con comentarios que describen cada operación.

El código P de la figura 8.6 (página 406) contiene varias instrucciones nuevas de código P. En primer lugar, las instrucciones `rdi` y `wri` (sin parámetros) implementan las sentencias enteras `read` y `write` construidas en TINY. La instrucción de código P `rdi` requiere que la dirección de la variable cuyo valor va a leerse se encuentre en la parte superior de la pila, y esta dirección es extraída como parte de la instrucción. La instrucción `wri` requiere que el valor que será escrito esté en la parte superior de la pila, y este valor se extrae como parte de la instrucción. Otras instrucciones de código P que aparecen en la figura 8.6 que aún no se han comentado son: la instrucción `lab`, que define la posición de un nombre de etiqueta; la instrucción `fjp` ("false jump"), que requiere un valor booleano en el tope o parte superior de la pila (el cual es extraído); la instrucción `sbi` (de "sustracción entera", en inglés), cuya operación es semejante a la de otras instrucciones aritméticas; y las operaciones de comparación `grt` (de "greater than") y `equ` ("equal to"), que requieren dos valores enteros en el tope de la pila (los cuales son extraídos), y que insertan sus resultados booleanos. Finalmente, la instrucción `stp` ("stop") correspondiente a la instrucción `halt` del anterior código de tres direcciones.

Figura 8.6
Código P para el programa TINY de la figura 8.1

```
lda x          ; carga dirección de x
rdi            ; lee un entero, almacena a la
               ; dirección en el tope de la pila (y la extrae)
lod x          ; carga el valor de x
ldc 0          ; carga la constante 0
grt            ; extrae y compara dos valores del tope
               ; inserta resultado Booleano
fjp L1         ; extrae resultado Booleano, salta a L1 si es falso
lda fact       ; carga dirección de fact
ldc 1          ; carga la constante 1
sto            ; extrae dos valores, almacenando el primero en la
               ; dirección representada por el segundo
lab L2         ; definición de etiqueta L2
lda fact       ; carga dirección de fact
lod fact       ; carga valor de fact
lod x          ; carga valor de x
mpi            ; multiplica
sto            ; almacena el tope a dirección del segundo y extrae
lda x          ; carga dirección de x
lod x          ; carga valor de x
ldc 1          ; carga constante 1
sbi            ; resta
sto            ; almacena (como antes)
lod x          ; carga valor de x
ldc 0          ; carga constante 0
equ            ; prueba de igualdad
fjp L2         ; salto a L2 si es falso
lod fact       ; carga valor de fact
wri            ; escribe tope de pila y extrae
lab L1         ; definición de etiqueta L1
stp
```

*Comparación del código P con el código de tres direcciones* El código P en muchos aspectos está más cercano al código de máquina real que al código de tres direcciones. Las instrucciones en código P también requieren menos direcciones: todas las instrucciones que hemos visto son instrucciones de "una dirección" o "cero direcciones". Por otra parte, el código P es menos compacto que el código de tres direcciones en términos de números de instrucciones, y el código P no está "autocontenido" en el sentido que las instrucciones funcionen implícitamente en una pila (y las localidades de pila implícitas son de hecho las direcciones "perdidas"). La ventaja respecto a la pila es que contiene todos los valores temporales necesarios en cada punto del código, y el compilador no necesita asignar nombres a ninguno de ellos, como en el código de tres direcciones.

*Implementación del código P* Históricamente, el código P ha sido en su mayor parte generado como un archivo del texto, pero las descripciones anteriores de las implementaciones de estructura de datos internas para el código de tres direcciones (cuádruples y triples) también funcionarán con una modificación apropiada para el código P.

# 8.2 TÉCNICAS BÁSICAS DE GENERACIÓN DE CÓDIGO

En esta sección comentaremos los enfoques básicos para la generación de código en general, mientras que en secciones posteriores abordaremos la generación de código para construcciones de lenguaje individuales por separado.

## 8.2.1 Código intermedio o código objetivo como un atributo sintetizado

La generación de código intermedio (o generación de código objetivo directa sin código intermedio) se puede ver como un cálculo de atributo similar a muchos de los problemas de atributo estudiados en el capítulo 6. En realidad, si el código generado se ve como un atributo de cadena (con instrucciones separadas por caracteres de retorno de línea), entonces este código se convierte en un atributo sintetizado que se puede definir utilizando una gramática con atributos, y generado directamente durante el análisis sintáctico o mediante un recorrido postorden del árbol sintáctico.

Para ver cómo el código de tres direcciones, o bien, el código P se pueden definir como un atributo sintetizado, considere la siguiente gramática que representa un pequeño subconjunto de expresiones en C:

*exp* → **`id`** = *exp* | *aexp*
*aexp* → *aexp* + *factor* | *factor*
*factor* → ( *exp* ) | **`num`** | **`id`**

Esta gramática sólo contiene dos operaciones, la asignación (con el símbolo =) y la adición (con el símbolo +).[3] El token **`id`** representa un identificador simple, y el token **`num`** simboliza una secuencia simple de dígitos que representa un entero. Se supone que ambos tokens tienen un atributo *strval* previamente calculado, que es el valor de la cadena, o lexema, del token (por ejemplo, "42" para un **`num`** o "xtemp" para un **`id`**).

*Código P* Consideraremos el caso de la generación del código P en primer lugar, ya que la gramática con atributos es más simple debido a que no es necesario generar nombres para elementos temporales. Sin embargo, la existencia de asignaciones incrustadas es un factor que implica complicaciones. En esta situación deseamos mantener el valor almacenado como el valor resultante de una expresión de asignación, ya que la instrucción de código P estándar `sto` es destructiva, pues el valor asignado se pierde. (Con esto el código P muestra sus orígenes de Pascal, en el cual no existen las asignaciones incrustadas.) Resolvemos este problema introduciendo una instrucción de **almacenamiento no destructiva** `stn` en nuestro código P, la que como la instrucción `sto`, supone que se encuentra un valor en la parte superior o tope de la pila y una dirección debajo de la misma; `stn` almacena el valor en la dirección pero deja el valor en el tope de la pila, mientras que descarta la dirección. Con esta nueva instrucción, una gramática con atributos para un atributo de cadena de código

3. La asignación en este ejemplo tiene la semántica siguiente: `x = e` almacena el valor de `e` en `x` y tiene el mismo valor resultante que `e`.

P se proporciona en la tabla 8.1. En esa figura utilizamos el nombre de atributo *pcode* para la cadena de código P. También empleamos dos notaciones diferentes para la concatenación de cadena: ++ cuando las instrucciones van a ser concatenadas con retornos de línea insertados entre ellas y || cuando se está construyendo una instrucción simple y se va a insertar un espacio.

Dejamos al lector describir el cálculo del atributo *pcode* en ejemplos individuales y mostrar que, por ejemplo, la expresión `(x=x+3)+4` tiene ese atributo

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

Tabla 8.1
Gramática con atributos de código P como un atributo de cadena sintetizado

| Regla gramatical | Reglas semánticas |
|---|---|
| $exp_1 \rightarrow$ **id** = $exp_2$ | $exp_1$ .*pcode* = "**lda**" \|\| **id** .*strval* ++ $exp_2$ .*pcode* ++ "**stn**" |
| *exp* → *aexp* | *exp* .*pcode* = *aexp* .*pcode* |
| $aexp_1 \rightarrow aexp_2$ + *factor* | $aexp_1$ .*pcode* = $aexp_2$ .*pcode* ++ *factor* .*pcode* ++ "**adi**" |
| *aexp* → *factor* | *aexp* .*pcode* = *factor* .*pcode* |
| *factor* → ( *exp* ) | *factor* .*pcode* = *exp* .*pcode* |
| *factor* → **num** | *factor* .*pcode* = "**ldc**" \|\| **num** .*strval* |
| *factor* → **id** | *factor* .*pcode* = "**lod**" \|\| **id** .*strval* |

***Código de tres direcciones*** Una gramática con atributos para código de tres direcciones de la gramática de expresión simple anterior se proporciona en la tabla 8.2. En esa tabla utilizamos el atributo de código que se llama *tacode* (para código de tres direcciones), y como en la tabla 8.1, ++ para concatenación de cadena con un retorno de línea y || para concatenación de cadena con un espacio. A diferencia del código P, el código de tres direcciones requiere que se generen nombres temporales para resultados intermedios en las expresiones, y esto requiere que la gramática con atributos incluya un nuevo atributo *name* para cada nodo. Este atributo también es sintetizado, pero para asignar nombres temporales recién generados a nodos interiores utilizamos una función *newtemp*( ) que se supone genera una secuencia de nombres temporales `t1`, `t2`, `t3`,. . . (se devuelve uno nuevo cada vez que se llama a *newtemp*( )). En este ejemplo simple sólo los nodos correspondientes al operador + necesitan nuevos nombres temporales; la operación de asignación simplemente utiliza el nombre de la expresión en el lado derecho.

Advierta en la tabla 8.2 que, en el caso de las producciones unitarias *exp* → *aexp* y *aexp* → *factor*, el atributo *name*, además del atributo *tacode*, se transportan de hijos a padres y que, en el caso de los nodos interiores del operador, se generan nuevos atributos *name* antes del *tacode* asociado. Observe también que, en las producciones de hoja *factor* → **num** y *factor* → **id**, el valor de cadena del token se utiliza como *factor.name*, y que (a diferencia

Tabla 8.2
Gramática con atributos para código de tres direcciones como un atributo de cadena sintetizado

| Regla gramatical | Reglas semánticas |
|---|---|
| $exp_1 \rightarrow$ `id` $= exp_2$ | $exp_1.name = exp_2.name$<br>$exp_1.tacode = exp_2.tacode$ ++ `id`$.strval \,\|\|\, $ "=" $\,\|\|\, exp_2.name$ |
| $exp \rightarrow aexp$ | $exp.name = aexp.name$<br>$exp.tacode = aexp.tacode$ |
| $aexp_1 \rightarrow aexp_2 + factor$ | $aexp_1.name = newtemp()$<br>$aexp_1.tacode = aexp_2.tacode$ ++ $factor.tacode$ ++ $aexp_1.name \,\|\|$ "=" $\,\|\| aexp_2.name \,\|\|$ "+" $\,\|\| factor.name$ |
| $aexp \rightarrow factor$ | $aexp.name = factor.name$<br>$aexp.tacode = factor.tacode$ |
| $factor \rightarrow$ ( $exp$ ) | $factor.name = exp.name$<br>$factor.tacode = exp.tacode$ |
| $factor \rightarrow$ `num` | $factor.name =$ `num`$.strval$<br>$factor.tacode =$ "" |
| $factor \rightarrow$ `id` | $factor.name =$ `id`$.strval$<br>$factor.tacode =$ "" |

del código P) no se genera ningún código de tres direcciones en tales nodos (utilizamos "" para representar la cadena vacía).

De nueva cuenta, dejamos al lector mostrar que, dadas las ecuaciones de atributo de la tabla 8.2, la expresión `(x=x+3)+4` tiene el atributo *tacode*

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

(Esto supone que *newtemp*( ) es llamado en postorden y genera nombres temporales comenzando con `t1`.) Advierta cómo la asignación `x=x+3` genera dos instrucciones de tres direcciones utilizando un elemento temporal. Esto es una consecuencia del hecho de que la evaluación de atributos siempre crea un elemento temporal para cada subexpresión, incluyendo los lados derechos de las asignaciones.

Visualizar la generación de código como el cálculo de un atributo de cadena sintetizado es útil para mostrar claramente las relaciones entre las secuencias de código de las diferentes partes del árbol sintáctico y para comparar los diferentes métodos de generación de código, pero es poco práctico como técnica para generación de código real, por varias razones. En primer lugar, el uso de la concatenación de cadena causa que se desperdicie una cantidad desmesurada de copiado de cadenas y memoria a menos que los operadores de la concatenación sean muy complejos. En segundo, por lo regular es mucho más deseable generar pequeños segmentos de código a medida que continúa la generación de código y escribir estos segmentos en un archivo o bien insertarlos en una estructura de datos (tal como un arreglo de cuádruples), lo que requiere acciones semánticas que no se adhieren a la síntesis postorden estándar de atributos. Finalmente, aunque es útil visualizar el código como puramente sintetizado, la generación de código en general depende mucho de atributos heredados, y esto complica en gran medida las gramáticas con atributos. Es por

esto que no nos preocupamos aquí por escribir ningún código (incluso pseudocódigo) para implementar las gramáticas con atributos de los ejemplos anteriores (pero vea los ejercicios). En vez de eso, en la siguiente subsección regresaremos a técnicas de generación de código más directas.

## 8.2.2 Generación de código práctica

Las técnicas estándar de generación de código involucran modificaciones de los recorridos postorden del árbol sintáctico implicado por las gramáticas con atributos de los ejemplos precedentes o, si no se genera un árbol sintáctico de manera explícita, acciones equivalentes durante un análisis sintáctico. El algoritmo básico puede ser descrito com o el siguiente procedimiento recursivo (para nodos de árbol con dos hijos como máximo, pero fácilmente extensibles a más):

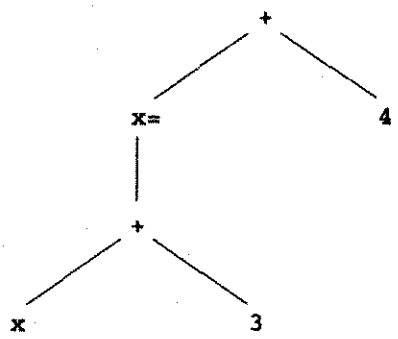
```
procedure genCode ( T: treenode );
begin
  if T no es nil then
    genere código para preparar en el caso del código del hijo izquierdo de T ;
    genCode(hijo izquierdo de T) ;
    genere código para preparar en el caso del código del hijo derecho de T ;
    genCode(hijo derecho de T) ;
    genere código para implementar la acción de T ;
end;
```

Observe que este procedimiento recursivo no sólo tiene un componente postorden (que genera código para implementar la acción de *T*) sino también un componente de preorden y un componente enorden (que genera el código de preparación para los hijos izquierdo y derecho de *T*). En general, cada acción que representa *T* requerirá una versión un poco diferente del código de preparación preorden y enorden.

Para ver de manera detallada cómo se puede construir el procedimiento *genCode* en un ejemplo específico considere la gramática para expresiones aritméticas simples que hemos estado utilizando en esta sección (véase la gramática en la página 407). Las definiciones en C para un árbol sintáctico abstracto de esta gramática se pueden proporcionar de la manera siguiente (compare con las de la página 111 del capítulo 3):

```
typedef enum {Plus,Assign} Optype;
typedef enum {OpKind,ConstKind,IdKind} NodeKind;
typedef struct streenode
      { NodeKind kind;
        Optype op; /* usado con OpKind */
        struct streenode *lchild,*rchild;
        int val; /* usado con ConstKind */
        char * strval;
          /* usado para identificadores y números */
      } STreeNode;
typedef STreeNode *SyntaxTree;
```

Con estas definiciones se puede dar un árbol sintáctico para la expresión `(x=x+3)+4` como se ve en seguida:




Advierta que el nodo designación contiene el identificador que se está asignando (en el campo **strval**), así que un nodo designación tiene solamente un hijo (la expresión que se asigna).[4]

Basados en esta estructura para un árbol sintáctico podemos escribir un procedimiento *genCode* para generar código P como se da en la figura 8.7. En esa figura haremos los comentarios siguientes acerca del código. En primer lugar, el código utiliza la función estándar de C **sprintf** para concatenar las cadenas en el elemento temporal local **codestr**. En segundo lugar se llama el procedimiento **emitCode** para generar una línea simple de código P, ya sea en una estructura de datos o en un archivo de salida; sus detalles no se muestran. Finalmente, los dos casos de operador (**Plus** y **Assign**) requieren dos órdenes diferentes de recorridos: **Plus** necesita sólo procesamiento de postorden, mientras que **Assign** requiere cierto procesamiento tanto preorden como postorden. De este modo, las llamadas recursivas pueden no escribirse del mismo modo para todos los casos.

Para mostrar que la generación de código, incluso con la variación necesaria en orden de recorrido, todavía se puede realizar durante un análisis sintáctico (sin la generación de un árbol sintáctico), mostramos un archivo de especificación Yacc en la figura 8.8 que corresponde directamente al código de la figura 8.7. (Observe cómo el procesamiento combinado preorden y postorden de las asignaciones se traduce en secciones de acción dividida en la especificación Yacc.)

Dejamos al lector escribir un procedimiento *genCode* y especificación Yacc para generar el código de tres direcciones como se especificó en la gramática con atributos de la tabla 8.2.

Figura 8.7
Implementación de un procedimiento de generación de código para código P correspondiente a la gramática con atributos de la tabla 8.1

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  /* CODESIZE = longitud máxima de 1 línea de código P  */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            genCode(t->lchild);
            genCode(t->rchild);
            emitCode("adi");
            break;
```

4. También almacenamos números como cadenas en el campo **strval** en este ejemplo.

Figura 8.7 Continuación

```
            case Assign:
              sprintf(codestr,"%s %s",
                                "lda",t->strval);
              emitCode(codestr);
              genCode(t->lchild);
              emitCode("stn");
              break;
            default:
              emitCode("Error");
              break;
          }
          break;
        case ConstKind:
          sprintf(codestr,"%s %s","ldc",t->strval);
          emitCode(codestr);
          break;
        case IdKind:
          sprintf(codestr,"%s %s","lod",t->strval);
          emitCode(codestr);
          break;
        default:
          emitCode("Error");
          break;
      }
    }
}
```

Figura 8.8
Especificación Yacc para la generación de código P de acuerdo con la gramática con atributos de la tabla 8.1

```
%{
#define YYSTYPE char *
    /* hace que Yacc utilice cadenas como valores */

/* otro código de inclusión ... */
%}

%token NUM ID

%%

exp     : ID
          { sprintf(codestr,"%s %s","lda",$1);
            emitCode(codestr); }
          '=' exp
          { emitCode("stn"); }
        | aexp
        ;
```

Figura 8.8 Continuación

```
aexp    : aexp '+' factor {emitCode("adi");}
        | factor
        ;

factor  : '(' exp ')'
        | NUM       { sprintf(codestr,"%s %s","ldc",$1);
                      emitCode(codestr); }
        | ID        { sprintf(codestr,"%s %s","lod",$1);
                      emitCode(codestr); }
        ;

%%
/* funciones de utilería ... */
```

## 8.2.3 Generación de código objetivo a partir del código intermedio

Si un compilador genera código intermedio, ya sea directamente durante un análisis sintáctico o desde un árbol sintáctico, entonces debe efectuarse otro paso en el código intermedio para generar el código objetivo final (por lo regular después de algún procesamiento adicional del código intermedio). Este paso puede ser bastante complejo por sí mismo, en particular si el código intermedio es muy simbólico y contiene poca o ninguna información acerca de la máquina objetivo o el ambiente de ejecución. En este caso, el paso de la generación de código final debe suministrar todas las ubicaciones reales de variables y temporales, más el código necesario para mantener el ambiente de ejecución. Una cuestión particularmente importante es la asignación apropiada de los registros y el mantenimiento de la información sobre el uso de los mismos (es decir, cuáles registros se encuentran disponibles y cuáles contienen valores conocidos). Aplazaremos un análisis detallado de tales cuestiones de asignación hasta más adelante en este capítulo. Por ahora comentaremos sólo técnicas generales para este proceso.

Por lo regular la generación de código a partir del código intermedio involucra alguna o ambas de las dos técnicas estándar: expansión de macro y simulación estática. La **expansión de macro** involucra el reemplazo de cada clase de instrucción del código intermedio con una secuencia equivalente de instrucciones de código objetivo. Esto requiere que el compilador se mantenga al tanto de las decisiones acerca de ubicaciones e idiomas de código en estructuras de datos separadas y que los procedimientos del macro varíen la secuencia de código como se requiera mediante las clases particulares de datos involucradas en la instrucción de código intermedio. De este modo, cada paso puede ser mucho más complejo que las formas simples de expansión de macro disponibles del preprocesador de C o ensambladores de macro. La **simulación estática** involucra una simulación en línea recta de los efectos del código intermedio y generación de código objetivo para igualar estos efectos. Esto también requiere de más estructuras de datos, y puede variar de formas muy simples de seguimiento empleadas en conjunto con la expansión de macro, hasta la altamente

sofisticada **interpretación abstracta** (que mantiene los valores algebraicamente a medida que son calculados).

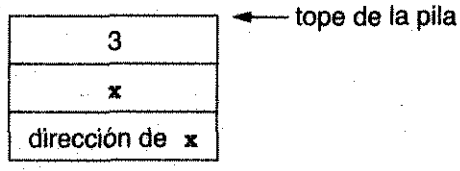
Podemos obtener alguna idea de los detalles de estas técnicas considerando el problema de traducir desde el código P al código de tres direcciones y viceversa. Consideremos la pequeña gramática de expresión que hemos estado utilizando como ejemplo de ejecución en esta sección, y consideremos la expresión `(x=x+3)+4`, cuyas traducciones en código P y código de tres direcciones se proporcionaron en las páginas 408 y 409, respectivamente. Consideremos primero la traducción del código P para esta expresión:

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

en su correspondiente código de tres direcciones:

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

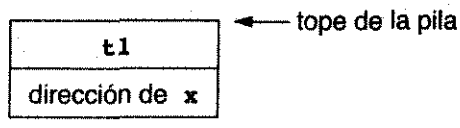
Esto requiere que realicemos una simulación estática de la pila de máquina P para encontrar equivalentes de tres direcciones del código dado. Hacemos esto con una estructura de datos de pila real durante la traducción. Después de las primeras tres instrucciones de código P, no se han generado todavía instrucciones de tres direcciones, pero la pila de máquina P se ha modificado para reflejar las cargas, y la pila tiene el aspecto siguiente:




Ahora, cuando se procesa la operación `adi`, se genera la instrucción de tres direcciones

```
t1 = x + 3
```

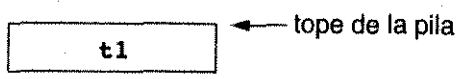
y la pila se cambia a



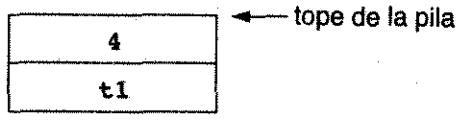

La instrucción `stn` provoca entonces que se genere la instrucción de tres direcciones

```
x = t1
```

y la pila se cambie a




La instrucción siguiente inserta la constante 4 en la pila:




Finalmente, la instrucción **adi** provoca que se genere la instrucción de tres direcciones

```
t2 = t1 + 4
```

y la pila se cambie a


Esto completa la simulación estática y la traducción.

Ahora consideraremos el caso de traducir del código de tres direcciones al código P. Si ignoramos la complicación agregada de los nombres temporales, esto puede hacerse mediante expansión simple de macro. Por consiguiente, una instrucción de tres direcciones

```
a = b + c
```

siempre se puede traducir en la secuencia de código P

```
lda a
lod b ; o ldc b si b es una constante
lod c ; o ldc c si c es una constante
adi
sto
```

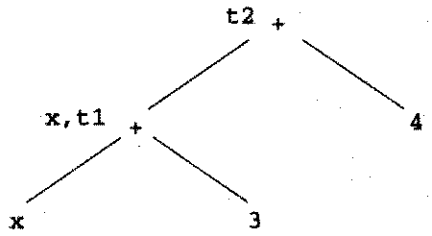
Esto resulta en la siguiente traducción (algo insatisfactoria) del anterior código de tres direcciones en código P:

```
lda  t1
lod  x
ldc  3
adi
sto
lda  x
lod  t1
sto
lda  t2
```

*(continúa)*

```
lod  t1
ldc  4
adi
sto
```

Si queremos eliminar los elementos temporales extra, entonces debemos utilizar un esquema más sofisticado que la expansión de macro pura. Una posibilidad es generar un nuevo árbol a partir del código de tres direcciones, indicando el efecto que el código tiene al etiquetar los nodos del árbol tanto con el operador de cada instrucción como con el nombre que se le asigna. Esto puede visualizarse como una forma de simulación estática, y el árbol resultante para el anterior código de tres direcciones es




Advierta cómo la instrucción de tres direcciones

```
x = t1
```

no provoca que se creen nodos extra en este árbol, pero sí que el nodo con nombre `t1` adquiera el nombre adicional `x`. Este árbol es similar, pero no idéntico, al árbol sintáctico de la expresión original (véase la página 411).[5] El código P se puede generar a partir de este árbol de manera muy semejante a como se genera el código P a partir de un árbol sintáctico, de la manera antes descrita, pero con elementos temporales eliminados al hacer asignaciones sólo a nombres permanentes de nodos interiores. De este modo, en el árbol de muestra, sólo se asigna `x`, los nombres `t1` y `t2` nunca se emplean en el código P generado, y el valor correspondiente al nodo raíz (con nombre `t2`) se deja en la pila de la máquina P. Esto produce exactamente la misma generación de código P que antes, aunque se utiliza `stn` en lugar de `sto` siempre que se realiza un almacenamiento. Alentamos al lector a escribir pseudocódigo o código en C para llevar a cabo este proceso.

## 8.3 GENERACIÓN DE CÓDIGO DE REFERENCIAS DE ESTRUCTURAS DE DATOS

### 8.3.1 Cálculos de direcciones

En la sección anterior vimos cómo se puede generar el código intermedio para asignaciones y expresiones aritméticas simples. En estos ejemplos todos los valores básicos eran constantes, o bien, variables simples (ya sea variables de programa tales como `x`, o temporales como `t1`). Las variables simples eran identificadas sólo por nombre: la traducción a código

5. Este árbol es un caso especial de una construcción más general denominada **DAG de un bloque básico**, la cual se describe en la sección 8.9.3.

objetivo requiere que estos nombres sean reemplazados por direcciones reales, las cuales podrían ser registros, direcciones de memoria absoluta (para globales), o desplazamientos de registro de activación (para locales, que posiblemente incluyan un nivel de anidación). Estas direcciones se pueden insertar en el momento en que es generado el código intermedio o aplazar la inserción hasta que se genere el código real (con la tabla de símbolos manteniendo las direcciones).

Sin embargo, existen muchas situaciones que requieren que se realicen los cálculos de las direcciones para localizar la dirección real en cuestión, y estos cálculos deben ser expresados directamente, incluso en código intermedio. Tales cálculos se presentan en subíndice de arreglo, campo de registro y referencias de apuntador. Comentaremos cada uno de estos casos por turno. Pero debemos comenzar por describir extensiones para el código de tres direcciones y el código P que nos permitan expresar tales cálculos de direcciones.

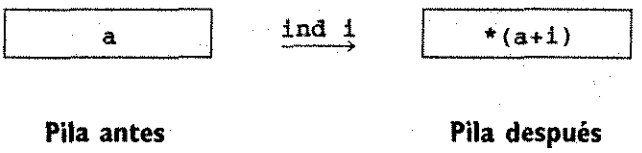
*Código de tres direcciones para cálculos de direcciones* En el código de tres direcciones, lo que se necesita no es tanto nuevas operaciones (las operaciones aritméticas habituales se pueden utilizar para calcular direcciones) sino maneras para indicar los modos de direccionamiento "dirección de" e "indirecto". En nuestra versión del código en tres direcciones utilizaremos la notación equivalente en C "&" y "*" para indicar estos modos de direccionamiento. Por ejemplo, supongamos que deseamos almacenar el valor constante 2 en la dirección de la variable **x** más 10 bytes. Expresaríamos esto en código de tres direcciones de la manera siguiente:

```
t1 = &x + 10
*t1 = 2
```

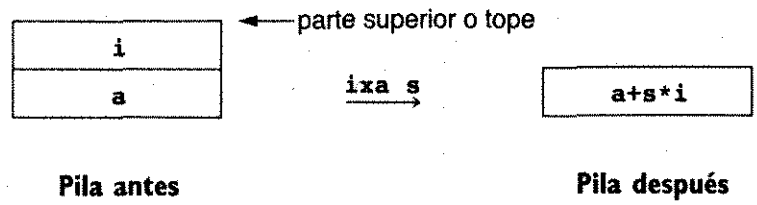
La implementación de estos nuevos modos de direccionamiento requiere que la estructura de datos para el código de tres direcciones contenga un nuevo campo o campos. Por ejemplo, la estructura de datos cuádruple de la figura 8.4 (página 403) se puede aumentar mediante un campo enumerado `AddrMode` con posibles valores `None`, `Address` e `Indirect`.

*Código P para cálculos de direcciones* En código P es común introducir nuevas instrucciones para expresar nuevos modos de direccionamiento (puesto que hay pocas direcciones explícitas a las cuales asignar modos de direccionamiento). Las dos instrucciones que introduciremos para este propósito son las siguientes:

1. `ind` ("indirect load"), que toma como parámetro un desplazamiento entero, supone que hay una dirección en la parte superior o tope de la pila, agrega el desplazamiento a la dirección y reemplaza a esta última en la pila con el valor de la ubicación resultante:




2. `ixa` ("indexed address"), la cual toma como parámetro un factor de escala entero, supone que se tiene un desplazamiento en la parte superior o tope de la pila y una dirección base debajo del mismo, multiplica el desplazamiento por el factor de escala, agrega la dirección base, extrae tanto el desplazamiento como la base de la pila e inserta la dirección resultante:




Estas dos instrucciones de código P, junto con la instrucción `lda` (load address) que se presentó anteriormente, permitirán realizar los mismos cálculos de dirección y las referencias a los modos de direccionamiento que para el código de tres direcciones.[6] Por ejemplo, el problema de muestra anterior (almacenar el valor constante 2 en la dirección de la variable `x` más 10 bytes) ahora se puede resolver en código P de la manera que se presenta a continuación:

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

Ahora volveremos a un análisis de arreglos, registros y apuntadores, seguido por un análisis de generación de código objetivo y un ejemplo extendido.

## 8.3.2 Referencias de arreglo

Una referencia de arreglo involucra la subindización de una variable de arreglo mediante una expresión para obtener una referencia o valor de un simple elemento del arreglo, como en el código en C

```
int a[SIZE]; int i,j;

...
a[i+1] = a[j*2] + 3;
```

En esta asignación la subindización de `a` mediante la expresión `i+1` produce una dirección (el objetivo de la asignación), mientras que la subindización de `a` mediante la expresión `j*2` produce el valor de la dirección calculada del tipo de elemento de `a` (a saber, `int`). Como los arreglos son almacenados de manera secuencial en la memoria, cada dirección debe ser calculada desde la **dirección base** de `a` (su dirección de inicio en memoria) y un desplazamiento que depende linealmente del valor del subíndice. Cuando se deseara obtener el valor más que la dirección, se debe generar un paso de indirección extra para obtener el valor de la dirección calculada.

El desplazamiento es calculado a partir del valor de subíndice de la manera siguiente. En primer lugar, debe hacerse un ajuste al valor del subíndice si el intervalo de subíndice no comienza en 0 (esto podría ocurrir en lenguajes como Pascal y Ada, pero no en C). En segundo lugar, el valor del subíndice ajustado debe ser multiplicado por un **factor de escala**

6. De hecho, la instrucción `ixa` podría ser simulada mediante operaciones aritméticas, sólo que en el código P estas operaciones son tipeadas (`adi` = multiplicación entera sólo) y de este modo no se pueden aplicar a direcciones. No insistimos en las limitantes de tipo del código P porque involucra parámetros extra que hemos suprimido por simplicidad.

que es igual al tamaño de cada uno de los elementos del arreglo en la memoria. Finalmente, el subíndice escalado resultante se agrega a la dirección base para obtener la dirección final del elemento del arreglo.

Por ejemplo, la dirección de la referencia de arreglo en C `a[i+1]` es[7]

```
a + (i + 1) * sizeof(int)
```

De manera más general, la dirección de un elemento de arreglo `a[t]` en cualquier lenguaje es

$$\textit{dirección\_base}(\texttt{a}) + (t - \textit{límite\_inferior}(\texttt{a})) * \textit{tamaño\_elemento}(\texttt{a})$$

Ahora volveremos a las maneras de expresar este cálculo de dirección en el código de tres direcciones y en el código P. Para hacer esto en una notación independiente de la máquina objetivo supondremos que la "dirección" de una variable de arreglo es su dirección base. De este modo, si `a` es una variable de arreglo, `&a` en código de tres direcciones es lo mismo que *dirección_base*(`a`), y en código P

```
lda a
```

carga la dirección base de `a` en la pila de máquina P. También, puesto que un cálculo de referencia de arreglo depende del tamaño del tipo de datos del elemento en la máquina objetivo, utilizaremos la expresión `elem_size(a)` para el tamaño de elemento del arreglo `a` en la máquina objetivo.[8] Como ésta es una cantidad estática (suponiendo tipeado estático), esta expresión será reemplazada por una constante en tiempo de compilación.

*Código de tres direcciones para las referencias de arreglo* Una manera en que se pueden expresar referencias de arreglo en código de tres direcciones es introducir dos nuevas operaciones, una que obtenga el valor de un elemento de arreglo

```
t2 = a[t1]
```

y otra que asigne la dirección de un elemento de arreglo

```
a[t2] = t1
```

(esto podría ser dado por los símbolos `=[]` y `[]=`). Si se utiliza esta terminología no es necesario expresar el cálculo de dirección real (y las dependencias de máquina tales como el tamaño del elemento desaparecen de esta notación). Por ejemplo, la sentencia de código fuente

```
a[i+1] = a[j*2] + 3;
```

se traduciría en las instrucciones de tres direcciones

```
t1 = j * 2
t2 = a[t1]
t3 = t2 + 3
t4 = i + 1
a[t5] = t3
```

7. En C, el nombre de un arreglo (tal como `a` en esta expresión) representa su propia dirección base.

8. Esto podría ser de hecho una función suministrada por la tabla de símbolos.

Sin embargo, todavía es necesario introducir modos de direccionamiento como los anteriormente descritos cuando tratamos con referencias de apuntador y campo de registro, de modo que tenga sentido tratar todos esos cálculos de dirección de manera uniforme. De esta manera, también podemos escribir los cálculos de direcciones de un elemento de arreglo directamente en código de tres direcciones. Por ejemplo, la asignación

```
t2 = a[t1]
```

también puede escribirse como (utilizando elementos temporales adicionales `t3` y `t4`)

```
t3 = t1 * elem_size(a)
t4 = &a + t3
t2 = *t4
```

y la asignación

```
a[t2] = t1
```

puede escribirse como

```
t3 = t2 * elem_size(a)
t4 = &a + t3
*t4 = t1
```

Finalmente, como un ejemplo más complejo, la sentencia de código fuente

```
a[i+1] = a[j*2] + 3;
```

se traduce en las instrucciones de tres direcciones

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3
t5 = t4 + 3
t6 = i + 1
t7 = t6 * elem_size(a)
t8 = &a + t7
*t8 = t5
```

*Código P para referencias de arreglo* Como se describió anteriormente, utilizaremos las nuevas instrucciones de dirección `ind` e `ixa`. La instrucción `ixa` de hecho fue construida precisamente con los cálculos de dirección de arreglo en mente, mientras que la instrucción `ind` se utiliza para cargar el valor de una dirección calculada con anterioridad (es decir, para implementar una carga indirecta). La referencia de arreglo

```
t2 = a[t1]
```

se escribe en código P como

```
lda t2
lda a
lod t1
ixa elem_size(a)
ind 0
sto
```

y la asignación de arreglo

```
a[t2] = t1
```

se escribe en código P como

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

Finalmente, el ejemplo anterior más complejo

```
a[i+1] = a[j*2] + 3;
```

se traduce en las siguientes instrucciones de código P:

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

*Un procedimiento para la generación de código con referencias de arreglo* Mostraremos aquí cómo las referencias de arreglo se pueden generar mediante un procedimiento de generación de código. Utilizaremos el ejemplo del subconjunto de expresiones C de la sección anterior (véase la gramática de la página 407), aumentado mediante una operación de subíndice. La nueva gramática que utilizaremos es la siguiente:

*exp* → *subs* = *exp* | *aexp*
*aexp* → *aexp* + *factor* | *factor*
*factor* → ( *exp* ) | **num** | *subs*
*subs* → **id** | **id** [ *exp* ]

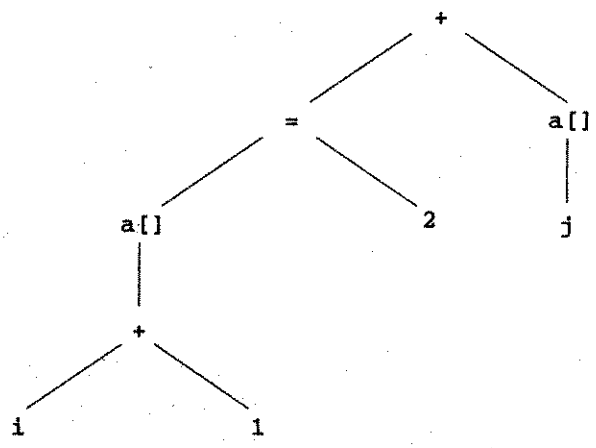
Advierta que el objetivo de una asignación ahora puede ser una variable simple o una variable subindizada (ambas incluidas en el no terminal *subs*). Utilizamos la misma estructura de datos para el árbol sintáctico que antes (véanse las declaraciones en la página 410), sólo que ahora existe una operación adicional **Subs** para la subindización:

```
typedef enum {Plus,Assign,Subs} Optype;
/* otras declaraciones como antes */
```

Además, como ahora puede estar una expresión de subíndice a la izquierda de una asignación, no es posible almacenar el nombre de la variable objetivo en el mismo nodo de asignación (ya que puede no haber tal nombre). En vez de eso, los nodos de asignación ahora tienen dos hijos como nodos extra: el hijo izquierdo debe ser un identificador o una expresión de subíndice. Los subíndices mismos sólo pueden ser aplicados a identificadores, de modo que almacenamos el nombre de la variable de arreglo en nodos subíndice. Así, el árbol sintáctico para la expresión

```
(a[i+1]=2)+a[j]
```

es




Un procedimiento de generación de código que produce código P para tales árboles sintácticos se proporciona en la figura 8.9 (compare con la figura 8.7, página 411). La diferencia principal entre este código y el de la figura 8.7 es que éste necesita tener un atributo heredado **isAddr** que distinga un identificador o expresión con subíndice a la izquierda de una asignación a partir de uno a la derecha. Si **isAddr** se establece a **TRUE**, entonces la *dirección* de la expresión debe ser devuelta; de otro modo, se devuelve el *valor*. Dejamos al lector verificar que este procedimiento genera el siguiente código P para la expresión `(a[i+1]=2)+a[j]`:

```
lda a
lod i
```

```
ldc 1
adi
ixa elem_size(a)
ldc 2
stn
lda a
lod j
ixa elem_size(a)
ind 0
adi
```

También dejamos al lector la construcción de un generador de código de tres direcciones para esta gramática (véanse los ejercicios).

Figura 8.9
Implementación de un procedimiento de generación de código para el código P correspondiente a la gramática de expresión de la página 422

```
void genCode( SyntaxTree t, int isAddr)
{ char codestr[CODESIZE];
  /* CODESIZE = longitud máxima de 1 línea de código P */
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            if (isAddr) emitCode("Error");
            else { genCode(t->lchild,FALSE);
                   genCode(t->rchild,FALSE);
                   emitCode("adi");}
            break;
          case Assign:
            genCode(t->lchild,TRUE);
            genCode(t->rchild,FALSE);
            emitCode("stn");
            break;
          case Subs:
            sprintf(codestr,"%s %s","lda",t->strval);
            emitCode(codestr);
            genCode(t->lchild,FALSE);
            sprintf(codestr,"%s%s%s",
               "ixa elem_size(",t->strval,")");
            emitCode(codestr);
            if (!isAddr) emitCode("ind 0");
            break;
          default:
            emitCode("Error");
            break;
        }
        break;
      case ConstKind:
```

Figura 8.9 Continuación

```
            if (isAddr) emitCode("Error");
            else
            { sprintf(codestr,"%s %s","ldc",t->strval);
              emitCode(codestr);
            }
            break;
          case IdKind:
            if (isAddr)
              sprintf(codestr,"%s %s","lda",t->strval);
            else
              sprintf(codestr,"%s %s","lod",t->strval);
            emitCode(codestr);
            break;
          default:
            emitCode("Error");
            break;
        }
    }
}
```

*Arreglos multidimensionales* Un factor que complica el cálculo de las direcciones de arreglo es la existencia, en la mayoría de los lenguajes, de arreglos en varias dimensiones. Por ejemplo, en C un arreglo de dos dimensiones (con diferentes tamaños de índice) puede declararse como

```
int a[15][10];
```

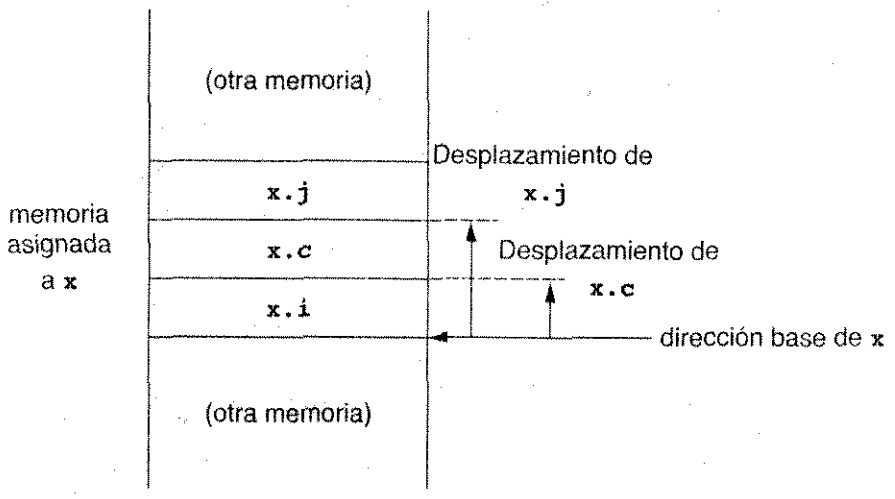
Tales arreglos pueden estar parcialmente subindizados, produciendo un arreglo de menos dimensiones, o subindizados por completo, produciendo un valor del tipo elemento del arreglo. Por ejemplo, dada la declaración anterior de `a` en C, la expresión `a[i]` implica subindización de manera parcial de `a`, produciendo un arreglo unidimensional de enteros, mientras que la expresión `a[i][j]` implicala subindización completa de `a` y produce un valor de tipo entero. La dirección de una variable de arreglo subindizada de manera parcial o completa se puede calcular mediante la aplicación recursiva de las técnicas recién descritas para el caso unidimensional.

## 8.3.3 Estructura de registro y referencias de apuntador

El cálculo de la dirección de un campo de estructura o registro presenta un problema similar al de calcular una dirección de arreglo subindizada. En primer lugar se calcula la dirección base de la variable de estructura; luego se encuentra el desplazamiento (generalmente fijo) del campo nombrado, y se suman los dos para obtener la dirección resultante. Considere, por ejemplo, las declaraciones en C

```
typedef struct rec
   { int i;
     char c;
     int j;
   } Rec;
...
Rec x;
```

Por lo general, la variable `x` está asignada en la memoria como lo muestra la siguiente ilustración, con cada campo (`i`, `c` y `j`) teniendo un desplazamiento desde la dirección base de `x` (la cual puede ser en sí misma un desplazamiento en un registro de activación):




Advierta que los campos están asignados linealmente (por lo regular desde la dirección más baja a la más alta), que el desplazamiento de cada campo es una constante, y que el primer campo (`x.i`) tiene desplazamiento cero. Advierta también que los desplazamientos de los campos dependen del tamaño de los diversos tipos de datos en la máquina objetivo, pero que no se encuentra involucrado un factor de escala como con los arreglos.

Para escribir código intermedio independiente del objetivo para cálculos de dirección de campo de estructura de registro, debemos introducir una nueva función que devuelva el desplazamiento en campo, dada una variable de estructura y el nombre del campo. Llamaremos a esta función `field_offset` y la escribiremos con dos parámetros, el primero será el nombre de la variable y el segundo el nombre del campo. De este modo, `field_offset (x,j)` devuelve el desplazamiento de `x.j`. Como con otras funciones similares, esta función puede ser suministrada mediante la tabla de símbolos. En cualquier caso, ésta es una cantidad en tiempo de compilación, de modo que el código intermedio real generado tendrá casos de llamadas a `field_offset` reemplazados por constantes.

Las estructuras de registro generalmente son utilizadas junto con apuntadores y asignación de memoria dinámica para implementar estructuras de datos dinámicas tales como listas y árboles. Así, describimos también cómo interactúan los apuntadores con cálculos de dirección de campo. Un apuntador, para los propósitos de esta exposición, establece simplemente un nivel adicional de indirección; pasamos por alto las cuestiones de asignación involucradas en la creación de valores de apuntador (éstas fueron comentadas en el capítulo anterior).

*Código de tres direcciones para las referencias de apuntador y estructura* Consideremos, en primer lugar, el código de tres direcciones para el cálculo de direcciones de campo: para calcular la dirección de `x.j` en un elemento temporal `t1` utilizamos la instrucción de tres direcciones

```
t1 = &x + field_offset(x,j)
```

Una asignación de campo tal como la sentencia de C

```
x.j = x.i;
```

puede ser traducida en el código de tres direcciones

```
t1 = &x + field_offset(x,j)
t2 = &x + field_offset(x,i)
 *t1 = *t2
```

Ahora consideremos los apuntadores. Supongamos, por ejemplo, que **x** es declarada como un apuntador para un entero, por ejemplo mediante la declaración en C

```
int * x;
```

Supongamos además que **i** es una variable entera normal. Entonces la asignación en C

```
*x = i;
```

puede ser traducida previamente en la instrucción de tres direcciones

```
*x = i
```

y la asignación

```
i = *x;
```

en la instrucción de tres direcciones

```
i = *x
```

Para ver cómo interactúa la indirección de apuntadores con los cálculos de dirección de campo consideremos el ejemplo siguiente de una estructura de datos de árbol y declaración de variable en C:

```
typedef struct treeNode
  { int val;
    struct treeNode * lchild, * rchild;
  } TreeNode;
...
TreeNode *p;
```

Ahora consideremos dos asignaciones típicas

```
p->lchild = p;
p = p->rchild;
```

Estas sentencias se producen en el código de tres direcciones

```
t1 = p + field_offset(*p,lchild)
*t1 = p
t2 = p + field_offset(*p,rchild)
p = *t2
```

*Código P para referencias de apuntador y estructura* Dada la declaración de **x** al principio de esta exposición (página 423), se puede hacer un cálculo directo de la dirección de **x.j** en código P de la manera siguiente:

```
lda x
lod field_offset(x,j)
ixa 1
```

La sentencia de asignación

```
x.j = x.i;
```

puede ser traducida en el código P

```
lda x
lod field_offset(x,j)
ixa 1
lda x
ind field_offset(x,i)
sto
```

Advierta cómo se utiliza la instrucción **ind** para obtener el valor de **x.i** sin calcular primero su dirección completa.

En el caso de los apuntadores (con **x** declarada como una **int***), la asignación

```
*x = i;
```

se traduce al código P

```
lod x
lod i
sto
```

y la asignación

```
i = *x;
```

se traduce al código P

```
lda i
lod x
ind 0
sto
```

Concluimos con el código P para las asignaciones

```
p->lchild = p;
p = p->rchild;
```

(véase la declaración de **p** en la página anterior). Esto se traduce al siguiente código P:

```
lod p
lod field_offset(*p,lchild)
ixa 1
lod p
sto
lda p
lod p
ind field_offset(*p,rchild)
sto
```

Dejamos los detalles de un procedimiento de generación de código que generará estas secuencias de código ya sea en código de tres direcciones o en código P para los ejercicios.

## 8.4 GENERACIÓN DE CÓDIGO DE SENTENCIAS DE CONTROL Y EXPRESIONES LÓGICAS

En esta sección describiremos la generación de código para varias formas de sentencias de control. Entre éstas se encuentran principalmente la sentencia if y la sentencia while estructuradas, las que veremos con detalle en la primera parte de esta sección. También incluimos en esta descripción el uso de la sentencia break (como en C), pero no comentaremos el control de bajo nivel como el de la sentencia goto, puesto que tales sentencias se pueden implementar con facilidad directamente en código intermedio o código objetivo. Otras formas de control estructurado, como la sentencia repeat (o la sentencia do-while), la sentencia for y la sentencia case (o sentencia switch), se dejan para los ejercicios. Otra técnica de implementación útil para la sentencia switch, denominada **tabla de salto**, también se describe en un ejercicio.

La generación de código intermedio para sentencias de control, tanto en código de tres direcciones como en código P, involucra la generación de **etiquetas** de una manera similar a la generación de nombres temporales en el código de tres direcciones, pero en este caso representan direcciones en el código objetivo a las que se harán los saltos. Si las etiquetas están por eliminarse en la generación de código objetivo, entonces surge un problema, ya que los saltos a localidades de código que ya no son conocidas se deben **reajustar** o reescribir retroactivamente. Comentaremos esto en la siguiente parte de esta sección.

Las expresiones lógicas, o booleanas, que se utilizan como pruebas de control, y que también se pueden emplear de manera independiente como datos, se analizan a continuación, particularmente respecto a la **evaluación de cortocircuito**, en la cual difieren de las expresiones aritméticas.

Finalmente, en esta sección presentaremos un procedimiento de generación de código de muestra en código P para sentencias if y while.

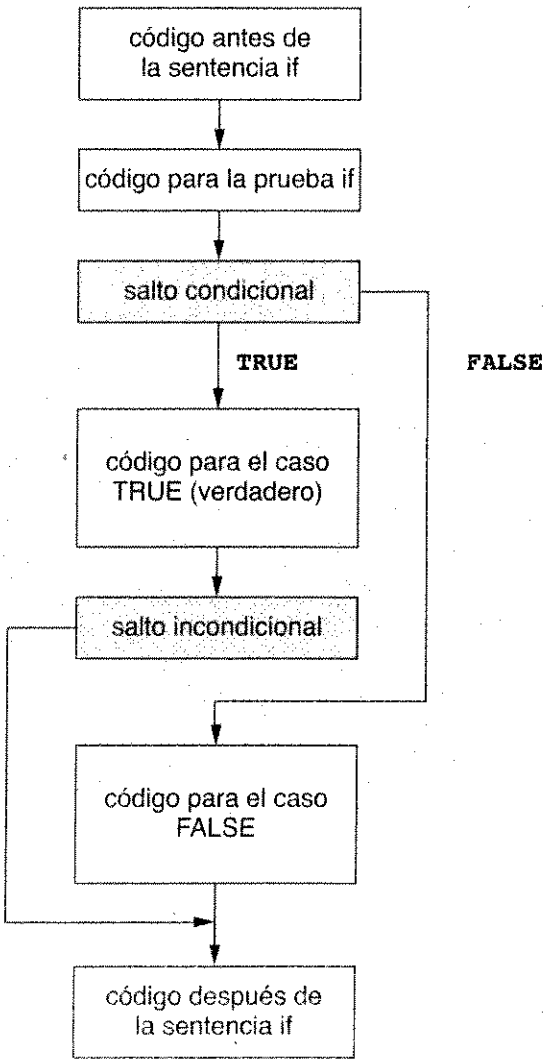
### 8.4.1 Generación de código para sentencias if y while

Consideraremos las siguientes dos formas de las sentencias if y while, que son similares en muchos lenguajes diferentes (pero que aquí daremos en una sintaxis tipo lenguaje C):

*sent-if* → `if (` *exp* `)` *sent* | `if (` *exp* `)` *sent* `else` *sent*
*sent-while* → `while (` *exp* `)` *sent*

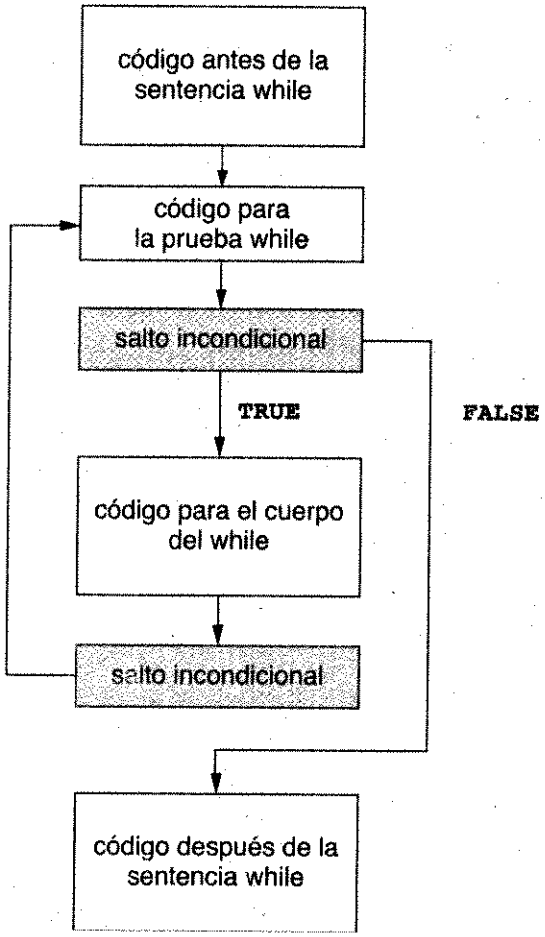
El problema principal en la generación de código para tales sentencias es traducir las características de control estructurado en un equivalente "no estructurado" que involucre saltos, los cuales se pueden implementar directamente. Los compiladores se encargan de generar código para tales sentencias en un orden estándar que permite el uso eficiente de un subconjunto de los saltos posibles que una arquitectura objetivo puede permitir. Arreglos de código típicos para cada una de estas sentencias se muestran en las figuras 8.10 (más abajo en esta misma página) y 8.11 (página 430). (La figura 8.10 muestra una parte else (el caso FALSE), pero esto es opcional, de acuerdo con la regla gramatical que se dio recientemente, y el arreglo mostrado es fácil de modificar para el caso de una parte else omitida.) En cada uno de esos arreglos existen sólo dos clases de saltos (saltos incondicionales y saltos cuando la condición es falsa) y el caso verdadero es siempre un caso "que queda en la nada" que no necesita saltos. Esto reduce el número de saltos que el compilador necesita generar. También significa que sólo son necesarias dos instrucciones de salto en el código intermedio. Los saltos falsos ya aparecieron en el código de tres direcciones de la figura 8.2 (como una instrucción `if_false..goto`) y en el código P de la figura 8.6 (como una instrucción `fjp`). Resta introducir los saltos incondicionales, los cuales escribiremos

Figura 8.10
Arreglo de código típico para una sentencia if



simplemente como instrucciones `goto` en código de tres direcciones y como el `ujp` (unconditional jump) en código P.

Figura 8.11
Arreglo de código típico para una sentencia while




*Código de tres direcciones para sentencias de control* Supondremos que el generador de código genera una secuencia de etiquetas con nombres tales como `L1, L2`, y así sucesivamente. Para la sentencia

`if` ( *E* ) *S1* `else` *S2*

se generará el siguiente patrón de código:

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

De manera similar, una sentencia while de la forma

```
while ( E ) S
```

provocaría que fuera generado el siguiente patrón de código de tres direcciones:

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

*Código P para sentencias de control* Para la sentencia

```
if ( E ) S1 else S2
```

se genera el siguiente patrón de código P:

```
<code to evaluate E>
fjp L1
<code for S1>
ujp L2
lab L1
<code for S2>
lab L2
```

y para la sentencia

```
while ( E ) S
```

se genera el siguiente patrón de código P:

```
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
```

Advierta que todas estas secuencias de código (tanto de código de tres direcciones como de código P) finalizan en una declaración de etiqueta, a la que podríamos llamar **etiqueta de salida** de la sentencia de control. Muchos lenguajes proporcionan una construcción de lenguaje que permite salir de los ciclos o bucles desde ubicaciones arbitrarias dentro del cuerpo del ciclo. Por ejemplo, C proporciona la sentencia **break** (que también se puede utilizar dentro de sentencias switch). En estos lenguajes se debe disponer de la etiqueta de salida para todas las rutinas de generación de código que se pueden llamar dentro del cuerpo de un ciclo, de manera que si se encuentra una sentencia de salida tal como un "break", se pueda generar un salto hacia la etiqueta de salida. Esto transforma a la etiqueta de salida en

un atributo heredado durante la generación del código, que debe almacenarse en una pila o bien pasarse como un parámetro a las rutinas de generación de código apropiadas. Más adelante en esta sección se ofrecerán más detalles al respecto, además de ejemplos de otras situaciones donde puede ocurrir esto último.

## 8.4.2 Generación de etiquetas y reajuste

Una característica de la generación de código para sentencias de control que puede provocar problemas durante la generación de código objetivo es el hecho de que, en algunos casos, deben generarse saltos a una etiqueta antes de que ésta se haya definido. Durante la generación de código intermedio, esto presenta pocos problemas, puesto que una rutina de generación de código simplemente puede llamar al procedimiento de generación de etiqueta cuando se necesita una para generar un salto hacia delante y grabar el nombre de la etiqueta (de manera local o en una pila) hasta que se conozca la ubicación de la etiqueta. Durante la generación del código objetivo, las etiquetas simplemente se pueden pasar a un ensamblador si se está generando código ensamblador, pero si se va a generar código ejecutable real, estas etiquetas deben transformarse en ubicaciones de código absolutas o relativas.

Un método estándar para generar tales saltos hacia delante consiste en dejar una brecha en el código donde el salto vaya a ocurrir, o bien, generar una instrucción de salto ficticia a una localidad falsa. Entonces, una vez que se conoce la ubicación del salto real, se utiliza para arreglar o **ajustar** el código omitido. Esto requiere que el código generado se mantenga en un "buffer" o "almacenamiento temporal" en la memoria, de manera que el ajuste se pueda hacer al vuelo, o que el código se escriba en un archivo temporal y después se reintroduzca y ajuste cuando sea necesario. En cualquier caso, puede ser necesario que los ajustes se almacenen de manera temporal en una pila o se mantengan localmente en procedimientos recursivos.

Durante el proceso de ajuste puede surgir otro problema, ya que muchas arquitecturas tienen dos variedades de saltos, un salto corto o ramificación (dentro de, digamos, 128 bytes de código) y un salto largo que requiere más espacio de código. En ese caso un generador de código puede necesitar insertar instrucciones `nop` cuando salte de manera breve, o hacer varios pasos para condensar el código.

## 8.4.3 Generación de código de expresiones lógicas

Hasta ahora no hemos dicho nada acerca de la generación de código para las expresiones lógicas o booleanas que se utilizan como pruebas en sentencias de control. Si el código intermedio tiene un tipo de datos booleano, y operaciones lógicas como **and** y **or**, entonces el valor de una expresión booleana se puede calcular en código intermedio exactamente de la misma manera que una expresión aritmética. Éste es el caso para el código P, y el código intermedio se puede diseñar de manera similar. Sin embargo, incluso si éste es el caso, la traducción en código objetivo por lo regular requiere que los valores booleanos se representen aritméticamente, ya que la mayoría de las arquitecturas no tienen un tipo booleano integrado. La manera estándar de hacer esto es representar el valor booleano **false** como 0 y **true** como 1. Entonces, los operadores estándar de bit **and** y **or** se pueden utilizar para calcular el valor de una expresión booleana en la mayor parte de las arquitecturas. Esto requiere que el resultado de operaciones de comparación como < se normalice a 0 o 1. En algunas arquitecturas esto requiere que 0 o 1 se carguen de manera explícita, puesto que el operador de comparación mismo sólo establece un código de condición. En ese caso necesitan generarse saltos adicionales para cargar el valor apropiado.

Si las operaciones lógicas son de **cortocircuito**, como en C, es necesario un uso adicional de los saltos. Una operación lógica es de cortocircuito si puede fallar la evaluación de su segundo argumento. Por ejemplo, si *a* es una expresión booleana que se calcula como falsa, entonces la expresión booleana *a* **and** *b* se puede determinar inmediatamente como falsa sin evaluar *b*. De manera similar, si se conoce que *a* es verdadera, entonces *a* **or** *b* se puede determinar como verdadera sin evaluar *b*. Las operaciones de cortocircuito son muy útiles para el codificador, ya que la evaluación de la segunda expresión ocasionaría un error si un operador no fuera de cortocircuito. Por ejemplo, es común escribir en C

```
if ((p!=NULL) && (p->val==0)) ...
```

donde la evaluación de `p->val` cuando `p` es nula podría causar una falla de memoria.

Los operadores booleanos de cortocircuito son semejantes a las sentencias if, sólo que éstos devuelven valores, y con frecuencia se definen utilizando **expresiones if** como

*a* **and** *b* ≡ **if** *a* **then** *b* **else false**

y

*a* **or** *b* ≡ **if** *a* **then true else** *b*

Para generar código que asegure que la segunda subexpresión será evaluada sólo cuando sea necesario, debemos emplear saltos exactamente de la misma manera que en el código para las sentencias if. Por ejemplo, el código P de cortocircuito para la expresión en C `(x!=0)&&(y==x)` es:

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
lod FALSE
lab L2
```

## 8.4.4 Un procedimiento de generación de código de muestra para sentencias if y while

En esta sección mostraremos un procedimiento de generación de código para sentencias de control utilizando la siguiente gramática simplificada:

*sent* → *sent-if* | *sent-while* | `break` | `other`
*sent-if* → `if(` *exp* `)` *sent* | `if(` *exp* `)` *sent* `else` *sent*
*sent-while* → `while(` *exp* `)` *sent*
*exp* → `true` | `false`

Por simplicidad, esta gramática utiliza el token **other** para representar sentencias no incluidas en la gramática (como la sentencia de asignación). También incluye únicamente las expresiones booleanas constantes **true** y **false**. Incluye una sentencia **break** para mostrar cómo una sentencia así se puede implementar utilizando una etiqueta heredada pasada como un parámetro.

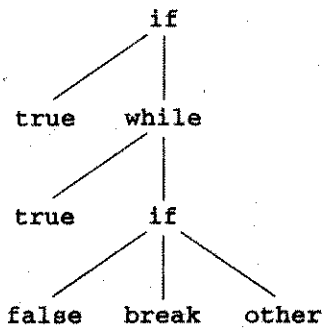
Con la siguiente declaración en C se puede implementar un árbol sintáctico abstracto para esta gramática:

```
typedef enum {ExpKind,IfKind,
      WhileKind,BreakKind,OtherKind} NodeKind;
typedef struct streenode
        { NodeKind kind;
          struct streenode * child[3];
          int val; /* usado con ExpKind */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

En esta estructura de árbol sintáctico, un nodo puede tener hasta tres hijos (un nodo if con una parte else), y los nodos de expresión son constantes con valor verdadero o falso (almacenado en el campo **val** como un 1 o un 0). Por ejemplo, la sentencia

```
if(true)while(true)if(false)break else other
```

tiene el árbol sintáctico




donde únicamente hemos mostrado los hijos no nulos de cada nodo.[9]

En la figura 8.12 se proporciona un procedimiento de generación de código que genera código P utilizando los **typedef** dados y la correspondiente estructura de árbol sintáctico. Acerca de este código hacemos las observaciones siguientes:

9. La ambigüedad del else ambiguo en esta gramática se resuelve mediante la regla estándar de la "anidación más cercana", como lo muestra el árbol sintáctico.

Figura 8.12
Procedimiento de generación de código para sentencias de control

```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
      if (t->val==0) emitCode("ldc false");
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);
      lab1 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab1);
      emitCode(codestr);
      genCode(t->child[1],label);
      if (t->child[2] != NULL)
      { lab2 = genLabel();
        sprintf(codestr,"%s %s","ujp",lab2);
        emitCode(codestr);}
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      if (t->child[2] != NULL)
      { genCode(t->child[2],label);
        sprintf(codestr,"%s %s","lab",lab2);
        emitCode(codestr);}
      break;
    case WhileKind:
      lab1 = genLabel();
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      genCode(t->child[0],label);
      lab2 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab2);
      emitCode(codestr);
      genCode(t->child[1],lab2);
      sprintf(codestr,"%s %s","ujp",lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);
      break;
    case BreakKind:
      sprintf(codestr,"%s %s","ujp",label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
  }
}
```

En primer lugar, el código supone, como antes, la existencia de un procedimiento `emitCode` (este procedimiento simplemente imprimiría la cadena que se le pase). El código también supone la existencia de un procedimiento `genLabel` sin parámetros que devuelve nombres de etiqueta en la secuencia `L1`, `L2`, `L3`, y así sucesivamente.

El procedimiento `genCode` aquí tiene un parámetro `label` extra que es necesario para generar un salto absoluto en el caso de una sentencia break. Este parámetro sólo se cambia en la llamada recursiva que procesa el cuerpo de una sentencia while. De este modo, una sentencia break siempre provocará un salto fuera de la sentencia while anidada más cercana. (La llamada inicial a `genCode` puede utilizar una cadena vacía como el parámetro `label`, y cualquier sentencia break que se encuentre fuera de una sentencia while generará entonces un salto a una etiqueta vacía y, en consecuencia, un error.)

Advierta también cómo las variables locales `lab1` y `lab2` se emplean para guardar nombres de etiqueta, para los cuales existen saltos y/o definiciones aún pendientes.

Por último, puesto que una sentencia other no corresponde a ningún código real, en este caso este procedimiento simplemente genera la instrucción en código no P "`Other`".

Dejamos al lector describir la operación del procedimiento de la figura 8.12 y mostrar que para la sentencia

```
if(true)while(true)if(false)break else other
```

se genera la secuencia de código

```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

## 8.5 GENERACIÓN DE CÓDIGO DE LLAMADAS DE PROCEDIMIENTOS Y FUNCIONES

Las llamadas de procedimientos y funciones son el último mecanismo de lenguaje que comentamos en términos generales en este capítulo. La descripción del código intermedio y del código objetivo para este mecanismo es, mucho más que otros mecanismos de lenguaje, complicada por el hecho de que diferentes máquinas objetivo utilizan mecanismos muy diferentes para realizar llamadas y por el hecho de que las llamadas dependen mucho de la organización del ambiente de ejecución. Por consiguiente, es difícil obtener una representación

de código intermedio que sea tan general como para emplearla en cualquier ambiente de ejecución o arquitectura objetivo.

## 8.5.1 Código intermedio para procedimientos y funciones

Los requerimientos para las representaciones de código intermedio de llamadas de funciones pueden describirse en términos generales como sigue. En primer lugar, existen en realidad *dos* mecanismos que necesitan descripción: la **definición** de función/procedimiento (también conocida como **declaración**) y la **llamada** de función/procedimiento.[10] Una definición crea un nombre de función, parámetros y código, pero la función no se ejecuta en ese punto. Una llamada crea valores reales para los parámetros (o **argumentos** para la llamada) y realiza un salto hacia el código de la función, el cual se ejecuta entonces y regresa. El ambiente de ejecución en el que ésta tiene lugar no se conoce cuando se crea el código para la función, excepto en su estructura general. Este ambiente de ejecución es construido en parte por el elemento que llama, y en parte por el código de la función llamada; esta división de la responsabilidad forma parte de la **secuencia de llamada** estudiada en el capítulo anterior.

El código intermedio para una definición debe incluir una instrucción que marque el inicio, o **punto de entrada**, del código para la función, y una instrucción que marque el final, o **punto de retorno**, de la función. De manera esquemática podemos escribir esto de la manera siguiente:

Instrucción de entrada
<código para el cuerpo de la función>
Instrucción de retorno

De la misma manera, una llamada de función debe tener una instrucción que indique el principio del cálculo de los argumentos (en preparación para la llamada) y luego una instrucción de llamada real que indique el punto en que los argumentos han sido construidos y el salto real hacia el código de la función puede tener lugar:

Instrucción de comienzo de cálculo de argumento
<código para calcular los argumentos>
Instrucción de llamada

Diferentes versiones del código intermedio tienen versiones muy diferentes de estas cuatro instrucciones agrupadas, en particular respecto a la cantidad de información acerca del ambiente, los parámetros y la función misma que es parte de cada instrucción. Ejemplos típicos de tal información incluyen el número, tamaño y ubicación de los parámetros; el tamaño del marco de pila; el tamaño de las variables locales y el espacio temporal, así como diversas indicaciones del uso del registro por la función llamada. Como es habitual, presentaremos código intermedio que contiene una mínima cantidad de información en las instrucciones mismas, con la idea de que cualquier información necesaria puede mantenerse separadamente en una entrada de la tabla de símbolos para el procedimiento.

---

**10.** A lo largo de este texto adoptamos el criterio de que las funciones y procedimientos representan en esencia el mismo mecanismo y, sin abundar en el tema, consideraremos que son lo mismo para los propósitos de esta sección. La diferencia es, por supuesto, que una función debe tener disponible un valor devuelto para el elemento que llama cuando salga y el elemento que llama debe saber dónde encontrarlo.

*Código de tres direcciones para procedimientos y funciones* En código de tres direcciones, la instrucción de entrada necesita dar un nombre al punto de entrada de procedimiento, semejante a la instrucción `label`; de este modo, es una instrucción de una dirección a la que llamaremos simplemente `entry`. De manera similar, llamaremos `return` a la instrucción de retorno. Esta instrucción también es una instrucción de una dirección: debe dar el nombre del valor devuelto, si existe uno.

Por ejemplo, considere la definición de la función en C

```
int f(int x, int y)
{ return x+y+1; }
```

Esto se traducirá al siguiente código de tres direcciones:

```
entry f
t1 = x + y
t2 = t1 + 1
return t2
```

En el caso de una llamada, de hecho necesitamos tres diferentes instrucciones de tres direcciones: una para señalar el inicio del cálculo del argumento, a la que llamaremos `begin_args` (y la cual es una instrucción de dirección cero); una instrucción que se utiliza de manera repetida para especificar los nombres de los valores de argumento, a la que denominaremos `arg` (y que debe incluir la dirección o nombre del valor del argumento); y finalmente, la instrucción de llamada real, que denotaremos simplemente como `call`, la cual también es una instrucción de una dirección (debe proporcionarse el nombre o punto de entrada de la función que se está llamando).

Por ejemplo, supongamos que la función `f` se definió en C como en el ejemplo anterior. Entonces, la llamada

```
f(2+3,4)
```

se traduce al código de tres direcciones

```
begin_args
t1 = 2 + 3
arg t1
arg 4
call f
```

Aquí enumeramos los argumentos en orden de izquierda a derecha. El orden podría, por supuesto, ser diferente. (Véase la sección 7.3.1, página 361.)

*Código P para procedimientos y funciones* La instrucción de entrada en código P es `ent`, y la instrucción de retorno es `ret`. De esta manera, la definición anterior de la función `f` de C se traduce al código P

```
ent f
lod x
```

```
lod y
adi
ldc 1
adi
ret
```

Advierta que la instrucción **ret** no necesita un parámetro para indicar cuál es el valor devuelto: se supone que el valor devuelto está en el tope de la pila de la máquina P al retorno.

Las instrucciones en código P para una llamada son la instrucción **mst** y la instrucción **cup**. La instrucción **mst** viene de "mark stack" y corresponde a la instrucción en código de tres direcciones **begin_args**. La razón por la que se denomina "mark stack" es que el código objetivo generado de una instrucción de esta naturaleza está involucrado con el establecimiento del registro de activación para la nueva llamada en la pila, es decir, los primeros pasos en la secuencia de llamada. Esto generalmente significa, entre otras cosas, que el espacio debe ser asignado o "marcado" en la pila para elementos tales como los argumentos. La instrucción **cup** en código P es la instrucción de "procedimiento de llamada de usuario" y corresponde directamente a la instrucción **call** del código de tres direcciones. La razón de que reciba este nombre es que el código P distingue dos clases de llamadas: **cup** y **csp**, o "call standard procedure". Un procedimiento estándar es un procedimiento "integrado" requerido por la definición del lenguaje, como los procedimientos **sin** o **abs** de Pascal (C no tiene procedimientos integrados de los cuales hablar). Las llamadas a procedimientos integrados pueden emplear conocimiento específico acerca de su funcionamiento para mejorar la eficiencia de la llamada (o inclusive eliminarla). No consideraremos más la operación **csp**.

Advierta que no introducimos una instrucción en código P equivalente a la instrucción **arg** en código de tres direcciones. En vez de eso se supone que todos los valores de argumentos aparecen en la pila (en el orden apropiado) cuando se encuentra la instrucción **cup**. Esto puede producir un orden un poco diferente para la secuencia de llamada que el correspondiente al código de tres direcciones (véanse los ejercicios).

Nuestro ejemplo de una llamada en C (la llamada **f(2+3,4)** para la función **f** descrita anteriormente) se traduce ahora en el siguiente código P:

```
mst
ldc 2
ldc 3
adi
ldc 4
cup f
```

(De nueva cuenta, calculamos los argumentos de izquierda a derecha.)

## 8.5.2 Un procedimiento de generación de código para definición de función y llamada

Como en secciones anteriores, deseamos mostrar un procedimiento de generación de código para una gramática de muestra con definiciones de función y llamadas. La gramática que utilizaremos es la que se muestra a continuación:

*program* → *decl-list exp*
*decl-list* → *decl-list decl* | ε
*decl* → **fn id** ( *param-list* ) = *exp*
*param-list* → *param-list* , **id** | **id**
*exp* → *exp* + *exp* | *llamada* | **num** | **id**
*llamada* → **id** ( *arg-list* )
*arg-list* → *arg-list* , *exp* | *exp*

Esta gramática define un programa como una secuencia de declaraciones de funciones, seguidas por una expresión simple. No hay variables o asignaciones en esta gramática, sólo parámetros, funciones y expresiones, las cuales pueden incluir llamadas de función. Todos los valores son enteros, todas las funciones devuelven enteros y todas las funciones deben tener por lo menos un parámetro. Existe sólo una operación numérica (aparte de la llamada de función): la adición entera. Un ejemplo de un programa definido mediante esta gramática es

```
fn f(x)=2+x
fn g(x,y)=f(x)+y
g(3,4)
```

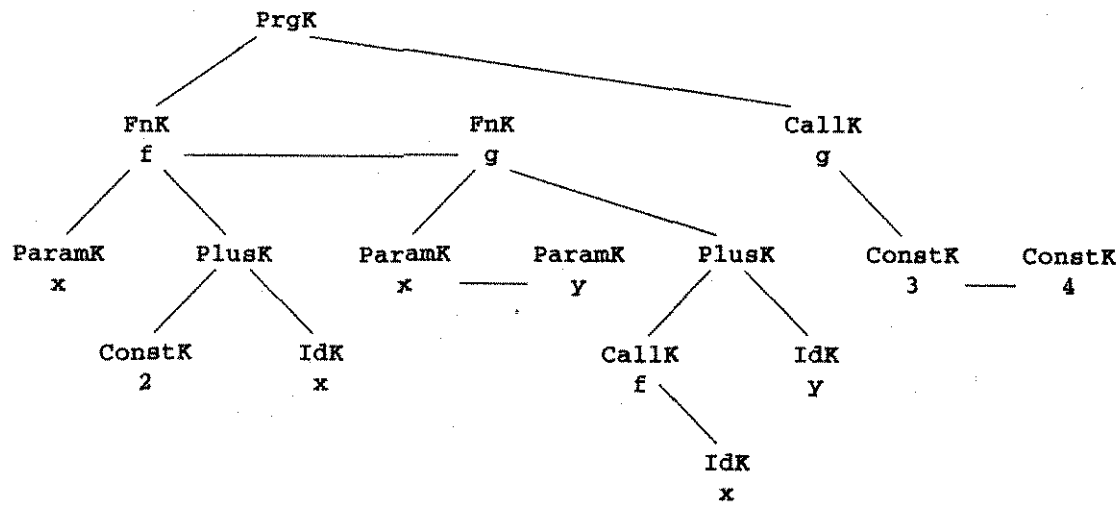
Este programa contiene dos definiciones de funciones seguidas por una expresión que es una llamada a **g**. También hay una llamada a **f** en el cuerpo de **g**.

Queremos definir una estructura de árbol sintáctico para esta gramática. Lo haremos utilizando las siguientes declaraciones en C:

```
typedef enum
  {PrgK,FnK,ParamK,PlusK,CallK,ConstK,IdK}
  NodeKind;
typedef struct streenode
        { NodeKind kind;
          struct streenode *lchild,*rchild,
                           *sibling;
          char * name; /* usado con FnK,ParamK,
                          CallK,IdK */
          int val; /* usado con ConstK */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

Tenemos 7 diferentes clases de nodos en esta estructura de árbol. Cada árbol sintáctico tiene una raíz que es un nodo **PrgK**. Este nodo es utilizado simplemente para agrupar las declaraciones y la expresión del programa en conjunto. Un árbol sintáctico contiene exactamente un nodo así. El hijo izquierdo de este nodo es una lista de hermanos de nodos **FnK**; el hijo derecho es la expresión del programa asociado. Cada nodo **FnK** tiene un hijo izquierdo que es una lista de hermanos de nodos **ParamK**. Estos nodos definen los nombres de los parámetros. El cuerpo de cada función es el hijo derecho de su nodo **FnK**. Los nodos de expresión se representan de la manera habitual, sólo que un nodo **CallK** contiene el nombre de la función llamada y tiene un hijo derecho que es una lista de hermanos de las expresiones de argumento. Por ejemplo, el programa de muestra anterior tiene el árbol sintáctico que se da en la figura 8.13. Por claridad incluimos en esa figura la clase de nodo correspondiente a cada nodo, junto con algunos atributos nombre/valor. Los hijos y hermanos se distinguen por la dirección (hermanos a la derecha, hijos debajo).

Figura 8.13
Árbol sintáctico abstracto para el programa de muestra en la página 440




Dada esta estructura de árbol sintáctico, en la figura 8.14 se proporciona un procedimiento de generación de código que produce código P. Acerca de éste código hacemos las siguientes observaciones. En primer lugar, el código para un nodo **PrgK** simplemente recorre el resto del árbol que tiene por debajo. El código para un **IdK**, **ConstK** o **PlusK** es casi idéntico al de los ejemplos anteriores.

Figura 8.14
Procedimiento de generación de código para definición de función y llamada

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  SyntaxTree p;
  if (t != NULL)
  switch (t->kind)
  { case PrgK:
      p = t->lchild;
      while (p != NULL)
      { genCode(p);
        p = p->sibling; }
      genCode(t->rchild);
      break;
    case FnK:
      sprintf(codestr,"%s %s","ent",t->name);
      emitCode(codestr);
      genCode(t->rchild);
      emitCode("ret");
      break;
    case ParamK: /* sin acciones */
      break;
    case ConstK:
      sprintf(codestr,"%s %d","ldc",t->val);
      emitCode(codestr);
      break;
    case PlusK:
      genCode(t->lchild);
      genCode(t->rchild);
      emitCode("adi");
      break;
```

Figura 8.14 Continuación

```
        case IdK:
          sprintf(codestr,"%s %s","lod",t->name);
          emitCode(codestr);
          break;
        case CallK:
          emitCode("mst");
          p = t->rchild;
          while (p!=NULL)
          { genCode(p);
            p = p->sibling; }
          sprintf(codestr,"%s %s","cup",t->name);
          emitCode(codestr);
          break;
        default:
          emitCode("Error");
          break;
    }
}
```

Restan los casos de **FnK, ParamK** y **CallK**. El código para un nodo **FnK** simplemente rodea el código para el cuerpo de la función (el hijo derecho) con **ent** y **ret**; los parámetros de la función nunca son visitados. En realidad, los nodos de parámetro nunca provocan que se genere código: los argumentos ya han sido calculados por el elemento que llama.[11] Esto explica también por qué no hay acciones para un nodo **ParamK** en la figura 8.14; en efecto, debido al comportamiento del código para **FnK**, no deberían ser alcanzados nodos **ParamK** en el recorrido del árbol, de modo que este caso en realidad podría ser desechado.

El caso final es **Callk**. El código para este caso resulta de una instrucción **mst**, que procede a generar el código para cada argumento, y finalmente de una instrucción **cup**.

Dejamos al lector demostrar que el procedimiento de generación de código de la figura 8.14 produciría el siguiente código P, dado el programa cuyo árbol sintáctico se muestra en la figura 8.13:

```
ent f
ldc 2
lod x
adi
ret
ent g
mst
```

11. Sin embargo, los nodos de parámetro tienen un importante papel de administración en el sentido que determinan las posiciones relativas, o desplazamientos, donde se pueden encontrar los parámetros en el registro de activación. Supondremos que esto es manejado en otra parte.

```
lod x
cup f
lod y
adi
ret
mst
ldc 3
ldc 4
cup g
```

## 8.6 GENERACIÓN DE CÓDIGO EN COMPILADORES COMERCIALES: DOS CASOS DE ESTUDIO

En esta sección examinaremos la salida de código ensamblador producida por dos diferentes compiladores comerciales para diferentes procesadores. El primero es el compilador para C de Borland versión 3.0 para procesadores Intel 80×86. El segundo es el compilador para C de Sun versión 2.0 para SparcStations. Mostraremos la salida de ensamblador de estos compiladores para los mismos ejemplos de código que utilizamos para ilustrar el código de tres direcciones y el código P.[12] Esto debería mejorar la comprensión de las técnicas de generación de código, así como de la conversión de código intermedio a código objetivo. También debería suministrar una comparación útil para el código de máquina producido por el compilador TINY, el cual se comentará en secciones posteriores.

### 8.6.1 El compilador C de Borland 3.0 para la arquitectura 80 × 86

Comenzamos nuestros ejemplos de la salida de este compilador con la asignación utilizada en la sección 8.2.1:

```
(x=x+3)+4
```

Supondremos que la variable **x** en esta expresión está almacenada localmente en el marco de pila.

El código ensamblado para esta expresión tal como se produce mediante el compilador Borland versión 3.0 para la arquitectura Intel 80×86 es como se muestra a continuación:

```
mov     ax,word ptr [bp-2]
add     ax,3
mov     word ptr [bp-2],ax
add     ax,4
```

En este código el registro de acumulador **ax** se utiliza como la ubicación temporal principal para el cálculo. La ubicación de la variable local **x** es **bp-2**, lo que refleja el uso del registro **bp** (base pointer ) como el apuntador de marco y el hecho de que las variables enteras ocupan dos bytes en esta máquina.

La primera instrucción mueve el valor de **x** hacia **ax** (los corchetes en la dirección **[bp-2]** indican una carga indirecta más que inmediata). La segunda instrucción agrega la

12. Para los propósitos de éste y la mayoría de otros ejemplos, las optimizaciones que estos compiladores realizan están desactivadas.