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 de caso del código producido para estas características mediante dos compiladores merciales: el compilador C de Borland para la arquitectura 80×86 y el compilador C 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
```

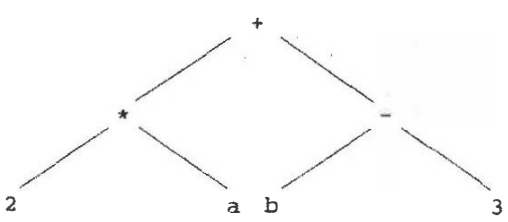
Esta instrucción expresa la aplicación del operador *op* a los valores de `y` y `z`, y la asignación de este valor para que sea el nuevo valor de `x`. Aquí *op* puede ser un operador aritmético como + o − o algún otro operador que pueda actuar sobre los valores de `y` y `z`.

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.

8.2 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**"
$exp \rightarrow aexp$	exp.*pcode* = $aexp$.*pcode*
$aexp_1 \rightarrow aexp_2$ + *factor*	$aexp_1$.*pcode* = $aexp_2$.*pcode* ++ *factor*.*pcode* ++ "**adi**"
$aexp \rightarrow factor$	$aexp$.*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 $exp \rightarrow aexp$ y $aexp \rightarrow factor$, 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* exp_1.*tacode* = exp_2.*tacode* ++ `id`.*strval* || "=" || exp_2.*name*
$exp \rightarrow aexp$	*exp*.*name* = *aexp*.*name* *exp*.*tacode* = *aexp*.*tacode*
$aexp_1 \rightarrow aexp_2$ + *factor*	$aexp_1$.*name* = *newtemp*() $aexp_1$.*tacode* = $aexp_2$.*tacode* ++ *factor*.*tacode* ++ $aexp_1$.*name* || "=" || $aexp_2$.*name* || "+" || *factor*.*name*
aexp → *factor*	*aexp*.*name* = *factor*.*name* *aexp*.*tacode* = *factor*.*tacode*
factor → (*exp*)	*factor*.*name* = *exp*.*name* *factor*.*tacode* = *exp*.*tacode*
factor → `num`	*factor*.*name* = `num`.*strval* *factor*.*tacode* = ""
factor → `id`	*factor*.*name* = `id`.*strval* *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.