Capítulo 7

Generación de código intermedio. Optimización

Bibliografía:

- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 8, 9, 10 (pag. 478-666).
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 8, páginas: 398-481.

1. Introducción.
2. Tipos de representaciones intermedias: Código de 3-direcciones.
3. Código intermedio como un atributo sintetizado.
4. Generación de código para expresiones y sentencias de control:
   a) Proposiciones de asignación.
   b) Expresiones aritméticas.
   c) Expresiones booleanas.
   d) Sentencias de control.
   e) Funciones.
5. Optimización de código:
   a) Bloques básicos y optimización local.

b) Eliminación de subexpresiones comunes.

c) Eliminación de código muerto.

d) Transformaciones aritméticas.

e) Empaquetamiento de variables temporales.

f) Mejoras en lazos.

7.1. Introducción

Como se comentó en el primer capítulo el proceso de la compilación se desglosa en dos partes: la parte que depende sólo del lenguaje fuente (etapa inicial o *front-end*) y la parte que depende sólo del lenguaje objeto (etapa final o *back-end*).

- *Etapa inicial:* corresponde con la parte de análisis (léxico, sintáctico y semántico).
- *Etapa final:* corresponde con la parte de síntesis (generación de código).

La etapa inicial traduce un programa fuente a una *representación intermedia* a partir de la cual la etapa final genera el código objeto.
De esta forma, los detalles que tienen que ver con las características del lenguaje objeto (código ensamblador, código máquina absoluto o relocalizable, . . .), la arquitectura de la máquina (número de registros, modos de direccionamiento, tamaño de los tipos de datos, memoria cache, ...), el entorno de ejecución (estructura de registros y memoria de la máquina donde se va a ejecutar el programa . . .) y el sistema operativo se engloban en la etapa final y se aislan del resto.
La generación de código es la tarea más complicada de un compilador. Las ventajas de utilizar esta representación intermedia, independiente de la máquina en la que se va a ejecutar el programa, son:

- Se puede crear un compilador para una nueva máquina distinta uniendo la etapa final de la nueva máquina a una etapa inicial ya existente. Se facilita la *redestinación*.

- Se puede aplicar, a la representación intermedia, un optimador de código independiente de la máquina.

La figura 7.1 muestra las dos etapas y como se relacionan entre sí a través de la representación intermedia.

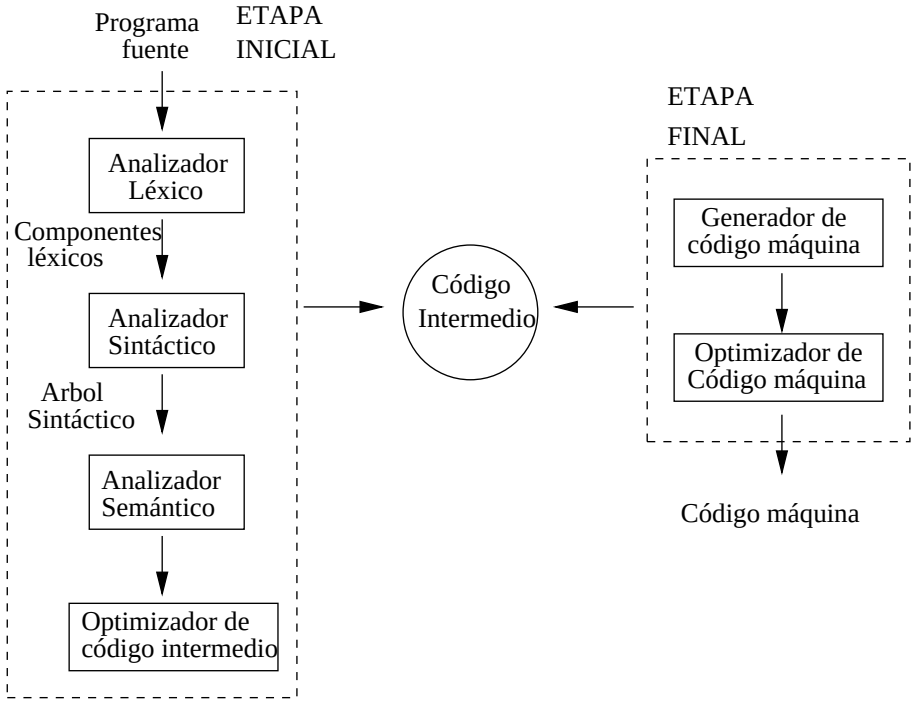


Figura 7.1: Etapa inicial y final de un compilador

En este capítulo veremos cómo traducir las construcciones de los lenguajes de programación como: las declaraciones, asignaciones y proposiciones de flujo de control a una representación intermedia. La mayor parte de las traducciones de estas proposiciones se pueden implantar durante el análisis sintáctico utilizando las técnicas de traducción vistas en en el diseño de esquemas de traducción dirigidos por la sintaxis (ETDS).

7.2. Tipos de representaciones intermedias: el código de 3-direcciones

Una *representación intermedia* es una estructura de datos que representa al programa fuente durante el proceso de la traducción a código objeto. Hasta ahora hemos usado el árbol de análi-

sis sintáctico como representación intermedia, junto con la tabla de símbolos que contenía información sobre los nombres (variables, constantes, tipos y funciones) que aparecían en el programa fuente.

Aunque el árbol de análisis sintáctico es una representación válida, no se parece ni remotamente al código objeto, en el que sólo se emplean saltos a direcciones en memoria en vez de construcciones de alto nivel, como sentencias `if-then-else`. Es necesario generar una nueva forma de representación intermedia. A esta representación intermedia, que se parece al código objeto pero que sigue siendo independiente de la máquina, se le llama *código intermedio*.

El código intermedio puede tomar muchas formas. Todas ellas se consideran como una forma de linearización del árbol sintáctico, es decir, una representación del árbol sintáctico de forma secuencial. El código intermedio más habitual es el código de 3-direcciones.

El **código de tres direcciones** es una secuencia de proposiciones de la forma general

```
x = y op z
```

donde `op` representa cualquier operador; `x,y,z` representan variables definidas por el programador o variables temporales generadas por el compilador. `y,z` también pueden representar constantes o literales. `op` representa cualquier operador: un operador aritmético de punto fijo o flotante, o un operador lógico sobre datos booleanos.

No se permite ninguna expresión aritmética compuesta, pues sólo hay un operador en el lado derecho. Por ejemplo, `x+y*z` se debe traducir a una secuencia, donde t_1, t_2 son variables temporales generadas por el compilador.

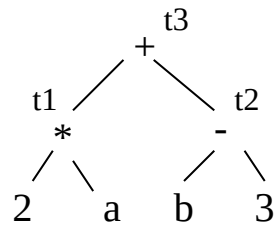
$t_1 = y * z$

$t_2 = x + t_1$

Las expresiones compuestas y las proposiciones de flujo de control se han de descomponer en proposiciones de este tipo, definiendo un conjunto suficientemente amplio de operadores. Se le llama *código de 3-direcciones* porque cada proposición contiene, en el

caso general, tres direcciones, dos para los operandos y una para el resultado. (Aunque aparece el nombre de la variable, realmente corresponde al puntero a la entrada de la tabla de símbolos de dicho nombre).
El código de tres direcciones es una representación linealizada (de izquierda a derecha) del árbol sintáctico en la que los nombres temporales corresponden a los nodos internos. Como estos nombres temporales se representan en la memoria no se especifica más información sobre ellos en este tipo de código. Normalmente se asignarán directamente a registros o se almacenarán en la tabla de símbolos.

`2*a+b-3`



Código de 3-direcciones:

t_1 = 2 * a

t_2 = b - 3

t_3 = t_1 + t_2

7.2.1. Tipos de proposiciones de 3-direcciones

La forma de código de 3-direcciones que hemos visto hasta ahora es insuficiente para representar todas las construcciones de un lenguaje de programación (saltos condicionales, saltos incondicionales, llamadas a funciones, bucles, etc), por tanto es necesario introducir nuevos operadores. El conjunto de proposiciones (operadores) debe ser lo suficientemente rico como para poder implantar las operaciones del lenguaje fuente.
Las proposiciones de *3-direcciones* van a ser en cierta manera análogas al código ensamblador. Las proposiciones pueden tener etiquetas simbólicas y existen instrucciones para el flujo de control (`goto`). Una etiqueta simbólica representa el índice de una proposición de *3-direcciones* en la lista de instrucciones.

Las proposiciones de 3-direcciones más comunes que utilizaremos:

1. Proposiciones de la forma `x = y op z` donde `op` es un operador binario aritmético, lógico o relacional.
2. Instrucciones de la forma `x = op y`, donde `op` es un operador unario (operador negación lógico, menos unario, operadores de desplazamiento o conversión de tipos).
3. Proposiciones de copia de la forma `x = y`, donde el valor de `y` se asigna a `x`.
4. Salto incondicional `goto etiq`. La instrucción con etiqueta `etiq` es la siguiente que se ejecutará.
5. Saltos condicionales como `if_false x goto etiq`.
6. `param x` y `call f` para apilar los parámetros y llamadas a funciones (los procedimientos se consideran funciones que no devuelven valores). También `return y`, que es opcional, para devolver valores. Código generado como parte de una llamada al procedimiento `p(x_1,x_2,...,x_n)`.

   `param` x_1
   `param` x_2
   `...`
   `param` x_n
   `call p,n`
7. Asignaciones con índices de la forma `x = y[i]`, donde se asigna a `x` el valor de la posición en `i` unidades de memoria más allá de la posición `y`. O también `x[i] = y` .
8. Asignaciones de direcciones a punteros de la forma `x = &y` (el valor de x es la dirección de y), `x = *y` (el valor de x se iguala al contenido de la dirección indicada por el puntero y) ó `*x = y` (el objeto apuntado por x se iguala al valor de y).

Ejemplo. Consideremos el código que calcula el factorial de un número. La tabla 7.1 muestra el código fuente y el código de 3-direcciones. Existe un salto condicional `if_false` que se usa para

traducir las sentencias de control `if-then, repeat-until` que contiene dos direcciones: el valor condicional de la expresión y la dirección de salto. La proposición `label` sólo tiene una dirección. Las operaciones de lectura y escritura, `read, write`, con una sola dirección. Y una instrucción de parada `halt` que no tiene direcciones.

read x;	read x
if 0<x **then**	t1 = 0 < x
fact:=1;	if_false t1 goto L1
repeat	fact=1
fact:=fact*x;	label L2
x:=x-1;	t2=fact * x
until x=0;	fact=t2
write fact;	t3=x-1
end;	x=t3
	t4=x==0
	if_false t4 goto L2
	write fact
	label L1
	halt

Cuadro 7.1: Código fuente y código de 3-direcciones para el cálculo del factorial

7.2.2. Implementación de código de tres direcciones

Una proposición de código de 3-direcciones se puede implantar como una estructura tipo registro con campos para el operador, los operandos y el resultado. La representación final será entonces una lista enlazada o un vector de proposiciones.

Implementación mediante cuádruplos

Un *cuádruplo* es una estructura tipo registro con cuatro campos que se llaman `(op, result, arg1, arg2)`. El campo `op` contiene un código interno para el operador.

Por ejemplo, la proposición de tres direcciones `x = y + z` se representa mediante el cuádruplo *(ADD, x,y, z)*. Las proposiciones con operadores unarios no usan el $arg2$. Los campos que no se usan se dejan vacíos o un valor NULL. Como se necesitan cuatro campos se le llama representación mediante *cuádruplos*.

Una posible implantación del programa que calcula el factorial mediante cuádruplos se muestra ahora en la parte izquierda de la tabla 7.2.

(a) Cuádruplos	(b) Tripletes
(read,x,–,–)	(0) (read,x,–)
(isbigger,t1,x,0)	(1) (isbigger,x,0)
(if_false,t1,L1,–)	(2) (if_false, (1), (11))
(assign,fact,1,–)	(3) (assign, fact,1)
(label,L2,–,–)	(4) (mult, fact,x)
(mult,t2,fact,x)	(5) (assign, (4), fact)
(assign,fact,t2,–)	(6) (sub, x,1)
(sub,t3,x,1)	(7) (assign, (6), x)
(assign,x,t3,–)	(8) (isequal, x, 0)
(isequal,t4,x,0)	(9) (if_false, (8),(4))
(if_false,t4,L2,–)	(10) (write,fact,–)
(write,fact,–,–)	(11) (halt,–,–)
(label,L1,–,–)	
(halt,–,–,–)	

Cuadro 7.2: Código de 3-direcciones mediante: (a) cuádruplos y (b) tripletes

Implementación mediante tripletes

Para evitar tener que introducir nombres temporales en la tabla de símbolos, se hace referencia a un valor temporal según la posición de la proposición que lo calcula. Las propias instrucciones representan el valor del nombre temporal. La implantación se hace mediante registros de sólo tres campos `(op, arg1, arg2)`.
La parte derecha de la tabla 7.2 muestra la implantación mediante tripletes del cálculo del factorial. Los números entre paréntesis representan punteros dentro de la lista de tripletes, en tanto que los punteros a la tabla de símbolos se representan mediante los nombres mismos.
En la notación de tripletes se necesita menor espacio y el compilador no necesita generar los nombre temporales. Sin embargo, en esta notación, trasladar una proposición que defina un valor temporal exige que se modifiquen todas las referencias a esa proposición. Lo cual supone un inconveniente a la hora de optimizar el código, pues a menudo es necesario cambiar proposiciones de lugar.

A partir de ahora nos vamos a centrar en la notación de cuádruplos, que es la que se implantará en la práctica 3. La figura 7.2 a continuación muestra en código C como se podría implantar la estructura de datos para los cuádruplos:

```
typedef enum {assign, add, mult,if_false, goto, label, read, write, isequal, ... }
OpKind;
typedef struct {
        int val;// para valores
        char *name;// para identificadores de variables
} Address;
typedef struct {
        OpKind op;
        Address result, arg1, arg2;
} Quad;
```

Figura 7.2: Posible implantación en C de la estructura cuádruplo

Por simplicidad se podrian considerar los campos `result, arg1, arg2` como punteros a caracter. En esta implantación sólo se permite que un argumento represente una constante entera o una cadena (la cadena representa el nombre de un temporal o una variable de la tabla de símbolos). Una alternativa a almacenar nombres en el cuádruplo es almacenar punteros a la tabla de símbolos, con lo que se evita tener que hacer operaciones de búsqueda en un procesado posterior.

7.3. Código intermedio como un atributo sintetizado

El código intermedio puede ser considerado como un atributo sintetizado. El código intermedio es visto como una cadena de caracteres y se puede diseñar un esquema de traducción dirigido por la sintaxis (ETDS) que genere dicho código al recorrer el árbol de análisis sintáctico en orden postfijo.
Consideremos como ejemplo la siguiente gramática simplificada que genera expresiones aritméticas. Dada la expresión $E \rightarrow E_1 +$

E_2 se calcula el valor del no-terminal E en un nombre temporal t. Por el momento, se crea un nuevo nombre cada vez que se necesita. Más tarde veremos un sencillo método para reutilizar los nombres temporales.
Cada no-terminal tiene dos atributos:

- `E.lugar`, nombre temporal que contendrá el valor de E, $(t_1, t_2, \ldots)$.
- `E.cod`, serie de todas las proposiciones de código de 3-direcciones que calculan E.

Ambos atributos son sintetizados. La función `nuevotemp()` genera nombres distintos t_1, t_2, ... cada vez que es llamada. Las llaves indican una instrucción de código de 3-direcciones. El símbolo // representa la concatenación de los trozos de código.

Producción	Regla Semántica
$S \rightarrow id := E$	$S.cod = E.cod//\{lexema(id) = E.lugar\}$
$E \rightarrow E_1 + E_2$	$E.lugar = nuevotemp();$ $E.cod = E_1.cod//E_2.cod//\{E.lugar = E_1.lugar + E_2.lugar\}$
$E \rightarrow (E_1)$	$E.lugar = E_1.lugar$ $E.cod = E_1.cod$
$E \rightarrow id$	$E.lugar = lexema(id)$ $E.cod =$ " "
$E \rightarrow num$	$E.lugar = lexema(num);$ $E.cod =$ " "

Cuadro 7.3: ETDS para expresiones aritméticas

Veamos el ejemplo `x=x+3+4`. La figura 7.3 muestra el árbol sintáctico. El código de 3-direcciones correspondiente es:

t_1 = x + 3

t_2 = t_1 + 4

x = t_2

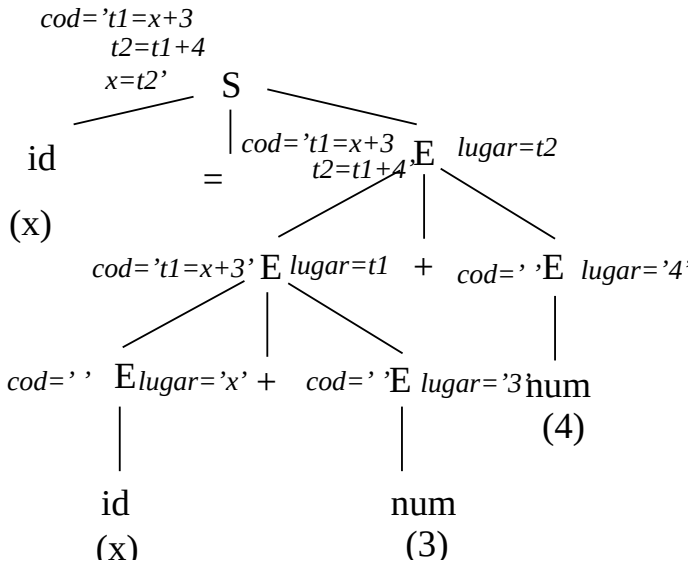


Figura 7.3: Árbol sintáctico para el ejemplo `x=x+3+4`

Reutilización de los nombres temporales

Hasta ahora se ha supuesto que la función `nuevotemp()` genera un nombre temporal cada vez que se necesita un temporal . Sin embargo, los nombres temporales utilizados para guardar valores intermedios de los cálculos de expresiones tienen que ser introducidos en la tabla de símbolos para guardar sus valores con el consiguiente aumento de espacio.

Los nombres temporales se pueden reutilizar mediante un sencillo método que consiste en llevar una cuenta c, iniciada a cero, de variables temporales. Cuando se utilice un nombre temporal como operando hay que decrementar el valor de c en 1. Siempre que se genere un nuevo nombre temporal se usa el valor del contador (`sprintf(simbolo,``t%d'',c)`) y se incrementa el valor de c en 1. Consideremos el ejemplo: $x = a * b + c * d - e * f$

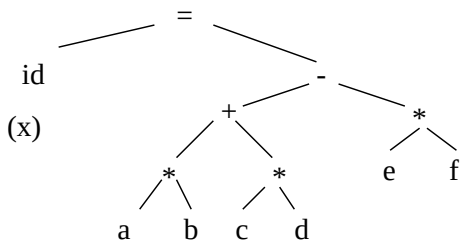


Figura 7.4: Árbol sintáctico para el ejemplo `x=a*b+c*d-e*f`

Proposición	Valor de c
	0
t0 = a * b	1
t1 = c * d	2
t0 = t0 + t1	1
t1 = e * f	2
t0 = t0 - t1	1
x = t0	0

Cuidado: este método no es aplicable para temporales que se utilizan más de una vez. Por ejemplo, al evaluar una expresión en una proposición condicional. A estos valores hay que asignarles nombres temporales creados con un nombre propio.

7.4. Traducción de proposiciones de asignación y expresiones aritméticas

La tabla 7.4 a continuación muestra como se puede realizar la traducción de expresiones aritméticas más complejas. La función

`gen_cuad(op, result, arg1, arg2)`

genera el correspondiente código de 3-direcciones en notación de cuádruplos. Cada vez que es llamada genera una lista de código con sólo un cuádruplo.

Producción	**Reglas Semánticas**
$S \rightarrow id := E$	$S.cod = E.cod//gen\_cuad(ASSIGN, id.lugar, E.lugar, --)$
$E \rightarrow E_1 + E_2$	$E.lugar = nuevotemp()$
	$E.cod = E_1.cod//E_2.cod//gen\_cuad(ADD, E.lugar, E_1.lugar, E_2.lugar)$
$E \rightarrow E_1 * E_2$	$E.lugar = nuevotemp()$
	$E.cod = E_1.cod//E_2.cod//gen\_cuad(MULT, E.lugar, E_1.lugar, E_2.lugar)$
$E \rightarrow -E_1$	$E.lugar = nuevotemp()$
	$E.cod = E_1.cod//gen\_cuad(UMINUS, E.lugar, E_1.lugar, --)$
$E \rightarrow (E_1)$	$E.lugar = E_1.lugar$
	$E.cod = E_1.cod$
$E \rightarrow id$	$E.lugar = lexema(id)$
	$E.cod =$ " "

Cuadro 7.4: ETDS para expresiones aritméticas

En la práctica las proposiciones de 3-direcciones se pueden escribir de forma consecutiva a un archivo de salida en lugar de construir la lista de proposiciones en el atributo `cod`, lo que facilita la implementación.
Veamos ahora como se puede implantar una función recursiva que genera el código de 3-direcciones, que llamaremos `genera_código()`. Al fin y al cabo se trata de un atributo sintetizado y podemos recorrer el árbol en modo postfijo. Supongamos que la función devuelve un registro llamado TAC (`Three-Address-Code`), que contiene dos campos:

- `lugar` (nombre temporal donde se almacena el valor de una

expresión)

- `cod` (puntero a la lista de cuádruplos que almacena el código).

Se utiliza el lexema, pero sería en realidad un puntero a la tabla de símbolos.

```
typedef enum {n_assign, n_mult, n_add, n_id, ...} NodeKind;
typedef struct streenode {
      NodeKind kind;
      int nchilds; // número de hijos
      struct streenode childs[MAXNUMCHILDS];
      // se puede usar una lista con primer hijo que apunta a hermanos
      int val; // para constantes numéricas
      char *strval; // para identificadores, deberia ser puntero a la
TS
      } STreeNode;
typedef STreeNode * SyntaxTreeRoot;
typedef struct {
      char lugar[10];
      lista_codigo *cod; //puntero a lista de cuádruplos
      } TAC;
```

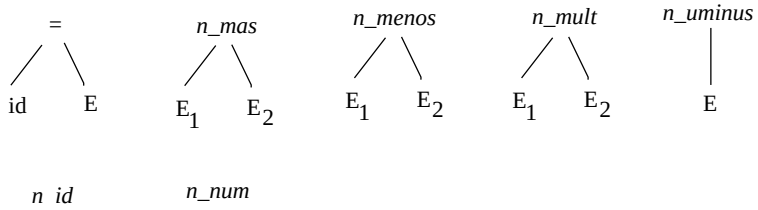


Figura 7.5: Tipos de nodo

```
TAC genera_código (STreeNode *nodo) {
      TAC datos;
      TAC aux1, aux2;
      lista_codigo *cod=NULL;
      datos.cod=NULL;
      switch (node->kind) {
            case n_assign: // childs[0]=id, childs[1]=E
                  aux1=genera_codigo(childs[1]);
                  cod=gen_cuad(assign, lexema(id), aux1.lugar,--);
                  datos.cod=concatena_codigo(aux1.cod,cod);
                  break;
            case n_add: // childs[0]=E1, childs[1]=E2
                  aux1=genera_codigo(childs[0]);
```

```
            aux2=genera_codigo(childs[1]);
            datos.lugar=nuevotemp();
            cod=gen_cuad(add, datos.lugar,aux1.lugar,aux2.lugar);
            datos.cod=concatena_codigo(aux1.cod,aux2.cod,cod);
            break;
        case n_mult: // childs[0]=E1, childs[1]=E2
            aux1=genera_codigo(childs[0]);
            aux2=genera_codigo(childs[1]);
            datos.lugar=nuevotemp();
            cod=gen_cuad(mult, datos.lugar,aux1.lugar,aux2.lugar);
            datos.cod=concatena_codigo(aux1.cod,aux2.cod,cod);
            break;
        case n_parentesis: // childs[1]=E
        // no haria falta crear este tipo de nodo
            datos=genera_codigo(childs[1]);
            break;
        case n_id:
            datos.lugar=lexema(id);
            datos.cod=NULL;
            break; }
        case n_num:
            datos.lugar=lexema(num);
            datos.cod=NULL;
            break; }
    return(datos);
}
```

7.5. Traducción de expresiones booleanas

Las expresiones booleanas se utilizan principalmente como parte de las proposiciones condicionales que alteran el flujo de control del programa, `if-then, if-then-else, while-do`. Las expresiones booleanas se componen de los operadores booleanos `and,or,not` aplicados a variables booleanas o expresiones relacionales.
A su vez, las expresiones relacionales son de la forma E_1 `oprel` E_2, donde E_1 y E_2 son expresiones aritméticas y `oprel` es cualquier operador relacional `<, >, <=, >=,...`.
Consideremos expresiones booleanas generadas por la gramática:

```
E → E or E
      | E and E
      | not E
      | ( E )
      | id oprel id
      | true
      | false       | id
```

Uno de los métodos para traducir expresiones booleanas a código de 3-direcciones consiste en codificar numéricamente los valores `true` y `false` y evaluar una expresión booleana como una expresión aritmética, siguiendo unas prioridades. A menudo se utiliza 1 para indicar `true` y 0 para indicar `false`.
Las expresiones booleanas se evalúan de manera similar a una expresión aritmética de izquierda a derecha. Supongamos el ejemplo: `a or b and not c`. La secuencia de código de 3-direcciones correspondiente es:
t_1 = `a or b`
t_2 = `not c`
t_3 = t_1 `and` t_2
La siguiente gramática en la tabla 7.5 muestra el ETDS para producir código de 3-direcciones para las expresiones booleanas.
La función para generar código se podría ampliar añadiendo nuevos casos a la sentencia `switch` correspondientes a los tipos de no-

Producción	**Reglas Semánticas**
$E \rightarrow E_1\ or\ E_2$	$E.lugar = nuevotemp()$
	$E.cod = E_1.cod//E_2.cod//gen\_cuad(or, E.lugar, E_1.lugar, E_2.lugar)$
$E \rightarrow E_1\ and\ E_2$	$E.lugar = nuevotemp()$
	$E.cod = E_1.cod//E_2.cod//gen\_cuad(and, E.lugar, E_1.lugar, E_2.lugar)$
$E \rightarrow\ not\ E_1$	$E.lugar = nuevotemp()$
	$E.cod = E_1.cod//gen\_cuad(not, E.lugar, E_1.lugar, --)$
$E \rightarrow (E_1)$	$E.lugar = E_1.lugar$
	$E.cod = E_1.cod$
$E \rightarrow\ id_1\ oprel\ id_1$	$E.lugar = nuevotemp();$
	$E.cod = gen\_cuad(oprel, E.lugar, lexema(id_1), lexema(id_2))$
$E \rightarrow true$	$E.lugar = nuevotemp();$
	$E.cod = gen\_cuad(assign, E.lugar, 1, --)$
$E \rightarrow false$	$E.lugar = nuevotemp();$
	$E.cod = gen\_cuad(assign, E.lugar, 0, --)$
$E \rightarrow id$	$E.lugar = lexema(id)$
	$E.cod =$ ““

Cuadro 7.5: ETDS para expresiones booleanas utilizando una representación numérica

dos asociados a los operadores lógicos. Por ejemplo, para el nodo correspondiente al operador lógico `and`, nodo tipo `n_and`, se insertaría el siguiente trozo de código:

```
case n_and: // childs[0]=E1, childs[1]=E2
  aux1=genera_codigo(childs[0]);
  aux2=genera_codigo(childs[1]);
  datos.lugar=nuevotemp();
  cod=gen_cuad(and, datos.lugar,aux1.lugar,aux2.lugar);
  datos.cod=concatena_codigo(aux1.cod,aux2.cod,cod);
  break;
```

7.6. Traducción de proposiciones de control

Pasemos a considerar ahora la traducción de proposiciones de control `if-then, if-then-else, while-do` generadas por la siguiente gramática:

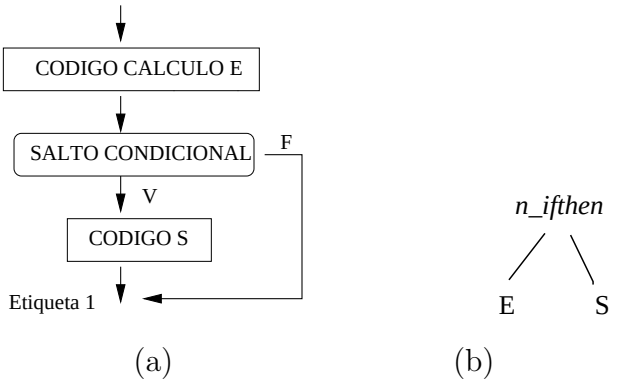
`S` → **if** `E` **then** S_1

| **if** `E` **then** S_1 **else** S_2

| **while** `E` **do** S_1

7.6.1. Proposición if-then

Supongamos una sentencia `if-then` de la forma $S \rightarrow$ `if` E `then` S_1, ver diagrama de flujo en figura 7.6. Para generar el código correspondiente a esta proposición habría que añadir a la función `genera_código()` un nuevo caso para la sentencia `switch` que contemple este tipo de nodo en el árbol sintáctico, nodo `n_ifthen`.



(a) (b)

Figura 7.6: (a) Diagrama de flujo para la proposición `if-then` y (b) tipo de nodo

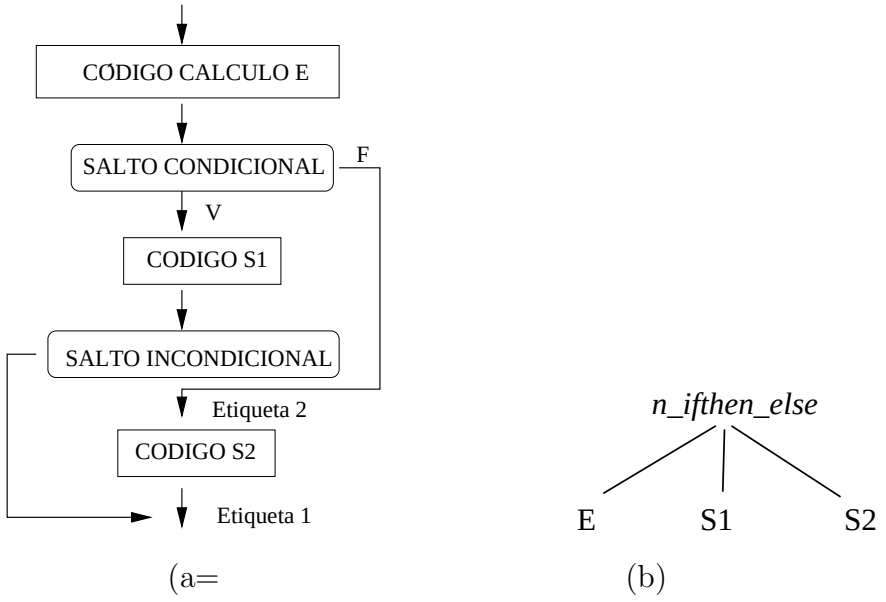
```
TAC datos;
TAC aux1, aux2;
lista_codigo *cod;
datos.cod=NULL;
direcciones dir; //usamos dirección al salto, en vez de etiquetas
     case n_ifthen: // childs[0]=E, childs[1]=S1
          aux1=genera_codigo(childs[0]);
          cod=gen_cuad(if_false,--, aux1.lugar, dir?);
          // aún no se sabe la direc. de salto
          aux2=genera_codigo(childs[1]);
```

```
dir=sgtedirlibre(); //relleno de retroceso
rellena(cod,arg3,dir)
datos.cod=concatena_codigo(aux1.cod,cod,aux2.cod);
break;
```

Supondremos que tenemos una función, `sigtedirlibre()`, que guarda el índice de la siguiente instrucción libre (la función `gen_cuad()` incrementa ese contador). En la implantación se ha optado por utilizar direcciones directamente a las instrucciones en vez de usar etiquetas.

7.6.2. Proposición if-then-else

Supongamos una sentencia `if-then-else` de la forma $S \rightarrow$ `if` E `then` S_1 `else` S_2 , cuyo diagrama de flujo tiene la forma representada en la figura 7.7. Para generar el código correspondiente a esta proposición habría que añadir a la función `genera_código()` un nuevo caso para la sentencia `switch` que contemple este tipo de nodo en el árbol sintáctico, nodo `n_ifthenelse`. Este fragmento de código se podría implantar de forma similar a como hemos hecho en la sección anterior. Ver ejercicios.

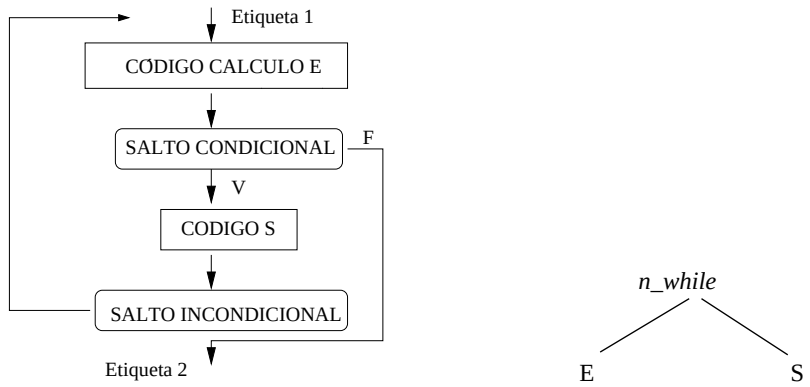


(a= (b)

Figura 7.7: (a)Diagrama de flujo para la proposición `if-then-else` y (b) tipo de nodo

7.6.3. Proposición while-do

Una sentencia `while -do` tiene la forma $S \rightarrow$ `while` E `do` S_1, cuyo diagrama de flujo viene representado en la figura 7.8. Para generar el código correspondiente a esta proposición habría que añadir a la función `genera_código()` un nuevo caso para la sentencia `switch` que contemple este tipo de nodo en el árbol sintáctico.



(a= (b)

Figura 7.8: (a)Diagrama de flujo para la proposición `while_do` y (b) tipo de nodo

Supondremos que tenemos una función, `sigtedirlibre()`, que guarda el índice de la siguiente instrucción libre (se asume que la función `gen_cuad()` incrementa ese contador).

```
TAC datos; datos.cod=NULL;
TAC aux1, aux2;
lista_codigo *cod1=NULL, *cod2=NULL;
direcciones dir1,dir2;
     case n_while: // childs[0]=E, childs[1]=S1
          dir1=sgtedirlibre();
          aux1=genera_codigo(childs[0]);
          cod1=gen_cuad(if_false,--, aux1.lugar, dir?);
          aux2=genera_codigo(childs[1]);
          cod2=gen_cuad(goto,--, dir1,--);// salto incondicional
a dir1
          dir2=sigtedirlibre();
          // rellena argumento de cod1, direccion salto condicional
```

```
rellena(cod1,arg3,dir2)
datos.cod=concatena_codigo(aux1.cod,cod1,aux2.cod,cod2);
break;
```

Se ha optado por utilizar direcciones directamente a las instrucciones en vez de etiquetas.

7.7. Traducción de funciones

En esta sección se describe en términos generales el mecanismo de generación de código de 3-direcciones para funciones. La generación de código para las funciones es la parte más complicada porque depende de la máquina objeto y de la organización del entorno de ejecución. La traducción de funciones implica dos etapas:

- **la definición de la función**. Una definición de función crea un nombre, parámetros y código del cuerpo de la función pero no se ejecuta la función en ese punto.
- **la llamada a la función** desde una parte del programa (trataremos a los procedimientos como funciones que no devuelven valores). Una llamada crea los valores actuales para los parámetros y realiza un salto al código de entrada de la función que se ejecuta y retorna al punto de la llamada.

Supongamos que el subárbol sintáctico correspondiente a una función tiene como raíz un nodo de tipo `n_call`, llamada a función, con una serie de hijos que corresponden a los argumentos, que en general, serán expresiones $E_1, \ldots, E_n$ a evaluar para obtener el valor actual de los n-argumentos.

A continuación se incluye una implantación para generar el código de 3-direcciones correspondiente a la llamada de la función en primer lugar. Posteriormente analizaremos el caso de la definición de la función. Supondremos que se han comprobado previamente las condiciones semánticas de la llamada (número y tipo de argumentos).

Ejemplo de llamada a la función

La llamada a una función de dos argumentos genera el siguiente código intermedio:
Se hace uso de una pila en donde se apilan los valores actuales de los parámetros de la función para después, como veremos en el

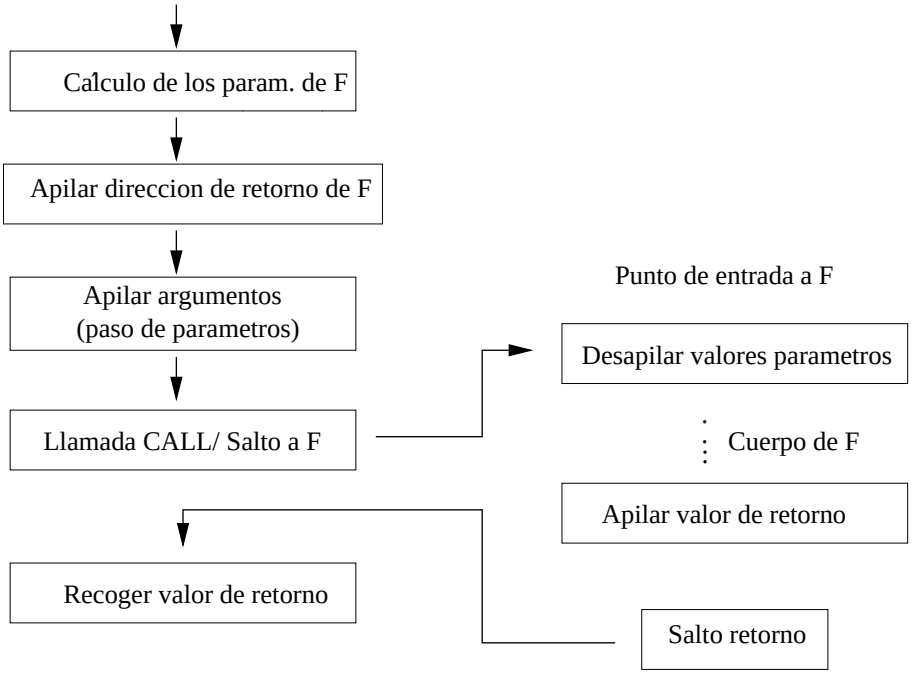


Figura 7.9: Diagrama de flujo para la llamada a función

```
ASIGN_C  t0  147  // asigna la dirección de retorno 147 a t0 (la siguiente al CALL)
PUSH     t0  –    // apila la dirección de retorno
PUSH     t1  –    // apila el valor de t1, primer arg. de la función
PUSH     t2  –    // apila el valor de t2, segundo arg. de la función
CALL     X   –    // salta a la dirección X, entrada a la función
POP      t3  –    // direccion 147, se desapila el valor devuelto por la función, se almacena en t3
```

Cuadro 7.6: Ejemplo de código de llamada a una función

siguiente apartado, en la implantación del cuerpo de la función, desapilarlos.

Nota: Respecto a la dirección de la llamada a la función (índice de la instrucción del código de entrada a la función), se podría conocer en el caso de que el código para la función ya se hubiera generado antes de la llamada (por ejemplo, se podría haber almacenado en la entrada de la tabla de símbolos correspondiente a esa función en un campo definido para ese propósito).

En el caso de que aún no se haya generado el código para la definición de la función tendríamos que hacer un relleno de retroceso en todas las llamadas a esa función. Una forma sería para cada función almacenar los índices de las instrucciones donde se hace una llamada a dicha función (por ejemplo, en un vector de índices) y una vez definida la función que ya se conoce la dirección de entrada recorremos el vector de índices y rellenamos con la dirección adecuada en todas las proposiciones de llamadas a esa función.

```
TAC datos;
datos.cod=NULL;
TAC aux[MAXNCHILDS];
cuadruplos cod1, cod2, codpush[MAXNCHILDS], codcall, codpop;
direcciones dir1,dir2; //direcciones a los saltos
char *t0, *t1; // varibales temporales

case n_call:
   // cálculo de los parámetros de la función
   for(i = 0; i<nchilds;i++)
         aux[i] = genera_código(childs[i]);
   t0 = nuevotemp();
   //asigna en t0 la direc. de retorno de la func., aún no se sabe
   cod1 = gen_cuad(ASSIGN,t0,dir1?);
   cod2 = gen_cuad(PUSH,–,t0,–); // apilamos la direc. de retorno
   // apilamos los argumentos
   for(i = 0; i<nchilds;i++)
         codpush[i] = gen_cuad(PUSH,–, aux[i].lugar,–));
   // llamada a la func., no se sabe aún la direcc. de su cuerpo
   codcall = gen_cuad(CALL, –, dir2?,–);
   // la siguiente dirección libre es la de retorno
   dir1 = sigtedirlibre();
   rellena(cod1,arg3,dir1);
   t1 = nuevotemp();
   // recogemos el valor devuelto por la funcion
   codpop = gen_cuad(POP,t1,–,–) ;
   datos.cod=concatena(losaux[i].cod,cod1,cod2,loscodpush[i],codcall,codpop);
   datos.lugar = t1;
   //para relleno de retroceso de dir2, vease Nota
   return (datos);
```

Figura 7.10: Implantación de la llamada a funciones

Cuerpo de la función

Para traducir el cuerpo de la función bastaría recorrer el árbol hasta encontrar un nodo de tipo `n_function` a partir del cual colgará el bloque de sentencias del cuerpo de la función. No haría falta generar la parte del árbol de los argumentos ya que éstos estarían almacenados en la tabla de símbolos. A partir de cierta dirección de código intermedio deberíamos tener un instrucción que sería la entrada a la función y que contendría:

```
POP     t10   // recoge el segundo argumento
POP     t11   // recoge el primer argumento
...           // operaciones de la función que almacena el resultado en t20 (por ejemplo)
POP     t15   // recoge de la pila el valor de la direcc. de retorno (147) y lo pone en t15 (por ejem.)
RETURN  t15   t20 // salida de la función: el entorno de ejecución deber primero saltar a la instrucc.
              cuya dirección es el valor de t15 y apilar el valor devuelto que es el de t20
```

Esto implica las siguientes tareas (ver parte derecha de la figura 7.9) :

1. Recoger la dirección libre actual y utilizarla como dirección para rellenar (relleno con retroceso) las llamadas a la función que estamos definiendo.
2. Generar código para desapilar los argumentos de la función y meterlos en las variables correspondientes.
3. Generar código para el bloque de sentencias S.
4. Desapilar la direccion de retorno.
5. Generar la salida de la funcion con RETURN.
6. Unir los trozos de código para tenerlos en una única lista de cuádruplos.

```
TAC datos;
datos.cod=NULL;
TAC aux[MAXNCHILDS];
cuadruplos codpoparg[MAXNPARAM], codpopdirreturn, codreturn;
direcciones dir; //direccion de retorno
char *t; // varibale temporal
case n_function:
   dir = sigtedirlibre();
   rellena(todas llamadas a la funcion con esta dir);
   // desapilamos los argumentos
   for(i = 0; i < nparams ;i++)
          codpoparg[i] = gen_cuad(POP, param[i].lugar, -, -));
       //genera codigo para el cuerpo de la función (k intrucciones)
   //y calculamos el valor a devolver
   for(k = 0; k < nchilds; k++)
          aux[k] = genera_código(childs[k]);
   t = nuevotemp();
   //desapilamos la direccion de retorno de la función
   codpopdirreturn = gen_cuad(POP, t, -, -);
   //salida de la funcion: el entorno de ejecución salta a la
   //direcc. de retorno en t y se apila el valor devuelto
   codreturn = gen_cuad(RETURN, t, valor devuelto, -);
  datos.cod=concatena(loscodpoparg[i],losaux[k].cod,codpopdirreturn,codreturn);
   datos.lugar = valor devuelto;
   return (datos);
```

Figura 7.11: Implantación del cuerpo de las funciones

VUESTRA práctica 3

- En la definición de la función vais a tener que colgar los argumentos en el árbol para saber cuantos parámetros teneis que desapilar cuando implementeis el cuerpo de la función, pues no teneis la tabla de simbolos con esa información.
- Para la llamada a las funciones tendreís que crear a mano un nodo n_CALL, producción `FunctionStm` → `id ( Arg )` . De lo contrario, lo confundiriaís con el nodo n_id.

- Para colgar los parámetros en la definición de la función tendreís que crear un nodo a mano n_PARAM, producción de `Arg`.

Vamos a hacer el siguiente ejemplo. Se ha considerado que primero se genera el código del programa principal y a continuación el código de las funciones.

```
module Ejemplo ;
var resultado : integer ;
function Suma ( integer a, integer b) : integer ;
var total: integer;
begin
  total = a + b ;
  return(total) ;
end Suma ;
begin (* programa principal *)
resultado = Suma ( 2, 4*8) ;
write resultado ;
end Ejemplo .
```

Listado de cuadruplos

```
00 (ASSIGN_C, t1, 2, NULL)
01 (MULT, t2, 4, 8)
02 (ASSIGN_C, t3, 7, NULL)
03 (PUSH, t3, NULL, NULL)
04 (PUSH, t1, NULL, NULL)
05 (PUSH, t2, NULL, NULL)
06 (CALL, 11, NULL, NULL)
07 (POP, t4, NULL, NULL)
08 (ASSIGN_V, resultado, t4, NULL)
09 (WRITE, resultado, NULL, NULL)
10 (HALT, NULL, NULL, NULL) // final del programa
11 (POP, b, NULL, NULL) // entrada a la función
12 (POP, a, NULL, NULL)
13 (ADD, t5, a, b)
14 (ASSIGN_V, total, t5, NULL)
```

```
15 (POP, t6, NULL, NULL) // desapilo la dirección de retorno, la 07
16 (RETURN, t6, total, NULL) // el entorno de ejecución salta a la direccion
// en t6 (la 07) y apila el valor a devolver
```

7.8. Optimización de código intermedio

La optimización de código intermedio se puede realizar:

- a nivel local: sólo utilizan la información de un bloque básico para realizar la optimización.
- a nivel global: que usan información de varios bloques básicos.

El término *optimización* de código es inadecuado ya que no se garantiza el obtener, en el sentido matemático, el mejor código posible atendiendo a maximizar o minimizar una función objetivo (tiempo de ejecución y espacio). El término de *mejora* de código sería más apropiado que el de optimización.
Nos concentraremos básicamente en la optimización de código de tres-direcciones, puesto que son siempre transportables a cualquier etapa final, son optimizaciones independientes de la máquina. Las optimizaciones a nivel de la máquina, como la asignación de registros y la utilización de instrucciones específicas de la máquina, se salen del contexto de esta asignatura.
La mayoría de los programas emplean el 90 % del tiempo de ejecución en el 10 % de su código. Lo más adecuado es identificar las partes del programa que se ejecutan más frecuentemente y tratar de que se ejecuten lo más eficientemente posible. En la práctica, son los lazos internos los mejores candidatos para realizar las transformaciones.
Además de la optimización a nivel de código intermedio, se puede reducir el tiempo de ejecución de un programa actuando a otros niveles: a nivel de código fuente y a nivel de código objeto.

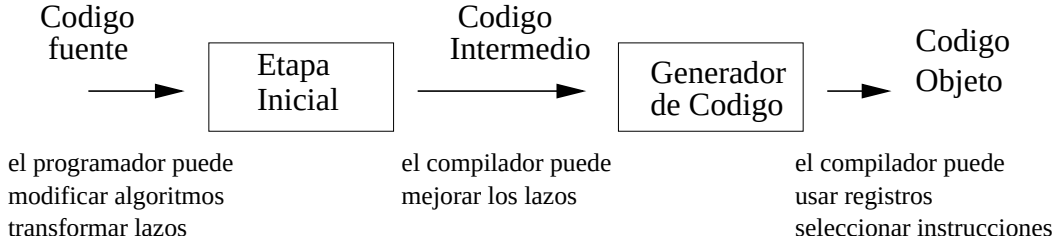


Figura 7.12: Niveles en los que el programador y el compilador pueden mejorar el código

Un *bloque básico* es una unidad fundamental de código. Es una secuencia de proposiciones donde el flujo de control entra en el principio del bloque y sale al final del bloque. Los bloques básicos pueden recibir el control desde más de un punto en el programa (se puede llegar desde varios sitios a una etiqueta) y el control puede salir desde más de una proposición (se podría ir a una etiqueta o seguir con la siguiente instrucción). Cuando aplicamos optimización dentro de un bloque básico sólo nos tenemos que preocupar sobre los efectos de la optimización en los valores de las variables a la entrada del bloque y los valores que tienen a la salida del bloque, que han de ser los mismos que en el código original sin transformar.
El algoritmo para particionar un programa en bloques se describe a continuación:

1. Encontrar todas las proposiciones que comienzan el principio de un bloque básico:
   - La primera sentencia del programa.
   - Cualquier proposición del programa que es el objetivo de un salto.
   - Cualquier proposición que sigue a una bifurcación.
2. Para cualquier proposición que comienza un bloque básico, el bloque consiste de esa proposición y de todas las siguientes hasta el principio del siguiente bloque o el final del programa.

El flujo de control de un programa puede visualizarse como un grafo dirigido de bloques básicos. A este grafo se le llama *grafo de flujo*. Como ejemplo consideremos este trozo de código escrito en *pseudoC* que suma los diez primeros números que se muestra en la tabla 7.7 (a):

void main()	i = 10
{	s = 0
int i,s;	label l1
i=10;	t0 = i > 0
s=0;	iffalse t0 goto l2
while i > 0 do	s = s + i
{	i = i - 1
s=s+i;	goto l1
i=i-1;	label l2
}	...
}	
(a)	(b)

Cuadro 7.7: Ejemplo (a) código fuente, (b) código de 3-direcciones

La figura 7.13 muestra el diagrama de flujo para el ejemplo anterior. Aparecen 4 bloques básicos.

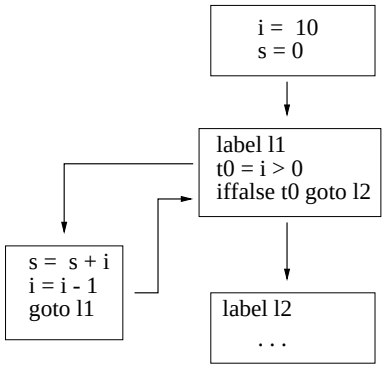


Figura 7.13: Diagrama de flujo

Algunas definiciones previas:

- Dada una proposición de 3-direcciones de la forma $a = b\ op\ c$ decimos que la proposición *referencia* b y c y que *define* a.
- Se dice que un nombre en un bloque básico *vive* al final del bloque si su valor es referenciado en otro bloque básico en el programa.
- Se dice que un nombre está *muerto* si no es referenciado en el resto del programa.

Se presentan algunas de las transformaciones más útiles para mejorar el código. Son transformaciones locales, se pueden realizar observando sólo las proposiciones de un bloque básico.

7.8.1. Eliminación de subexpresiones comunes

Si una expresión se calcula más de una vez, se puede remplazar el cálculo de la segunda por el valor de la primera expresión. Consideremos el siguiente ejemplo del código 7.8 (a). Vemos que la expresión $t3{*}t1$ se calcula dos veces, por tanto podemos escribir el código de la figura 7.8 (b):

t1 = 4 - 2	t1 = 4 - 2
t2 = t1 / 2	t2 = t1 / 2
t3 = a * t2	t3 = a * t2
t4 = t3 * t1	t4 = t3 * t1
t5 = t4 + b	t5 = t4 + b
t6 = t3 * t1	t6 = t4
t7 = t6 + b	t7 = t6 + b
c = t5 * t7	c = t5 * t5
(a)	(b)

Cuadro 7.8: Eliminación de subexpresiones comunes

Esto sólo es posible si los operandos que están implicados en el cálculo de la expresión no han modificado su valor en las proposiciones intermedias.

7.8.2. Propagación de copias

La propagación de copias considera las proposiciones de la forma $a = b$. Después de esta sentencia sabemos que a y b tienen el mismo valor, por tanto, podemos remplazar cada vez que aparezca a por b, con la esperanza de que podamos remplazar todas las ocurrencias de a hasta que se convierta en un nombre muerto y se pueda entonces eliminar la proposición de copia.
A partir del código anterior podemos eliminar la proposición de copia $t6 = t4$. Sustituimos $t6$ por $t4$. Vease figura 7.9 (a). Ahora se puede ver que hay de nuevo una subexpresión común que puede ser eliminada, obteniendo el código que se muestra en 7.9 (b). Y finalmente realizando otra vez propagación de copias, obtenemos 7.9 (c):

t1 = 4 - 2	t1 = 4 - 2	t1 = 4 - 2
t2 = t1 / 2	t2 = t1 / 2	t2 = t1 / 2
t3 = a * t2	t3 = a * t2	t3 = a * t2
t4 = t3 * t1	t4 = t3 * t1	t4 = t3 * t1
t5 = t4 + b	t5 = t4 + b	t5 = t4 + b
t6 = t4	t6 = t4	t6 = t4
t7 = t4 + b	t7 = t5	t7 = t5
c = t5 * t7	c = t5 * t7	c = t5 * t5
(a)	(b)	(c)

Cuadro 7.9: Propagación de copias y eliminación de subexpresiones comunes

Vemos que el haber hecho una optimización puede dar lugar a que sea posible aplicar nuevas optimizaciones.

7.8.3. Eliminación de código muerto

Podemos tener proposiciones que definen un nombre que nunca más es referenciado, está muerto. Estas proposiciones pueden entonces ser eliminadas. Dada la proposición $a = b\ op\ c$, se dice que es *código muerto o inactivo* si a no es referenciada. En general, el código muerto aparece como consecuencia de la propagación de copias y es esto lo que hace que la técnica de propagación de copias sea tan útil. Veamos como aplicar esta técnica al ejemplo anterior en la figura 7.9 (c).
Vemos que $t6$ y $t7$ no tienen ninguna referencia a partir de su definición, por tanto pueden ser eliminadas. Obtenemos el código que aparece en la figura 7.10. Se ha supuesto que todas las variables no-temporales están vivas, se hace referencia a ellas en el resto del programa.

```
t1 = 4 - 2
t2 = t1 / 2
t3 = a * t2
t4 = t3 * t1
t5 = t4 + b
c = t5 * t5
```

Cuadro 7.10: Eliminación de código muerto

Aunque es poco probable que el programador introduzca código muerto o inactivo, éste puede aparecer como resultado de transformaciones anteriores.

7.8.4. Transformaciones aritméticas

Se pueden hacer uso de transformaciones algebraicas simples para reducir la cantidad de computación transformando operaciones más costosas por otras menos costosas. Existen tres tipos de transformaciones algebraicas básicas.

Cálculo previo de constantes

Se trata de calcular a nivel de compilación el valor previo de constantes en vez de hacerlo en tiempo de ejecución que retardaría la ejecución de un programa. A esta optimización se le llama *cálculo previo de constantes* (en inglés *constant folding*). El código de la figura 7.10 puede ser mejorado dando lugar al código de la figura 7.11 (a). Haciendo una propagación de copias y eliminación de código muerto tenemos el código de la figura 7.11 (b). Realizando de nuevo un cálculo previo de constantes obtenemos 7.11 (c). Y finalmente podemos hacer de nuevo la propagación de copias y la eliminación de código muerto, obteniendo el código de la figura 7.11 (d).

t1 =2	t2 = 2 / 2	t2 = 1	t3 = a * 1
t2 = t1 / 2	t3 = a * t2	t3 = a * t2	t4 = t3 * 2
t3 = a * t2	t4 = t3 * 2	t4 = t3 * 2	t5 = t4 + b
t4 = t3 * t1	t5 = t4 + b	t5 = t4 + b	c = t5 * t5
t5 = t4 + b	c = t5 * t5	c = t5 * t5	
c = t5 * t5			
(a)	(b)	(c)	(d)

Cuadro 7.11: Cálculo previo de constantes

Hemos pasado de seis proposiciones a tener cuatro, eliminado una substracción y una división en el proceso.

Transformaciones algebraicas

Podemos hacer uso de identidades algebraicas para simplificar el código. Las principales identidades son:

$$x+0 = 0+x = x; \qquad x-0 = x; \qquad x*1 = 1*x = x; \qquad \frac{x}{1} = x$$

Partiendo de la figura 7.11 (d) podemos obtener el código de la figura 7.12 (a). De nuevo si usamos propagación de copias y eliminación de código muerto obtenemos el código de la figura 7.12 (b) :

t3 = a	t4 = a * 2
t4 = t3 * 2	t5 = t4 + b
t5 = t4 + b	c = t5 * t5
c = t5 * t5	
(a)	(b)

Cuadro 7.12: Identidades algebraicas

Reducción de intensidad

En la mayoría de las máquinas las operaciones de multiplicación y división son substancialmente más costosas que las operaciones de suma y resta. Y a su vez, las potencias son más costosas que las multiplicaciones y divisiones. Por tanto, siempre que sea posible es conveniente sustituir un operador más costoso por otro menos costoso. A esto se le conoce como *reducción de la intensidad*. Las identidades más comunes son:

$$x^2 = x * x; \qquad 2 * x = x + x$$

En nuestro ejemplo de la figura 7.12 (b) podemos obtener el código 7.13

```
t4 = a + a
t5 = t4 + b
c = t5 * t5
```

Cuadro 7.13: Reducción de intensidad

Otra transformación típica es usar desplazamientos de *bits* cuando se divida o se multiplique por potencias de dos.

7.8.5. Empaquetamiento de temporales

Se trata de reusar los temporales con el fin de ahorrar espacio. Después de haber optimizado el código es normal pensar que se puedan usar menos temporales y sería conveniente entonces renombrar estos temporales.

Supongamos que tenemos una tabla de temporales disponibles (por ejemplo de $t1$ a $t9$), de manera que marcamos si un temporal está vivo en un cierto punto del bloque básico. En general, es posible remplazar dos temporales por uno único si no existe ningún punto donde los dos temporales están vivos a la vez. Para cada temporal lo remplazamos por el primer temporal en la tabla que está muerto en todos los lugares en el que el temporal bajo consideración está vivo.

Veamos nuestro ejemplo de la figura 7.13. $t4$ se define en la primera proposición y está vivo en la segunda, entonces lo remplazamos por el primer terminal muerto de la tabla, $t1$, obteniendo el código de la figura 7.14 (a). Por otro lado, $t5$ es definido en la segunda proposición y vivo en la proposición 3. Esto no interacciona con el temporal $t1$, que esta vivo sólo en la segunda proposición, por tanto $t5$ puede ser sustituido por $t1$. Obteniendo el código de la figura 7.14 (b).

t1 = a + a	t1 = a + a
t5 = t1 + b	t1 = t1 + b
c = t5 * t5	c = t1 * t1
(a)	(b)

Cuadro 7.14: Empaquetamiento de temporales

Comparando este código con el original que tenía ocho proposiciones, siete temporales, dos adiciones, una substracción, cuatro multiplicaciones y una división, lo hemos reducido a tres proposiciones que implican dos adiciones y una multiplicación.

7.8.6. Mejora de los lazos

Traslado de código

Una modificación importante que disminuye la cantidad de código en un lazo es el traslado de código. Esta transformación toma una expresión que produce el mismo resultado independientemente del número de veces que se ejecute un lazo y la coloca antes del lazo. Por ejemplo:

```
while (i<=limite-2) ...
```

Se puede transformar en:

```
t=limite -2;
while (i<=t) ...
```

Variables de inducción

Se trata de identificar las variables que permanecen ligadas entre sí en la ejecución, están relacionadas entre sí de forma que conocido el valor de una se puede obtener el de la otra. Supongamos el siguiente fragmento de código donde se ha supuesto que los elementos de la matriz ocupan 4 bytes y ya se ha realizado cierta optimación.

```
suma=0;                          suma=0;
j=0;                             j=0;
while j<10                       label l1
      {                          t1= j<10
      suma = suma + a[j];        iffalse t1 goto l2
      j=j+1;                     t2=4*j
      }                          t3=a[t2]
...                              suma=suma+t3
                                 j=j+1
                                 goto l1
                                 label l2
                                 ...
```

Cuadro 7.15: (a) código fuente y (b) código de 3-direcciones

donde $a[t2]$ en la tabla 7.15 (b) significa la dirección de a más un desplazamiento $t2$. Sabemos que $t2$ no cambia en ningún momento salvo en $t2 = 4 * j$, por tanto justo después de la proposición $j = j + 1$ se cumple que $t2 = 4 * j + 4$, y se puede sustituir la asignación $t2 = 4 * j$ por $t2 = t2 + 4$, debiendo inicializar el valor de $t2 = 0$.

Siempre que haya variables de inducción se pueden hacer sustituciones de este tipo y eliminar todas menos una.

7.9. Ejercicios

1. (0.25 ptos) Escribe el pseudocódigo correspondiente, usando la notación que hemos visto a lo largo del capítulo, para la implementación de la generación de código de tres direcciones medinate cuadrúplos para la construccion `if-then-else`.

2. (0.25 ptos) Escribe el pseudocódigo correspondiente, usando la notación que hemos visto a lo largo del capítulo, para la implementación de la generación de código de tres direcciones medinate cuadrúplos para la construcción `repeat_until`.

3. (0.25 ptos) Escribe el pseudocódigo correspondiente, usando la notación que hemos visto a lo largo del capítulo, para la implementación de la generación de código de tres direcciones medinate cuadrúplos para la construcción `switch`.

4. (0.3 ptos) Supongamos una nueva construcción de los lenguajes de programación que llamaremos **do-while-do** (hacer-mientras-hacer). Esta construcción nace de forma natural, como las construcciones **while-do, do-while** que ya conocéis. Esta construcción surge para implementar la idea en que hay casos en los que no se desea salir de la iteración al principio, ni al final de la iteración, sino a la mitad, después de que se ha hecho cierto procesamiento.

   Por ejemplo para implementar el código:

```
dowhiledo
        read(X);
        while (no_final_fichero)
            process(X);
end dowhiledo
```

   La sintaxis de esta construcción es:

   $$S \rightarrow \textbf{ dowhiledo } S* \textbf{ while } E \ S* \textbf{ end dowhiledo } \mid \epsilon$$

   donde se ha usado el operador * para indicar cero o más repeticiones. Se pide:

- Dibujar el diagrama de flujo de esta construcción.
- Dibujar la forma del árbol abstracto de análisis sintáctico que usarías para su traducción a código de 3-direcciones.
- Escribir el pseudocódigo de la función `generar_código` para traducir este tipo de sentencias a una lista de cuádruplos.
- Desafortunadamente ningún lenguaje común de programación implementa esta construcción. ¿A qué crees que se debe esto?. ¿Qué construcción(es) usa el lenguaje C para implementar esta idea?

7.10. Ejemplo de generación de código para una calculadora usando PCCTS

Supongamos una pequeña calculadora que realizar operaciones aritméticas sencillas según la gramática:

entrada	→ (ecuacion)+
ecuacion	→ **id** = expresion **;**
expresion	→ termino (**"+"** termino | **"-"** termino)*
termino	→ factor (**"*"** factor | **"/"** factor)*
factor	→ "(" expresion ")" | dato
dato	→ **num** | **id** | **- num** | **- id**

Se pide implementar un traductor que traduzca las expresiones aritméticas a código de 3-direcciones, implementado mediante cuáduplos de la forma `(operador, resultado, arg1, arg2)`. Los tipos de operadores son:

(ASSIGN, result, arg1,NULL)	asigna arg1 a result
(ADD, result, arg1,arg2)	suma arg1, arg2 y lo alamacena en result
(SUB, result, arg1,arg2)	resta arg1, arg2 y lo alamacena en result
(MULT, result, arg1,arg2)	multiplica arg1, arg2 y lo alamacena en result
(DIV, result, arg1,arg2)	divide arg1, argg2 y lo alamacena en result
(NEG, result, arg1,NULL)	mult. por (-1) arg1, y lo alamacena en result
(HALT,NULL, NULL, NULL)	final de programa

Se ha implementado la clase cuadruplo (Cuad.h). Se ha implementado la clase TAC *(Three-Address-Code)*, a partir de una lista de cuaduplos y se ha introducido el `lugar` como un dato protegido adicional. Se ha implementado el método generar código en la clase AST. Fichero con la especificacion de la gramática p3.g.
Para la entrada: `a=3+2;`
`b=a*2;`
`c=a+b+2*6;`
`d=-1+a;`

Obtenemos la salida:
(ENTRADA (= a (+ 3 2)) (= b (* a 2)) (= c (+ (+ a b) (* 2 6))) (= d (+ (UMINUS 1) a)))
Numero de lineas analizadas 6

```
(ADD,t0,3,2)
(ASSIGN,a,t0,NULL)
(MULT,t1,a,2)
(ASSIGN,b,t1,NULL)
(ADD,t2,a,b)
(MULT,t3,2,6)
(ADD,t4,t2,t3)
(ASSIGN,c,t4,NULL)
(NEG,t5,1,NULL)
(ADD,t6,t5,a)
(ASSIGN,d,t6,NULL)
(HALT,NULL,NULL,NULL)
```

Numero de cuadruplos 12