UNIVERSIDAD NACIONAL AU
DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONAL
Y DE POSGRADO DEL COLEGIO DE CIENCIAS
Y HUMANIDADES

INSTITUTO DE INVESTIGACIONES EN
MATEMATICAS APLICADAS Y EN SISTEMAS

REGULARIZACION DE TIKHONOV A PARTIR DE DATOS

TESIS

QUE PARA OBTENER EL GRADO DE
MAESTRIA EN CIENCIAS
DE LA COMPUTACION
P R E S E N T A :
ING. ROBERTO SOLIS ORTA

DIRECTOR DE TESIS
DR. ALBERTO TORALBA ESPINOSA

MEXICO, D. F. DICIEMBRE 1999

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

ÍNDICE

CAPÍTULO I

INTRODUCCIÓN

La función de un compilador consiste en traducir un programa escrito en algún lenguaje (lenguaje fuente) a un lenguaje diferente (lenguaje objeto). Al programa que recibe como entrada el compilador se le llama programa fuente y normalmente está escrito en algún lenguaje de programación de alto nivel, mientras que al programa de salida se le llama programa objeto y usualmente está escrito en lenguaje de máquina.

El proceso de traducción es complejo y para facilitar su comprensión se divide conceptualmente en varias fases, cada fase realiza cierta transformación sobre el programa de entrada hasta obtener finalmente el programa objeto: el analizador léxico, por ejemplo, recibe como entrada un archivo de caracteres y se encarga de agruparlos en palabras, números, signos de puntuación y operadores del lenguaje fuente; el analizador sintáctico toma estos elementos y forma con ellos árboles de sintaxis, que son transformados en representación intermedia por la fase de generación de código intermedio y, finalmente, el generador de código traduce las instrucciones de lenguaje intermedio a código de máquina para producir como salida el programa objeto.

Las primeras fases del compilador: análisis léxico, análisis sintáctico, análisis semántico, generación de código intermedio y optimización de código intermedio, realizan tareas que dependen principalmente del lenguaje fuente, mientras que las últimas fases: generación y optimización de código objeto, dependen fuertemente de la máquina para la cual se genera código y son casi independientes del lenguaje fuente. Al primer grupo de fases se le denomina la parte frontal del compilador *(front end)*, mientras que al segundo grupo se le llama la parte posterior del compilador *(back end)*.

Agrupar las fases del compilador en estas dos clases tiene implicaciones prácticas interesantes. Por ejemplo, si se desea tener compiladores para algún lenguaje de programación en dos máquinas diferentes, no es necesario escribir dos compiladores completos distintos, sino

que se escribe solamente una parte frontal y dos partes posteriores (una para cada máquina), que se agregan a la parte frontal para tener los dos compiladores completos. En principio, para tener compiladores de n lenguajes diferentes para k máquinas distintas, no es necesario construir n x k compiladores completos; en vez de ello se escriben n partes frontales diferentes (una para cada lenguaje fuente), k partes posteriores (una por cada máquina) y se combinan en todas las formas posibles para tener los compiladores deseados.

La parte frontal y la posterior se comunican entre sí mediante una representación intermedia que puede estar formada por árboles, gráficas dirigidas sin ciclos *(dags)*, notación prefija, tríples, cuádruples, etcétera. La parte frontal del compilador traduce instrucciones de lenguaje fuente a representación intermedia, mientras que la parte posterior traslada cada construcción de la representación intermedia a lenguaje de máquina.

Actualmente existen herramientas tales como generadores de analizadores léxicos y generadores de analizadores sintácticos que facilitan la construcción de las fases correspondientes a la parte frontal del compilador; sin embargo, para simplificar la construcción de las fases de la parte posterior todavía no hay herramientas parecidas disponibles. En Estados Unidos, Alemania y otros países se está investigando la forma de simplificar el desarrollo de la parte de generación de código objeto de un compilador; en el siguiente capítulo se mencionan algunos de los trabajos reportados en este campo.

Cuando se intenta generar código objeto a partir de representación intermedia, hay tres problemas que deben ser considerados, pues su solución determina la calidad (rapidez o tamaño) del código resultante: decidir en que orden se traducirá cada parte de una instrucción del código intermedio a lenguaje objeto, escoger las instrucciones de máquina que se utilizarán para traducir cada instrucción de representación intermedia y, por último, elegir los registros que se utilizarán en las instrucciones de máquina seleccionadas. La mayoría de los algoritmos que se han propuesto hasta el momento para generar código de máquina a partir de lenguaje intermedio separan los dos primeros problemas del tercero, de manera tal que la solución que proponen al problema de generación de código se divide en dos fases: la primera fase crea un programa objeto incompleto (en cada instrucción falta indicar los registros del procesador que serán empleados), la segunda fase completa el código objeto eligiendo los registros que se utilizarán en cada instrucción; puesto que los procesadores tienen un número limitado de registros, en algunos casos se encontrará que el conjunto de instrucciones seleccionado por la primera fase requiere más registros que los disponibles en el procesador y por lo tanto debe modificarse tal conjunto, ocasionado en algunos casos que el código resultante no sea muy bueno.

El objetivo de la presente tesis consistió en diseñar un algoritmo que considere los tres problemas anteriores al resolver el problema de generación de código y crear con tal algoritmo

una herramienta que facilite la construcción de la parte posterior de un compilador. El algoritmo de generación de código desarrollado, además, es capaz de tomar en cuenta la presencia de subexpresiones comunes en el programa de entrada (subexpresiones que aparecen varias veces en la entrada y para las cuales no debe emitirse código cada vez que se encuentren, para así producir el código más eficiente posible).

La herramienta construida recibe como entrada un programa escrito en representación intermedia junto con un conjunto de tablas que contienen la especificación de algún procesador y produce como salida un programa equivalente al de entrada escrito en lenguaje ensamblador del procesador que se especificó. La herramienta puede generar código de alta calidad, comparable al producido por compiladores comerciales, para procesadores de propósito general con instrucciones de dos direcciones (en el capítulo 4 se presenta el modelo que establece con precisión la clase de procesadores que pueden ser manejados por la herramienta).

Este reporte está organizado en la siguiente forma. En el capítulo 2 se explican algunos de los algoritmos de generación de código existentes. En el capítulo 3 se presenta el sistema desarrollado, describiéndose la parte frontal construida para un subconjunto del lenguaje C, así como los módulos encargados de manipular la representación intermedia para ponerla en la forma requerida por el generador de código. En el capítulo 4 se describe el algoritmo de generación de código, indicando antes que clase de procesadores puede manejar y como se da la especificación de uno. En el capítulo 5 se presentan los resultados de las pruebas realizadas con el generador de código y se dan las conclusiones.

ALGORITMOS DE GENERACIÓN DE CÓDIGO EXISTENTES

Los principales problemas que deben resolverse cuando se genera código objeto a partir de representación intermedia son:

1. Elegir el orden de evaluación de las instrucciones del código intermedio.
2. Seleccionar las instrucciones de máquina que serán emitidas.
3. Asignar los registros que se usarán en las instrucciones de máquina seleccionadas.

La posibilidad de traducir una instrucción de lenguaje intermedio a código objeto en formas diferentes, la variedad de modos de direccionamiento de la memoria y la existencia de registros asimétricos (registros que sólo pueden usarse en ciertas instrucciones de máquina), son aspectos que complican la tarea de generación de código.

El problema de la generación de código óptimo consiste en hallar el código objeto de menor costo equivalente a un programa fuente dado. El costo del código se define como su tamaño en bytes o bien como el tiempo que el procesador necesita para ejecutarlo.

Algunos investigadores han tratado de averiguar qué tan difícil es generar código óptimo, encontrando que para máquinas con instrucciones de dos direcciones, generar código óptimo para programas con subexpresiones comunes, es un problema NP completo. Más aún, se ha demostrado que si existiera una máquina con un número infinito de registros, generar código óptimo para ella no sería más sencillo que generar código para una máquina real; es decir que, aún suponiendo que la asignación de registros sea trivial, el problema consistente en hallar el orden de evaluación óptimo y hacer la mejor selección de instrucciones posible, es un problema intratable.

Se han propuesto varios algoritmos para generar código de alta calidad. Todos ellos imponen diversas restricciones al problema de generación de código, intentando con esto que los programas construidos con esos algoritmos sean razonablemente rápidos. La mayor parte de estos trabajos hacen énfasis en el problema de selección de instrucciones, dedicando poca atención a los problemas de asignación de registros y determinación del orden de evaluación.

Para la elección de instrucciones algunos algoritmos utilizan programación dinámica, intentando con esto formar código con el costo más pequeño posible, mientras que otros prefieren usar criterios heurísticos buscando que la selección se efectúe muy rápidamente.

A fin de facilitar la creación de generadores de código para diferentes procesadores, los algoritmos separan la especificación de la máquina del método de generación de código; de esta forma, con sólo cambiar la especificación de la máquina se tendrá un generador de código diferente. En este capítulo se describen varios algoritmos de generación de código a fin de dar un panorama general del estado del arte actual y comprender por qué en esta tesis se dio énfasis a ciertos aspectos del problema de generación de código.

2.1 Algoritmo de Graham-Glanville [Grah82, Gana82,Grah84]

Un algoritmo de generación de código necesita conocer ciertas características del procesador para el que se emitirá código tales como las clases de registros que existen, los modos de direccionamiento de la memoria, las instrucciones de máquina, etcétera. Esta información, que puede suministrarse en varias formas, es utilizada por el algoritmo para elegir el código que se emitirá para cada instrucción del programa de entrada.

En el método de Graham-Glanville, la descripción de la máquina se especifica como una gramática independiente del contexto y un esquema de traducción dirigida por sintaxis. Un analizador sintáctico para esta gramática puede tomar como entrada el programa en representación intermedia y emitir código objeto a medida que reconoce cada una de las construcciones gramaticales presentes en el mismo.

Cada regla de producción de la gramática representa una instrucción de lenguaje intermedio; el lado izquierdo de la producción corresponde al resultado, el lado derecho corresponde a la operación y operandos. Asociada a cada regla de producción se da un conjunto de instrucciones de máquina equivalentes a la instrucción de representación intermedia y, opcionalmente, ciertas condiciones semánticas que deben ser cumplidas para poder aplicar la producción al estar realizando análisis sintáctico. Cada vez que el analizador sintáctico aplica alguna de las reglas de producción, se emite el código asociado a esa producción, de manera que al terminar el análisis sintáctico ya se habrá completado el programa objeto.

La producción mostrada enseguida, por ejemplo, indica que al sumar un valor almacenado en un registro con una constante el resultado queda en un registro. El código en lenguaje ensamblador del procesador 8086 de Intel que debe emitirse para realizar tal operación se indica en la segunda columnay será emitido al estar realizando el análisis sintáctico de la entrada sólo si la condición semántica asociada es satisfecha, es decir si el segundo operando tiene el valor constante 1.

Producción	Código	Condiciones semánticas
Registro --> Suma Registro Constante	inc Reg	Constante = 1

Debido a que, en general, hay muchas formas diferentes de traducir una instrucción de representación intermedia a lenguaje de máquina, pueden existir varias producciones asociadas a la misma operación de código intermedio, ocasionando esto que la gramática resultante sea ambigua.

En el método de Graham-Glanville se utiliza un generador de analizadores sintácticos para crear un analizador sintáctico, que será el encargado de emitir código objeto. Puesto que la gramática es ambigua, cuando se están creando las tablas del analizador sintáctico se deben resolver conflictos que resultan debido a que es posible aplicar más de una regla de producción para formar el árbol de sintaxis de algunas entradas. Se usan ciertos criterios heurísticos para resolver estas situaciones, intentando siempre que el código resultante sea lo más eficiente posible.

Cuando el conflicto surge entre hacer una reducción o tomar otro símbolo del lado derecho de una producción *(shift-reduce conflict)*, se decide tomar el siguiente símbolo, prefiriendo así utilizar las producciones más largas, esperando con esto que la secuencia de instrucciones de máquina sea lo más pequeña posible. Por la mismo razón, los conflictos que se presentan cuando hay más de una reducción posible *(reduce-reduce conflict)*, son resueltos en favor de la reducción correspondiente a la producción más larga. Si las producciones involucradas tienen la misma longitud se prefiere utilizar la que aparece primero en la gramática (debido a esto las producciones deben ordenarse, colocando primero aquellas que tengan asociado el código de máquina de menor costo).

Al estar creando las tablas del analizador sintáctico se deben prever todos los problemas

que puedan presentarse durante la generación de código. Puede suceder, por ejemplo, que en la gramática se encuentren las siguientes producciones :

X —> Y
Y —> Z
Z —> X

el analizador sintáctico se quedará en un ciclo infinito si intenta aplicar consecutivamente las reglas. Este problema puede detectarse durante la construcción del analizador sintáctico, encontrando la cerradura transitiva de las producciones.

Es posible también que el analizador sintáctico se bloquee, no pudiendo utilizar ninguna regla de producción para completar el análisis del programa de entrada. Hay dos razones por las cuales puede ocurrir bloqueo:

1. Dado que la gramática es ambigua, para ciertas entradas el analizador sintáctico encontrará que es posible aplicar varias reglas de producción; la decisión de cuál regla de producción usar, se realiza con base en las longitudes de las producciones o a los costos de las instrucciones asociadas. Es posible, entonces, que se elija incorrectamente la producción y que el analizador sintáctico llegue a un estado en el cual deba señalarse un error porque ninguna producción puede aplicarse para continuar el análisis sintáctico de la entrada. Puesto que los programas en representación intermedia no pueden tener errores sintácticos (están formados por instrucciones válidas del lenguaje intermedio), al llegar a un estado de error, el analizador sintáctico no puede proseguir el análisis del programa de entrada y se bloquea. A esta clase de bloqueo se le llama bloqueo sintáctico.

2. Es posible también, que el analizador llegue a un estado en el cual las únicas reglas de producción que pueden ser utilizadas tienen acciones semánticas asociadas que no se cumplen, por lo tanto ninguna de ellas se puede aplicar y el análisis no puede proseguir. A esta clase de bloqueo se le llama bloqueo semántico.

Ambas clases de bloqueo pueden corregirse agregando producciones a la gramática. En el caso de bloqueo semántico se agregan producciones que no tienen asociadas condiciones semánticas y por tanto siempre pueden aplicarse; para eliminar el bloqueo sintáctico, se agregan reglas con el fin de eliminar de la tabla del analizador las entradas que estan marcadas con acciones de error, de manera que siempre pueda continuarse el análisis sintáctico. Existen algoritmos que permiten detectar y corregir los bloqueos sintácticos al momento de estar construyendo las tablas del analizador sintáctico, sin embargo los bloqueos semánticos se deben corregir manualmente.

2.1.1 Ventajas del método de Graham-Glanville

1. El algoritmo de Graham-Glanville tiene complejidad temporal lineal, por lo cual los generadores de código construidos con esta técnica son muy rápidos.

2. Si se desea construir un generador de código para algún procesador, sólo se necesita escribir la gramática que describe a tal procesador. Las herramientas que existen actualmente permiten detectar los ciclos y bloqueos sintácticos presentes en las producciones e incluso modifican la gramática para eliminarlos. El generador de código se construye automáticamente a partir de la gramática, utilizando un generador de analizadores sintácticos.

3. Se han realizado varios generadores de código con este método; las pruebas que se han efectuado con ellos muestran que el código generado es de muy buena calidad.

4. Se ha demostrado que si la descripción de la máquina es correcta, entonces el algoritmo de Graham-Glanville siempre emitirá código correcto.

2.1.2 Limitaciones del método de Graham Glanville

1. La gramática para describir una máquina puede tener cientos de reglas de producción, por lo que no es muy sencillo construir un nuevo generador de código.

2. El método de análisis sintáctico determina cuáles producciones serán usadas para emitir código, por lo tanto el algoritmo sólo considera una manera de traducir el programa fuente a lenguaje de máquina. Por esta razón, en algunos casos se puede emitir código que no resulte muy bueno.

3. Para algunas construcciones de representación intermedia, no puede generarse código de alta calidad usando análisis sintáctico independiente del contexto. Por ejemplo se emitiría la misma secuencia de instrucciones de máquina para la expresión lógica a & b, aunque esta se utilizara en el programa fuente en instrucciones diferentes como c = (a & b), o bien, if (a & b). El código debería ser distinto pues en el segundo caso el resultado de la operación lógica no se debe almacenar en la memoria, como ocurre con la primera instrucción.

El algoritmo tampoco puede generar código correctamente para los saltos condicionales o incondicionales, pues es necesario conocer la distancia del salto. Para el procesador 8086, por ejemplo, los saltos condicionales sólo pueden hacerse a una instrucción que diste menos de 128 bytes de la instrucción de salto.

4. El código intermedio debe ser transformado, en ciertos casos, para hacer posible la generación de código correcto. Por ejemplo, se debe hacer explícito el control de flujo; si en el programa fuente aparece la instrucción if (a > b && c > d), en el código intermedio debe agregarse un salto después de evaluar a > b, de esta manera, si el resultado de la comparación es falso, no se intenta evaluar la segunda parte de la condición puesto que no es necesario.

   Los operadores de la representación intermedia que no pueden traducirse fácilmente a lenguaje de máquina, deben modificarse para facilitar la emisión de código. Si, por ejemplo, la operación especificada en alguna instrucción de lenguaje intermedio puede ser efectuada por alguna función de biblioteca, se debe cambiar la instrucción por una llamada a tal función. Estas transformaciones del código intermedio no son completamente independientes de la máquina, por lo cual es posible que deba reescribirse la parte del generador de código que realiza tales transformaciones cada vez que se quiera generar código para una máquina diferente.

5. El lenguaje intermedio, que es de muy bajo nivel, contiene información sobre la máquina para la cual se emitirá código. Las fases de la parte frontal, por ejemplo, asocian una dirección de la memoria a cada variable del programa fuente; esta información se envía al generador de código a través de la representación intermedia.

   Que el programa intermedio tenga información sobre la máquina, implica que si se desea generar código para un procesador diferente o si se desea cambiar la forma de los registros de activación usados en las llamadas a funciones, será necesario cambiar la forma de la representación intermedia también.

6. El algoritmo de Graham-Glanville no incorpora asignación de registros. En este método, asignar registros y elegir instrucciones se efectúan como pasos separados, ocasionando que algunas veces se deban hacer respaldos innecesarios en la memoria *(spilling)* de los valores guardados en algunos registros.

7. Las subexpresiones comunes no son detectadas, por lo cual los valores que se dejan guardados en registros no se vuelven a utilizar. Cada vez que un valor se necesita en un registro, se emite una instrucción de carga.

2.1.3 Generadores de código construidos con el algoritmo de Graham-Glanville

El algoritmo de Graham-Glanville se ha utilizado en la construcción de varios generadores de código; podemos clasificar esos trabajos en dos grupos:

1. Los que utilizan principalmente sintaxis para describir la máquina, empleando pocas o ninguna condición semántica y
2. Los que basan la descripción del procesador en condiciones y atributos semánticos asociados a las reglas gramaticales.

Ambos enfoques tienen ventajas y desventajas que serán mencionadas enseguida.

2.1.3.1 Enfoque sintáctico

Al enfatizar el empleo de producciones gramaticales sobre acciones y atributos semánticos para dar la descripción de un procesador, el problema principal que se presenta consiste en que la gramática resultante es demasiado grande. Para comprender por qué, consideremos el siguiente ejemplo.

El micropocesador 68000 de Motorola permite realizar operaciones sobre valores que ocupan un byte, dos bytes *(word)* o cuatro bytes *(longword)*. Para indicar el tamaño de los operandos, al código en lenguaje ensamblador de la operación se le agrega la especificación del tamaño en la siguiente forma:

1. Si los operandos ocupan un byte cada uno, al código de la operación se le añade un punto seguido de la letra B.
2. Si los operandos son de dos bytes, se agrega un punto seguido de la letra W.
3. Si cada operando es de cuatro bytes, se agrega un punto seguido de la letra L.

En lenguaje ensamblador de esta máquina, sub representa la operación de resta. Para restar dos valores de un byte, el código correspondiente es sub.B; para restar dos operandos de

dos bytes cada uno, sub.W; y para restar valores de cuatro bytes se debe escribir sub.L. Las reglas gramaticales que especifican la forma de restar dos valores guardados en registros se indican en la tabla 2.1.

Producción	Código
Registro.1 -> Restabyte Registro.1 Registro.2	Sub.B reg.2, reg.1
Registro.1 -> Restaword Registro.1 Registro.2	Sub.W reg.2, reg.1
Registro.1 -> Restalong Registro.1 Registro.2	Sub.L reg.2, reg.1

Tabla 2.1

La gramática que describe al procesador 68000 de Motorola es grande debido a que por cada operador cuyos operandos puedan ocupar uno, dos o cuatro bytes, es necesario agregar tres producciones diferentes. Por la misma razón, si un procesador tiene registros asimétricos, por cada operación binaria que pueda tomar ambos operandos de registro, en el peor caso es necesario escribir k * k producciones diferentes, donde k es el número de clases de registros que tenga el procesador. Cada producción tendrá la forma

Registro_clase_i --> Operación Registro_clase_j Registro_clase_m

Registro_clase_j y Registro_clase_m tienen k valores posibles cada uno por lo que hay k * k combinaciones posibles. El registro utilizado para el resultado debe ser igual al utilizado por el segundo operando.

Para tratar de reducir el número de reglas de producción en la descripción de la máquina, se factoriza la gramática, juntando las producciones que tengan forma parecida. Para el procesador 68000 de Motorola, por ejemplo, las operaciones de multiplicación y división de enteros con signo y sin signo necesitan que ambos operandos sean de dos bytes cada uno; el primer operando se debe poner en un registro de datos, y el segundo puede estar en la memoria, en un registro de datos o bien, ser una constante. El resultado de la operación se deja en el registro del primer operando. Las siguientes producciones especifican las clases de operandos válidos

```
REGISTRO.1 --> MUL_DIV_WORD  REGISTRO.1  REGISTRO.2
REGISTRO.1 --> MUL_DIV_WORD  REGISTRO.1  MEMORIA
REGISTRO.1 --> MUL_DIV_WORD  REGISTRO.1  CONSTANTE
```

Cuatro producciones más indican cuáles son los operadores de multiplicación y división

```
MUL_DIV_WORD --> Muls
MUL_DIV_WORD --> Mulu
MUL_DIV_WORD --> Divs
MUL_DIV_WORD --> Divu
```

Si para cada operador se hubieran escrito las tres producciones que indican cuales pueden ser los operandos, la gramática, en vez de siete, tendría doce producciones.

Cuando se utiliza solamente sintaxis para formar la descripción de la máquina, la gramática resultante tiene muchas reglas de producción; la ventaja de esto, sin embargo, consiste en que es posible utilizar las herramientas existentes para detectar y corregir ciclos y bloqueos, puesto que tales herramientas funcionan sólo si no hay condiciones semánticas asociadas a las producciones de la gramática.

En [Grah84] se reporta la construcción de generadores de código para tres máquinas diferentes usando el algoritmo de Graham-Glanville. Para probar los generadores se usó la parte frontal de un compilador del lenguaje C. La tabla 2.2 muestra algunas características de las gramáticas que describen a esas máquinas.

	VAX-11	68000 Motorola	RISC-II
Número de producciones	661	529	287
Número de estados en la tabla del analizador sintáctico	1257	807	479
Conflictos			
shift-reduce	24482	8082	1628
reduce-reduce	8336	9003	2679
Número de bloqueos detectados	25	12	5

Tabla 2.2

2.3.2 Enfoque semántico

Ganapathi y Fischer [Gana82] extendieron el algoritmo de Graham-Glanville, haciendo que la descripción de la máquina se especifique mediante una gramática con atributos y no usando solamente una gramática independiente del contexto. La presencia de atributos permite establecer en forma limitada el contexto en el que se utiliza una instrucción, haciendo posible detectar subexpresiones comunes, tomar en cuenta efectos laterales de una instrucción y facilitar la asignación de registros.

Al estar realizando el análisis sintáctico del programa en representación intermedia, se evalúan los atributos asociados a los diferentes símbolos usados en la gramática; estos atributos pueden usarse para facilitar la asignación de registros como se muestra enseguida.

Para el procesador 8086 de Intel, por ejemplo, la multiplicación de dos números de dos bytes cada uno, requiere que el primer operando se guarde en el registro ax, el segundo operando puede ser un registro, o bien estar guardado en la memoria y, finalmente, el resultado se deja en la pareja de registros dx,ax (pues el resultado puede ser de 32 bits), la parte alta del resultado se deja en el registro dx y la parte baja en el registro ax. Las producciones necesarias para emitir código correspondiente a la multiplicación y a la suma se muestran en la tabla 2.3.

Producción	Condiciones semánticas
Registro --> Multiplica Registro.1 Operando.2	Registro.reg = dx:ax Registro.1.reg = ax Operando.2 = registro o memoria ALTERA Registro.1
Registro --> Suma Registro.1 Operando.2	Registro.reg = Registro.1.reg Operando.2 = registro, memoria o constante ALTERA Registro.1

Tabla 2.3

El símbolo no terminal Registro (Registro y Registro.1 son el mismo símbolo no terminal, la numeración se coloca sólo para comprender más fácilmente las condiciones semánticas) tiene asociado un atributo llamado reg, cuyo valor indica cuál es el registro de la máquina que se usa para guardar el valor del operando .

Para poder aplicar alguna de las producciones anteriores, es necesario que las condiciones semánticas asociadas a ellas se cumplan. Así, para la primera producción, el segundo operando debe ser un registro o estar guardado en la memoria y el primer operando debe estar almacenado en el registro ax. Para cumplir la segunda condición semántica de la primera producción se revisa que el registro ax esté libre (si no es así, su valor se pasa a un registro diferente o se guarda en una celda de la memoria) y se almacena el valor del primer operando en él.

Considérese la expresión (a * b) + c, para producir el código correspondiente, el analizador sintáctico debe utilizar la primera producción, emitiendo código para guardar el valor de a en el registro ax y después emite código para multiplicar este valor con b. En el atributo reg del símbolo no terminal REGISTRO se recuerda que el resultado de la operación queda en la pareja de registros dx:ax; en esta forma, al usar la segunda producción para emitir el código de la suma ya se conoce el registro donde se encuentra guardado el primer sumando.

El analizador sintáctico tiene un descriptor de registros donde se indica, entre otras cosas, qué valor está guardado en cada registro. Cuando una variable o constante se debe almacenar en un registro, se revisa primero el descriptor para saber si tal valor ya está guardado en uno, evitando así, en algunos casos emitir operaciones de carga a registro. El operador ALTERA indica que el contenido de un registro será modificado al efectuar el código asociado a la producción, de manera tal que si el registro almacenaba el valor de una subexpresión común, la próxima vez que se encuentre tal subexpresión su valor deberá ser calculado de nuevo o tomado de la memoria.

Si una instrucción de máquina afecta los bits del registro de banderas del procesador, el uso de una condición semántica permite al analizador sintáctico guardar esa información en el atributo de algún símbolo y utilizarla para tratar de generar código más eficiente.

El empleo de condiciones semánticas, además, permite reducir el tamaño de la gramática. Por ejemplo, para el procesador 68000 de Motorola la resta de dos valores de cualquier tamaño válido guardados en registros se puede especificar con una sola producción.

PRODUCCIÓN	CÓDIGO
REGISTRO → RESTA REGISTRO.2 REGISTRO.1	SUB.$ registro.2,registro.1

El símbolo no terminal REGISTRO tiene un atributo tamaño, que indica el número de bytes que ocupa el valor que representa; el símbolo $ debe reemplazarse por el descriptor del

tamaño correcto de los operandos. Las acciones semánticas necesarias para dar valor al atributo y para encontrar el valor de $ son las siguientes:

```
REGISTRO.tamaño = REGISTRO.1 .tamaño
SI REGISTRO.tamaño = 1 ENTONCES $ = B
OTRO SI REGISTRO.tamaño = 2 ENTONCES $ = W
    OTRO $ = L
```

Por cada operador cuyos operandos puedan ser de los tres tamaños válidos, será necesario formar solamente una producción con una acción semántica similar a la anterior, obteniéndose una gramática con menos producciones que la creada sin el empleo de condiciones semánticas. Si el procesador tiene registros asimétricos, el empleo de condiciones semánticas también puede ayudar a mantener pequeño el tamaño de la gramática.

Las desventajas del enfoque semántico consisten en que el generador de código es complejo, pues debe evaluar e interpretar correctamente las condiciones semánticas y, además, no es posible detectar y corregir automáticamente los bloqueos que puedan ocurrir durante la generación de código.

En [Bird82] y en [Land82] se reporta la construcción de generadores de código para compiladores del lenguaje Pascal usando el enfoque semántico. En el primer trabajo se probó el generador de código para la Amdahl-470 y en el segundo se generó código para dos procesadores diferentes, el 68000 de Motorola y el 7.000 de Siemens. La tabla 2.4 contiene información sobre las gramáticas que describen a los procesadores.

	Amdhal 470	Motorola 68000	Siemens 7.000
Número de producciones	248	130	150
Número de estados de la tabla del analizador sintáctico	810	154	149
Número de operadores semánticos	29	64	87

Tabla 2.4

La comparación de esta tabla con la tabla 2.2 muestra que el número de producciones de la gramática es mucho menor si se emplean condiciones semánticas.

2.2 ALGORITMO DE CHRISTOPHER-HATCHER-KUKUK [Chri84]

Cuando se desea construir un generador de código utilizando el algoritmo de Graham-Glanville, los principales problemas que se deben resolver al estar escribiendo la gramática que describe al procesador, consisten en detectar y corregir los ciclos, bloqueos sintácticos y bloqueos semánticos. Christopher, Hatcher y Kukuk sugieren que estos problemas se presentan por el método de análisis sintáctico que se usa y para evitarlos proponen la utilización del algoritmo de Earley en el generador de código.

El algoritmo de Earley es un método de análisis sintáctico capaz de manejar gramáticas ambiguas. Para un programa de entrada, el algoritmo de Earley construye en paralelo todas las posibles derivaciones de manera tal, que los bloqueos sintácticos y semánticos no pueden ocurrir, puesto que si hay una derivación para la entrada el algoritmo la encontrará. Este algoritmo, además, puede manejar correctamente conjuntos de producciones con ciclos de la forma $X \rightarrow Y$, $Y \rightarrow Z$, $Z \rightarrow X$. En consecuencia, si se utiliza este método de análisis sintáctico no es necesario contar con herramientas especiales para detección y corrección de ciclos y bloqueos, ya que tales problemas son resueltos automáticamente durante el análisis sintáctico.

Se define un ítem como una producción con un marcador colocado en alguna posición del lado derecho que indica cuánto de la producción se ha utilizado en algún momento del análisis sintáctico. Por ejemplo, si una gramática tiene la producción $S \rightarrow a\ b$, siendo S el símbolo inicial y a, b símbolos terminales, dada una entrada el analizador sintáctico se encarga de revisar si está o no bien formada de acuerdo a la gramática. Al comenzar el análisis no ha sido leído ningún símbolo de la entrada; para indicar tal estado se coloca el marcador (que se denotará con un punto) al principio de la producción: $S \rightarrow . \ a\ b$; se compara después el primer símbolo de la entrada con el primero del lado derecho de la producción y si son iguales se debe correr el punto un lugar a la derecha para indicar cuánto de la producción ha sido utilizada hasta este momento; el nuevo ítem es $S \rightarrow a . b$.

El algoritmo de Earley encuentra todas las derivaciones de un programa (o, equivalentemente, todos los árboles de análisis sintáctico), formando lo que se denomina conjuntos configuración. Si el programa está formado por n - 1 símbolos entonces será necesario formar n conjuntos configuración $C_1, C_2, \ldots C_n$; el conjunto configuración C_i representa todas las derivaciones cuyos primeros i-1 símbolos terminales son iguales a los del programa de entrada.

Los conjuntos configuración se construyen inductivamente, es decir cada conjunto configuración, salvo el primero, se forma a partir de los conjuntos configuración anteriores. Un conjunto configuración está formado por parejas [I,M], cuyo primer elemento I es un ítem y el segundo elemento es el número de conjunto configuración utilizado para crear la pareja.

El algoritmo que construye los conjuntos configuración para un programa de entrada es el siguiente:

Algoritmo conjuntos_configuración

1. Si el símbolo inicial de la gramática es S, entonces, en C_1 se agregan todas las parejas [I,0], donde I es un ítem formado con una producción de S cuyo primer símbolo del lado derecho es un símbolo no terminal o un símbolo terminal igual al primer símbolo de la entrada y se coloca el marcador al inicio del lado derecho de la producción.

   Si en algún ítem a la derecha del marcador se encuentra un símbolo no terminal N, se agregan a C_1 las parejas [I,1], donde I está formado con una producción de N que tiene el marcador al inicio del lado derecho de la producción. Sólo se consideran las producciones de N que comienzan con un símbolo no terminal o bien con un símbolo terminal que es igual al primer símbolo de la entrada.

2. Los conjuntos C_j, para $2 \leq j \leq n$, se obtienen a partir de C_{j-1}, de la siguiente forma.

   i. Se lee el siguiente símbolo de la entrada y se buscan los ítems de C_{j-1} que tengan situado inmediatamente a la derecha del marcador el símbolo leído. Todas las parejas con esos ítems se copian en C_j, moviendo el marcador una posición a la derecha; el segundo elemento de la pareja no cambia.

   ii. Si en la pareja [I,k] de C_j el ítem I tiene el marcador situado al final de la producción ($N \rightarrow w$.), entonces se buscan en C_k las parejas [J,m] en las cuales el ítem J tenga el marcador situado inmediatamente a la izquierda del no terminal N; esas parejas se copian en C_j, moviendo antes el marcador una posición a la derecha.

   iii. Si en alguno de los ítems de C_j, a la derecha del marcador está un símbolo no terminal N, se agregan a C_j las parejas [I,J], donde I se forma con una producción de N colocando el marcador al inicio del lado derecho de la producción. Sólo se toman las producciones de N cuyo primer símbolo del lado derecho sea un símbolo no terminal o bien un símbolo terminal igual al siguiente símbolo de la entrada.

Termina_algoritmo.

El programa que sera analizado y los lados derechos de las producciones normalmente se

presentan en notación prefija. Con objeto de mostrar cómo se forman los conjuntos configuración, usemos el algoritmo anterior para la entrada:

+ a + b 1

y la gramática mostrada en la tabla 2.5.

Producción
R --> + R constante
R --> + R 1
R --> + R_1 R_2
R --> variable

Tabla 2.5

Primero se toman las producciones de R cuyo lado derecho comience con el símbolo '+' ya que éste es el primer símbolo de la entrada o bien que comience con un símbolo no terminal. Se forman los ítems colocando el punto al inicio del lado derecho de tales producciones y se agregan a C_1 las parejas formadas con estos ítems usando como segundo elemento al cero.

C_1 = {[R → . + R constante,0], [R → . + R 1,0]. [R → . + R_1 R_2,0] }.

Se busca en C_1 las producciones que tiene el símbolo '+' inmediatamente a la derecha del punto y se agregan a C_2 las parejas correspondientes, moviendo antes el marcador una posición a la derecha:

C_2 = {[R → + . R constante,0], [R → + . R 1,0],[R → + . R_1 R_2,0]}.

Puesto que hay producciones que tienen el símbolo no terminal R colocado a la derecha del punto, se buscan las producciones de R que comiencen con un símbolo no terminal o con una variable (en el ejemplo, el siguiente símbolo de la entrada es una variable), encontrándose la cuarta producción, que se utiliza para agregar a C_2 la pareja [R --> . variable,2]. Los demás conjuntos configuración se listan enseguida:

C3 = {[R → variable . , 2],[R → + R . constante, 0], [R → + R . 1, 0], [R → + R_1 . R_2, 0],
[R → . + R constante, 3], [R → . + R 1 , 3]. [R → . + R_1 R_2 , 3]}

C4 = {[R → + . R constante , 3], [R → + . R 1, 3], [R → + . R_1 R_2, 3], [R → . variable, 4]}

C5 = {[R → variable . , 4],[R → + R . constante, 3],[R → + R . 1, 3],[R → + R_1 . R_2, 3]
[R → . variable, 5]}

$C6 = \{[R \rightarrow + R\ 1\ .\ , 3], [R \rightarrow + R \text{ constante}, 3], [R \rightarrow + R_1\ R_2\ .\ , 0]\}$

Los árboles de análisis sintáctico correspondientes a la entrada pueden obtenerse a partir de los conjutos configuración usando el siguiente algoritmo.

Algoritmo árboles_de_análisis_sintáctico

1. Se coloca como raíz del árbol al símbolo inicial de la gramática.
   A la variable k se le da el valor de n (n es el número de conjuntos configuración).
2. Se busca la siguiente hoja del árbol que esté situada más a la derecha.

   i. Si la hoja es un símbolo terminal entonces se hace k = k - 1.
   ii. Si la hoja es un símbolo no terminal N, entonces, se busca en C_k la producción que tenga a N del lado izquierdo, el marcador situado al final de la producción y con el número de conjunto configuración más pequeño (si hay varias parejas con el mismo número de conjunto configuración, entonces es posible formar varios árboles de análisis sintáctico). Por cada símbolo del lado derecho de esa producción, se crea un nodo que se coloca como hijo del nodo correspondiente a N.

3. Se repite el paso 2 hasta que k sea igual a 1.

Termina_algoritmo

Para formar los árboles de análisis sintáctico del ejemplo anterior, primero se hace k = 6 y se coloca a R como raíz del árbol (ver figura 2.1a). La única hoja es la raíz y como tiene asociada un símbolo no teminal, se busca en C_6 las producciones de R con el punto colocado al final del lado derecho. La pareja elegida es $[R \rightarrow + R_1\ R_2\ .\ , 0]$ puesto que el número de conjunto configuración asociado es el más pequeño. Utilizando la producción indicada se forma el árbol de la figura 2.1b.

Se toma la hoja situada más a la derecha y como representa un símbolo no terminal, se buscan en C_6 las producciones de ese símbolo que tengan colocado el marcador al final del lado derecho (ya no se considera el ítem anterior). Hay dos producciones que cumplen con la condición anterior :

i. $R \rightarrow + R\ 1$
ii. $R \rightarrow + R$ constante

por consiguiente es posible formar los dos árboles de sintaxis mostrados en la figura 2.1c.

Continuando con el algoritmo, al final se forman los árboles de la figura 2.1d. En [Aho72] se demuestra que el algoritmo de Earley construye los árboles de sintaxis en tiempo $O(n^3)$ si el programa de entrada tiene subexpresiones comunes y en tiempo $O(n^2)$ si no las tiene.

Los árboles obtenidos difieren solamente en la producción que fué utilizada en el nodo señalado con la flecha (ver figura 2.1d), es por ello que el algoritmo de Earley no crea dos árboles completamente separados, sino una gráfica formada compartiendo los nodos comunes de los árboles.

Al especificar la gramática que describe algún procesador, a cada producción se le asocia una acción semántica que indica:

1. El conjunto de instrucciones de máquina que será emitido cuando se utilice la regla de producción durante la generación de código.
2. El costo de las instrucciones de máquina asociadas a la producción.
3. El lugar donde se deja el resultado de la operación representada por la producción. El resultado puede quedar en una celda de la memoria o en un registro de datos, por ejemplo.

Para la gramatica anterior, la tabla 2.6 indica cual es el código, costo y resultado para cada producción.

Producción	Código	Costo	Resultado
R --> + R constante	add R,constante	5	Registro R
R --> + R 1	inc R	3	Registro R_1
R --> + R_1 R_2	add R_1,R_2	4	Registro R_1
R --> variable	mov R,variable	7	Registro R

Tabla 2.6

Usando programación dinámica y los costos asociados a las producciones, se encuentra el árbol de sintaxis con el cual se puede generar el código de menor costo. En el ejemplo anterior, se elige el árbol de la izquierda (ver fig 2.1).

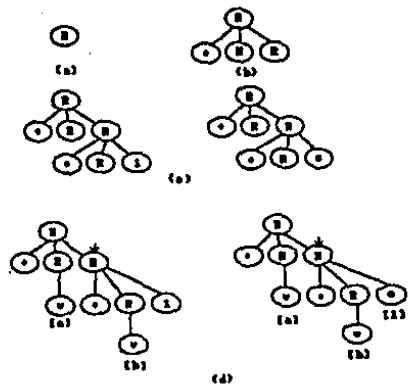

Figura 2.1

Una vez construidos los árboles, el algoritmo de Christopher-Hatcher-Kukuk emite el código asociado a las producciones utilizadas en la creación del árbol de sintaxis elegido.

Las ventajas del algoritmo de Christopher-Hatcher-Kukuk sobre el algoritmo de Graham-Glanville son las siguientes:

1. El método de análisis sintáctico elimina los ciclos, bloqueos sintácticos y bloqueos semánticos.
2. El empleo de programación dinámica para elegir el código que será emitido asegura que el generador siempre producirá código muy bueno.

Las limitaciones del método son las mismas que se mencionaron para el algoritmo de Graham-Glanville.

En [Chri84] se reporta la creación de tres generadores de código utilizando el algoritmo de Earley para realizar el análisis sintáctico. Los resultados muestran que aunque el código producido es muy bueno, los generadores de código construidos con este método son al menos doce veces más lentos que un generador de código diseñado con técnicas tradicionales. Esta es la razón principal por la cual este algoritmo de generación de código no podría aplicarse en la construcción de compiladores comerciales.

2.3 ALGORITMO DE SCHEUNEMANN [Hora85]

Si el programa en lenguaje intermedio se presenta en forma de árboles de expresión, cada uno de los nodos representa un valor que debe almacenarse, al menos temporalmente, en algún lugar para poder efectuar correctamente las operaciones indicadas en cada árbol. Al emitir código de máquina para las operaciones representadas en los árboles, cada valor asociado a un nodo debe mantenerse en algún medio de almacenamiento del computador (la memoria o un registro, por ejemplo) hasta que sea utilizado por una operación.

Según la tesis de Scheunemann, el principal problema que debe resolverse al generar código a partir de árboles, consiste en decidir dónde se dejará el resultado de la evaluación de cada nodo.

Considérese por ejemplo el árbol de la figura 2.2, que representa la expresión a = b + c * d. Si se desea generar código en lenguaje ensamblador del procesador 68000 de Motorola para este árbol, primero se debe determinar cuáles valores se tomarán de la memoria y cuáles se colocarán en registros. Hay 2^6 maneras diferentes de hacer la elección puesto que hay seis nodos que representan un valor (la raíz es el único nodo que no tiene asociado valor alguno) y por cada valor hay dos posibilidades : dejarlo en la memoria o en un registro. La figura muestra una de las posibles elecciones; cada arco tiene una etiqueta que indica si el nodo que está inmediatamente debajo se evalúa en un registro o en la memoria.

Una vez elegidas las etiquetas, el proceso de emisión de código se puede efectuar fácilmente, lo único que se debe hacer es buscar las instrucciones de máquina que tomen los operandos de la memoria o de un registro según se indica en el árbol y que dejen el resultado en el lugar señalado.

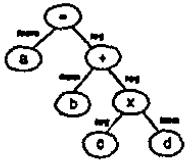

Figura 2.2

Por ejemplo, para generar código correspondiente al nodo que tiene el operador de multiplicación, se buscan las instrucciones de lenguaje ensamblador que realicen la multiplicación tomando el primer operando de un registro, el segundo operando de la memoria y que dejen el

resultado de la operación en un registro. El 68000 tiene una sola instrucción que puede ser utilizada en este caso, por consiguiente esa es la instrucción que se emite para este nodo.

A las etiquetas que indican dónde debe guardarse el valor representado por un nodo, se les denomina clases de almacenamiento. Conocidas la clases de almacenamiento para los nodos, el problema de selección de instrucciones se simplifica, pues en general habrá pocas formas de traducir a lenguaje de máquina cada una de las operaciones respetando las clases de almacenamiento elegidas.

Es necesario definir cuidadosamente las clases de almacenamiento a fin de poder utilizar en el código generado la mayor cantidad posible de recursos del procesador. Así , por ejemplo, si se desea emitir código para el procesador 68000 de Motorola no será conveniente usar sólo dos clases de almacenamiento: registro y memoria, puesto que este procesador tiene registros de diferentes clases y permite formar la dirección de una celda de la memoria de diversas maneras. Lo adecuado en este caso es crear una clase de almacenamiento distinta por cada tipo de registro y una por cada modo de direccionamiento de la memoria, logrando así que las clases de almacenamiento reflejen lo más fielmente los recursos de memoria disponibles en esa máquina.

Dos de los modos de direccionamiento de la memoria permitidos por el procesador 68000 de Motorola son el modo absoluto y el modo indirecto con desplazamiento. El segundo modo de direccionamiento forma una dirección completa de dos partes: una dirección base y un desplazamiento relativo a la dirección base; este modo es adecuado para tomar de la memoria el valor de una variable local, la dirección base corresponde a la dirección del inicio del registro de activación y el desplazamiento corresponde a la posición de la variable local dentro del registro de activación. En el modo absoluto se debe proporcionar la dirección completa de la celda, a fin de poder leer el valor guardado allí; este modo de direccionamiento es adecuado para las variables globales, puesto que ellas no se ponen dentro de un registro de activación. Si solamente se elige una clase de almacenamiento para los valores mantenidos en la memoria, en algunas ocasiones se emitira código redundante. Supóngase, por ejemplo, que la clase de almacenamiento elegida exige para poder leer un valor de la memoria, que se proporcione la dirección completa de la celda donde tal valor está almacenado, entonces cuando se desee tomar el valor de una variable local será necesario emitir una instrucción para sumar la dirección base con el desplazamiento; tal instrucción no es necesaria si se permiten dos clases de almacenamiento, una por cada uno de los modos de direccionamiento mencionados.

El primer paso para generar código de alta calidad a partir de árboles de expresión, consiste en hallar las clases de almacenamiento óptimas para los valores representados por los nodos, es decir, las clases de almacenamiento con las que se emitiría el código de menor costo. Si de alguna forma pueden encontrarse tales clases de almacenamiento, el paso siguiente

consiste en elegir las instrucciones de máquina que serán emitidas. Para facilitar la tarea de selección de instrucciones, se construye una tabla que indica, para cada operación y combinación de clases de almacenamiento de operandos y resultado, las instrucciones de máquina equivalentes y el costo asociado a las mismas. Esta tabla se denomina tabla de patrones *(template table)*.

Al utilizar muchas clases de almacenamiento se logra modelar adecuadamente el procesador, sin embargo, el problema que se presenta consiste en que la tabla de patrones es muy grande. Si se formaron k clases de almacenamiento diferentes, entonces por cada operación binaria deben crearse k^3 entradas en la tabla para tomar en cuenta todas las posibles combinaciones de clases de almacenamiento para operandos y resultado. Si existen m operaciones distintas, la tabla tendrá $k^3 * m$ entradas. Así, por ejemplo, si hay diez clases de almacenamiento y la máquina tiene 30 operaciones diferentes, la tabla tendría 30000 entradas; tomando en cuenta que en cada entrada debe indicarse el código de máquina y el costo asociado,se llegará a la conclusión de que la tabla ocuparía una cantidad de memoria que podría hacer impráctico este algoritmo.

Para mantener pequeño el tamaño de la tabla, Scheunemann muestra que para algunas combinaciones de clases de almacenamiento de operandos y resultado, las instrucciones de máquina equivalentes realizan tareas que tienen poco o nada que ver con la operación. Por ejemplo para el procesador 68000 de Motorola, el código que se emitirá para realizar la suma de dos valores almacenados en la memoria, dejando el resultado en otra celda de la memoria tendría la siguiente forma (para formar las direcciones de la memoria se utiliza direccionamiento absoluto):

```
move operando1 , registro1
add  operando2 , registro1
move registro1 , resultado
```

Solamente la segunda instrucción tiene relación con la operación de suma, mientras que la primera y tercera instrucciónes realizan conversiones entre clases de almacenamiento. Dada una operación, se define una combinación prima como aquella combinación de clases de almacenamiento de operandos y resultado para la cual el código asociado no realiza conversiones entre clases de almacenamiento. En la tabla de patrones sólo se guardarán las combinaciones primas.

Para considerar todas las combinaciones de clases de almacenamiento posibles de operandos y resultado, se crea una nueva tabla, llamada tabla de conversión, que indica el código de máquina que debe emitirse al cambiar de una clase de almacenamiento a otra.

Dado un árbol de expresión, para tratar de encontrar las clases de almacenamiento óptimas se asocia a cada nodo un vector de k elementos, siendo k el número de clases de almacenamiento diferentes que fueron definidas para la máquina. En la posición i del vector asociado a un nodo, se indicará el costo mínimo para evaluar el nodo, dejando el resultado de la evaluación en la clase de almacenamiento i. Para calcular los vectores de costos, se realiza un recorrido en posterior del árbol, llenando los vectores de cada nodo en la siguiente forma:

1. Si el nodo es una hoja, se determina la clase de almacenamiento que más conviene utilizar para formar la dirección de la celda de la memoria asignada al valor representado por el nodo (esta información puede tomarse de otra tabla que indique además, el costo de tomar un valor de la memoria usando tal clase de almacenamiento). Si un nodo representa una variable local, por ejemplo, en el procesador 68000 de Motorola el modo de direccionamiento que conviene utilizar es el indirecto con desplazamiento.

   Después, se consulta la tabla de conversión para encontrar el costo de colocar el valor en las restantes clases de almacenamiento.

2. Para los nodos internos, se buscan en la tabla de patrones todas las entradas correspondientes a la operación indicada por el nodo. Si la entrada de la tabla indica que el operador es binario, que el primer y segundo operandos deben ser de las clases de almacenamiento k_1 y k_2 respectivamente y que el resultado se deja en la clase k_3, entonces se llena la entrada del vector correspondiente a k_3 con el valor obtenido sumando el costo de tener el primer operando en la clase k_1 (este valor se toma del arreglo de costos asociado al nodo correspondiente al primer operando), más el costo de tener el segundo operando en la clase k_2 más el costo de la operación indicado en la tabla. Si en k_3 ya había un valor calculado previamente, se toma el menor de los dos.

   Usando la tabla de conversión se procede a calcular el costo de tener el resultado de la operación en las clases de almacenamiento que no fueron consideradas por la tabla de patrones.

En cada entrada del vector asociado a un nodo interno, además del costo, se indican las clases de almacenamiento elegidas para los hijos, de forma tal que después de llenar los vectores, haciendo un recorrido descendente del árbol, es posible determinar las clases de almacenamiento "óptimas" (con tales clases de almacenamiento se emite código óptimo solamente si no hay subexpresiones comunes y, además, el número de registros del procesador es suficiente para evaluar cada árbol sin tener que mover valores de registro a memoria o entre registros).

En [Hors85] se muestra que el algoritmo para encontrar las clases de almacenamiento "óptimas" es de orden O(n * m * k), donde n es el número de nodos del árbol, m es el número máximo de entradas de la tabla de patrones asociadas a una operación y k es el número de clases de almacenamiento.

En [Hors85] se menciona que fueron construidos generadores de código para tres máquinas: VAX-11, IBM/370 y para el procesador 8086 de Intel. Para la IBM/370, el generador de código construido con el algoritmo de Scheunemann se agregó a la parte frontal de un compilador de Pascal existente; el código emitido fué más pequeño que el generado por un compilador comercial disponible para esa máquina. La tabla de patrones para la IBM/370 tuvo menos de 300 entradas y ocupó en total cerca de 5Kb de memoria.

El algorimto de Scheunemann genera código muy bueno y la descripción de la máquina es bastante pequeña, sin embargo su principal limitación consiste en no incorporar asignación de registros. Después de hallar las clases de almacenamiento "óptimas" para los nodos de un árbol de expresión, se intenta generar código, pero si no hay suficientes registros para realizar las operaciones indicadas en el árbol, será necesario volver a calcular los vectores de costos forzando ahora a que algunos valores ya no sean mantenidos en registros, sino en la memoria. En algunos casos, el código emitido por el algoritmo podría mejorarse si se tomara en cuenta la asignación de registros al estar eligiendo las clases de almacenamiento "óptimas". El algoritmo tampoco toma en cuenta la presencia de subexpresiones comunes en el programa de entrada.

2.4 ALGORITMO DE AHO-JOHNSON [Aho76]

Es posible generar código óptimo para árboles de expresión utilizando programación dinámica; tal afirmación fue demostrada por Aho y Johnson, quienes diseñaron un algoritmo de generación de código que permite emitir código óptimo, para una clase amplia de máquinas, en tiempo lineal. El algoritmo de Aho-Johnson, a diferencia de los algoritmos de generación de código ya descritos en este capítulo, además de resolver los problemas de selección de instrucciones y determinación del orden de evaluación, incorpora asignación de registros.

El algoritmo de Aho-Johnson solamente puede generar código para máquinas con las siguientes características:

1. Todos los registros son idénticos.
2. Las instrucciones sólo puedan ser de los siguientes tipos :
   - Instrucciones de la forma r ← E, donde r es un registro y E es una operación cuyos operandos pueden estar en registros o en la memoria. Si E tiene operandos que se toman de registros, entonces alguno de esos registros debe ser igual a r. Las operaciones no deben tener efectos colaterales tales como modificar el valor de alguna celda de la memoria o alterar los bits del registro de banderas.
   - Instrucción de carga a registro r ← m, donde m representa una dirección de la memoria.
   - Instrucción de almacenamiento en memoria m ← r.
3. Todas las celdas de la memoria son idénticas y los modos de direccionamiento de la memoria no pueden utilizar registros para formar una dirección.

Dado un árbol de expresión, el algoritmo de Aho-Johnson hace tres recorridos sobre el mismo para generar código. Primero, para cada nodo se determina el costo mínimo de evaluarlo suponiendo que sólo hay k registros libres; este costo se calcula para todos los posibles valores de k. En el segundo paso se identifican los subárboles cuyo valor necesita almacenarse en la memoria y los que requieren todos los registros de la máquina para su evaluación. En este paso se crea una lista que indica el orden en que se generará código para los subárboles; los subárboles que necesitan todos los registros de la máquina se colocan al principio de la lista a fin de generar código primero para ellos (cuando ningún registro está ocupado todavía). En el último recorrido se toman los subárboles en el orden determinado por el paso anterior y se emiten las instrucciones de máquina correspondientes.

Se dice que una instrucción I cubre a un subárbol T, si es posible generar código para T de manera tal que la última instrucción sea I. Si la instrucción I cubre a un subárbol T, entonces, al construir el árbol para la instrucción I, la raíz y todos los nodos internos de tal árbol deben ser idénticos a los nodos correspondientes de T. Además, si la instrucción I tiene algún operando que se toma de un registro, en T debe existir el subárbol correspondiente que se evalúa en ese mismo registro y si la instrucción I tiene un operando que se toma de la memoria, entonces T debe tener un subárbol que se almacena en la misma celda de la memoria.

Cuando la instrucción I cubra a un árbol T, los subárboles de T que deban evaluarse en la memoria según la instrucción I se denotarán por el conjunto $\{T_1, T_2, \ldots, T_h\}$ y los que deban almacenarse en registros se indicarán con el conjunto $\{S_1, S_2, \ldots S_k\}$.

Para realizar el primer paso del algoritmo de Aho-Johnson, a cada nodo del árbol se le asocia un vector C de n+1 elementos, siendo n el número de registros de la máquina. En C_0 se indica el costo mínimo de evaluar el nodo en la memoria, mientras que en C_i, para i mayor a cero, se calcula el costo mínimo de evaluar el nodo usando solamente i registros. Debe mencionarse en este momento que en el algoritmo propuesto por Aho y Johnson en [Aho76] se toma como costo el número de instrucciones de máquina, por lo cual, el algoritmo generará programas que estén formados por el menor número posible de instrucciones.

El vector C se calcula haciendo un recorrido en orden posterior de cada árbol y procediendo para cada nodo N como indica el siguiente algoritmo.

Algoritmo costos (N)

1. Se inicializa cada elemento del vector C con un valor muy grande (infinito).
2. Si el nodo N es una hoja, entonces debe representar a una variable o constante que está almacenada en la memoria, por lo cual se hace C_0 igual a cero.
3. Si el nodo N es interno, se toman todas las instrucciones de la máquina que cubren al subárbol cuya raíz es N y para cada una de ellas se determinan los subárboles $T_1, T_2, \ldots T_h$ que deben mantenerse en la memoria y los subárboles $S_1, S_2, \ldots S_k$ que se deben evaluar en los registros.

   Los subárboles T_i se evalúan al principio para que tengan todos los registros de la máquina disponibles; como el valor de esos subárboles se guarda en la memoria, su evaluación no deja ningún registro bloqueado.

   Los subárboles S_i por el contrario dejan el resultado de su evaluación en registros, de manera tal que si hay m registros libres y los subárboles se evalúan en orden, después de generar código para S_1 quedarán solamente m-1 registros libres. Cuando se vaya a generar código para S_i, habrá sólo m-i+1 registros que pueden ser utilizados. Es importante

determinar en qué orden se generará codigo para los subárboles S_i, ya que los subárboles que se consideran primero pueden usar más registros que los subárboles evaluados al final; lo que conviene es emitir código primero para los subárboles más complejos.

Para llenar el vector de costos se toman todas las permutaciones p de los subárboles $S_1, S_2, \ldots S_k$ y para todas las entradas j del vector de costos comprendidas entre k y n (al menos se necesitan k registros para poder evaluar los subárboles), se calcula la entrada C_j con la fórmula :

$$C_j(N) = \text{mínimo}\ \{C_j(N),\ \sum_{i=1}^{k} C_{j-i+1}(S_{p(i)}) + \sum_{i=1}^{k} C_0(T_i) + 1\}$$

el costo de evaluar los subárboles T_i en la memoria, más el costo de evaluar los subárboles S_i en los registros, más el costo de realizar la operación (este costo es uno, puesto que se intenta minimizar el número de instrucciones emitidas), se compara con el valor anterior de C_j eligiéndose el menor de los dos valores.

En cada entrada del arreglo de costos se pueden almacenar la instrucción y la permutación utilizadas para obtener el valor del costo mínimo; esta información se usará más adelante.

Se repite el paso 3 para todas las instrucciones que cubren al subárbol cuya raiz es N.

4. Para dejar el resultado de un subárbol en la memoria, es posible evaluarlo primero en un registro y después almacenar el valor del registro en una celda de la memoria. Si el valor de un subárbol se guarda en la memoria, entonces conviene emitir código para tal subárbol al principio del programa, a fin de poder usar todos los registros de la máquina si es necesario. El valor de C_0 se elige en la siguiente forma :

$$C_0(N) = \text{mínimo}\ \{C_0(N), C_n(N) + 1\}$$

$C_n(N) + 1$ representa el costo de evaluar el subárbol utilizando todos los registros de la máquina dejando el resultado en un registro ($C_n(N)$) más el costo de almacenar el valor de ese registro en la memoria.

5. De la misma manera, para dejar el resultado de un subárbol en un registro, es posible evaluarlo primero en la memoria y después cargar el valor correspondiente en algún registro. Para tomar en cuenta esta posibilidad, se calculará el valor de C_j para toda j entre 1 y n con la expresión:

$$C_j(N) = \text{mínimo}\ \{C_j(N), C_0(N) + 1\}$$

$C_0(N) + 1$ representa el costo de poner el resulado del subárbol en la memoria ($C_0(N)$) más el costo de almacenarlo en un registro.

Termina_algoritmo

Una vez calculados los arreglos de costos, es necesario identificar los subárboles cuyo valor será almacenado en la memoria, puesto que se debe emitir primero código para ellos. El siguiente algoritmo construye una lista X con las raíces $x_1, x_2, \ldots x_s$ de los subárboles que deben evaluarse en la memoria; la lista está ordenada de manera tal que si en la evaluación del subárbol x_i se usa el valor de x_j, entonces x_j aparece antes que x_i. El algoritmo es recursivo y tiene dos parámetros, el primero es un apuntador al nodo que se está considerando y el segundo indica el número de registros disponibles para generar el código correspondiente a tal nodo. En la invocación inicial, el primer argumento apunta a la raíz de un árbol y el segundo argumento tiene el valor de n (este valor indica que todos los registros pueden usarse al emitir código para el árbol).

algoritmo marcar (nodo, n_reg)

1. Si no hay instrucción asociada a nodo en la posición n_reg del vector de costos, entonces termina la llamada.
2. Sea z ← E la instrucción asociada a C_{n_reg} y p la permutación óptima. Se determina el conjunto de subárboles $S_1, S_2, .. S_k$ que se deben evaluar en registros (los subárboles están ordenados según la permutación p) y el conjunto $T_1, T_2, \ldots T_h$ formado por los subárboles que deben almacenarse en la memoria.

   2.1. Si n_reg es cero, entonces para cada subárbol S_i invocar al algorimo marcar, pasando como primer argumento el apuntador a S_i y como segundo parámetro el número de registros libres para la evaluación de S_i.

   Si n_reg es cero, significa que el subárbol apuntado por nodo se debe evaluar en la memoria. Puesto que los subárboles cuyo resultado se almacena en la memoria serán colocados al principio, cuando todos los registros están libres, el segundo parámetro debe tener el valor n-i+1(*).

   2.2. Si n_reg no es cero, entonces para cada subárbol S_i se invoca al algoritmo marcar, pasando como primer argumento el apuntador a S_i y como segundo parámetro n_reg-i+1.

(*) El algoritmo reportado en [Aho76] tiene un error en esta parte, pues el valor que pasan en el segundo parámetro es n_reg-i+1, es decir 1-i (porque n_reg es cero), ocasionando que en algunas llamadas el segundo argumento tome valores negativos, lo cual no tiene sentido.

2.3 Para cada subárbol T_i, el algoritmo se invoca recursivamente, usando como primer parámetro un apuntador al subárbol y cero como segundo argumento.

3. Si n_reg es cero, entonces se debe agregar el subárbol cuya raíz es apuntada por nodo al final de la lista X. Obsérvese que un subárbol es agregado a la lista hasta después de haber analizado todos sus hijos, de manera tal que si alguno de los hijos también debe evaluarse en la memoria entonces aparecerá antes en la lista.

Termina_algoritmo

Lo único que falta, después de efectuar el algoritmo marcar, es emitir código. Se agrega la raíz del árbol al final de la lista X y se procede a generar código para cada uno de los subárboles de la lista.

Con el modelo de máquina definido, un subárbol sólo puede dejarse en la memoria si primero se genera código para dejar el valor del mismo en un registro y después se emite una instrucción de almacenamiento a la memoria. Se toman en orden los subárboles x_i de la lista X, para cada uno de ellos se genera código que deja el resultado en un registro; después, se emite una instrucción para almacenar el valor del registro en una celda de la memoria, y se marca ese registro como libre. Finalmente, el árbol se modifica reemplazando el subárbol x_i por un nodo que represente la dirección de la celda de la memoria donde el valor fue almacenado.

Procediendo en la manera anterior, se puede generar código para cada árbol respetando el orden de evaluación establecido en el segundo paso. Resta solamente presentar el algoritmo que emite código para un subárbol dejando el resulado en un registro. El algoritmo tiene dos parámetros, el primero es un apuntador a un nodo y el segundo indica el número de registros que pueden ser utilizados al emitir código. En la invocación inicial el primer parámetro apunta a la raíz del árbol para el que se va a emitir código y el segundo tiene el valor n, el número de registros que tiene el procesador.

Algoritmo código (nodo,n_reg)

1. Se consulta el vector de costos de nodo para determinar la instrucción y permutación utilizadas en la entrada C[n_reg]. Se identifica después a los subárboles S_1, S_2, ..., S_k que deben evaluarse en registros.
2. Se genera código para cada uno de los subárboles S_i, en el orden indicado por la permutación, haciendo para cada uno de ellos una invocación al algoritmo código, pasando como primer parámetro el apuntador al subárbol y como segundo n_reg-i+1 ,que es el número de registros disponibles para la evaluación de S_i. En cada llamada a código, se devuelve un registro r_i, donde está almacenado el resultado del subárbol.

3. Si k no es cero, entonces se elige alguno de los registros r_i para almacenar ahí el resultado de la evaluación de nodo. Si k es cero entonces se elige alguno de los registros libres para almacenar el resultado.
4. Se emite la instrucción asociada a la entrada C[n_reg], sustituyendo los valores de los registros r_i en la parte adecuada de la instrucción. Después,se liberan todos los registros, menos aquél donde se guardó el resultado de la operación y se devuelve ese registro como resultado de la llamada.

Termina_algoritmo

En [Aho76] se demuestra que el algoritmo anterior genera código óptimo para árboles de expresión en tiempo O(N*c), donde N es el número de nodos del árbol y c es una constante que depende del número de registros de la máquina, del número máximo de operandos de una operación y del número máximo de instrucciones de máquina que cubren a un subárbol.

CAPÍTULO 3

GENERACIÓN DE CÓDIGO A PARTIR DE *DAGS*

La mayoría de los algoritmos de generación de código diseñados hasta la fecha tiene dos limitantes principales; primero, no se considera la existencia de diversas clases de registros en el procesador y, segundo, no se toma en cuenta la presencia de subexpresiones comunes en el programa de entrada. En esta tesis se presenta un algoritmo eficiente de generación de código que toma en cuenta estos dos aspectos.

Para poder probar el algoritmo de generación de código era necesario tener un programa que pudiera traducir de algún lenguaje de alto nivel a código intermedio. Como no se contaba con ningún compilador para PC que diera esta clase de salida, fue necesario escribir las fases de la parte frontal para un compilador de un subconjunto del lenguaje de programación C.

El código intermedio que decidió utilizarse fue código de tres direcciones; una instrucción de tres direcciones tiene la forma general

$$a \leftarrow b \ op \ c$$

donde b y c son operandos, op es la operación y a el resultado; en una instrucción de tres direcciones puede faltar alguno de estos elementos, sin embargo en cada instrucción a lo más puede haber una operación.

Para representar las instrucciones de tres direcciones se utilizaron cuádruples; un cuádruple es un registro con 4 campos en los cuales se almacena cada parte de una instrucción.

La parte frontal, entonces, lo que produce como salida es un conjunto de cuádruples. Este conjunto es dividido en bloques básicos; un bloque básico es un conjuntos de instrucciones sin saltos, salvo al final (la última instrucción puede ser un salto) y a las cuales se puede llegar solamente pasando antes por la primera instrucción del bloque básico (no hay saltos cuyos destinos sean instrucciones que están dentro del bloque, sólo la primera instrucción puede ser el destino de un salto). Cada bloque básico es transformado en *dags*, lo que permite descubrir las subexpresiones comunes presentes en el bloque básico (cada nodo que tenga más de un padre representa una subexpresión común). Después, los *dags* son divididos en árboles y se genera código a partir de ellos.

El sistema desarrollado consta de tres módulos, cada módulo se encarga de realizar alguna fase del proceso de traducción de lenguaje fuente a lenguaje de máquina como se muestra en la figura 3.1

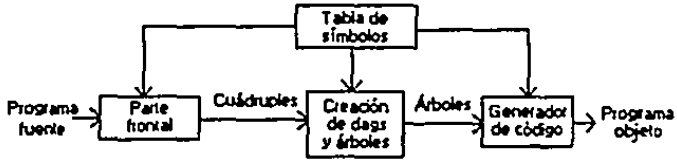

Figura 3.1

La parte frontal, el generador de código y el módulo encargado de crear los *dags* y los árboles utilizan la información contenida en la tabla de símbolos para poder realizar su trabajo. La parte frontal se encarga de guardar en la tabla de símbolos los identificadores que forman parte del programa de entrada y las variables temporales creadas durante la formación de los cuádruples. Las otras partes del sistema: el generador de código y el módulo encargado de crear los *dags* y los árboles, utilizan la información almacenada por la parte frontal en la tabla de símbolos y además agregan a la misma ciertos atributos adicionales de los identificadores. Por ejemplo, al crear los *dags*, es conveniente asociar a cada identificador un apuntador al nodo que representa su valor y al generar código, resulta útil guardar en la tabla de símbolos el modo de direccionamiento usado para tomar de la memoria el valor de cada identificador.

El algoritmo de generación de código diseñado en esta tesis tiene dos partes. Primero, con los cuádruples se forman *dags* para detectar las subexpresiones comunes presentes en el programa de entrada; los *dags* son divididos en árboles que se ordenan intentando que al generar código, se obtenga el más eficiente posible. Segundo, se genera código a partir de los árboles tomando en cuenta las subexpresiones comunes y la presencia de registros asimétricos en el procesador. La primera parte del algoritmo se presenta en las secciones 3.2 y 3.3, la segunda parte se describe en el capítulo 4.

3.1 Parte frontal

La parte frontal, que traduce un programa escrito en un subconjunto de C a código de tres direcciones, se escribió en C y se compiló en un PC/AT utilizando el compilador TurboC de Borland, versión 2.0. Para construir el analizador léxico se utilizó el generador de analizadores léxicos llamado LEX (Lesk75) y para construir el analizador sintáctico se utilizó el generador de

analizadores sintácticos llamado YACC (John75). El analizador sintáctico no incluye recuperación de errores, si recibe como entrada un programa erróneo, señala el primer error y después termina su ejecución.

La parte frontal realiza las siguientes optimizaciones locales *(peephole optimization)* sobre el conjunto de cuádruples:

1. Si hay un salto a la siguiente instrucción, tal salto es eliminado.

```
salta_a E1
E1:
```

2. Si el destino de un salto es una instrucción de salto

```
salta_a  E1
   ...
E1: salta_a  E2
```

el destino del primer salto es cambiado al destino del segundo

```
salta_a  E2
  ...
E1: salta_a E2
```

3. Un salto condicional seguido de un salto incondicional es cambiado por un solo salto condicional. Por ejemplo,el siguiente conjunto de instrucciones:

```
salta_si_mayor_que_a  E1
salta_a E2
E1:
```

es transformado en la única instrucción:

```
salta_si_menor_o_igual_a  E2
E1:
```

4. Si los operandos de una instrucción artimética o lógica son constantes, se reemplaza la operación por el resultado de la misma *(constant folding)*.
5. Se elimina código muerto (código que no puede ser alcanzado). Por ejemplo, en el siguiente fragmento de programa:

```
salta_a E1
a = b + 1
E1:
```

la instrucción a = b + 1 nunca se efectuará, por lo tanto es eliminada.

Cada vez que la parte frontal encuentra un identificador, este se almacena en la tabla junto con los siguientes atributos:

1. El tipo, que indica si el identificador representa una variable simple, un arreglo, una estructura, o el nombre de una función.
2. Un bit que señala si el identificador es local o global.
3. Un bit que establece si el identificador es un temporal o una variable del programa fuente.
4. Un bit que indica si el identificador representa un campo de una estructura.
5. Si el identificador representa un arreglo, entonces, un atributo indica el número de elementos que tiene.

Esta información es utilizada por la parte frontal para detectar los errores semánticos del programa fuente. La parte frontal, además, se encarga de asignar la posición que ocuparán las variables locales dentro de los registros de activación; para calcular este valor, es necesario que la parte frontal conozca las siguientes características de la máquina para la que se va a generar código:

1. El número de *bytes* que ocupa cada uno de los tipos de datos definidos en el lenguaje fuente. Esta información es proporcionada a la parte frontal utilizando un arreglo llamado Tamaño_tipo; la entrada i-ésima del arreglo indica el tamaño en *bytes* que ocupa el i-ésimo tipo de datos del lenguaje fuente.

2. Las restricciones impuestas sobre las direcciones de la memoria que puede ocupar un valor de cada tipo. Por ejemplo, el procesador 68000 de Motorola exige que un valor de dos bytes se guarde en una dirección par.

   Un arreglo llamado Restricciones_sobre_dirección indica, para cada tipo de datos del lenguaje fuente, las restricciones que impone el procesador sobre las direcciones que pueden ser usadas para almacenar un valor de cada tipo.

3.2 Creación de *dags* a partir de cuádruples

El programa en representación intermedia, está formado por cuádruples; cada cuádruple representa una instrucción de tres direcciones. Una instrucción de tres direcciones tiene la forma r = a op b, donde r es una variable, a y b son variables o valores constantes y op representa una operación. Existen instrucciones de tres direcciones en las cuales alguno de esos elementos puede faltar; por ejemplo, si se omite el primer operando se obtiene la instrucción r = op b, en la cual op representa un operador unario.

Las instrucciones de tres direcciones consideradas en esta tesis tienen alguna de las siguientes formas:

1. Operación binaria con resultado, r = a op b.
2. Operación unaria con resultado, r = op b.
3. Operación sin operandos ni resultado, op.
4. Asignación, r = a.
5. Operación unaria sin resultado, op a.

Las restantes combinaciones de operandos, resultado y operador, se pueden poner en la forma especificada por alguna de las cinco instrucciones anteriores.

Para traduducir un programa escrito en C a lenguaje intermedio, es necesario primero definir los operadores de representación intermedia que serán utilizados. Las clases de operadores manejadas por la parte frontal para construir el conjunto de cuádruples son las siguientes:

1. Operadores aritméticos.
2. Operadores que denotan saltos condicionales e incondicionales.
3. Operadores para invocación a función , regreso de una llamada y paso de parámetros.
4. Operadores para referenciar un elemento de variable estructurada. La gramática permite dos clases de variables estructuradas: arreglos y estructuras. Este tipo de operadores permite elegir algún elemento de un arreglo o campo de una estructura.
5. Operadores para definición de variables. Un operador de esta clase indica que una variable fue definida en el programa fuente. Los cuádruples con este tipo de operadores permiten reservar espacio en la memoria para las variables.
6. Operadores que señalan una parte del programa fuente. En ciertas partes del programa es necesario emitir directivas al ensamblador para que éste pueda producir el programa ejecutable. Por ejemplo, algunos ensambladores requieren que al principio del programa se coloque una directiva indicando cual es la dirección de la primera instrucción del programa

objeto. Se utiliza un cuádruple para marcar el comienzo del programa y de esta forma poder emitir tal directiva en el lugar correcto del archivo objeto.

3.2.1 Formación de *dags*

Un grafo dirigido sin ciclos *(dag)* permite identificar las subexpresiones comunes presentes en el conjunto de cuádruples, de manera tal que si una instrucción aparece varias veces en el programa de entrada se genere código una sola vez para evaluarla. De la misma forma, si el *dag* muestra que una variable es utilizada en varias instrucciones diferentes, al generar código se buscará mantener el valor de esa variable en un registro, intentanto así que el programa objeto sea lo más eficiente posible.

Más adelante se presenta el algoritmo que permite crear un *dag* a partir de un conjunto de cuádruples; en el *dag* que produce este algoritmo, un nodo puede representar una variable, un operador o un valor constante. Cada nodo que represente una variable tendrá asociado un campo que apunte a la entrada de la tabla de símbolos que guarde la información de tal variable; en un nodo correspondiente a una constante se almacenará el valor de la constante (sólo se manejan constantes de tipo entero) y en los nodos que representen una operación, habrá un campo en el nodo que indique cuál es el operador correspondiente.

Todos los identificadores que forman parte de los cuádruples están guardados en la tabla de símbolos. Cada entrada de la tabla de símbolos tiene un apuntador al nodo del *dag* que represente el valor del identificador. Al estar formando el *dag*, los valores constantes que aparezcan como operandos de algún cuádruple se guardarán en una tabla de constantes junto con un apuntador al nodo que represente el valor de cada constante. La tabla de símbolos y la tabla de constantes ayudan a detectar subexpresiones comunes.

Al crear el *dag* para la instrucción r = a op b, se forman tres nodos, uno para cada operando y otro para la operación; a este último nodo se le asocia el identificador r como se muestra en la figura 3.2, indicando que el valor de r está representado por tal nodo.

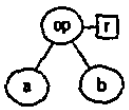

Figura 3.2

Un nodo que tiene varios identificadores asociados representa el valor de una expresión común que es utilizada tantas veces como identificadores tiene asociados. Al generar código para

un nodo de ese tipo se deben emitir instrucciones para guardar el valor representado por el nodo en las direcciones de memoria asignadas a los identificadores. Si un nodo del *dag* no tiene ningún identificador asociado, entonces seguramente su valor es utilizado solamente una vez para evaluar alguna expresión más compleja y no necesita salvarse en la memoria.

Dado un conjunto de cuádruples, el *dag* correspondiente se forma tomando cada cuádruple y creando, si es necesario, los nodos para los operandos y para el operador. Si ya existen en el *dag* nodos para los operandos o para la operación, se evita la creación de nuevos nodos. Por ejemplo, para el fragmento de programa en C:

```
j = i
a[ i ] = b + 1
a[ j ] = 2
y = a[ i ] + 5
```

el conjunto equivalente de cuádruples es el siguiente:

```
j = i
t1 = b + 1
t2 = a [ ] i
t2 = t1
t3 = a [ ] j
t3 = 2
t4 = a [ ] i
y = t4 + 5
```

Se toma el primer cuádruple, que tiene solamente un operando, y se determina si en el *dag* ya existe un nodo que represente el valor de ese operando; para decidir esto, se busca en la tabla de símbolos la entrada donde está guardada la variable i, si el apuntador asociado a tal entrada es nulo, entonces no hay un nodo creado para i. Como apenas se está tomando el primer cuádruple no hay ningún nodo en el *dag* todavía, por lo que es necesario crear un nuevo nodo que se etiqueta con i y se le asocia el identificador j como se muestra en la figura 3.3a. En la tabla de símbolos se indica que los valores de las variables j e i están representados por este nodo.

Para el segundo cuádruple es necesario formar tres nodos: dos para los operandos y uno para la operación. Al nodo correspondiente a la suma se le asocia el identificador t1. El *dag* formado con éste y los dos siguientes cuádruples se muestra en la figura 3.3b; las variables temporales fueron omitidas.

Al considerar los dos siguientes cuádruples se necesita crear un nuevo nodo al que se le asocia a[j] (ver figura 3.3c). Para el penúltimo cuádruple, dado que ya existe en el *dag* el nodo que representa el valor de a[i], solamente se debe agregar a tal nodo el identificador t4 como se muestra en 3.3d. Finalmente, para el último cuádruple, debido a que el nodo para t4 ya existe, sólo se debe crear un nodo para el valor constante 5 y otro para la operación de suma como se indica en la figura 3.3e.

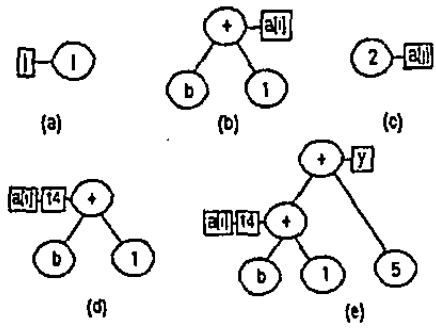

Figura 3.3

Obsérvese que según el *dag* de la figura 3.3e, a la variable y se le debe asignar el valor de b + 1 + 5; sin embargo, de acuerdo al programa fuente el valor correcto que debe tomar y es 2 + 3 (puesto que i es igual a j, en la tercera instrucción del programa fuente a[i] también toma el valor de 2).

Al estar formando el *dag* es posible que el valor de una variable cambie como resultado de alguna operación, como ocurre con a[i] en el ejemplo anterior y por lo tanto el nodo que representa el valor de esa variable no puede utilizarse como subexpresión común.

Se dice que una instrucción mata el valor representado por un nodo, si al ejecutar esa instrucción cambia el valor asociado al nodo. Un nodo cuyo valor ha sido muerto, deja de ser expresión común y no puede volver a utilizarse.

En el ejemplo anterior, la instrucción a[j] = 2 mata el valor del nodo asociado con a[i], por lo cual al tomar el penúltimo cuádruple, debe formarse un nuevo *dag* que represente el valor de a[i]. El *dag* correcto para el conjunto anterior de cuádruples se muestra en la figura 3.4.

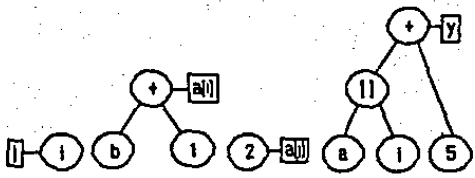

Figura 3.4

Al emitir código se pueden considerar los nodos en orden diferente al cual fueron creados. Supóngase, por ejemplo, que antes de ejecutar las operaciones indicadas en el *dag* de la figura 3.4 los valores de las variables son los siguientes: i = 1, b = 3, a[j] = 1. Si se evalúan los árboles de izquierda a derecha como aparecen en la figura, los valores que se obtendrán para las variables serán: a[i] = 2, a[j] = 2, y = 7. Podrían también, ejecutarse las operaciones del segundo árbol antes que las indicadas en el primer árbol y los valores finales de las variables serían los mismos que en el caso anterior; sin embargo no podría evaluarse el tercer árbol antes que el segundo, pues los valores que se obtendrían son: a[i] = 4, a[j] = 4, y = 9.

Puesto que la calidad del código producido depende del orden elegido para evaluar los nodos del *dag* , se intentará elegir el orden que genere el código de menor costo posible. Al generar código, sin embargo, debe tomarse en cuenta que el orden relativo de evaluación de algunos nodos no puede ser alterado o de lo contrario no se emitirá código correcto.

Si una instrucción I mata un valor v, entonces, al generar código los nodos para v y para I deben tomarse en el mismo orden relativo al cual aparecen en el programa fuente. Si la instrucción I aparece antes que un uso del valor v, entonces se debe emitir código para el nodo correspondiente a I antes de emitir código para la instrucción donde se usa v y viceversa, si el valor v es usado antes de aparecer I en el programa de entrada, entonces debe generase código para la instrucción que emplea el valor v antes de producir el código para I.

Estas restricciones de orden de evaluación son señaladas en el *dag* colocando un arco del nodo que debe evaluarse después al nodo que debe evaluarse antes. En la figura 3.4, por ejemplo, se debe generar código para el nodo que representa el valor de a[j] después de emitir código para el nodo correspondiente a la suma de b con 1; los arcos que señalan esta y las restantes restricciones de orden de evaluación para el *dag* anterior, se muestran en la figura 3.5.

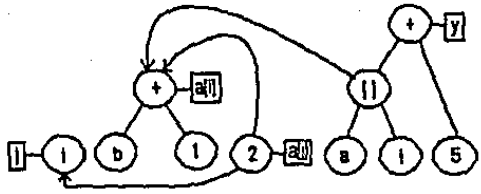

Figura 3.5

Supóngase, por ejemplo, que se desea formar el *dag* correspondiente al siguiente fragmento de programa en C:

```
b = 1 + a;
a = 2 + c;
c = i + 4;
d = b + c;
```

El subárbol para la primera instrucción se muestra en la figura 3.6a. La segunda instrucción altera el valor de la variable a, por lo cual se debe indicar en el *dag* que el nodo que representa el valor de tal variable es muerto por esta instrucción (ver figura 3.6b). Al asignar a c el valor de i + 4, se mata el nodo que representaba el valor de la variable c, formándose el *dag* mostrado en la figura 3.6c.

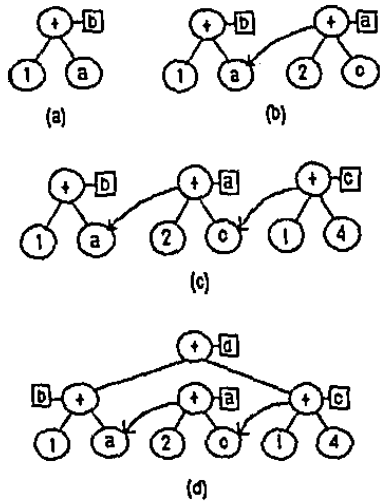

Figura 3.6

Finalmente, se considera la última instrucción; como ya existe un nodo que representa el valor de b y otro que corresponde al valor de c, sólo se crea el nodo para sumar estos valores y así resulta el *dag* de la figura 3.6d.

El algoritmo de generación de código diseñado en esta tesis requiere que el *dag* sea dividido en árboles. El *dag* de la figura 3.6d está formado por los dos árboles mostrados en la figura 3.7. Obsérvese en la figura, que no es posible establecer un orden de evaluación para estos árboles, pues los arcos de restricción de orden indican que el primer árbol debe ser evaluado antes que el segundo y que el segundo árbol debe evaluarse antes que el primero. Para evitar estas situaciones, no se permitirá que ningún ancestro de un nodo muerto pueda ser utilizado como subexpresión común.

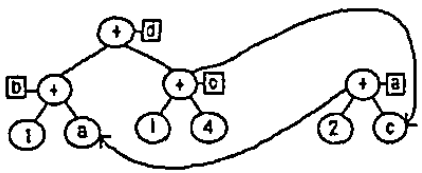

Figura 3.7.

En el ejemplo anterior, al formar el subárbol para la última instrucción del programa, no se tomará el nodo existente para la variable b, pues su hijo derecho representa un valor muerto. En este caso se debe crear un nuevo nodo para b y formar el *dag* de la figura 3.8.

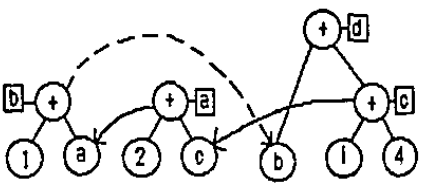

Figura 3.8.

El nodo que representa el valor de la variable b no se considera como muerto, pues ninguna instrucción altera ese valor. Al generar código para el primer árbol, podría dejarse el valor de b en un registro y al emitir código para el tercer árbol el valor de b sería tomado de ese

registro. La variable b es una subexpresión común, aunque la raíz del primer árbol no pueda usarse para formar alguna otra parte del *dag*.

Para indicar en el *dag* que b es una subexpresión común, los nodos que representan los usos de esa variable se ligan mediante un arco (el arco punteado mostrado en la figura 3.8). A este tipo de arcos que señalan los usos de una subexpresión común se les denominará ligas de siguiente uso.

Cada nodo del *dag* tiene los siguientes campos:

1. Un campo que indica qué identificador, constante u operador representa el nodo.
2. Un bit que señala si el valor del nodo ha sido muerto por alguna instrucción.
3. Un bit que determina si alguno de los descendientes del nodo ha sido muerto.
4. Una lista de identificadores cuyo valor está representado por el nodo.
5. Los apuntadores a los hijos del nodo.
6. Los apuntadores de siguiente uso.
7. Un apuntador al siguiente nodo creado y uno al nodo anterior. Además de formar parte de un *dag*, los nodos se encuentran dentro de una lista doblemente ligada, colocándose los nodos en esta lista en el orden en que fueron creados.

El siguiente algoritmo se encarga de formar el *dag* correspondiente a los cuádruples de un bloque básico.

Algoritmo crear_dag

Se toma cada cuádruple del bloque básico y con cada uno de ellos se procede en la siguiente forma:

1. Para cada operando se determina si existe o no en el *dag* un nodo que represente su valor; para esto se consulta la tabla de símbolos o la tabla de constantes, según el operando sea una variable o una constante.

   En caso de no existir el nodo que representa el valor de un operando, o bien si el nodo existe pero está muerto o alguno de sus descendientes ha sido muerto, será necesario crear un nuevo nodo. Al crear un nuevo nodo, se llenan sus campos con la información requerida y se ajusta la tabla de símbolos o la de constantes para indicar que el nuevo nodo representa el valor de una variable o de una constante. Si es necesario, se colocan las ligas de siguiente uso.
2. Si ya estaban creados los nodos que representan los valores de los operandos, entonces se busca en el *dag* un nodo que represente el operador del cuádruple y cuyos hijos sean

los nodos encontrados en el paso 1. Si tal nodo no existe, se crea uno nuevo colocándole como hijos los nodos correspondientes a los operandos.

En este paso no se hace nada si el cuadruple no tiene operador.

3. Sea r la variable usada como resultado del cuádruple. En la tabla de símbolos se busca el nodo que represente el valor de r; si este nodo existe se elimina r de su lista de identificadores y después se agrega r a la lista de identificadores del nodo que representa el valor del resultado del cuádruple (el nodo hallado en el paso 3, si es que el cuádruple tiene operador, o el nodo hallado en el paso 1, en otro caso). Se ajusta la tabla de símbolos para indicar cuál es el nuevo nodo que representa el valor de r.
4. Si el cuádruple que acaba de considerarse mata el valor representado por algún nodo, entonces deben colocarse las restricciones de orden de evaluación correspondientes y marcar como muertos los nodos cuyo valor es alterado por la instrucción indicada en el cuádruple. Las instrucciones que matan el valor de algún nodo del *dag* son las siguientes:

   i) Una llamada a función mata todos los nodos que representen el valor de una variable global; esto es porque dentro del cuerpo de la función puede alterarse el valor de cualquier variable global.

   ii) Una asignación a un elemento de un arreglo mata todos los nodos que representen el valor de alguno de los elementos del mismo arreglo.

   iii) Una asignación a una variable simple mata a todos los nodos que representen el valor de esa misma variable.

   iv) Una asignación a un elemento de una estructura mata a todos los nodos que representen el valor de ese mismo elemento de la estructura.

Para determinar qué nodos se deben marcar como muertos, se recorre hacia atrás la lista donde aparecen los nodos por orden de creación. Si el último cuádruple fue una invocación a función, entonces cada vez que se encuentre un nodo representando el valor de una variable global, se marca como muerto y se coloca una liga de restricción de orden de evaluación entre el nodo que corresponde a la llamada a función y el que representa el valor de la variable global. Si al estar recorriendo la lista de nodos se encuentra un nodo que represente una llamada a función, se coloca una liga de restricción de orden de evaluación entre los nodos que representan las llamadas a función y no se sigue recorriendo la lista hacia atrás, puesto que si hay algún otro nodo que corresponda al valor de una variable global, tal nodo ya debe estar muerto.

Cuando el cuádruple que acaba de considerarse representa la asignación de un valor a una variable simple o a un elemento de un arreglo, se recorre la lista de nodos hacia

atrás hasta llegar al principio de la lista o bien hasta encontrar otra asignación a la misma variable o al mismo arreglo, matando los nodos y colocando las ligas de restricción de orden de evaluación que sean necesarias.

Cada vez que un nodo es muerto, todos sus ancestros son marcados para que no sean considerados como posibles subexpresiones comunes.

5. Es posible que deban colocarse algunas ligas de restricción de orden de evaluación si alguno de los nodos creados para los operandos o para el resultado representa el valor de una variable simple, de un elemento de una estructura o de un elemento de un arreglo:

   I) Si el nodo creado para un operando representa el valor de una variable simple o de un elemento de una estructura, se agrega una liga de restricción de orden entre ese nodo y cualquier otro nodo creado antes que represente una asignación a la misma variable o elemento de estructura.

   II) Si alguno de los nodos de los operandos representa el valor de una variable global, se colocan ligas de restricción entre ese nodo y los nodos que representen llamadas a funciones.

   III) Si alguno de los nodos representa un elemento de un arreglo, se colocan ligas de restricción hacia los nodos que representen asignaciones a ese mismo arreglo.

Termina algoritmo.

3.3 División del *dag* en árboles

El algoritmo de generación de código diseñado en esta tesis requiere que la entrada se dé como un conjunto de árboles de expresión, por lo tanto, es necesario dividir en árboles el conjunto de *dags* formados con los bloques básicos del programa fuente.

Para dividir un *dag* en árboles, se crean n-1 nodos por cada nodo que tenga n padres. Un nodo con n padres representa una subexpresión común que es utilizada n veces; cada uno de los nuevos nodos representará el valor de la subexpresión común y será colocado como hijo de alguno de los n padres.

En el *dag* mostrado en la figura 3.8a, por ejemplo, el nodo que representa el valor de la variable a tiene tres padres. Al dividir el *dag* en árboles será necesario crear dos nuevos nodos que representen el valor de a y se colocará cada uno de estos nodos como hijo de alguno de los tres padres como se muestra en la figura 3.8b. Los arcos de líneas discontinuas son ligas de siguiente uso e indican que los nodos señalados representan el mismo valor.

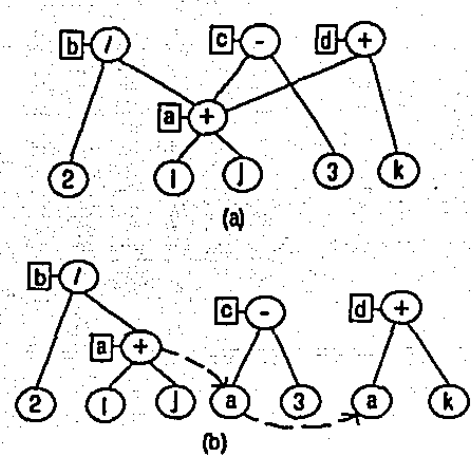

Figura 3.8

Cada uno de los nodos creados al estar dividiendo un *dag* en árboles, representa a la variable usada para almacenar el valor de una subexpresión común. Si un nodo con más de un padre tiene una lista con varios identificadores asociados, alguno de ellos será asignado a los nodos creados durante la formación de los árboles; se preferirá elegir un identificador que represente el valor de una variable del programa fuente a un identificador temporal creado durante la formación de los cuádruples.

Por otra parte, si un nodo con varios padres no tiene ningún identificador asociado, entonces será necesario crear una variable temporal que se asignará a los nuevos nodos.

Al crear nuevos nodos es posible que deban agregarse algunas ligas de restricción de orden de evaluación a fin de emitir código en el orden correcto para ellos. Supóngase, por ejemplo, que en algún *dag* un nodo N tiene varios padres y al dividir el *dag* en árboles se crean los nodos $n_1, n_2, ..., n_k$; sea r el identificador que representa el valor de N y que se asigna a los nuevos nodos. Para decidir si es necesario agregar nuevas ligas de restricción de orden, se toman todas las ligas de restricción de orden que comienzan o que terminan en N y para cada una de ellas se procede en la siguiente forma:

1. Si hay una restricción del nodo N a un nodo M y además se presenta alguna de las siguientes condiciones:

   i. r es un identificador global y M representa una llamada a función,

   ii. r representa un elmento de un arreglo y M una asignación a ese mismo arreglo, o

ii. r representa un elmento de un arreglo y M una asignación a ese mismo arreglo, o

iii. r es una variable simple y M representa una asignación a esa misma variable,

entonces se agrega una liga de restricción de orden de cada nodo n_i al nodo M.

2. Si hay una restricción de orden de un nodo M al nodo N y además se cumple alguna de las condiciones anteriores, entonces se agrega una liga de restricción de orden de M a cada nodo n_i.

3.4 Orden de evaluación

El algoritmo encargado de generar código para un árbol debe respetar las ligas de restricción de orden de evaluación a fin de garantizar que el código producido sea correcto. Para crear el programa objeto el algoritmo de generación de código hace un recorrido en orden posterior de cada uno de los árboles, emitiendo para cada nodo visitado las instrucciones en lenguaje de máquina; al hacer el recorrido de un árbol, el algoritmo necesita saber en que orden se deben visitar los hijos de cada uno de los nodos para no violar las restricciones de orden señaladas por las ligas.

Cada nodo del *dag* tiene un campo donde se indica si los hijos pueden evaluarse en cualquier orden o bien si primero debe considerarse el hijo izquierdo o el hijo derecho. Para llenar este campo, se recorre cada uno de los árboles hasta encontrar un nodo N que tenga una o más ligas de restricción de orden de evaluación; para cada liga se determina el nodo M que debe ser evaluado antes que N y se busca después el primer ancestro P común a N y M. Si tal ancestro existe, entonces uno de los hijos de P debe evaluarse antes que el otro: si N aparece en la rama izquierda del subárbol cuya raíz es P, entonces el hijo derecho de P debe evaluarse antes que el hijo izquierdo; en cambio, si N aparece en la rama derecha del subárbol, entonces debe evaluarse primero el hijo izquierdo de P.

Que N y M no tengan ancestro común, significa que esos nodos pertenecen a diferentes árboles; en este caso el árbol al que pertenece N debe evaluarse después del árbol donde está colocado el nodo M.

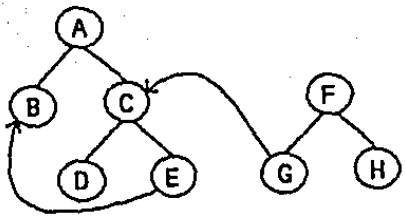

Figura 3.10

Para la pareja de árboles mostrado en la figura 3.10, hay dos nodos con restricciones de orden de evaluación: el nodo E y el nodo G. Se toma primero el nodo E que tiene una liga de restricción hacia el nodo B y se busca el ancestro común para B y E. Con el fin de facilitar la búsqueda del ancestro común, cada nodo tiene un campo que indica su nivel y un apuntador al padre. El nodo B está en el nivel uno, mientras que el nodo E se encuentra colocado en el nivel dos; por lo tanto, si B y E tienen un ancestro común, éste debe encontrarse en el nivel uno (si B es ancestro de E) o en el nivel cero.

El algoritmo que determina el ancestro común de dos nodos utiliza dos apuntadores, uno se coloca apuntando al nodo B y el otro apunta inicialmente al nodo E. El apuntador colocado en el nodo E se mueve un nivel hacia arriba siguiendo la liga que apunta al padre, este movimiento se realiza debido a que el nodo E está colocado un nivel más abajo que el nodo B; después de este ajuste los dos apuntadores están colocados en nodos situados al mismo nivel. Cada apuntador se mueve hacia arriba un nivel cada vez hasta que ambos apunten al mismo nodo (en cuyo caso este nodo es el ancestro común) o bien hasta llegar a las raíces de dos árboles distintos (si se presenta esta situación, entonces, los nodos pertenecen a diferentes árboles).

Utilizando el procedimiento anterior, se determina que los nodos B y E tienen como ancestro común al nodo A; puesto que E está colocado en la rama derecha, entonces primero debe evaluarse el hijo izquierdo
de A.

Hay otra liga de restricción, entre los nodos G y C; puesto que tales nodos pertenecen a diferentes árboles, siguiendo el poceso descrito anteriormente se determina que el árbol cuya raíz es F debe ser evaluado después del árbol con raíz A.

Si un nodo M mata a un nodo N y además el padre de N es ancestro del nodo M, al generar código, el nodo N debe ser evaluado en registro. Para comprender por qué, considérese el árbol de la figura 3.11; la liga de restricción de orden indica que el hijo izquierdo de la raíz debe ser evaluado antes que el hijo derecho. Si, por ejemplo, el valor inicial de b es 5, el valor que debe

obtenerse para la variable a después de realizar la suma indicada en el árbol es 7. Obsérvese que al efectuar la operación indicada en la rama derecha del árbol, en la celda de la memoria reservada para b se almacena el valor 2; si se decide dejar el primer operando en la memoria, cuando se genere código para realizar la suma, ambos operandos valdrían 2 y el resultado de la operación indicaría que, en este caso, el valor de la variable a es 4.

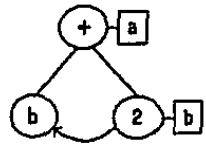

Figura 3.11

Para generar código correcto, el primer operando debe colocarse en un registro, de manera tal que su valor no sea alterado al realizar la operación indicada por el hijo derecho de la raíz.

Cada nodo tiene un campo que indica si el valor representado por el nodo debe ser evaluado en un registro o bien puede ser mantenido en un registro o en la memoria al estar generando código.

3.4.1 Orden de evaluación de los árboles

Al generar código para un programa con subexpresiones comunes, se intentará mantener los valores que son utilizados varias veces en algún lugar donde puedan ser tomados rápidamente por el procesador, a fin de que el código producido sea lo más eficiente posible. Puesto que el medio de almacenamiento con el menor tiempo de acceso son los registros, el valor de una subexpresión común se colocará en alguno de los registros del procesador y se intentará no modificar el contenido de ese registro hasta que termine de generarse código para las instrucciones que utilicen el valor de la subexpresión común.

Para la mayor parte de los procesadores, en casi todas las instrucciones de máquina alguno de los operandos o el resultado se deja en un registro. Si el procesador tiene pocos registros es muy probable que en algún momento no exista un registro libre para efectuar la siguiente operación y, así, alguno de los registros ocupados debe ser liberado almacenando temporalmente su valor en la memoria. Estas instrucciones para guardar el valor de un registro en

la memoria no serían necesarias si el procesador tuviera un número ilimitado de registros. Parece evidente, entonces, que el costo del código es una función no creciente con el número de registros del procesador.

Si al ejecutar un programa en lenguaje de máquina, el valor de una subexpresión común se mantiene en un registro, ese registro se bloquea durante cierto tiempo, pues lo que se desea es que su contenido no se altere sino hasta después de utilizar su valor en todas las instrucciones que lo necesitan. Al bloquear un registro, hay menos registros disponibles para ejecutar las instrucciones del programa objeto y, por consiguiente, es posible que se deba respaldar en la memoria el valor de alguno de ellos para liberarlo. A fin de reducir al mínimo la probabilidad de que en algún momento no exista un registro disponible se intentará mantener próximos los usos de una subexpresión común, de forma tal que el registro que almacene el valor de la subexpresión común sea bloqueado durante el menor tiempo posible.

El algoritmo de generación de código diseñado en esta tesis decide el orden de evaluación para los nodos de un mismo árbol, pero no determina el orden en que serán considerados los árboles formados con los cuádruples de un bloque básico. Por tal razón, antes de generar código los árboles se ordenan de forma tal que sean satisfechas las restricciones de orden de evaluación y los usos de una misma subexpresión común queden en árboles adyacentes (de esta forma, si el valor de una subexpresión común se mantiene en un registro entre dos usos, tal registro es bloqueado a lo más durante la generación de código para dos árboles).

Para decidir el orden de evaluación de los árboles, primero se forma una gráfica en la cual cada nodo represente un árbol diferente. A cada nodo R se le asocian dos listas ligadas; en la primera lista se colocan los árboles que deben ser evaluados antes que el árbol representado por R y en la otra los árboles que comparten las mismas subexpresiones comunes que R. Para formar esta gráfica, se procede en la siguiente forma:

Algoritmo crea_gráfica _de _árboles

1. Se recorren los nodos de cada árbol hasta encontrar un nodo N con ligas de restricción de orden o ligas de siguiente uso.
2. Se toma una de las ligas del nodo N, si la liga se dirige hacia el nodo M entonces se buscan los árboles a los que pertenecen N y M.
3. Si los nodos pertenecen a diferentes árboles, en la gráfica se coloca una liga de restricción de orden o de siguiente uso, según sea el caso, entre los nodos que repesenten a esos árboles.
4. Se repiten los pasos 2 y 3 hasta terminar con todas las ligas de restricción y de siguiente uso del nodo N.

Termina algoritmo.

Usando el algoritmo anterior, para el conjunto de árboles mostrado en la figura 3.12a se forma la gráfica de árboles de la figura 3.12b. Las ligas de siguiente uso se muestran con arcos de líneas discontinuas, en cada arco se indica la subexpresión común representada por el mismo.

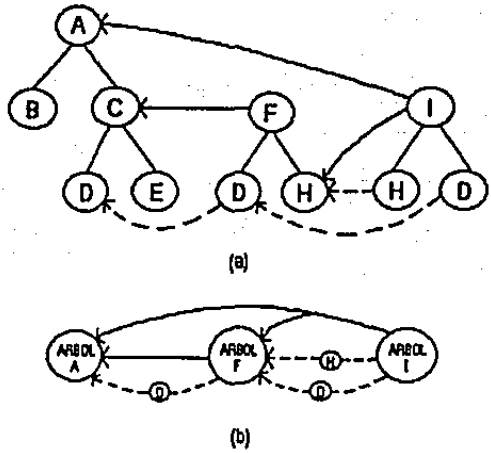

Figura 3.12

Una vez construída la gráfica de árboles, se determina el orden de evaluación de los mismos haciendo un ordenamiento topológico de los nodos de la gráfica de árboles con respecto a las ligas de restricción de orden de evaluación, intentando mantener los usos de una subexpresión común en árboles adyacentes.

Cada nodo de la gráfica de árboles tiene dos campos, en los cuales se indica cuántas ligas de restricción de orden terminan y cuántas salen del árbol representado por el nodo; a la primera cantidad se le llamará el número de restricciones de entrada y a la segunda, el número de restricciones de salida. El algoritmo que se presenta enseguida forma una lista L con los nodos de la gráfica de árboles; los nodos se colocan en la lista en el orden en que se debe emitir código para los árboles que representan.

Algoritmo orden_árboles.

1. Inicialmente L está vacía.
2. Se agregan a L los nodos sin subexpresiones comunes y sin restricciones de entrada.
3. Se busca el nodo con el menor número de subexpresiones comunes, el mayor número de restricciones de salida y sin restricciones de entrada. Este nodo se coloca al principio de L.

   Cada vez que un nodo N sea colocado en la lista L se debe ajustar el campo que indica el número de restricciones de entrada para algunos de los nodos en la gráfica de árboles. Si el nodo N tiene una liga de restricción de orden de evaluación que termine en el nodo M, entonces debe decrementarse en uno el número de restricciones de entrada de M.
4. El siguiente nodo que se agrega a la lista L no debe tener restricciones de entrada. Si hay varios nodos sin restricciones de entrada, se asigna un valor a cada uno de ellos en la siguiente forma:

   4.1 Se hace valor igual a 0.

   4.2 Se recorren las listas de subexpresiones comunes del nodo N y para cada una de ellas se hace lo siguiente:

   si el nodo N comparte una subexpresión común E con el primer nodo de L entonces
   si los usos de la subexpresión común E indicados en los árboles de L han quedado siempre en árboles adyacentes entonces
   si el nodo N representa el último uso de la subexpresión común entonces
   valor = valor + 1
   otro valor = valor + 0.9
   otro valor = valor + 0.8
   otro
   si el nodo N comparte una subexpresión común E con el segundo nodo de L y además, esta es la primera vez que los usos de E no quedan en árboles adyacentes entonces
   valor = valor + 0.7

5. Se elige el nodo para el cual el valor asignado sea mayor y se coloca al principio de la lista L; si varios nodos comparten el mayor valor, se elige el que tenga más restricciones de salida.
6. Se repite el paso 4 hasta colocar todos los nodos en la lista L.

Termina algoritmo.

El algoritmo anterior se utilizará para determinar el orden de evaluación de los nodos de la gráfica de árboles mostrada en la figura 3.13a.

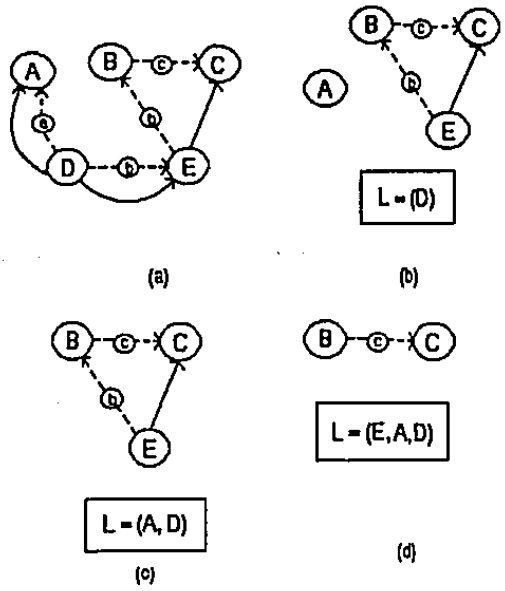

Figura 3.13

El nodo que se coloque al final de la lista L solamente será adyacente a un nodo M, por lo cual no es conveniente colocar al final de L un nodo N con muchas subexpresiones comunes, pues lo más probable es que no todas ellas sean compartidas por N y M y, por lo tanto, los usos de algunas de las subexpresiones comunes de N queden en árboles no adyacentes. Por esta razón el algoritmo coloca al final de L los nodos sin subexpresiones comunes y sin restricciones de entrada.

En la gráfica de la figura 3.13a los únicos nodos sin restricciones de entrada son B y D; ambos nodos comparten subexpresiones comunes con otros nodos de la gráfica, por lo que se aplica el paso 3 del algoritmo para decidir cuál de ellos se coloca primero en L. El nodo B y el nodo D tienen tres ligas de siguiente uso, pero, debido a que D tiene dos restricciones de salida y B ninguna, se decide colocar primero a D en la lista.

Nótese que si primero se coloca al nodo B en L, dado que este nodo no tiene ninguna restricción de salida, el único nodo que puede colocarse enseguida en la lista es D. Este parece ser un mal orden de evaluación, puesto que B y D no comparten subexpresiones comunes y por tanto no hay razón para desear que la evaluación del árbol representado por B siga inmediatamente a la del árbol representado por D. Sería preferible colocar al nodo C o al nodo E junto a B, puesto que de esta forma los usos de alguna de las subexpresiones comunes c o b quedarían en árboles adyacentes.

En cambio, al elegir como primer nodo a D se eliminan las restricciones de entrada de los nodos A y E, pudiendo ahora escoger el segundo nodo entre B, A y E. Mientras más restricciones de entrada sean eliminadas, el conjunto de nodos que pueden ser elegidos es mayor y se incrementa la probabilidad de poder colocar los usos de subexpresiones comunes en árboles adyacentes. Por esta razón, en los pasos 3, 4 y 5 del algoritmo se prefiere colocar primero en L al nodo con el mayor número de restricciones de salida.

La figura 3.13b muestra que después de colocar al nodo D en la lista, los nodos sin restricciones de entrada son A, B y E. Los pasos 4 y 5 se repiten hasta colocar todos los nodos en la lista L; se preferirá colocar en la lista primero alguno de los nodos que comparten subexpresiones comunes con D, es decir, el nodo A o el nodo E. El algoritmo de generación de código intentará mantener el valor de una subexpresión común en un registro, de forma tal que ese registro será bloqueado mientras existan instrucciones que utilicen el valor que está almacenado en él; los usos de la subexpresión común se mantendrán lo más próximos posibles para reducir al mínimo la probabilidad de tener que respaldar temporalmente en la memoria el valor de ese registro. Cada vez que un registro sea respaldado en la memoria, se deberán emitir dos instrucciones: una instrucción de almacenamiento a memoria para liberar el registro y una instrucción de carga a registro para restaurar el valor del mismo.

Los nodos D y A comparten la subexpresión común a, mientras que D y E comparten la subexpresión b. Dado que el nodo A representa el último uso del valor de la variable a, pero E no representa el último uso de b, entonces se preferirá colocar al nodo A junto a D.

Al evaluar el árbol representado por A inmediatamente antes del árbol representado por D, todos los usos de la variable a quedarán en árboles adyacentes y lo más probable es que el valor de a pueda ser mantenido en un registro mientras existan instrucciones que utilicen su valor. En cambio, si el nodo E es evaluado junto a D, los dos primeros usos de b quedan en árboles adyacentes y se debería colocar ahora el nodo B junto a E para que todos los usos de la variable b queden muy cerca unos de otros. Es posible, sin embargo, que el nodo B no pueda colocarse junto a E, ya sea porque B tiene restricciones de entrada o bien porque E comparte varias subexpresiones comunes con algún otro nodo N y por lo tanto, conviene más colocar a N junto a

E. La probabilidad de emitir código para guardar el valor de un registro en la memoria es mayor si se colocan juntos los nodos E y D, que si se coloca A junto a D.

Al diseñar el algoritmo heurístico para ordenar los árboles, el razonamiento anterior sugirió que el peso asignado al hecho de colocar juntos dos nodos que representen los últimos usos de una subexpresión común debe ser mayor al peso asociado a colocar adyacentes nodos que no representen los últimos usos de una subexpresión común.

Después de agregar el nodo A en la lista L quedan tres nodos en la gráfica, de los cuales B y E no tienen restricciones de entrada. Ninguno de los dos nodos comparte una subexpresión común con el primer nodo de L, sin embargo E y D comparten la subexpresión b, por lo cual ahora se elige colocar en L al nodo E. Obsérvese que si dos nodos no aparecen adyacentes en la lista y ambos representan usos de una subexpresión común, dejar el valor de esa subexpresión común en un registro entre los usos implica bloquear el registro durante mucho tiempo y por lo tanto se incrementa la probabilidad de que alguna instrucción necesite usar ese registro y, entonces, se deba respaldar su valor en la memoria. Por esta razón, el algoritmo prefiere agregar al principio de L un nodo que comparte una subexpresión común con el primer nodo de la lista, a insertar en la misma un nodo que comparte una subexpresión común con el segundo nodo de L.

La figura 3.13d muestra que después de colocar a E en L, quedan en la gráfica dos nodos sin restricciones de entrada. Se coloca primero a B en L, puesto que B y E comparten una subexpresión común. El orden de evaluación elegido por el algoritmo se indica en la lista siguiente:

L = (C, B, E, A, D)

los nodos se deben evaluar de izquierda a derecha.

CAPÍTULO 4

ALGORITMO DE GENERACIÓN DE CÓDIGO

El algoritmo de Aho-Johnson [Aho76] es capaz de generar código óptimo para árboles de expresión, sin embargo, el modelo de máquina que proponen los autores tiene las siguientes limitaciones que impiden aplicar el método en la generación de código para máquinas reales:

1. Todos los registros del procesador deben ser idénticos. Prácticamente ningún procesador real cumple esta condición.
2. Para formar la dirección de una celda de la memoria no debe utilizarse ningún registro. Normalmente los procesadores permiten varios modos de direccionar la memoria y algunos modos requieren el empleo de registros de dirección.
3. El conjunto de instrucciones que el algoritmo puede manejar es muy restringido, por lo cual al generar código sólo podrían usarse algunas de las instrucciones de un procesador.

El algoritmo de generación de código desarrollado en esta tesis extiende al algoritmo de Aho-Johnson en dos aspectos principales:

1. Se amplía el modelo de máquina para poder modelar adecuadamente procesadores reales.
2. Se toma en cuenta la presencia de subexpresiones comunes en el programa de entrada.

En las primeras dos secciones de este capítulo se describen los tipos de procesadores que pueden ser manejados por el algoritmo y la forma en que se debe dar al algoritmo la especificación de un procesador. En el resto del capítulo se hace una descripción detallada del algoritmo de generación de código.

4.1 Modelo de máquina

El algoritmo propuesto puede generar código para procesadores con las siguientes características:

1. Registros asimétricos. El procesador puede tener registros de k clases diferentes $c_1, c_2, \ldots c_k$; todos los registros de una misma clase deben ser idénticos. Por ejemplo, el algoritmo es capaz de generar código para el procesador 68000 de Motorola cuyos registros se dividen en dos clases: registros de dirección y registros de datos; si en alguna instrucción se requiere un registro de datos, se podrá utilizar cualquier registro de la segunda clase, mientras que si se necesita un registro de dirección, será posible utilizar cualquier registro del primer grupo.
2. Las instrucciones del procesador pueden ser de alguno de los cinco tipos que se indican enseguida (en la notación utilizada, R representa cualquiera de los registros de la máquina, M denota una dirección de memoria y E representa alguna operación junto con sus operandos):

    i. Instrucciones cuyo resultado queda en un registro: R ← E. El resultado puede dejarse en el registro ocupado por alguno de los operandos o en un registro diferente.
    ii. Instrucciones cuyo resultado se deja en la memoria: M ← E. El resultado debe dejarse en la celda de la memoria que ocupa alguno de los operandos de la instrucción.
    iii. Instrucciones de carga a memoria: M ← R.
    iv. Instrucciones de almacenamiento en registro: R ← M.
    v. Instrucciones de transferencia entre registros: R ← R.

3. Solamente se manejan operaciones sin operandos, operaciones unarias y operaciones binarias.
4. Los modos de direccionamiento de la memoria pueden utilizar uno o más registros del procesador para formar una dirección.
5. Se permiten instrucciones que tomen un operando o almacenen el resultado en más de un registro; por ejemplo, en el procesador 8086 de Intel, el resultado de una multiplicación se deja en la pareja de registros dx:ax.

6. Se permiten restricciones en cuanto a la dirección que puede ocupar un dato en la memoria; por ejemplo, en el procesador 68000 de Motorola un valor de 16 bits debe ser colocado en una dirección par.

4.2 Especificación del procesador

El generador de código desarrollado recibe como entrada un programa escrito en representación intermedia junto con la descripción de un procesador y produce como salida un programa equivalente al de entrada escrito en lenguaje ensamblador del procesador que se especificó.

La descripción de un procesador se proporciona al generador de código mediante diez tablas cuyo contenido se describe a continuación.

1. Tabla de instrucciones. Normalmente existen varias maneras de traducir una instrucción de código intermedio a lenguaje de máquina, cada una de esas maneras tiene asociado un costo diferente (el costo puede ser el tamaño en bytes que ocupa el código o bien el número de ciclos de reloj que necesita el procesador para ejecutar las instrucciones de máquina). Tal información (las diversas formas de traducir a lenguaje de máquina cada una de las instrucciones de representación intermedia) está almacenada en la tabla de instrucciones.

El algoritmo de generación de código analiza el contenido de esta tabla para elegir las instrucciones que deben ser emitidas a fin de que el código objeto resultante sea lo menos costoso posible.

Cada entrada de la tabla de instrucciones tiene los siguientes campos:

i. El operador de representación intermedia.
ii. Los tipos de los operandos. Un operando puede ser una constante, un valor guardado en registro o un valor almacenado en la memoria.
iii. El tipo del resultado. El resultado de una operación puede dejarse en la memoria o en un registro. Es posible que el resultado ocupe el registro o la celda de la memoria utilizada por alguno de los operandos de la instrucción.
iv. El código en lenguaje ensamblador equivalente a la instrucción de representación intermedia.
v. El costo del código . Al llenar las tablas para probar el algoritmo de generación de código (ver capítulo 5), se eligió como costo el número de ciclos de reloj requeridos para la ejecución de las instrucciones de máquina.

vi. Un conjunto de bits que contienen información sobre el operador de representación intermedia y sobre el código equivalente en lenguaje de máquina. Entre otras cosas estos bits indican lo siguiente:

- Si el operador es conmutativo.
- Si hay ciertas condiciones que deben cumplirse para que el código asociado pueda emitirse. Por ejemplo, la instrucción inc del procesador 8086 de Intel puede utilizarse para traducir la operación de suma solamente si uno de los operandos es el valor constante 1.

Los restantes bits son utilizados para facilitar la asignación de registros y generación de código y se describen en el apéndice A.

2. Tabla de clases de registros. Como se mencionó en la sección anterior, el algoritmo de generación de código es capaz de emitir código para procesadores con registros asimétricos, en esta tabla se indican las clases de registros que tiene el procesador, así como cuales registros pertenecen a cada clase.

Algunas instrucciones de máquina pueden manejar un grupos de registros como si fueran uno solo. Por ejemplo, para la operación de multiplicación el procesador 8086 de Intel toma el primer operando de la pareja de registros dx:ax. Existen, entonces, dos tipos de clases de registros: clases cuyos elementos son grupos de registros (la pareja dx:ax, por ejemplo) y clases cuyos elementos son registros simples.

3. Tabla de nombres de registros. A cada registro de la máquina se le asocia un número entero; cada vez que en las tablas se hace referencia a un registro, se utiliza el código numérico asociado. La tabla de nombres de registros, relaciona los códigos numéricos con los nombres de los registros que deben ser utilizados al escribir el programa objeto.

4. Tabla de conversión entre clases de registros. Al estar emitiendo código puede ser necesario pasar un valor almacenado en cierta clase de registro a una clase diferente. En esta tabla se especifica el código que debe ser emitido para hacer la conversión entre cada una de las clases de registros que fueron definidas. Junto con el código se indica el costo del mismo.

5. Tabla de modos de direccionamiento. La mayor parte de los procesadores permite varias maneras (llamadas modos de direccionamiento) de formar la dirección de una celda de la memoria; en esta tabla se indica cuales son tales modos. Cada entrada de la tabla tiene cinco campos donde se especifica la siguiente información:

i. El modo de direccionamiento de la memoria.
ii. El costo de tomar un valor almacenado en la memoria utilizando el modo de direccionamiento especificado.
iii. Los registros requeridos por el modo de direccionamiento para formar la dirección.
iv. El código que debe emitirse para formar la dirección de una celda de la memoria utilizando el modo de direccionamiento.
v. El formato utilizado en lenguaje ensamblador para cada modo de direccionamiento.

6. Tabla de conversión. En esta tabla se indica el código que debe ser emitido para pasar a un registro un valor que está almacenado en la memoria y viceversa. Junto con el código, se especifica el costo del mismo.

7. Tabla de costo de formación de dirección. El algoritmo de generación de código diseñado es capaz de tomar en cuenta las subexpresiones comunes presentes en el programa de entrada. Si un valor se utiliza muchas veces, se intentará guardarlo en un registro hasta que sea utilizado por todas las instrucciones que lo requieran; sin embargo a veces esto no será posible debido a la cantidad limitada de registros del procesador. Si no es posible mantener el valor de una expresión común en registro, entonces se intentará utilizar el modo de direccionamiento de la memoria con el cual se pueda tomar el valor de la subexpresión común con el menor costo posible.

Si un valor almacenado en la memoria se utiliza varias veces, es posible que la primera vez que se tome tal valor de la memoria el costo asociado sea mayor que el de leer ese mismo valor las siguientes veces; esto es, debido a que al formar por primera vez la dirección de la celda de la memoria, puede ser necesario cargar una parte de la dirección en alguno de los registros R de la máquina. Una vez cargado el registro R, formar la dirección de la celda las siguientes veces tiene un costo menor, siempre y cuando R no se modifique.

La tabla de costo de formación de dirección indica para cada modo de direccionamiento de la memoria cuál es el costo de formar la dirección de una celda de la memoria la primera vez y las siguientes veces.

8. Tabla de tipos de datos. Los lenguajes de programación de alto nivel permiten definir en un programa varios tipos de variables. Por ejemplo, en el lenguaje C es posible definir variables simples (enteros, caracteres, reales, etcétera) y variables estructuradas (arreglos y estructuras); las variables, además, pueden ser locales o globales.

Al traducir un programa escrito en algún lenguaje de programación a lenguaje de máquina, es necesario asignar celdas de la memoria para cada una de las variables del programa fuente. Dependiendo del tipo de variable, para direccionar la celda de la memoria que se le asignó, puede

convenir utilizar un modo de direccionamiento o un modo diferente. Así, en el procesador 8086 de Intel, por ejemplo, a las variables globales enteras de un programa escrito en C, se les reserva espacio en alguno de los segmentos de datos de la memoria y por lo tanto, lo más conveniente es utilizar el modo de direccionamiento directo para leer su valor.

En la tabla de tipos de datos se indica cuál es el modo o los modos de direccionamiento de memoria que más conviene utilizar para tomar la dirección de la celda de la memoria asignada a valores de cada uno de los tipos del lenguaje fuente.

9. Tabla de tamaños de tipos. Esta tabla señala cuantos bytes ocupa en la memoria cada uno de los tipos de datos definidos en el lenguaje fuente. El contenido de esta tabla se determina en base a los tamaños de datos que pueden manejar el procesador y la memoria.

10. Tabla de restricciones sobre las direcciones de la memoria. Algunos computadores no permiten que datos de cierto tamaño se coloquen en cualquier dirección de la memoria. Por ejemplo, el procesador 68000 de Motorola sólo permite almacenar un valor de 16 bits en una dirección par.

La tabla de restricciones sobre las direcciones de la memoria indica cuales son las restricciones impuestas por el procesador sobre las direcciones que pueden ser ocupadas por valores de cada uno de los tipos de datos permitidos por el lenguaje fuente.

4.3 Algoritmo de generación de código

El algoritmo de generación de código realiza varios recorridos sobre los árboles de expresión que recibe como entrada antes de poder emitir código:

1. En el primer recorrido, utilizando programación dinámica se determina para cada nodo N el costo mínimo del código que sería necesario emitir para realizar las operaciones indicadas en el subárbol cuya raíz es N, considerando que solamente hay $i_1, i_2, ..., i_k$ registros de cada clase disponibles. Este cálculo se realiza para todos los valores posibles de $i_1, i_2, ..., i_k$.

   Para poder realizar la tarea anterior, es necesario asociar a cada nodo N una matriz de costos C de k dimensiones (siendo k el número de clases de registros del procesador). El tamaño de la i-ésima dimensión es $n_i + 1$, donde n_i es el número de registros de la clase i. En la entrada $C[i_1, i_2, ..., i_k]$, se colocará el costo mínimo de evaluar el nodo N utilizando solamente $i_1, i_2, ..., i_k$ registros de cada clase.

Además del costo, en cada entrada de la matriz se indica cuál es el código que se debe emitir para realizar la operación indicada en el nodo.

2. La información colocada en las matrices de costos durante el paso anterior señala las diversas formas de emitir código para cada uno de los nodos. En un segundo recorrido realizado sobre los árboles, se elige el código de menor costo que se puede producir para cada nodo.
3. Se buscan los nodos que representan subexpresiones comunes y se intenta mantener el valor indicado por cada uno de ellos en un registro mientras existan instrucciones que utilicen tal valor (esto se hace debido a que tomar un valor de registro es normalmente menos costoso que tomar el mismo valor de la memoria). Nótese que al mantener una subexpresión común en un registro, ese registro se bloquea y, por lo tanto, se incrementa la probabilidad de tener que emitir una instrucción para almacenar temporalmente el valor de algún registro en la memoria debido a que no existan suficientes registros libres para evaluar una expresión, y así, el código resultante puede ser más costoso que el obtenido sin mantener la subexpresión común en registro. Para prevenir que en este recorrido la calidad del código se degrade, cada vez que se considere una subexpresión común, se comparará el costo del código obtenido antes de mantener su valor en un registro con el costo que se obtendría después de hacerlo, eligiéndose el menor de los dos.
4. En el siguiente recorrido se realiza la asignación de registros, tarea que es relativamente sencilla, pues en los tres pasos anteriores se revisó que las instrucciones de máquina no intenten utilizar más registros que los disponibles en el procesador.
5. Finalmente se procede a escribir el código objeto en el archivo de salida.

En las siguientes subsecciones se explica en detalle cada una de las partes del algoritmo de generación de código.

4.3.1 Cálculo de la matriz de costos

Para llenar las matrices de costos asociadas a los nodos es necesario consultar la información contenida en las tablas de especificación del procesador. Por ejemplo, si un nodo representa una operación aritmética, se buscará en la tabla de instrucciones el código que se debe emitir para realizar tal operación y el costo asociado a ese código se utiliza para llenar las entradas de la matriz de costos del nodo.

Cada entrada de la matriz de costos asociada a un nodo N consta de dos listas, L_M y L_R. La lista L_M tiene tantos elementos como modos de direccionamiento hay en el procesador y L_R tiene tantas entradas como clases de registros fueron definidas. En L_M se indica el costo de

evaluar el nodo en la memoria utilizando cada uno de los modos de direccionamiento; mientras que en L_R se coloca el costo de evaluar el nodo en cada una de las clases de registros del procesador. Además del costo, en cada entrada de las listas hay un conjunto de campos en los que se almacena la siguiente información:

1. El código que se debe emitir para evaluar el nodo. Puesto que el código se toma de las tablas de especificación del procesador, en este campo puede indicarse solamente qué tablas (y que posición de cada tabla) se deben consultar para obtener el código.
2. El orden de evaluación de los operandos (recuérdese que el orden de evaluación de los nodos afecta a la calidad del código producido).
3. El número de registros de cada clase requeridos para almacenar el valor del primer operando.

Los elementos de la lista L_M asociada a la entrada $C[i_1,i_2,...i_k]$ que indican el costo de evaluar el nodo en la memoria, serán denotados por $C_{Mj}[i_1,i_2,...i_k]$, para $1 \leq j \leq m$, donde m es el número de modos de direccionamiento de la memoria. Los elementos de L_R usados para guardar el costo de evaluar el nodo en registro se denotarán mediante $C_{Rj}[i_1,i_2,..i_k]$, para $1 \leq j \leq k$, donde k es el número de clases de registros del procesador.

El algoritmo utilizado para llenar las matrices de costos asociadas a los nodos de un árbol se presenta a continuación:

Algoritmo costos (raíz)

Se hace un recorrido en posterior del árbol y para cada nodo N se llena la matriz de costos en la siguiente forma:

1. En cada entrada de la matriz, el campo de costo de cada elemento de las listas L_R y L_M se inicializa a infinito.

2. Cálculo de costos para un nodo hoja.

   Si el nodo es una hoja, hay tres posibilidades: que el nodo represente una operación sin operandos, que represente el valor de una variable o bien el valor de una constante.

   2.1. Cálculo de costos para una variable o constante.

   Si el nodo representa el valor de una variable o el valor de una constante, se busca en la tabla de tipos de datos el modo o los modos de direccionamiento que se pueden utilizar para ese tipo de variable o constante.

Si hay varios modos de direccionamiento que pueden emplearse para formar la dirección de la celda de la memoria asignada a una variable V y, además, V es utilizada varias veces, entonces se elige el modo de direccionamiento con el que sea más barato tomar su valor de la memoria. El costo de tomar de la memoria el valor de la variable V viene dado por la expresión:

(costo de formar la dirección la primera vez) +
(costo de formar la dirección las siguientes veces) *
(número de usos de la variable V - 1)

los costos de formación de la dirección vienen dados en la tabla de costos de formación de dirección.

Una vez elegido el modo o los modos de direccionamiento que se pueden utilizar, para cada uno de tales modos M_j se procede en la siguiente forma:

2.1.1. Se busca en la tabla de modos de direccionamiento la entrada E correspondiente al modo M_j.

2.1.2. Se toma de esa entrada el costo C_1 y el número de registros $r_1, r_2, ..., r_k$ de cada clase requeridos para formar la dirección utilizando el modo de direccionamiento M_j.

2.1.3. En las entradas $C_{Mj}[i_1,i_2,...i_k]$, tales que $i_1 \geq r_1$, $i_2 \geq r_2$,..., $i_k \geq r_k$, correspondientes al modo de direccionamiento M_j, se guarda la siguiente información:

- costo: C_1
- código: instrucciones tomadas de la posición E de la tabla de modos de direccionamiento

2.1.4. Se busca en la tabla de conversión la entrada E_1, que indica, cuál es el código que se debe emitir para pasar a un registro de la clase h el valor guardado en una celda de la memoria cuya dirección se forma utilizando el modo de direccionamiento M_j. Sea C_2 el costo de tal código.

Los campos de $C_{Rh}[i_1,i_2,...,i_k]$ (para todas las clases h presentes en la tabla de conversión y todos los valores $i_1 \geq r_1$, $i_2 \geq r_2$,..., $i_h \geq \max\{r_h,1\}$,..., $i_k \geq r_k$) se llenan con los siguientes valores:

- costo: $C_1 + C_2$

- código: instrucciones tomadas de la entrada E_1 de la tabla de costo de conversión

2.1.5. Si en alguna entrada $C_{Rj}[i_1,i_2,...,i_k]$ el campo de costo es infinito, entonces para todos los valores de i comprendidos entre 1 y k se calcula:

$C_{Rj}[i_1,i_2,...,i_k]$.costo = Mínimo {$C_{Rj}[i_1,i_2,...,i_k]$.costo, $C_{Ri}[i_1,i_2,...,i_k]$.costo + (costo de conversión $R_i \rightarrow R_j$)}

el costo de conversión entre las clases de registros j e i se toma de la tabla de conversión entre clases de registros. Si el segundo término es el menor, entonces, además del costo, en la entada $C_{Rj}[i_1,i_2,...,i_k]$ también se almacena el código tomado de la tabla de conversión entre clases de registros para realizar la conversión especificada.

2.2 Cálculo de costos para un operador sin operandos

Si el nodo representa un operador sin operandos, entonces, se busca en la tabla de instrucciones la entrada E correspondientes a tal operador. De esta entrada se determina el mínimo número de registros $r_1, r_2, ..., r_k$ de cada clase necesarios para realizar la operación y el costo C_1 del código equivalente.

En cada una de las entradas $C[i_1,i_2,...,i_k]$ tales que $i_1 \geq r_1, i_2 \geq r_2, ..., i_k \geq r_k$ se llenan los campos de $C_{Rj}[i_1,i_2,...i_k]$ con los siguientes valores:

- costo: C_1
- código: instrucciones tomadas de la posición E de la tabla de instrucciones.

Puesto que un operador sin operandos no representa valor alguno, se toma la convención de almacenar el costo del código equivalente a tal operador en las listas L_R de la matriz de costos.

3. Cálculo de costos para un nodo interno

Si el nodo N es un nodo interno, entonces, debe representar un operador op unario o binario; se buscan en la tabla de instrucciones las entradas correspondientes a ese operador y para cada una de tales entradas E se procede en la siguiente forma:

3.1. Se revisa si las restricciones indicadas en la entrada E de la tabla de instrucciones son satisfechas por el subárbol cuya raíz es N; si las restricciones no se satisfacen,

entonces se busca la siguiente entrada de la tabla de instrucciones correspondiente al operador op y se brinca al paso 3.1.

Entre las restricciones que *pueden encontrarse* en la tabla de instrucciones están: verificar que alguno de los hijos del nodo N represente el valor de cierta constante, o bien, *revisar* que ambos hijos de N representen el mismo valor.

3.2. Se determina el mínimo número de registros $r_1, r_2, ..., r_k$ de cada clase necesarios para ejecutar el código asociado a la entrada E de la tabla de instrucciones. Los valores r_i se calculan en la siguiente forma:

$$r_i = \text{Máximo } \{r1_i + r2_i + ra_i, res_i\}$$

donde $r1_i$ y $r2_i$ representan el número de registros de la clase i requeridos para almacenar el valor del primer y del segundo operando respectivamente, res_i es el número de registros de la clase i necesarios para almacenar el valor del resultado y ra_i es el número de registros adicionales de la clase i requeridos para ejecutar el código (estos registros adicionales no son usados para almacenar el valor de los operandos ni del resultado). Los valores de $r1_i$, $r2_i$, res_i y ra_i se obtienen de la tabla de instrucciones, de la tabla de modos de direccionamiento y de la tabla de clases de registros.

3.3. Para cada una de las entradas $C[i_1,i_2,...,i_k]$ de la matriz de costos asociada a N, tales que $i_1 \geq r_1$, $i_2 \geq r_2$, ..., $i_k \geq r_k$, se llenan las listas L_M y L_R en la siguiente forma:

3.3.1. Si el operador es binario, entonces hay dos posibilidades en cuanto al orden de evaluación de sus hijos: evaluar primero el hijo izquierdo o bien evaluar primero el hijo derecho; se consideran ambas posibilidades a fin de que el código resultante sea lo más eficiente posible.

Los pasos que se describen a continuación se repetirán dos veces si op es un operador binario; en la primera vez, se considerará que el operando que se evalúa primero es el hijo izquierdo del nodo y, en la segunda vez, se supondrá que primero será evaluado el hijo derecho de N.

3.3.2. Para la evaluación del primer hijo de N (se denotará tal hijo por h_1), hay $i_1, i_2, ..., i_k$ registros disponibles, por lo tanto se consulta la entrada $C[i_1,i_2,...,i_k]$ de la matriz de costos de h_1 a fin de conocer el código que debe emitirse para evaluar ese hijo. Si h_1 es el hijo izquierdo de N, entonces se consulta la entrada E de la tabla de instrucciones para determinar el tipo T del primer operando; si

h_1 es el hijo derecho de N, entonces, en T se coloca el tipo del segundo operando.

Si T representa un modo de direccionamiento, se busca en la lista L_M asociada a la entrada $C[i_1,i_2,..,i_k]$de la matiz de costos de h_1 el costo C_1 de evaluar h_1 en la memoria utilizando el modo de direccionamiento T. Si T representa una clase de registro entonces se debe buscar en la lista L_R el costo C_1 de evaluar h_1 en tal clase.

Si C_1 es infinito, entonces, se busca en la tabla de instrucciones la siguiente entrada correspondiente al operador op y se salta al paso 3.1.

3.3.3. Para la evaluación del segundo hijo (tal hijo será denotado por h_2) solamente hay disponibles i_1-rl_1, i_2-rl_2, ..., i_k-rl_k registros (esto es, porque el valor del primer operando ocupa $rl_1, rl_2, .., rl_k$ registros); la determinación del costo C_2 del código necesario para evaluar h_2, se realiza de forma similar al cálculo de C_1.

3.3.4. Si la entrada E de la tabla de instrucciones indica que el resultado debe almacenarse en la memoria utilizando el modo de direccionamiento M_j, entonces se busca en la lista L_M asociada a la posición $C[i_1,i_2,...,i_k]$ de la matriz de costos del nodo N la entrada $C_{Mj}[i_1,i_2,..i_k]$ correspondiente a tal modo. El campo de costo se calcula en la siguiente forma:

$$C_{Mj}[i_1,i_2,...,i_k].costo = Mínimo \{C_{Mj}[i_1,i_2,...,i_k].costo, C_1 + C_2 + (tabla\ de\ instrucciones[E].costo)\}$$

si el valor del segundo término es el menor, entonces en la entrada $C_{Mj}[i_1,i_2,...,i_k]$ se almacena también la siguiente información:

- código: instrucciones tomadas de la posición E de la tabla de instrucciones
- el orden de evaluación de los operandos
- el número de registros $(rl_1,rl_2,...,rl_k)$ requeridos para guardar el valor del primer operando.

3.3.5. Si el resultado de la operación debe dejarse en un registro, el cálculo del costo procede en manera análoga al caso anterior.

3.3.6. Si el operador op es conmutativo, entonces, se intercambian los hijos del nodo N y se repiten los pasos desde el 3.3 (no es necesario intercambiar realmente los hijos, sino solamente los apuntadores h_1 y h_2 a ellos).

3.4. Se toma la siguiente entrada E de la tabla de instrucciones correspondiente al operador op y se salta al paso 3.1. Si ya no hay más entradas en la tabla para tal operador, entonces se ejecuta el paso 3.5.

3.5. Si alguno de los elementos de las listas L_R asociadas a las entradas de la matriz del nodo N tiene en el campo de costo el valor de infinito, entonces, se usará la siguiente expresión para calcular el campo de costo de cada una de tales entradas:

$$C_{Rj}[i_1,i_2,...,i_k].costo = Mínimo \{ C_{Rj}[i_1,i_2,...,i_k].costo, C_{Ri}[i_1,i_2,...,i_k].costo + (costo\ de\ conversión\ R_i \rightarrow R_j)\}$$

en la expresión anterior se consideran todas las clases de registros i tales que $i <> j$. Si el valor del segundo término es el menor, entonces, en la entrada $C_{Rj}[i_1,i_2,...,i_k]$ se almacena el código requerido para hacer la conversión entre las clases de registros j e i, código que se toma de la tabla de conversión entre clases de registros.

3.6. Para evaluar un nodo en la memoria, es posible primero evaluarlo en un registro y después almacenar el valor de ese registro en la memoria. Para tomar en cuenta esta posibilidad, se aplica la siguiente fórmula a cada entrada de la matriz de costos para calcular el campo de costo de cada uno de los elementos de la lista L_M:

$$C_{Mi}[i_1,i_2,...,i_k].costo = Mínimo \{C_{Mi}[i_1,i_2,...,i_k].costo, C_{Rj}[i_1,i_2,...,i_k].costo + (costo\ de\ conversión\ R_j \rightarrow M_i)\}$$

el cálculo anterior se realiza para todas las clases de registros j, $1 \leq j \leq k$ y para todos los valores posibles de $i_1,i_2,...,i_k$. Si el segundo término es el menor, entonces, además del costo, en la entrada $C_{Mi}[i_1,i_2,...,i_k]$ se indica el código necesario para hacer la conversión entre la clase de registro R_j y el modo de direccionamiento M_i; el código se toma de la tabla de conversión.

3.7. Para evaluar un nodo en un registro, es posible evaluarlo primero en la memoria y después cargar el valor del nodo en alguno de los registros del procesador. La siguiente expresión se utiliza para calcular el campo de costo de cada uno de los elementos de las listas L_R:

$$C_{Ri}[i_1,i_2,...,i_k].costo = Mínimo \{ C_{Ri}[i_1,i_2,...,i_k].costo, C_{Mj}[i_1,i_2,...,i_k].costo + (costo\ de\ conversión\ M_j \rightarrow R_i)\}$$

la expresión anterior se aplica para todos los modos de direccionamiento j, $1 \leq j \leq m$ y se consideran todos los valores posibles de $(i_1,i_2,...i_k)$. Si el segundo término es el menor, entonces, también debe almacenarse en la entrada $C_{Ri}[i_1,i_2,...,i_k]$ el código tomado de la tabla de conversión para pasar el valor almacenado en la memoria a un registro de la clase i.

3.8 Finalmente, se vuelven a considerar los elementos de la lista L_M, utilizando en cada uno de ellos la expresión:

$$C_{Mi}[i_1,i_2,...,i_k] = \text{Mínimo } \{C_{Mi}[i_1,i_2,...,i_k], C_{Mi}[n_1,n_2,...,n_k]\}$$

donde $n_1,n_2,...,n_k$ es el número de registros de cada clase del procesador. El segundo término representa el costo de evaluar el nodo en la memoria utilizando el modo de direccionamiento M_i y teniendo libres todos los registros del procesador; si el segundo término es el menor, para asegurar que la evaluación del nodo tenga todos los registros del procesador disponibles, será necesario evaluar el subárbol cuya raíz es N antes de considerar cualquier otro nodo del árbol.

Termina algoritmo.

4.3.2 Selección de Instrucciones

La información colocada durante el primer paso del algoritmo de generación de código en las matrices de costos indica diferentes formas de emitir código para las operaciones representadas por los nodos de los árboles. Para garantizar que se producirá código de buena calidad, se debe elegir de todas las posibilidades aquella que tenga el costo asociado más pequeño; esta tarea es realizada por el segundo paso del algoritmo de generación de código.

Es posible que el código elegido por el algoritmo para algún subárbol S utilice todos los registros del procesador, en tal caso se decidirá emitir código para S antes de generar código para cualquier otro nodo del árbol (de esta forma se garantiza que todos los registros estarán desocupados). En este paso, además de elegir las instrucciones que serán emitidas para cada nodo, también se formará una lista L con los subárboles que requieren todos los registros del procesador y por lo tanto que deben ser evaluados al principio.

El siguiente algoritmo, que es invocado para cada árbol de un bloque básico, se encarga de elegir las instrucciones y determinar el orden de evaluación de los nodos de cada uno de los árboles.

Algoritmo selección_de_código_y_orden_de_evaluación (raíz)

1. Inicialmente la lista L es vacía.
2. Se busca en las listas L_M y L_R asociadas a la posición $C[n_1,n_2,...n_k]$ de la matriz de costos del nodo raíz la entrada que tenga el costo asociado más pequeño (n_i es el número de registros de la clase i del procesador). Para tal entrada se determina el tipo T del resultado.
3. Se invoca a la función selección_de_instrucciones(raíz,T,$n_1,n_2,...,n_k$), que se encargará de elegir el conjunto de instrucciones que deben ser emitidas para el árbol y de llenar la lista L.
4. Finalmente se modifica el árbol para permitir que los subárboles cuyas raíces están indicadas en L sean evaluados cuando todos los registros del procesador estén libres, esta tarea es realizada por la función ordenar_subárboles, que recibe dos argumentos, el apuntador a la raíz del árbol y la lista L.

Termina algoritmo.

El algoritmo selección_de_instrucciones es recursivo, en cada llamada se encarga de elegir las instrucciones que serán emitidas para el nodo del árbol apuntado por el primer parámetro y de decidir si el subárbol cuya raíz es tal nodo debe ser agregado o no a la lista L. El primer argumento de la función selección_de_instrucciones es un apuntador al nodo que se va a considerar, el segundo argumento establece cuál es el modo de direccionamiento o la clase de registro en la que debe quedar el valor representado por el mismo y los restantes parámetros indican cuantos registros están disponibles para la evaluación del nodo.

Algoritmo selección_de_instrucciones (N,T,$i_1,i_2,...,i_k$)

1. Se inicializa la lista I como vacía; en esta lista se guardarán las instrucciones elegidas para el nodo N.
2. Se busca en la lista L_M o en la lista L_R asociada a la posición $C[i_1,i_2,...,i_k]$ de la matriz de costos del nodo apuntado por N la entrada correspondiente al modo de direccionamiento o a la clase de registro T, según sea el caso. De tal entrada se determina:

   - El conjunto J de instrucciones asociadas a la entrada.
   - El hijo H que debe ser evaluado primero (esto se hace solamente si el nodo N representa un operador binario).
   - El número de registros $j_1,j_2,...,j_k$ de cada clase disponibles para evaluar el segundo hijo.

3. Se agregan las instrucciones de J a la lista I.
4. Se toma el conjunto de instrucciones J y para cada una de ellas se procede en la siguiente forma:

4.1 Si en el conjunto J hay una instrucción de almacenamiento a memoria $R_i \rightarrow M_j$, entonces se hace $T = R_i$ y se salta al paso 2.

4.2 Si en el conjunto J hay una instrucción de carga a registro $M_i \rightarrow R_j$, entonces se hace $T = M_i$ y se salta al paso 2.

4.3 Si en el conjunto J hay una instrucción $M_i \leftarrow M_n$ (es decir el costo de evaluar el nodo en el modo de direccionamiento M_i utilizando $i_1,i_2,...,i_k$ registros es igual al costo de evaluar el nodo utilizando $n_1,n_2,...n_k$ registros), entonces se hace $(i_1,i_2,...i_k) = (n_1,n_2,...,n_k)$ y se salta al paso 2.

4.4 Si no se cumple ninguna de las condiciones anteriores, entonces se busca en J la instrucción tomada de la tabla de instrucciones y se determinan T_1 y T_2, que son las clases de registros o modos de direccionamiento utilizados para el primer y el segundo hijo de N respectivamente.

5. Si el nodo tiene al menos un hijo, entonces se hace la llamada recursiva: selección_de_instrucciones(H,T_1,i_1,i_2,...,i_k), para elegir el conjunto de instrucciones que serán emitidas para el primer hijo del nodo N; el parámetro H es un apuntador al hijo que debe ser evaluado primero.
6. Si el nodo representa un operador binario, entonces, se hace la siguiente llamada: selección_de_instrucciones(H_2,T_2,j_1,j_2,...,j_k), para escoger las instrucciones que serán emitidas para el segundo hijo del nodo, que es apuntado por H_2.
7. Si en el conjunto de instrucciones I elegido para el nodo N se incluye la instrucción $M_i \leftarrow M_n$, entonces, el nodo N es agregado a la lista L puesto que la evaluación de N utiliza todos los registros del procesador.
8. Se guarda en el nodo N el conjunto de instrucciones I. Esta información será utilizada cuando se escriba el código objeto en el archivo de salida.

Termina algoritmo.

Obsérvese que un nodo N es agregado a la lista L hasta después de haber considerado todos sus hijos, es por ello que si alguno de los descendientes de N se coloca también en L, entonces aparecerá en la lista antes que N. Es posible, entonces, generar código para los subárboles en el orden en que estos aparecen en L, puesto que en tal orden la evaluación de los nodos que aparecen más abajo en el árbol precederá a la de los nodos que aparecen arriba; se deja al final la emisión de código para los nodos que no aparecen en ninguno de los subárboles de L.

El siguiente algoritmo se encarga de sacar de un árbol los subárboles indicados en la lista L.

Algoritmo ordenar_subárboles (raíz,L)

1. Por cada nodo N de la lista L se crea un nuevo árbol y se realizan las siguientes modificaciones sobre el árbol apuntado por raíz (ver figura 4.1), para que la información contenida en el mismo no sea alterada:

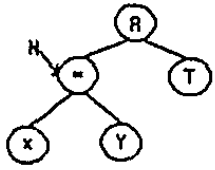

Figura 4.1

1.1 Si el nodo N representa una operación de asignación (ver figura 4.1), entonces, al crear el nuevo árbol, se harán las modificaciones mostradas en la siguiente figura sobre el árbol cuya raíz es R.

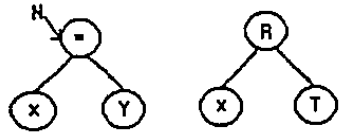

Figura 4.2

1.2 Si N no representa una operación de asignación, entonces será necesario crear un identificador temporal y modificar el árbol R como se muestra en la figura 4.3.

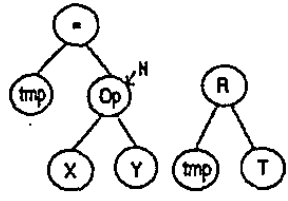

Figura 4.3

En ambos casos los árboles resultantes serán evaluados de izquierda a derecha. Para cada nodo creado en los pasos 1.1 o 1.2 se deberá calcular su matriz de costosasí como la matriz de costos para sus ancestros, utilizando el algoritmo descrito en la sección anterior.

Termina algoritmo.

Al crear nuevos árboles, es posible que el orden de evaluación resultante no sea válido. Por ejemplo en el árbol de la figura 4.4, el nodo A debe ser evaluado después del nodo D como lo indica la liga de restricción de orden colocada entre tales nodos.

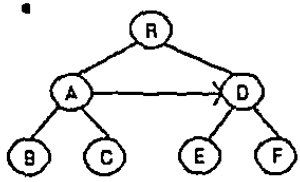

Figura 4.4

Si el nodo A se encuentra en la lista L, entonces, el árbol anterior será transformado en los dos árboles mostrados en la figura 4.5 (se ha supuesto que el nodo A representa una operación de asignación).

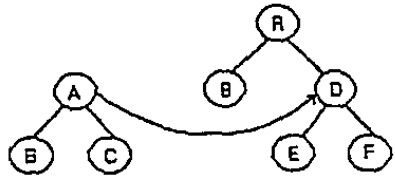

Figura 4.5

Obsérvese que si los árboles se evalúan de izquierda a derecha, entonces, se violaría la liga de restricción de orden de evaluación. Para resolver este problema, se decide crear un nuevo

árbol cuya raíz sea el nodo D y se evalúa tal árbol antes de considerar el árbol A como se muestra en la figura 4.6 (de nuevo se ha supuesto que D representa una operación de asignación, si no fuera así, se hubiera tenido que *crear* un nuevo identificador temporal).

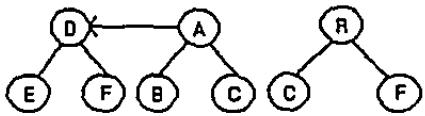

Figura 4.6

4.3.3 Manejo de subexpresiones comunes

El algoritmo de generación de código propuesto toma en cuenta la presencia de subexpresiones comunes en los bloques básicos para tratar de emitir código lo más eficiente posible. La estrategia que se utiliza consiste en tratar de mantener el valor de una subexpresión común en registro mientras haya instrucciones que utilicen tal valor.

Considérese la figura 4.7, en la cual el nodo E representa un valor que es utilizado dos veces; se intentará guardar el valor representado por el nodo E en uno de los registros del procesador, de forma tal que los nodos R y B tomen de ese registro el valor de su primer operando.

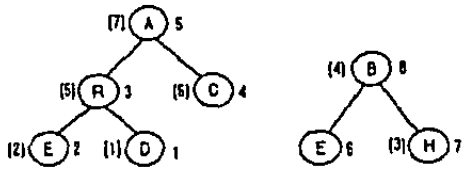

Figura 4.7

Hay dos formas posibles para lograr que el valor de la expresión común sea tomada del mismo registro en los dos árboles:

A. Que el nodo E del primer árbol se evalúe en el registro r y no se permita que tal registro sea alterado hasta generar código para el nodo B.

B. Que el nodo E del primer árbol se evalúe en un registro y después se altere el orden de evaluación de los nodos, emitiendo código para los nodos H y B antes de generar código para el resto de los nodos del primer árbol. Esta posibilidad será considerada solamente si las siguientes condiciones se cumplen:

- el nodo B es la raíz del segundo árbol
- los nodos que representan los usos de E pertenecen a diferentes árboles
- las ligas de restricción de orden no son violadas.

Supóngase que los números que aparecen a la derecha de los nodos representan el orden de evaluación de los mismos; si se utiliza la primera estrategia, la evaluación de los nodos R, C, A y H debe proceder con un registro libre menos (puesto que el registro r, donde queda almacenado el valor del nodo E no debe ser alterado). En cambio, si se utiliza la segunda estrategia (el orden de evaluación está dado en este caso por los números entre paréntesis) el registro r solo es bloqueado durante la evaluación de los nodos H y B, puesto que después de evaluar el nodo E del primer árbol se evalúan los nodos H, B y R (el nodo B hace uso del registro r, pero no puede alterar su valor). Nótese que para aplicar la segunda estrategia, se pide que el nodo B sea raíz, puesto que de esta forma la evaluación del nodo B no deja ningún registro ocupado y al generar código para los nodos del primer árbol que faltaban: R , C y A se tendrán todos los registros disponibles.

Cuando ambas estrategias sean aplicables, se preferirá utilizar aquella con la cual el registro r sea bloqueado durante la evaluación del menor número de nodos.

Enseguida se presenta el algoritmo utilizado para decidir si alguna de las estrategias anteriores puede ser aplicada para mejorar la calidad del código obtenido por el segundo paso del algoritmo de generación de código.

Algoritmo subexpresiones_comunes

Se buscan las parejas de nodos (n_i,n_j) que representen usos consecutivos de una subexpresión común (para descubrir estos nodos se utilizan las ligas de siguiente uso creadas al estar formando el *dag*). Se ordenan las parejas según la diferencia entre el orden de evaluación de los nodos n_i y n_j, de forma tal que primero aparezcan aquellas parejas en las cuales los nodos se evalúan muy cerca uno del otro.

Se consideran las parejas en el orden anterior y para cada una de ellas se realiza lo siguiente:

1. Supóngase que en la pareja (n_i,n_j) se evalúa primero el nodo n_i. Si alguna de las siguientes condiciones se cumple:

   - el padre del nodo n_j no es un nodo raíz, o bien
   - los nodos n_i y n_j pertenecen al mismo árbol, o bien
   - existe una liga de restricción de orden de evaluación indicando que alguno de los nodos del primer árbol que se evalúan después de n_i debe evaluarse antes de un nodo del segundo árbol.

   entonces se debe saltar al paso 2.

   Si ninguna de las condiciones se comple, entonces, se procede en la siguiente forma: sea n_1 el número de nodos que son evaluados entre el nodo n_i y el padre de n_j (este valor representa el número de nodos cuya evaluación debe proceder con un registro menos si se aplica la estrategia A mencionada al principio de esta sección) y sea n_2 el número de nodos que se evalúan entre n_i y su padre más el número de nodos del árbol donde se encuentra el nodo n_j (n_2 representa el número de nodos que deben ser evaluados con un registro menos si se utiliza la estrategia B). Si n_1 es menor a n_2, entonces, se salta al paso 2, en otro caso, se salta al paso 3.
2. En este paso se considera la posibilidad de evaluar el nodo n_i en un registro y no alterar el valor de tal registro hasta que sea utilizado por el padre de n_j. Para lograr que el valor del nodo n_i quede en un registro r, será necesario modificar las matrices de costos de los nodos que se evalúan entre n_i y n_j en la siguiente forma:

   2.1 Para asegurar que el nodo n_i quede evaluado en registro, en todas las listas L_M asociadas a su matriz de costos se da al campo de costo el valor infinito.

   2.2 La matriz de costos del padre del nodo n_i se recalcula aplicando el primer paso del algoritmo de generación de código, pero considerando solamente aquellas entradas de la tabla de instrucciones, que tomen de registro el valor del operando correspondiente al nodo n_i; de tales entradas se distinguen dos grupos:

   - Las entradas de la tabla de instrucciones en las cuales el resultado no se deja en el registro ocupado por el valor de n_i.
   - Las entradas en las que el código asociado modifica el registro r donde se almacenó el valor del nodo n_i. Para que el valor de n_i permanezca en registro, será necesario emitir una instrucción para copiar el valor de r en un registro diferente y al costo encontrado por el primer paso del algoritmo de generación

de código se debe sumar el costo de la instrucción de movimiento entre registros.

2.3 Se recalculan las matrices de costos de los ancestros del padre de n_i.

2.4 Si el registro r donde queda el valor del nodo n_i después de evaluar el nodo R es un registro de la clase h, entonces, para los nodos que se evalúan entre el padre de n_i y el nodo n_j se hace la siguiente modificación a sus matrices de costos:

- En las listas L_M y L_R asociadas a la entrada $C[i_1,i_2,...,n_h,...i_k]$ de cada nodo, tales que $0 \leq i_j \leq n_j$ (siendo n_j el número de registros de la clase j del procesador), al campo de costo se le da el valor de infinito; de esta forma, se asegura que al evaluar estos nodos no se utilizarán todos los registros de la clase h del procesador (si se intentara utilizar todos los registros de la clase h, el costo que se obtendría sería infinito) y por lo tanto habrá un registro de la clase h que no sería alterado por el código emitido para estos nodos.
- Si para algún nodo N, en la lista L_M de la entrada $C[i_1,i_2,...,i_k]$ de su matriz de costos, la instrucción asociada es $M_j \leftarrow M_N$, entonces deberá recalcularse tal entrada de la matriz utilizando el primer paso del algoritmo de generación de código, pero haciendo antes $n_h = n_h-1$, donde n_h es el número de registros de la clase h del procesador.

2.5 En la entrada de la lista L_R correspondiente a la clase de registros h de la matriz de costos del nodo n_j, el campo de costo se pone a cero y se elimina el código asociado. Después de esto se deben recalcular las restantes entradas de la matriz de costos de n_j utilizando el primer paso del algoritmo de generación de código.

2.6. Se calcula la matriz de costos del padre del nodo n_j, pero considerando solamente aquellas entradas de la tabla de instrucciones que tomen de registro el operando correspondiente a n_j.

2.7 Se vuelve a calcular la matriz de costos de cada uno de los nodos ancestros del padre de n_j.

2.8 Se consideran los árboles situados entre el árbol al que pertenece el nodo n_i y el árbol del que forma parte n_j; si la suma de los costos obtenidos en tales árboles mediante la aplicación de los pasos 2.1 al 2.7 es menor que el obtenido por el segundo paso del algoritmo de generación de código, entonces, el nuevo código es preferido al anterior (el costo del código elegido para un árbol se obtiene buscando el elemento con el costo más pequeño en las listas L_M y L_R asociadas a la entrada

$C[n_1,n_2,...,n_k]$ de la matriz de costos del nodo raíz); en otro caso se restauran las matrices de costos que fueron alteradas.

Si el nuevo código es preferido, entonces se debe repetir el segundo paso del algoritmo de generación de código sobre todos aquellos árboles cuyos nodos sufrieron algún cambio en sus matrices de costos.

2.9 Se toma la siguiente pareja de nodos que representen usos consecutivos de una subexpresión común y se salta al paso 1.

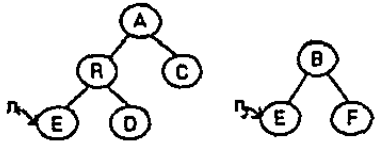

Figura 4.8

3. En este paso se considera la posibilidad de evaluar el nodo n_i en un registro r y alterar el orden de evaluación de los nodos, de forma tal (ver figura 4.8) que se emita código para los subárboles R y B antes de generar código para los nodos A y C; lo que se intenta con esto es que el registro r sea bloqueado durante el menor tiempo posible. Se considerarán dos posibilades:

   i. Evaluar el nodo n_i y, después, saltar a evaluar los nodos F y B, para posteriormente regresar a evaluar el resto de los nodos del primer árbol.
   ii. Evaluar el nodo R dejando el valor del nodo E en un registro y, después, saltar a evaluar los nodos F y B antes de completar la evaluación del primer árbol.

En cada caso se determina el costo del código resultante y se elige de las dos alternativas la mejor. El costo que sería obtenido con la estrategia (i) se calcula en el paso 3.1 y el que se obtendría con la segunda estrategia se calcula en el paso 3.2:

3.1 Para asegurar que el nodo n_i quede evaluado en un registro, en todas las listas L_M asociadas a su matriz de costos se da al campo de costo el valor infinito. Como resultado de esta modificación será necesario recalcular las matrices de costos de los ancestros del nodo n_i.

Después de evaluar el nodo n_i en el registro r, se salta a evaluar el segundo árbol; será necesario modificar las matrices de costos del nodo apuntado por n_j y del nodo B de la siguiente manera:

3.1.1 Si el registro r es de la clase h, entonces, en la entrada correspondiente a la clase de registro h de las listas L_R asociadas a la matriz del nodo n_j, el campo de costo se pone a cero y se elimina el código asociado a cada entrada; se recalculan las restantes entradas de la matriz utilizando el algoritmo costos.

3.1.2 Se vuelven a calcular las entradas de la matriz de costos del nodo B utilizando el primer paso del algoritmo de generación de código, pero considerando solamente aquellas entradas de la tabla de instrucciones que toman de registro el operando correspondiente al nodo n_j. Estas entradas se dividen en dos grupos:

- Aquellas en las que el código asociado no deja el resultado de la operación en el registro r ocupado por el valor de la expresión común E.
- Las entradas en las cuales el código asociado altera el valor almacenado en el registro r. En este caso, además del código indicado en la tabla de instrucciones será necesario emitir una instrucción para copiar el valor del registro r en un registro diferente (esto es, porque al regresar a evaluar el primer árbol, se desea que el valor del nodo n_i esté almacenado en alguno de los registros del procesador); por esta razón, al costo obtenido por el primer paso de algoritmo de generación de código debe sumarse el costo de la instrucción de movimiento entre registros.

3.1.3 Se suma el costo obtenido para los árboles de los que forman parte los nodos n_i y n_j; enseguida se restauran las matrices de costos que fueron alteradas.

3.2 En las listas L_M del nodo n_i el costo se hace infinito. Después, se recalcula la matriz de costos del nodo R, pero considerando solamente aquellas entradas de la tabla de instrucciones en las cuales el operando correspondiente al nodo n_i sea tomado de registro. Tales entadas se dividen en dos grupos y para cada uno se procede como se hizo en el paso 3.1.2.

Las instrucciones elegidas en el nodo R dejan el valor del nodo n_i en algún registro r del procesador. Como resultado de la modificación de la matriz de costos de R será necesario recalcular las matrices de costos de sus ancestros.

Antes de evaluar el nodo A, se salta al segundo árbol y se elige el código para el nodo B, aprovechando que el valor de E ya está guardado en registro:

3.2.1 Si el registro r donde está almacenado el valor de E es de la clase h, entonces, en las entradas de las listas L_R de la matriz de costos del nodo n_j correspondientes a tal clase, el campo de costo se hace igual a cero y se elimina el código asociado.

3.2.2 Se recalcula la matriz de costos del nodo B considerando solamente aquellas entradas de la tabla de instrucciones en las que el operando correspondiente al nodo n_j sea tomado de un registro.

3.2.3 Se suma el costo obtenido para los árboles de los que forman parte los nodos n_i y n_j; este valor es comparado contra el costo obtenido en los pasos 2, 3.2 y contra el obtenido en el segundo paso del algoritmo de generación de código. De los tres se elige el más pequeño y en las matrices de costos se coloca la información que indica como fue obtenido tal costo.

Si el código elegido es el que se obtuvo en el paso 3.2 o en el paso 3.3, entonces debe repetirse el segundo paso del algoritmo de generación de código sobre todos los árboles cuyos nodos hayan sufrido alteración en su matriz de costos.

3.3 Se toma la siguiente pareja de nodos que represente usos consecutivos de una subexpresión común y se salta al paso 1.

Termina algoritmo.

4.3.4 Asignación de registros

Una vez elegidas las instrucciones que serán emitidas para cada uno de los nodos, lo único que resta antes de poder escribir el código objeto en el archivo de salida consiste en decidir que registros serán utilizados en cada instrucción.

Para facilitar la tarea de asignar registros, se utiliza un descriptor de registros que indica en cada momento que registros están libres -y por lo tanto pueden ser asignados para realizar alguna instrucción- y cuales almacenan un valor que se utilizará más adelante -y por consiguiente no deben ser modificados-. Inicialmente en el descriptor todos los registros se marcan como libres.

El siguiente algoritmo se encarga de seleccionar los registros que se utilizarán en las instrucciones elegidas para los nodos de un árbol.

Algoritmo asignación_registros (raíz)

1. Si el nodo apuntado por raíz es una hoja, entonces puede representar el valor de una variable, el valor de una constante o bien un operador sin operandos.

   1.1 Si el nodo representa el valor de una variable o el de una constante, entonces se examina la lista I asociada al nodo para determinar las tablas que deben ser consultadas a fin de obtener las instrucciones de máquina elegidas para el nodo durante el segundo paso del algoritmo de generación de código.

   Si el valor del nodo se debe tomar de la memoria, se consulta la tabla de modos de direccionamiento para saber cuáles registros son requeridos para formar la dirección de la celda de memoria ocupada por el valor del nodo según el modo de direccionamiento elegido. Si, por ejemplo, se necesita un registro de la clase h, primero se revisa la tabla de clases de registros para saber que registros pertenecen a tal clase y, después, se consulta el descriptor de registros para elegir el primer registro de la clase h que esté libre; todos los registros utilizados para formar la dirección se deben marcar como ocupados en el descriptor de registros.

   Por otra parte, si el valor del nodo se debe almacenar en un registro, entonces hay dos posibilidades:

   - Que el nodo represente el valor de una subexpresión común y, además, que ya se haya decidido en que registro debe almacenarse su valor. En este caso no debe hacerse nada.
   - Que aún no se haya decidido en cuál registro se almacenará el valor del nodo, en este caso, después de asignar los registros utilizados por el modo de direccionamiento elegido para tomar de la memoria el valor del mismo, se debe consultar la tabla de conversión (la posición que debe consultarse está indicada en la lista I) para saber cuál es la clase de registro en la que debe quedar almacenado el valor del nodo. Conocida la clase, se elige de la misma un registro r que esté libre; este registro se marca como ocupado, mientras que los registros utilizados para formar la dirección se marcan como libres.

     Si el nodo representa el valor de una subexpresión común que decidió dejarse en registro entre usos consecutivos, entonces, siguiendo la liga de siguiente uso se busca el nodo más cercano que representa el valor de la misma subexpresión común y en tal nodo se indica que debe utilizarse el registro r para almacenar su valor.

1.2 Si el nodo representa un operador sin operandos, la lista I asociada indicará que se debe consultar la tabla de instrucciones para determinar si el código elegido para el nodo necesita utilizar algún registro del procesador. Si un registro de cierta clase es requerido, entonces, se consulta la tabla de clases de registros y el *descriptor* de registros para elegir un registro de la clase pedida que esté libre.

2. Si el nodo apuntado por raíz es un nodo interno, entonces se procede en la siguiente forma:

   2.1 Se se hace la llamada recursiva: asignación_registros (h_1), donde h_1 es un apuntador al hijo que debe evaluarse primero. Si el nodo tiene un solo hijo, entonces h_1 apunta a ese hijo.

   2.2 Si el nodo apuntado por raíz tiene dos hijos, entonces se hace la llamada recursiva asignacion_registros (h_2), donde h_2 apunta al hijo que debe evaluarse en segundo lugar.

   2.3 Parte del código elegido para un nodo interno está formado por instrucciones que deben ser tomadas de la tabla de instrucciones; se revisa la lista I asociada al nodo para saber que entrada de la tabla debe ser consultada para obtener tales instrucciones.

      2.3.1 Si la tabla de instrucciones indica que el valor del nodo debe ser tomado de la memoria, entonces la dirección que ocupará el resultado debe ser igual a la dirección ocupada por alguno de los operandos. Si, por ejemplo, el resultado debe dejarse en la misma dirección donde está guardado el primer operando, entonces, del hijo que corresponda a tal operando se copia lo siguiente:

         - los registros utilizados por el modo de direccionamiento
         - la información adicional requerida para formar la dirección del operando y del resultado. Por ejemplo, si el hijo correspondiente al primero operando es una variable local, entonces,es necesario conocer la posición (*offset*) que esa variable ocupa dentro del registro de activación.

      2.3.2 Si el resultado debe dejarse en registro, se consideran dos posibilidades:

         - Que el resultado deba dejarse en el registro ocupado por alguno de los operandos; en este caso, solamente se copia del nodo hijo correspondiente a tal operando el registro que debe ser utilizado.

- Que el resultado deba dejarse en un registro de cierta clase no ocupado por los operandos; en este caso, utilizando la tabla de clases de registros y el descriptor de registros se elige el primer registro libre de la clase requerida; tal registro se debe marcar como ocupado.

Si el nodo representa el valor de una subexpresión común que decidió dejarse en registro entre usos consecutivos, entonces, siguiendo la liga de siguiente uso se busca el nodo más cercano que representa el valor de la misma subexpresión común. En tal nodo se indica que se utilizará el mismo registro empleado para almacenar el valor del nodo apuntado por raíz.

2.3.3 Se consulta la tabla de instrucciones para determina si algún registro adicional es requerido para poder ejecutar el código elegido, si es así, se asigna un registro libre de la clase requerida y se marca como ocupado en el descriptor de registros.

2.3.4 Se marcan como libres los registros ocupados por los operandos, salvo aquellos registros que sean utilizados también para almacenar el valor del resultado.

2.4 En la lista I asociada al nodo apuntado por raíz, puede haber otras instrucciones además de aquellas tomadas de la tabla de instrucciones, para cada una de ellas se procede en la siguiente forma:

2.4.1 Si el valor del resultado está almacenado en el registro r y en la lista I hay una instrucción de movimiento a memoria ($R_j \rightarrow M_i$), se debe consultar la tabla de conversión en la posición indicada por la lista I a fin de averiguar el modo de direccionamiento que se utilizará para almacenar el valor del resultado.

Después, de la tabla de modos de direccionamiento se determina si es necesario algún registro para formar la dirección según el modo de direccionamiento elegido y si es así, se escoge un registro libre de la clase requerida y se marca como ocupado en el descriptor de registros. El registro r debe ser marcado como libre.

2.4.2 Si en la lista I hay una instrucción de carga a registro ($M_j \rightarrow R_i$), la tabla de conversión debe ser revisada para saber en que clase de registro quedará el valor del resultado; se elige un registro de tal clase y se marca como ocupado. Los registros utilizados para formar la dirección de la celda de la memoria que

ocupa el resultado *antes* de ejecutar esta instrucción son marcados como libres.

2.4.3 Si el resultado está *guardado* en un registro y en la lista I hay una instrucción para pasar el valor del resultado a un registro de una clase diferente, entonces debe consultarse la tabla de conversión entre clases de registros en la posición indicada por la lista I para conocer la clase de registro C donde el resultado será almacenado. Se marca como libre el registro donde está guardado el valor del resultado y, *después*, se elige un registro de la clase C, marcándolo como ocupado en el descriptor de registros.

Termina algoritmo.

4.3.5 Emisión de código

El último paso del algoritmo de generación de código consiste en escribir en el archivo objeto el código elegido en cada nodo. Se consideran los nodos en el orden escogido durante el segundo y tercer pasos del algoritmo y para cada uno de ellos se escriben en el archivo de salida las instrucciones indicadas en su lista I. Para escribir correctamente el código será necesario reemplazar los nombres de los registros que fueron elegidos durante la selección de registros en ciertas partes de las instrucciones, así como buscar en la tabla de símbolos información adicional para algunos nodos, tal como la posición *(offset)* que ocupan las variables locales dentro del registro de activación de una función.

En el apéndice A se describe el lenguaje utilizado para indicar en las tablas de especificación del procesador cuál es el código en lenguaje ensamblador asociado a cada entrada; tal lenguaje permite indicar donde se debe reemplazar: el nombre de un registro, el *offset* de una variable o alguna otra información antes de escribir el código en el archivo de salida.

4.4 Eficiencia del algoritmo de generación de código

La complejidad temporal del algoritmo de generación de código es $O(N*n_1*n_2*...*n_k*m*r)$, donde N es el número de nodos de los árboles, n_i es el número de registros de la clase i del procesador, m es el número máximo de entradas en la tabla de instrucciones para un mismo operador y r es el máximo entre el número de clases de registros y el número de modos de direccionamiento de la memoria. La complejidad espacial del algoritmo es $O(N*r*n_1*n_2*...*n_k)$.

Aparentemente el algoritmo es demasiado costoso tanto en espacio como en tiempo para poder utilizarlo en la construcción de la parte posterior de un compilador comercial, sin embargo, se pueden hacer las siguientes modificaciones en el algoritmo para reducir sus requerimientos de memoria y de tiempo.

1. En primer lugar, se pueden eliminar las listas L_R y L_M asociadas a cada entrada $C[i_1,i_2,...,i_k]$ de la matriz de costos de cada nodo. En lugar de ellas, se asocia a cada entrada de la matriz una lista que tiene inicialmente dos elementos, en el primer elemento se almacena el costo de evaluar el nodo en registro y en el seguno se coloca el costo de evaluar el nodo en memoria; cada elemento tiene un conjunto de campos en los cuales se almacena, además del costo, la siguiente información:

   - la clase de registro donde se guarda el valor del nodo o el modo de direccionameinto empleado para formar la dirección de la celda de la memoria utilizada para almacenar el valor del mismo
   - el conjunto de instrucciones elegidas para evaluar el nodo
   - el orden de evaluación de los operandos
   - el número de registros de cada clase requeridos para almacenar el valor del primer operando (esta información es almacenada solamente si el nodo representa un operador binario).

   El algoritmo costos sufre las siguientes modificaciones debido a la eliminación de las listas L_R y L_M:

   i. Para llenar la matriz de costos de un nodo que represente el valor de una variable o de una constante, primero se busca en la tabla de tipos de datos el modo o los modos de direccionamiento que se pueden utilizar para el tipo de variable o constante representado por el nodo. Si hay más de un modo de direccionamiento aplicable, entonces, a la lista asociada a cada entrada $C[i_1,i_2,...,i_k]$ de la matriz de costos se le deben agregar algunos elementos adicionales (un elemento por cada modo de direccionamiento extra aplicable). Luego, se calcula el costo de evaluar el nodo en cada uno de los modos de direccionamiento encontrados como se indica en al algoritmo costos (ver sección 4.3.1).

      Se busca en la tabla de conversión el código que debe ser emitido para almacenar el valor representado por el nodo en un registro. Normalmente el código es el mismo para cualquier clase de registro donde se desee almacenar el valor, es por ello que en el elemento de la lista asociada a la entrada $C[i_1,i_2,...,i_k]$ donde se indica el costo de evaluar el nodo en registro, se coloca una indicación de que el código es válido para todas las clases de registros del procesador.

   ii. Para llenar la matriz de costos de un nodo n que represente un operador op, se buscan en la tabla de instrucciones las entradas correspondientes a tal operador y

para cada una de ellas se procede como se indica en el algoritmo costos, con la siguiente diferencia: si en una entrada de la tabla de instrucciones se indica, por ejemplo, que alguno de los operandos debe estar almacenado en la clase de registro h y al consultar la matriz de costos del hijo correspondiente a tal operando se observa que su valor se guardó en una clase diferente, entonces será necesario consultar la tabla de conversión entre clases de registros para determinar el código que se debe emitir para colocar el valor del operando en la clase de registro requerida. Este código es agregado al nodo n y el costo del mismo se suma al costo del código obtenido por el algoritmo costos.

El paso 3.9 del algoritmo costos no se debe realizar; en los pasos 3.10 y 3.11 sólo se consideran los modos de direccionamiento y clases de registros que están indicados en las listas asociada a cada entrada $C[i_1,i_2,...,i_k]$ de la matriz de costos.

2. Al probar el algoritmo de generación de código, se ha observado que normalmente en cada nodo, los elementos de la matriz de costos son idénticos, es decir, que las listas asociadas a las entradas $C[i_1,i_2,...,i_k]$ son iguales.

   Para reducir el desperdicio de memoria, se decidió que cada entrada $C[i_1,i_2,...,i_k]$ fuera un apuntador a una lista, de forma tal que si en todas las entradas de una matriz se debe almacenar la misma lista, será necesario reservar espacio una sola vez para dicha lista y se hace que todas las entradas de la matriz apunten a ella.
3. Observando la forma en la que se realiza la selección de instrucciones, se notará que en el primer paso del algoritmo de generación de código no es necesario llenar toda la matriz de costos de un nodo.

   Para elegir las instrucciones en el nodo raíz, solamente se consulta la entrada $C[n_1,n_2,...,n_k]$ de su matriz de costos (siendo n_i el número de registros de la clase i), por consiguiente el algoritmo costos, sólo debería llenar esa entrada de la matriz. Para el primer hijo de la raíz hay $n_1,n_2,...,n_k$ registros disponibles, por esta razón, en tal nodo se consultará solamente la entrada $C[n_1,n_2,...n_k]$ de su matriz; para el segundo hijo hay $n_1-r_1,n_2-r_2,...,n_k-r_k$ registros disponibles (r_i es el número de registros de la clase i requeridos para almacenar el valor del primer operando), por eso sólo se examinará la entrada $C[n_1-r_1,n_2-r_2,...,n_k-r_k]$ de la matriz de costos de tal hijo.

   Si $u_1,u_2,...,u_k$ es el máximo número de registros de cada clase requeridos para almacenar el valor de un operando, para los hijos de la raíz solamente será necesario llenar las entradas $C[i_1,i_2,...,i_k]$ tales que: $n_j-u_j \leq i_j \leq n_j$, para todos los valores de j: $1 \leq j \leq k$.

   De la misma forma para los nodos situados en el nivel m (considerando que la raíz está colocada en el nivel cero), solamente se calculan las entradas de la matriz de costos

$C[i_1,i_2,...,i_k]$ tales que: máximo$\{0,n_j-m*u_j\}$ <= i_j <= n_j, para todos los valores de j: $1 \leq j \leq k$.

4. Es posible reducir el número de dimensiones de la matriz de costos almacenando en cada entrada de la matriz el número de registros de cierta clase utilizados por el código elegido para evaluar el nodo, de esta forma no será necesario reservar una dimensión en la matriz de costos para tal clase de registro. Por ejemplo, si un procesador tiene muchos registros de la clase h, de forma tal que sea poco probable que el código elegido para algún árbol requiera todos esos registros, entonces no tiene caso reservar en la matriz de costos una dimensión para tal clase de registro (la razón por la cual el algoritmo utiliza una matriz, es para asegurarse de que el código elegido nunca intente utilizar más registros que los disponibles en el procesador).

Si se decide guaradar en cada entrada de la matriz el número N_h de registros de la clase h utilizados por el código, entonces el algoritmo costos debe ser modificado en la siguiente forma:

i. Para un nodo hoja que representa el valor de una constante o de una variable, se buscan en la tabla de tipos de datos los modos de direccionamiento aplicables al valor representado por el nodo; para cada modo se consulta la tabla de modos de direccionamiento, determinándose de ella el número N_h.

Se consulta después la tabla de conversión para conocer el código que debe ser emitido para pasar el valor representado por el nodo a un registro. Si la conversión requiere utilizar m registros de la clase h, en la matriz de costos se indica que para tener el valor del nodo en registro, el número total de registros de la clase h es $m + N_h$.

ii. Para un nodo que represente un operador, el algoritmo costos debe consultar la tabla de instrucciones y las matrices de costos de los hijos del nodo para determinar:

- N_1, el número de registros de la clase h utilizados para evaluar el primer operando,
- N_2, el número de registros de la clase h utilizados para evaluar el segundo operando
- N_3, el número de registros de la clase h utilizados para almacenar el valor del primer operando (si el primer operando queda en registro) o bien para formar la dirección de la celda de la memoria donde se guarda el valor del primer operando (si el primer operando se deja en la memoria)

- N_4, el número de registros de la clase h utilizados para almacenar el valor del segundo operando (si el segundo operando se deja en un registro) o bien para formar la dirección de la celda de la memoria donde se guarda el valor del segundo operando (si el segundo operando se deja en la memoria)
- N_5, el número de registros adicionales de la clase h requeridos para poder ejecutar la operación indicada en el nodo

con tal información se puede calcular el número de registros de la clase h requeridos por el nodo en la siguiente forma:

$$N_h = \text{Máximo } \{N_1, N_2 + N_3, N_3 + N_4 + N_5\}$$

Si N_h es mayor al número de registros de la clase h del procesador, entonces, debe hacerse lo siguiente:

- Si $N_h = N_2 + N_3$, entonces, antes del código para evaluar el segundo operando, se emiten instrucciones para guardar en la memoria los valores almacenados en algunos de los registros de la clase h utilizados por el primer operando. Después del código requerido para evaluar el segundo operando, se emiten instrucciones para restaurar los valores de los registros que se almacenaron en la memoria.
- Si $N_h = N_3 + N_4 + N_5$, entonces, será necesario pasar el valor del primer operando o el valor del segundo operando a un registro que no sea de la clase h.

Haciendo las modificaciones anteriores sobre el algoritmo, la complejidad temporal del algoritmo resultante es $O(N^*n_1^*n_2^*...^*n_j^*m)$, donde j es el número de dimensiones de las matrices de costos que resultan después de hacer la consideración 4. Puesto que no es necesario llenar todas las entradas de la matriz de costos de cada nodo (ver consideración 3) el algoritmo modificado es muy eficiente en tiempo.

La complejidad espacial del algoritmo es $O(N^*n_1^*n_2^*...^*n_j)$. En cada entrada de la matriz, sin embargo, sólo es necesario reservar espacio suficiente para almacenar un apuntador.

En el siguiente capítulo se reportan las pruebas realizadas con el generador de código utilizando el algoritmo modificado; en las tablas de resultados se incluye el tiempo requerido por el programa para generar código para diversos programas de prueba, los resultados muestran que el algoritmo es de velocidad comparable a la de los generadores de código utilizados en los compiladores comerciales.

RESULTADOS Y CONCLUSIONES

5.1 Resultados

El generador de código construido se utilizó para emitir código en lenguaje ensamblador de dos procesadores diferentes: el 8086 de Intel y el 68000 de Motorola. Para comparar el código obtenido con el generador de código contra el producido por compiladores comerciales del lenguaje C para los procesadores anteriores, se midió el tiempo de ejecución del código generado por cada uno de ellos para los siguientes programas de prueba:

1. Un programa recursivo que realiza búsqueda binaria sobre un arreglo.
2. Un programa recursivo que encuentra los términos de la serie de Fibonacci.
3. Un programa iterativo para encontrar números primos utilizando el método de la criba de Eratóstenes.
4. Un programa iterativo para multiplicar dos matrices.
5. Un programa recursivo para ordenar los elementos de un vector utilizando el algoritmo de *mergesort*.
6. Un programa recursivo para ordenar los elementos de un vector utilizando el algoritmo de *quicksort*.

Estos programas de prueba permiten tener una estimación de la calidad del código producido por el generador de código para: ciclos, llamadas a funciones y bloques de instrucciones con subexpresiones comunes.

El generador de código desarrollado produce como salida un programa en lenguaje ensamblador, que para poder ser ejecutado necesita primero ensamblarse y ligarse. El código emitido por el generador de código para el procesador 8086 de Intel fue ensamblado con el macroensamblador de Microsoft versión 5.0 y ligado con el ligador de Microsoft versión 3.6. El

código obtenido para el procesador 68000 de Motorola fue ensamblado y ligado con las utilierías que para este fin provee el sistema operativo UNIX, versión V.3.0L para AT&T.

Las mediciones de los tiempos de ejecución del código producido por los compiladores comerciales y por el generador de código para el procesador 8086 de Intel fueron realizadas en una PC/XT BPM y las mediciones de los tiempos de ejecución para el código del procesador 68000 de Motorola se realizaron en una PC/AT&T UNIX.

Las tablas 5.1 y 5.2 reportan los tiempos de ejecución medidos; entre paréntesis, se indica el cociente que resulta al dividir el tiempo de ejecución del código obtenido por el generador de código entre el tiempo de ejecución del código producido por cada compilador. En la columna cuyo encabezado es Gencod se reportan los tiempos de ejecución del código producido por el generador de código.

	Gencod	Compilador de Microsoft (4.0)	Compilador de Microsoft (5.0)	TurboC (2.0)
Fibonacci	909	1483 (0.61)	1406 (0.65)	908 (1.00)
Primos	154	154 (1.00)	171 (0.90)	154 (1.00)
Búsqueda binaria	17	19 (0.89)	22 (0.77)	17 (1.00)
Multiplicación de matrices	162	167 (0.97)	280 (0.58)	165 (0.98)
Mergesort	65	77 (0.84)	83 (0.78)	66 (0.98)
Quicksort	50	57 (0.88)	236 (0.21)	50 (1.00)

Tabla 5.1 Tiempo de ejecución del código producido para el procesador 8086 de Intel. El tiempo se da en centésimas de segundo.

	Gencod	CC (compilador que proporciona el sistema operativo UNIX, versión V.03L)
Fibonacci	17.616	21.399 (0.82)
Primos	3.750	4.067 (0.92)
Multiplicación de matrices	7.733	9.083 (0.85)
Mergesort	2.300	2.517 (0.91)
Búsqueda binaria	0.500	0.550 (0.91)

Tabla 5.2 Tiempo de ejecución del código producido para el procesador 68000 de Motorola. El tiempo se da en segundos.

Algunos de los compiladores comerciales utilizados para las pruebas tienen varias opciones de optimización de código. Al compilar los programas de prueba se intentó utilizar aquellas opciones que representan optimizaciones parecidas a las realizadas por el generador de código, es decir, sólo optimizaciones realizadas con las instrucciones de un mismo bloque básico.

Además de estimar la calidad del código producido por el generador de código, también se comparó su velocidad de compilación contra la de los compiladores comerciales. Debido a que los programas de prueba elegidos para medir el tiempo de ejecución del código son muy pequeños (menos de 100 líneas cada uno) y puesto que el tiempo de compilación reportado en la tabla 5.3 incluye el tiempo necesario para cargar el compilador o el generador de código en la memoria y el tiempo requerido para crear los archivos de salida, se decidió escribir dos programas de prueba más grandes y medir el tiempo de compilación para cada uno de ellos. El programa Sintac, que es una parte del analizador sintáctico utilizado en la parte frontal desarrollada para poder probar el generador de código, tiene 1326 líneas y el programa Opt_cod, que incluye las funciones utilizadas por la parte frontal para optimizar el conjunto de cuádruples y dividirlo en bloques básicos, tiene 998 líneas.

Los resultados de estas mediciones se presentan en la tabla 5.3; entre paréntesis, se indica el cociente que resulta al dividir el tiempo de compilación del generador de código entre el tiempo de compilación de cada compilador comercial.

	Gencod	Compilador de Microsoft (4.0)	Compilador de Microsoft (5.0)	TurboC (2.0)
Fibonacci	440	1565 (0.28)	747 (0.59)	412 (1.07)
Multiplicación de matrices	879	1993 (0.44)	818 (1.07)	495 (1.78)
Búsqueda binaria	675	1966 (0.34)	818 (0.83)	456 (1.48)
Quicksort	841	2296 (0.37)	840 (1.00)	544 (1.55)
Mergesort	1021	2526 (0.40)	862 (1.18)	522 (1.96)
Primos	577	1697 (0.34)	785 (0.74)	434 (1.33)
Sintac	9273	11027 (0.84)	3056 (3.03)	2135 (4.34)
Opt_cod	7039	8572 (0.82)	2594 (2.71)	1749 (4.02)

Tabla 5.3. Tiempo de compilación en centésimas de segundo (el código producido es para el procesador 8086 de Intel).

Sólo se pudo medir el tiempo de compilación en la PC/XT, las pruebas no pudieron efectuarse en la PC/AT&T debido a que esta máquina tenía poco espacio disponible en disco, por lo cual el programa generador de código no pudo cargarse en ella (el programa tiene cerca de 30 mil líneas).

Debe hacerse notar que el tiempo de compilación reportado en la tabla anterior para los compiladores comerciales incluye tiempo de preproceso; la parte frontal desarrollada durante la tesis no tiene fase de preproceso, por lo tanto el tiempo de compilación indicado para Gencod no incluye este tiempo. Por otra parte, el analizador léxico y el analizador sintáctico utilizados en la parte frontal fueron creados utilizando un generador de analizadores léxicos (LEX) y un generador de analizadores sintácticos (YACC), por lo cual no son muy eficientes; es posible reescribir estas fases, así como algunas de las funciones utilizadas para realizar la generación de código para reducir el tiempo de compilación reportado para Gencod.

5.2 Conclusiones

En esta tesis se propone un algoritmo de generación de código basado en el algoritmo de Aho-Jonson [Aho76] capaz de producir código en lenguaje ensamblador para una clase amplia de procesadores. El algoritmo genera código de alta calidad, comparable al producido por compiladores comerciales, además de que la especificación de un procesador es bastante pequeña y simple de construir.

Para probar el algoritmo se escribió la parte frontal de un compilador del lenguaje C que produce como código intermedio, cuádruples. El conjunto de cuádruples es dividido en bloques básicos y se crea un *dag* para cada bloque.

Se propone un algoritmo para dividir los *dags* en árboles y para ordenar los árboles resultantes, de manera tal que el algoritmo de generación de código pueda emitir código eficiente para programas con subexpresiones comunes. Son necesarias más pruebas para determinar cuanto afecta el orden de evaluación de los árboles a la calidad del código resultante.

Las extensiones realizadas en el algoritmo de Aho-Johnson para poder manejar procesadores reales dieron lugar a un algoritmo de complejidades temporal y espacial lineales; las pruebas realizadas indican que el algoritmo es de velocidad comparable a la de los algoritmos de generación de código utilizados en los compiladores comerciales y, además, el código producido es tan bueno como el generado por estos (al realizar la implantación no se buscó la eficiencia, por lo cual hay muchos sitios en los cuales el programa puede modificarse para hacerlo más rápido).

Es de notarse la facilidad con la cual pudo cambiarse la especificación del procesador para que la herramienta generara código de un procesador diferente. La especificación del procesador 68000 de Motorola, por ejemplo, pudo escribirse en menos de dos días (en [LAND82] se reporta que fueron requeridos 3 meses hombre para escribir la especificación del procesador 68000 de Motorola para el sistema CGS diseñado por los autores). Además, las tablas de especificación de los procesadores son bastante pequeñas; para los procesadores elegidos la tabla más grande tiene menos de 80 entradas, lo cual hace relativamente simple verificar que la especificación de una máquina sea correcta (en [LAND82] se indica que una de las tablas utilizadas para describir el procesador 68000 de Motorola tiene cerca de 1300 líneas; en [GRAH84] se menciona que la descripción del procesador 68000 de Motorola tiene 525 producciones).

Por todo lo anteriormente indicado se piensa que el algoritmo de generación de código propuesto en el presente trabajo podría utilizarse para construir partes posteriores de compiladores comerciales.

Tablas de especificación del procesador

En este apéndice se presentan las tablas que contienen la descripción de los procesadores 8086 de Intel y 68000 de Motorola, dando antes una breve explicación del contenido de cada una de ellas.

A.1 Descripción de los campos de las tablas

1. Tabla de instrucciones. Cada entrada de la tabla tiene los siguientes *campos*:

i. Operación. Indica cuál es la operación de lenguaje intermedio. En el apéndice C se describen los operadores de representación intermedia.

ii. Primer operando. Indica de que tipo es el primer operando de la instrucción. Los valores que puede tomar este campo son los siguientes:

- Cualquiera de los modos de direccionamiento de la memoria.
- Cualquiera de las clases de registros del procesador.
- M_Q, este valor denota cualquier modo de direccionamiento de la memoria.
- R_Q, este valor denota cualquier registro del procesador.
- INMEDIATO, este valor señala que el operando es una constante.
- BANDERAS, con este valor se indica que el operando está implícitamente almacenado en los bits del registro de banderas del procesador.
- NULO, este valor señala que la instrucción no tiene operando.

iii. Segundo operando. Indica de que tipo es el segundo operando de la instrucción. Este campo puede tomar los mismos valores que el campo anterior.

iv. Resultado. En este campo se indica de que tipo es el resultado de la operación. Los valores que puede tomar este campo son los siguientes:

- Cualquiera de los modos de direccionamiento de memoria.
- Cualquiera de las clases de registros del procesador.
- M_I, este valor indica que el modo de direccionamiento del resultado es el mismo que el del hijo izquierdo
- M_D, señala que el modo de direccionamiento del resultado es el mismo que el utilizado en el hijo derecho.
- R_I, este valor indica que el resultado se deja en el registro ocupado por el hijo izquierdo.
- R_D, indica que el resultado se deja en el registo ocupado por el hijo derecho.
- BANDERAS, este valor señala que el resultado de la operación queda almacenado en el registro de banderas del procesador.
- NULO, indica que la operación no tiene resultado.

v. Registros adicionales. Además de los registros utilizados para formar la dirección de los operandos y del resultado o para almacenar los valores de los mismos, es posible que la ejecución del código de máquina requiera algunos registros adicionales. Si es así, en este campo se indica cuales son tales registros.

vi. Código. En este campo se guarda el código en lenguaje ensamblador equivalente a la operación de representación intermedia. El contenido de este campo será escrito en el archivo de salida durante la fase de emisión de código.

En algunas partes de una instrucción puede ser necesario colocar cierta información desconocida en el momento de llenar la tabla de instrucciones, tal como: el nombre del registro utilizado para almacenar el valor de alguno de los operandos, o bien la posición *(offset)* de una variable local dentro del registro de activación de una función.

La siguiente notación se emplea para saber que partes de la instrucción deben ser copiadas sin cambio en el archivo de salida y cuales deben ser reemplazadas por cierta información:

a. Lo que aparezca entre paréntesis angulares debe ser reemplazado por lo que se indica enseguida:

Se reemplaza por:

- $<M_1>$: dirección del primer operando.
- $<M_2>$: dirección del segundo operando.
- <L> : espacio en bytes ocupado por las variables locales de una función (el nombre de la función está indicado por el primer operando).
- $<C_1>$: valor del primer operando (el primer operando es una constante entera).
- $<C_2>$: valor del segundo operando.
- $<R_1>$: registro ocupado por el primer operando.
- $<R_2>$: registro ocupado por el segundo operando.
- $<D_1>$: registro utilizado para formar la dirección del primer operando.
- $<D_2>$: registro utilizado para formar la dirección del segundo operando.
- $<N_1>$: nombre del primer operando (el primer operando es una variable).
- $<N_2>$: nombre del segundo operando.
- $<O_1>$: *offset* del primer operando (el primer operando es una variable local).
- $<O_2>$: *offset* del segundo operando.
- $<E_1>$: etiqueta representada por el primer operando.

b. Lo que aparezca fuera de los paréntesis angulares debe ser copiado sin cambio en el archivo de salida.

vii. Costo. El valor que se almacena en este campo corresponde al costo del código. En las tablas que se presentan más adelante, se utilizó como costo el número de ciclos de reloj que requiere la ejecución del código.

viii. En este campo se almacena un conjunto de bits que guardan la siguiente información:

- Si el operador es o no conmutativo.
- Si hay varios modos de direccionamiento que pueden utilizarse para formar la dirección de la celda de la memoria donde se deja el resultado de la operación.
- Si alguna función de usuario debe ser invocada durante la fase de generación de código.
- Si hay ciertas condiciones que deben ser cumplidas para poder emitir el código indicado en una entrada de la tabla, se manejan dos tipos de condiciones:

    1. Que alguno de los operandostenga cierto valor (en este caso se debe indicar cuál es el valor que debe tener el operando)
    2. Que los operandos sean iguales.

ix. Este campo y los dos siguientes se utilizan solamente si el resultado de la operación se guarda en la memoria. En este campo se indica si el modo de direccionamiento utilizado por el resultado emplea o no los mismos registros usados para formar la dirección de la celda de la memoria donde está almacenado el valor de alguno de los operandos. Este campo puede tomar los siguientes valores:

- REGISTROS_DIRECCIÓN_IZQUIERDO, indica que para formar la dirección de la celda de la memoria donde se guarda el valor representado por el nodo se utilizan los mismos registros empleados para formar la dirección del hijo izquierdo.
- REGISTROS_DIRECCIÓN_DERECHO, indica que para formar la dirección de la celda de la memoria donde se guarda el valor representado por el nodo, se utilizan los mismos registros empleados para formar la dirección del hijo derecho.
- REGISTRO_DATO_IZQUIERDO, indica que para formar la dirección del resultado se utiliza el registro donde se almacenó el valor del primer operando.
- REGISTRO_DATO_DERECHO, indica que para formar la dirección del resultado se utiliza el registro donde se almacenó el valor del segundo operando.

x. En este campo se indica si el resultado tiene el mismo *offset* que alguno de los operandos.

xi. En este campo se indica si el resultado tiene asignada la misma celda de la memoria que alguno de los operandos (este campo es utilizado solamente si la celda reservada para el resultado no pertenece al registro de activación de una función).

Estos tres últimos campos tienen información redundante, se utilizan solamente para facilitar la escritura del código en el archivo objeto.

2. Tabla de clases de registros. Cada entrada de la tabla, que representa a una clase de registros del procesador, tiene los siguientes campos:

i. El número de registros que pertenecen a la clase.

ii. Una lista con los registros que pertecen a la clase.

iii. El tipo de registros que pertenecen a la clase. Este campo puede tomar dos valores:

- COMPUESTO, indica que los registros señalados en la lista se consideran como si fueran un solo registro.
- SIMPLE, indica que cada elemento de la lista representa un registro diferente.

3. Tabla de nombres de registros. En esta tabla se almacenan los nombres de los registros. A cada registro se le asigna un código numérico, tal código es igual a la posición que ocupa cada registro dentro de esta tabla.

4. Tabla de conversión entre clases de registros. Esta tabla se divide en dos partes; en la primera se indica el código que debe emitirse para pasar un valor almacenado en una clase de registros SIMPLE a otra clase de registros SIMPLE y en la segunda parte se indica el código para convertir una clase de registros COMPUESTO a una clase SIMPLE y viceversa.

Cada entrada de la tabla correspondiente a la parte de conversión entre las clases de registro SIMPLE tiene los siguientes campos:

i. Clase de registro fuente.
ii. Clase de registro destino. Se desea pasar un valor almacenado en la clase fuente a la clase destino.
iii. Costo. En este campo se especifica el costo del código necesario para hacer la conversión.
iv. Código. Para indicar el código se utiliza la misma notación empleada en la tabla de instrucciones, con las siguientes diferencias:

- $<R_1>$ representa el registro fuente y
- $<R_2>$ representa el registro destino.

Para la conversión entre las clases de registro COMPUESTO y SIMPLE, además de los cuatro campos anteriores cada entrada tiene un campo adicional.

v. Al hacer la conversión $R_1 \rightarrow R_2$, donde R_1 es un registro de alguna de las clases del tipo COMPUESTO y R_2 denota cualquiera de los registros del procesador, el valor almacenado en el grupo de registros R_1 debe pasarse a un sólo registro R_2; dependiendo de la clase a la que pertenezca R_1, puede ser conveniente que R_2 sea algún registro en particular del procesador. Por ejemplo, para el procesador 8086 de Intel, si se desea pasar un valor almacenado en la pareja de registros dx:ax a un registro de la clase SIMPLE, lo más adecuado es que este último registro sea ax, pues de esta forma no deberá emitirse código. En este campo, entonces, lo que se indica es si R_2 debe ser algún registro en particular del procesador.

5. Tabla de tipos de datos. En esta tabla se indican los modos de direccionamiento que se pueden utilizar para formar las direcciones de las celdas de la memoria asignadas los diferentes tipos de datos del lenguaje fuente. Cada entrada de la tabla tiene dos campos:

i. El tipo de datos del lenguaje fuente.

ii. El modo de direccionamiento que se puede utilizar para formar la dirección de la celda de la memoria asignada a un valor del tipo de datos anterior.

6. Tabla de modos de direccionamiento. En esta tabla se indica cuáles son los modos de direccionamiento del procesador. Cada entrada tiene los siguientes campos:

i. El modo de direccionamiento. A cada modo de direccionamiento se le asigna un código numérico, este código es el que aparece en las tablas siempre que se haga referencia a un modo de direccionamiento.
ii. El tipo de datos. En la tabla de tipos de datos se indican los modo de direccionamiento que pueden utilizarse para cada tipo de datos del lenguaje fuente. Si hay varios modos que se pueden utilizar para un mismo tipo de datos, entonces, se deberá utilizar una entrada por cada tipo de datos.
iii. Código. En esta campo se indica el código que debe ser emitido para poder formar la dirección de una celda de la memoria utilizando cada modo de direccionamiento. Por ejemplo, para poder utilizar el modo de direccionamiento indirecto, es necesario cargar la dirección en alguno de los registros del procesador.
iv. Costo. Aquí se especifica el costo del código anterior más el costo de formar la dirección de una celda de la memoria utilizando cada modo de direccionamiento.
v. Los registros utilizados por el modo de direccionamiento. En este campo se indican solamente los registros de las clases a las cuales se les asignó una dimensión en las matrices de costos.
vi. Los registros utilizados por el modo de direccionamiento pertenecientes a las clases a las que no se les asignó una dimensión en las matrices de costos.
vii. El formato utilizado por cada modo de direccionamiento.

7. Tabla de conversión. En esta tabla se indica como pasar un valor que está almacenado en la memoria a un registro y viceversa. Cada entrada de la tabla tiene los siguientes campos:

i. El modo de direccionamiento utilizado para tomar un valor de la memoria.
ii. La clase de registro en la que se desea almacenar el valor.
iii. El código necesario para hacer la transferencia.
iv. El costo del código.

8. Tabla de costo de formación de dirección. Cada entrada de esta tabla tiene los siguientes campos:

i. El modo de direccionamiento de la memoria.
ii. El costo de tomar por primera vez un valor almacenado en la memoria utilizando cada modo de direccionamiento.
iii. El costo de tomar un valor almacenado en la memoria, una vez que se llenaron los registros de dirección requeridos por el modo de direccionamiento.

9. Tabla de tamaños de tipos. En esta tabla se especifica el número de *bytes* que ocupa en la memoria un valor de cada uno de los tipos de datos simples del lenguaje fuente. Cada entrada de la tabla corresponde a un tipo de datos diferente y tiene un sólo campo en donde se indica el número de *bytes* que ocupa un valor de tal tipo.

10. Tabla de restricciones sobre las direcciones de la memoria. En esta tabla se indica cuales son las restricciones impuestas por el procesador sobre las direcciones de la memoria que puede ocupar un valor del tipo indicado por cada entrada (cada entrada corresponde a un tipo de datos simple diferente). Las entradas de la tabla tienen dos campos:

i. Divisor.
ii. Residuo.

Las únicas celdas de la memoria que pueden ser utilizadas son aquellas cuya dirección D es tal que D MÓDULO Divisor = Residuo.

A.2 Tablas con la descripción del procesador 8086 de Intel

A continuación se presentan las tablas que contienen la descripción del procesador 8086 de Intel. Las abreviaturas utilizadas en las tablas son las siguientes:

Rai	:	REGISTRO_DIRECCIÓN_IZQUIERDO
Rad	:	REGISTRO_DIRECCIÓN_DERECHO
Rdi	:	REGISTRO_DATO_IZQUIERDO
Rdd	:	REGISTRO_DATO_DERECHO
Sim	:	SIMÉTRICO
Est_global	:	VARIABLE ESTRUCTURADA GLOBAL
Est_local	:	VARIABLE ESTRUCTURADA LOCAL
Bi	:	BASE_ÍNDICE

Ind	:	INDIRECTO
Idx	:	ÍNDICE
Dir	:	DIRECTO
Inm	:	INMEDIATO
Hi	:	HIJOS IGUALES
B	:	BANDERAS
Vm	:	VARIOS MODOS
o1	:	OFFSET DEL PRIMER OPERANDO
o2	:	OFFSET DEL SEGUNDO OPERANDO
oc	:	OFFSET CERO
o+	:	SUMA DE LOS OFFSETS DE LOS OPERANDOS
ap1	:	ETIQUETA UTILIZADA PARA FORMAR LA DIRECCIÓN DEL PRIMER OPERANDO
Fun	:	INVOCAR A FUNCIÓN DE USUARIO

A.2.1 Tabla de instrucciones

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Retn	Nulo	Nulo	Nulo	0	0	8	0	0	0	epílogo
Finf	Dir	Nulo	Nulo	0	0	2	0	0	0	<N1> endp
Princf	Dir	Nulo	Nulo	0	0	2	0	0	0	<N1> proc near prólogo <L>
Finprog	Nulo	Nulo	Nulo	0	0	0	0	0	0	finp
Princprog	Nulo	Nulo	Nulo	0	0	0	0	0	0	include inicio.asm inicod
Ini_datos	Nulo	Nulo	Nulo	0	0	2	0	0	0	inidatos
Fin_datos	Nulo	Nulo	Nulo	0	0	2	0	0	0	Datos ends inicod
Def_dato	Mq	Inm	Nulo	0	Op2=2	2	0	0	0	<N1> dw 0
Def_dato	Mq	Inm	Nulo	0	0	3	0	0	0	<N1> db <C2> dup (0)
Asigna	Mq	Rq	Mi	0	Vm	9	0	0	0	mov word ptr <M1>,<R2>
Asigna	Mq	Rq	Rd	0	Vm	9	0	0	0	mov word ptr <M1>,<R2>
Asigna	Mq	Inm	Md	0	Vm	10	0	0	0	mov word ptr <M1>,<C2>

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Asigna	Mq	Inm	Mi	0	Vm	10	0	0	0	mov word ptr <M1>,<C2>
Asigna	Mq	Mq	Mi	0	Hi	0	0	0	0	
Asignad	Mq	Mq	Md	0	0	8	Rad	0	0	mov word ptr <M1>,<D2>
Arr	Ind	Mq	Mi	0	Vm	9	0	0	0	add <D1>,word ptr <M2>
Arr	Base	R4	Ind	0	Vm	7	Rdd	0	0	add <R2>,<D1>
										add <R2>,bp
Arr	Base	R5	Bi	0	Vm	0	Rdd	0	o1	
Arr	Dir	R4	Ind	0	Vm	8	Rdd	0	0	add <R2>,offset <N1>
Arr	Dir	R5	Ind	0	Vm	0	Rdd	ap1	0	
Arr	Ind	Rq	Mi	0	Vm	3	0	0	0	add <D1>,<R2>
Punto	Ind	Ind	Mi	0	Vm	3	0	0	0	add <D1>,<D2>
Punto	Base	Ind	Md	0	Vm	7	0	0	0	add <D2>,<D1>
										add <D2>,bp
Punto	Dir	Ind	Md	0	Vm	9	0	0	0	add <D2>,offset<N1>
Punto	Dir	Ind	Idx	0	Vm	0	Rai	ap1	0	
Punto	Base	Ind	Bi	0	Vm	0	Rad	0	o1	
Punto	Base	Bi	Md	0	Vm	0	0	0	o+	
Punto	Ind	Bi	Md	0	Vm	3	0	0	0	add <D2>,<D1>
Punto	Bi	Bi	Mi	0	Vm	3	0	0	o+	add <D2>,<D1>
Suma	Rq	Rq	Ri	0	Sim	3	0	0	0	add <R1>,<R2>
Suma	Rq	Rq	Rd	0	Sim	3	0	0	0	add <R2>,<R1>
Suma	Rq	Mq	Ri	0	Sim	9	0	0	0	add <R1>,word ptr <M2>
Suma	Mq	Rq	Mi	0	Sim	16	0	0	0	add word ptr<M1>,<R2>
Suma	Rq	Inm	Ri	0	Sim	4	0	0	0	add <R1>,<C2>
Suma	Mq	Inm	Mi	0	Sim	17	0	0	0	add word ptr<M1>,<C2>
Suma	Rq	Inm	Ri	0	Op2=1 Sim	2	0	0	0	inc <R1>
Suma	Mq	Inm	Mi	0	Op2=1 Sim	15	0	0	0	inc word ptr <M1>
Resta	Rq	Rq	Ri	0	0	3	0	0	0	sub <R1>,<R2>
Resta	Rq	Mq	Ri	0	0	9	0	0	0	sub <R1>,word ptr <M2>
Resta	Mq	Rq	Mi	0	0	16	0	0	0	sub word ptr <M1>,<R2>

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Resta	Rq	Inm	Ri	0	0	4	0	0	0	sub <R1>,<C2>
Resta	Mq	Inm	Mi	0	0	17	0	0	0	sub word ptr <M1>,<C2>
Resta	Rq	Inm	Ri	0	Op2=1	2	0	0	0	dec <R1>
Resta	Mq	Inm	Mi	0	Op2=1	15	0	0	0	dec word ptr <M1>
Incr	Mq	Nulo	Mi	0	0	15	0	0	0	inc word ptr <M1>
Incr	Rq	Nulo	Ri	0	0	2	0	0	0	inc <R1>
Decr	Mq	Nulo	Mi	0	0	15	0	0	0	dec word ptr <M1>
Decr	Rq	Nulo	Ri	0	0	2	0	0	0	dec <R1>
Menos unario	Mq	0	Mi	0	0	16	0	0	0	neg word ptr <M1>
Menos unario	Rq	0	Ri	0	0	3	0	0	0	neg <R1>
Multiplica	R1	Mq	R6	0	Sim	134	0	0	0	imul word ptr <M2>
Multiplica	R1	Rq	R6	0	Sim	128	0	0	0	imul <R2>
Multiplica	Rq	Inm	Ri	0	Op2=2 Sim	2	0	0	0	sal <R1>,1
Multiplica	Mq	Inm	Mi	0	Op2=2 Sim	15	0	0	0	sal word ptr <M1>,1
Divide	R6	Rq	R1	0	0	165	0	0	0	idiv <R2>
Divide	R6	Mq	R1	0	0	171	0	0	0	idiv word ptr <M2>
Divide	Rq	Inm	Ri	0	Op2=2	2	0	0	0	sar <R1>,1
Divide	Mq	Inm	Mi	0	Op2=2	15	0	0	0	sar word ptr <M1>,1
Compara	Rq	Rq	B	0	0	3	0	0	0	cmp <R1>,<R2>
Compara	Rq	Mq	B	0	0	9	0	0	0	cmp <R1>,word ptr <M2>
Compara	Rq	Inm	B	0	0	4	0	0	0	cmp <R1>,<C2>
Compara	Mq	Inm	B	0	0	10	0	0	0	cmp word ptr <M1>,<C2>
Compara	Mq	Rq	B	0	0	9	0	0	0	cmp word ptr <M1>,<R2>
Mayor	Etiq	B	Nulo	0	0	4	0	0	0	jl <E1>
Igual	Etiq	B	Nulo	0	0	4	0	0	0	je <E1>
Distinto	Etiq	B	Nulo	0	0	4	0	0	0	jne <E1>
Mayor o igual	Etiq	B	Nulo	0	0	4	0	0	0	jge <E1>

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Menor o igual	Etiq	B	Nulo	0	0	15	0	0	0	jmp <E1>
Ins_return	R1	Nulo	Nulo	0	0	8	0	0	0	epflogo
Param	Rq	Nulo	Nulo	0	0	11	0	0	0	push <R1>
Param	Mq	Nulo	Nulo	0	0	16	0	0	0	push word ptr <M1>
Call	Dir	Inm	R1	R7	0	19	0	0	0	call near ptr <N1> saca <C2>

En esta tabla prólogo, epflogo, ini_cod, ini_datos, finp y saca son nombres de macros definidas en el archivo inicio.asm y contienen instrucciones para: formar un registro de activación, eliminar un registro de activación, marcar el inicio o el final de un segmento de datos, marcar el inicio del programa y para sacal información del stack , respectivamente.

A.2.2 Tabla de clases de registros

Clase	Número de registros	Tipo de registros	Lista de registros
Clase 1	1	Simple	ax
Clase 2	1	Simple	dx
Clase 3	4	Simple	bx, cx, si, di
Clase 4	3	Simple	bx, si, di
Clase 5	2	Simple	si, di
Clase 6	2	Compuesto	ad, dx
Clase 7	5	Compuesto	bx, cx, dx, si, di

A.2.3 Tabla de nombres de registros

1	2	3	4	5	6
ax	bx	cx	dx	si	di

A.2.4 Tabla de conversión entre clases de registros

Conversión entre clases de registros Simple.

Registro fuente	Registro destino	Costo	Código
R4	R3	0	
R5	R3	0	
R5	R4	0	
Rq	Rq	2	mov <R2>,<R1>

Conversión entre regisros de la clase Simple y registros de la clase Compuesto.

Registro fuente	Registro destino	Resultado	Costo	Código
Rq	R6	R1	5	cwd
R1	R6	R6	5	cwd
R2	R6	R6	7	mov ax <R1>
R3	R6	R6	7	mov ax,<R1> \n cwd
R4	R6	R6	7	mov ax,<R1> \n cwd
R5	R6	R6	7	mov ax,<R1> \ncwd
R6	Rq	R1	0	
R6	R1	R1	0	
R6	R2	R2	5	mov <R2>,ax
R6	R3	R3	5	mov <R2>,ax
R6	R4	R4	5	mov <R2>,ax
R6	R5	R5	5	mov <R2>,ax

A.2.5 Tabla de tipos de datos

Tipo de variable	Modo de direccionamiento	Tipo de valor
Simple local	Base	Dato
Simple global	Directo	Dato
Offset local	Indirecto	Dirección
Offset local	Base	Dirección
Offset global	Indirecto	Dirección
Offset global	Directo	Dirección

A.2.6 Tabla de modos de direccionamiento

Modo de direccionamiento	Tipo de datos	Registros en matriz	Registros en entrada	Costo	Código	Formato
Base	Nulo	0	0	9		[bp<O>]
Directo	Nulo	0	0	8		<N>
Índice Base	Nulo	R3	R5	26	mov <D>,[bp<O>]	<N>[<D>]
Índice Indirecto	Nulo	R3	R5	28	mov <D>[bp<O>]	[bp+<D><O>]
	Est global	R3	R4	18	mov <D>,offset<N>	[<D>]
Indirecto	Est local	R3	R4	22	mov <D>,[bp<O>]	[<D>]
Inmediato	Nulo	0	0	0		<C>

A.2.7 Tabla de conversión

Fuente	Destino	Costo	Código
Base	Rq	8	mov <R>,word ptr [bp<O>]
Directo	Rq	8	mov <R>,word ptr <N>
Índice	Rq	8	mov <R>,word ptr <N>[<D>]
Base índice	Rq	8	mov <R>,word ptr [bp+<D><O>]
Indirecto	Rq	8	mov <R>,word ptr [<D>]
Inmediato	Rq	4	mov <R>,<C>
Rq	Base	9	mov word ptr [bp<O>],<R>
Rq	Directo	9	mov word ptr <N>,<R>
Rq	Índice	9	mov word ptr <N>[<D>],<R>
Rq	Base índice	9	mov word ptr [bp+<D><O>],<R>
Rq	Indirecto	9	mov word ptr [<D>],<R>

A.2.8 Tabla de costo de formación de dirección

La información contenida en esta tabla permite elegir el modo de direccionamiento más adecuado para formar la dirección de la celda de la memoria asiganda a un valor (este valor debe corresponder a una variable o a una constante del programa fuente) de cada tipo de datos del lenguaje fuente. La tabla de tipos de datos indica que para formar la dirección de las celdas de la memoria asignadas a las variables estructuradas, es posible utilizar dos modos de direccionamiento: base e indirecto y para las variables estructuradas globales hay también dos posibles modos de direccionamiento: directo e indirecto.

Formar la dirección de una celda de la memoria utilizando modo de direccionamiento base o directo tiene el mismo costo, no importa cuantas veces se haya formado tal dirección anteriormente. En cambio si se desea formar la dirección de una celda de la memoria utilizando modo indirecto, el costo de formar la dirección la primera vez es mayor al costo de formar la dirección las siguientes veces debido a que la primera vez es necesario cargar la dirección de la celda de la memoria en alguno de los siguientes registros: bx, si o di. Para una variable estructurada local o global la dirección de la celda de la memoria que se le asigne puede determinarse a tiempo de *ensamblado*, es por ello que si se utiliza modo de direccionamiento indirecto para formar la dirección de tal celda, será necesario emitir antes una instrucción para cargar un valor constante (la dirección de la celda) en alguno de los registros

mencionados anteriormente; tal instrucción tiene un costo de 4 (ver tabla).

Debido a que los modos de direccionamiento índice y base índice no se pueden utilizar para formar la dirección de una variabe, no es necesario determinar el costo de formar una direccción con tales modos de direccionamiento.

Modo de direccionamiento	Costo de primer acceso	Costos de siguientes accesos
Base	9	9
Directo	6	6
Índice	9	9
Base índice	12	12
Indirecto	9	5

A.2.9 Tabla de tamaños de tipos

Entero	Caracter
2	1

A.2.10 Tabla de restricciones sobre las direcciones de la memoria

	Divisor	Residuo
Entero	1	0
Caracter	1	0

A.3 Tablas con la descripción del procesador 68000 de Motorola

En estas tablas, además de las abreviaturas utilizadas para las tablas del procesador 8086 de Intel, se utilizan las siguientes:

Abs : Absoluto
Indsp : Indirecto con desplazamiento
Index : Indirecto con índice

A.3.1 Tabla de instrucciones

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Retn	Nulo	Nulo	Nulo	0	0	16	0	0	0	unlk %fp\nrts
Finf	Abs	Nulo	Nulo	0	0	2	0	0	0	
Princf	Abs	Nulo	Nulo	0	0	2	0	0	0	global <N1>\n<N1>:\n
										link %fp,&-<L>
Finprog	Nulo	Nulo	Nulo	0	0	0	0	0	0	
Princprog	Nulo	Nulo	Nulo	0	0	0	0	0	0	
Ini_datos	Nulo	Nulo	Nulo	0	0	2	0	0	0	data 1
Fin_datos	Nulo	Nulo	Nulo	0	0	2	0	0	0	text
Def_dato	Mq	Inm	Nulo	0	Op2=2	2	0	0	0	comm <N1>,2
Def_dato	Mq	Inm	Nulo	0	0	3	0	0	0	comm <N1>,<C2>
Asigna	Mq	Rq	Mi	0	Vm	4	0	0	0	mov.l %<R2>,<M1>
Asigna	Mq	Rq	Rd	0	Vm	4	0	0	0	mov.l %<R2>,<M1>
Asigna	Mq	Inm	Mi	0	Op2=0	4	0	0	0	clr.l <M1>
Asigna	Mq	Inm	Md	0	Vm	12	0	0	0	mov.l &<C2>,<M1>
Asigna	Mq	Inm	Mi	0	Vm	12	0	0	0	mov.l &<C2>,<M1>
Asigna	Mq	Mq	Mi	0	Hi	0	0	0	0	
Asigna	Mq	Mq	Mi	0	Vm	4	0	0	0	mov.l <M2>,<M1>
Asigna	Mq	Mq	Md	0	Vm	4	0	0	0	mov.l <M2>,<M1>
Asignad	Mq	Mq	Md	0	0	4	Rad	0	0	mov.l %<D2>,<M1>
Arr	Ind	Mq	Mi	0	Vm	6	0	0	0	add.l <M2>,%<D1>
Arr	Indsp	R3	Ind	R1	Vm	24	Rdd	0	0	mov.l &<O1>,%<Ra>
										add.l %<Ra>,%<R2>
										add.l %fp,%<R2>

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Arr	Indsp	R3	Indx	0	Vm	0	Rdd	0	o1	
Arr	Abs	R3	Ind	R1	Vm	18	Rdd	0	0	mov.l &<N1>,%<Ra> add.l %<Ra>,%<R2>
Arr	Abs	Inm	Dir	0	Vm	0	0	ap1	o2	
Arr	Ind	Rq	Mi	0	Vm	6	0	0	0	add.l %<R2>,%<D1>
Arr	Ind	Rq	Indx	0	Vm	0	Ra1	0	oc	
Punto	Ind	Ind	Mi	0	Vm	6	0	0	0	add.l %<D2>,%<D1>
Punto	Indsp	Ind	Md	R1	Vm	24	0	0	0	mov.l &<O1>,%<Ra> add.l %<Ra>,%<D2> add.l %fp,%<D2>
Punto	Abs	Ind	Md	R1	Vm	18	0	0	0	mov.l &<N1>,%<Ra> add.l %<Ra>,%<D2>
Punto	Indsp	Ind	Indx	0	Vm	0	Ra2	0	o1	
Punto	Indsp	Indx	Md	0	Vm	0	0	0	o+	
Punto	Ind	Indx	Md	0	Vm	6	0	0	0	add.l %<D1>,%<D2>
Punto	Indx	Indx	Mi	0	Vm	6	0	0	o+	add.l %<D1>,%<D2>
Suma	Rq	Rq	Ri	0	Sim	6	0	0	0	add.l %<R2>,%<R1>
Suma	Rq	Rq	Rd	0	Sim	6	0	0	0	add.l %<R2>,%<R1>
Suma	Rq	Mq	Ri	0	Sim	6	0	0	0	add.l <M2>,%<R1>
Suma	Mq	Rq	Mi	0	Sim	12	0	0	0	add.l %<R2>,<M1>
Suma	Rq	Inm	Ri	0	Sim	8	0	0	0	add.l &<C2>,%<R1>
Suma	Mq	Inm	Mi	0	Sim	12	0	0	0	add.l &<C2>,<M1>
Suma	Rq	Inm	Ri	0	Op2<7	1	0	0	0	add.l &<C2>,%<R1>
Suma	Mq	Inm	Mi	0	Op2<7	4	0	0	0	add.l &<C2>,<M1>
Resta	Rq	Rq	Ri	0	0	6	0	0	0	sub.l %<R2>,%<R1>
Resta	Rq	Mq	Ri	0	0	6	0	0	0	sub.l <M2>,%<R1>
Resta	Mq	Rq	Mi	0	0	12	0	0	0	sub.l %<R2>,<M1>
Resta	Rq	Inm	Ri	0	0	8	0	0	0	sub.l &<C2>,%<R1>
Resta	Mq	Inm	Mi	0	0	12	0	0	0	sub.l &<C2>,<M1>
Resta	Rq	Inm	Ri	0	Op2<7	1	0	0	0	sub.l &<C2>,%<R1>
Resta	Mq	Inm	Mi	0	Op2<7	4	0	0	0	sub.l &<C2>,<M1>
Incr	Mq	Nulo	Mi	0	0	4	0	0	0	add.l &1,<M1>
Incr	Rq	Nulo	Ri	0	0	1	0	0	0	add.l &1,<R1>

Operación	Op1	Op2	Res	Adic	Bits	Costo	Dir res	Ap res	Off res	Código
Decr	Mq	Nulo	Mi	0	0	4	0	0	0	sub.l &1,<M1>
Decr	Rq	Nulo	Ri	0	0	1	0	0	0	sub.l &1,<R1>
Menos unario	Mq	Nulo	Mi	0	0	12	0	0	0	neg.l <M1>
Menos unario	Rq	Nulo	Ri	0	0	6	0	0	0	neg.l %<R1>
Multiplica	R1	Mq	Ri	0	Sim	70	0	0	0	muls.w <M2>,%<R1>
Multiplica	R1	Rq	Ri	0	Sim	70	0	0	0	muls.w %<R2>,%<R1>
Multiplica	Rq	Inm	Ri	0	Op2=2	10	0	0	0	lsl.l &1,%<R1>
Multiplica	Rq	Inm	Ri	0	Fun 1 Sim	24	0	0	0	
Divide	R1	Rq	Ri	0	0	158	0	0	0	divs.w %<R2>,%<R1>
Divide	R1	Mq	Ri	0	0	158	0	0	0	divs.w <M2>,%<R1>
Divide	Rq	Inm	Ri	0	Op2=2	8	0	0	0	lsr.l &1,%<R1>
Compara	Rq	Rq	B	0	0	6	0	0	0	cmp.l %<R1>,%<R2>
Compara	Rq	Mq	B	0	0	6	0	0	0	cmp.l %<R1>,<M2>
Compara	Rq	Inm	B	0	0	6	0	0	0	cmp.l %<R1>,&<C2>
Compara	Mq	Inm	B	0	0	4	0	0	0	cmp.l <M1>,&<C2>
Mayor	Etiq	B	Nulo	0	0	10	0	0	0	bgt <E1>
Menor	Etiq	B	Nulo	0	0	10	0	0	0	blt <E1>
Igual	Etiq	B	Nulo	0	0	10	0	0	0	beq <E1>
Distinto	Etiq	B	Nulo	0	0	10	0	0	0	bne <E1>
Mayor o igual	Etiq	B	Nulo	0	0	10	0	0	0	bge <E1>
Menor o igual	Etiq	B	Nulo	0	0	10	0	0	0	ble <E1>
Goto	Etiq	Nulo	Nulo	0	0	10	0	0	0	bra <E1>
Ins_return	R2	Nulo	Nulo	0	0	8	0	0	0	unlk %fp\nrts
Param	Rq	Nulo	Nulo	0	0	2	0	0	0	mov.l %<R1>,-(%sp)
Param	Mq	Nulo	Nulo	0	0	4	0	0	0	mov.l <M1>,-(%sp)
Call	Abs	Inm	R2	R4	Op2=0	18	0	0	0	jsr <N1>
Call	Abs	Inm	R2	R4	0	30	0	0	0	jsr <N1>\nadd.l &<K2>,%sp

A.3.2 Tabla de clases de registros

Clase	Número de registros	Tipo de registros	Lista de registros
Clase 1	7	Simple	d0,d1,d2,d3,d4,d5,d6
Clase 2	1	Simple	d7
Clase 4	6	Simple	a0,a1,a2,a3,a4,a5
Clase 6	13	Compuesto	d0,d1,d2,d3,d4,d5,d6 d7,a0,a1,a2,a3,a4,a5

A.3.3 Tabla de nombres de registros

1	2	3	4	5	6	7	8	9	10	11	12	13	14
d0	d1	d2	d3	d4	d5	d6	d7	a0	a1	a2	a3	a4	a5

A.3.4 Tabla de conversión entre clases de registros

Conversión entre clases de registros Simple.

Registro fuente	Registro destino	Costo	Código
Rq	Rq	4	"mov.l %<R1>,%<R2>"

Conversión entre regisros de la clase Simple y registros de la clase Compuesto.

Registro fuente	Registro destino	Resultado	Costo	Código
Rq	R6	R6	0	
R6	Rq	R1	0	

A.3.5 Tabla de tipos de datos

Tipo de variable	Modo de direccionamiento	Tipo de valor
Simple local	Indirecto con desplazamiento	Dato
Simple global	Absoluto	Dato
Offset local	Indirecto	Dirección
Offset local	Indirecto con desplazamiento	Dirección
Offset global	Indirecto	Dirección
Offset global	Absoluto	Dirección

A.3.6 Tabla de modos de direccionamiento

Modo de direccionamiento	Tipo de datos	Registros en matriz	Registros en entrada	Costo	Código	Formato
Absoluto	Nulo	0	0	12		<N>
Indirecto con desplazamiento	Nulo	0	0	12		<O>(%fp)
Indirecto	Est global	0	R3	32	mov.l &<N>,%<D>	(%<D>)
Indirecto	Est local	R1	R3	42	mov.l %fp,%<D> mov.l &<O>,%<R> add.l %<R>,%<D>	(%<D>)
Indirecto con índice	Nulo	0	R3	30	mov.l <O>(%fp),%<D>	<O>(%<Da>, %<D>.l)
Inmediato	Nulo	0	0	8		&<C>
Direc	Nulo	0	0	12		<N><O>

A.3.7 Tabla de conversión

Fuente	Destino	Costo	Código
Absoluto	Rq	16	mov.l <N>,%<R>
Indirecto con desplazamiento	Rq	16	mov.l <O>(%fp),%<R>
Indirecto	Rq	12	mov.l (%<D>),%<R>
Indirecto con índice	Rq	18	mov.l <O>(%fp,%<D>),%<R>
Direc	Rq	16	mov.l <N><O>,%<R>
Inmediato	Rq	12	mov.l &<C>,%<R>
Rq	Absoluto	16	mov.l %<R>,<N>
Rq	Indirecto con desplazamiento	16	mov.l %<R>,<O>(%fp)
Rq	Indirecto	12	mov.l %<R>,(%<D>)
Rq	Indirecto con índice	18	mov.l %<R>,<O>(%fp,%<D>)
Rq	Direc	16	mov.l %<R>,<N><O>

A.3.8 Tabla de costo de formación de dirección

Puesto que el modo de direccionamiento indirecto con índice no se utiliza para formar la dirección de una variable, no es necesario determinar el costo de formar una dirección con tal modo de direccionamiento.

Modo de direccionamiento	Costo de primer acceso	Costos de siguientes accesos
Absoluto	12	12
Indirecto con desplazamiento	12	12
Indirecto	24	8
Indirecto con índice	14	14

A.3.9 Tabla de tamaños de tipos

Entero	Caracter
4	1

A.3.10 Tabla de restricciones sobre las direcciones de la memoria

	Divisor	Residuo
Entero	2	0
Caracter	1	0

APÉNDICE B

SINTAXIS DEL LENGUAJE MANEJADO POR LA PARTE FRONTAL

En este apéndice se presenta la gramática que describe el subconjunto del lenguaje de programación C que es manejado por la parte frontal que se desarrolló durante la tesis.

En las producciones, los símbolos no terminales aparecen con itálicas y los símbolos terminales con negritas.

programa	→	*definiciones*
definiciones	→	*definición*
	|	*definiciones definición*
definición	→	*tipo definición_función*
	|	*definición_función*
	|	*declaración_global*
tipo	→	**int**
	|	**struct id**
definición_función	→	*encabezado* **{** *declaraciones instrucciones* **}**
encabezado	→	**id ( )**
	|	**id (** *lista_identificadores* **)** *declaración_parámetros*
lista_identificadores	→	**id**
	|	*lista_identificadores* **, id**
declaración_parámetros	→	*declaración_parámetros declaración_parámetro*
	|	ε
declaración_parámetro	→	*tipo lista_identificadores* **;**
	|	ε
declaraciones	→	*declaraciones declaración*
	|	ε
declaración	→	*tipo lista* **;**
	|	**struct** *id_opcional* **{** *lista_campos* **}** *lista_opcional* **;**

declaración_global → *tipo lista_elementos* ;
| **struct** *id_opcional* { *lista_campos* }
lista_opcional ;

id_opcional → **id**
| ε

lista_opcional → *lista_elementos*
| ε

lista → **id**
| **id [número_entero]**
| **id ( )**
| *lista* , **id**
| *lista* , **id [número_entero]**
| *lista* , **id ( )**

lista_campos → *campo*
| *lista_campos campo*

campo → *tipo lista_elementos* ;

lista_elementos → **id**
| **id [número_entero]**
| *lista_elementos* **id**
| *lista_elementos* **id [número_entero]**

instrucciones → *instrucciones instrucción*
| ε

instrucción → *expresión* ;
| ;
| **break** ;
| **return** ;
| **return** *expresión* ;
| *instrucción_compuesta*
| **if** (*expresión*) *instrucción*
| **if** (*expresión*) *instrucción* **else** *instrucción*
| **while** (*expresión*) *instrucción*

instrucción_compuesta → { *instrucción* }

expresión → *variable*
| **número_entero** .
| **Id** (*argumentos_opcionales*)
| ++ *variable*
| -- *variable*
| *expresión* + *expresión*
| *expresión* - *expresión*
| *expresión* * *expresión*
| *expresión* < *expresión*
| *expresión* > *expresión*
| *expresión* >= *expresión*
| *expresión* <= *expresión*
| *expresión* == *expresión*
| *expresión* != *expresión*
| *expresión* && *expresión*
| *expresión* || *expresión*
| - *expresión*
| ! *expresión*
| *variable* = *expresión*

argumentos_opcionales → *lista_argumentos*
| ε

lista_argumentos → *expresión*
| *lista_argumentos* , *expresión*

variable → **Id**
| **Id** . *desc_campos*
| **Id** [*expresión*]
| **Id** [*expresión*] . *desc_campos*

desc_campos → **Id**
| **Id** [*expresión*]
| **Id** . *desc_campos*
| **Id** [*expresión*] . *desc_campos*

APÉNDICE C

OPERADORES DE REPRESENTACIÓN INTERMEDIA

En este apéndice se presenta el conjunto de operadores de representación intermedia manejados por la parte frontal y el generador de código.

Operador	Descripción
SUMA	Suma.
RESTA	Resta.
MULTIPLICA	Multiplicación.
DIVIDE	División.
INCR	Operación unaria de incremento.
DECR	Operación unaria para decremento.
MENOS_UNARIO	Menos unario.
ASIGNA	Asignación.
COMPARA	Comparación de dos valores.
MAYOR, MENOR, IGUAL, DISTINTO, MAYOR_IGUAL, MENOR_IGUAL	Cada uno de estos operadores representa un salto condicional. En el *dag*, un nodo correspondiente a alguno de estos operadores tiene dos hijos: el primer hijo contiene una etiqueta que indica cuál es el destino del salto, mientras que el segundo hijo es un subárbol que representa la condición que debe ser cumplida para efectuar el salto.
GOTO	Salto incondicional. En el *dag* un nodo correspondiente a un salto incondicional tiene un hijo que indica el destino del salto.

RETN	Regreso de una *llamada* a *función*. Este operador no tiene *operandos*.
INS_RETURN	Regreso de una llamada a función. Este operador se utiliza cuando se devuelve un valor como resultado de una llamada. En el *dag* un nodo con este operador tiene un hijo que representa el valor devuelto por la llamada.
PARAM	Este operador sirve para indicar que el valor de una expresión se utiliza como parámetro en una llamada a función. En el *dag* un nodo correspondiente a este operador tiene un hijo que representa el valor del *parámetro*.
CALL	Este operador sirve para señalar una llamada a función. En el *dag* un nodo con este operador tiene dos hijos: el primer hijo representa el nombre de la función y el segundo hijo indica el número de parámetros de la llamada.
PUNTO	Este operador permite tomar el valor de un campo de alguna estructura. En el *dag* un nodo con este operador tiene dos hijos: el primer hijo representa el nombre de la estructura y el segundo hijo el nombre del campo.
ARR	Este operador es utilizado para tomar alguno de los elementos de un arreglo. En el *dag* un nodo con este operador tiene dos hijos: el primer hijo representa el nombre del arreglo y el segundo la posición que ocupa el elemento, cuyo valor se desea tomar, dentro del arreglo.
ASIGNAD	Asignación de dirección. Aunque en la gramática presentada en el apéndice anterior no se manejan apuntadores, si un elemento de un arreglo o un campo de una estructura es una subexpresión común, entonces, será necesario conocer la dirección de tal elemento o campo.

FINF	Con este operador se *marca* el final de una función. En el *dag* un nodo con este *operador* tiene un hijo que indica el nombre de la *función*.
PRINCF	Con este operador se marca el inicio de una función. En el *dag* un nodo con este operador tiene un hijo que indica el nombre de la función.
FINPROG	Con este operador se marca el final de un programa.
PRINCPROG	Este operador señala el inicio de un programa.
INI_DATOS	Este operador señala el inicio de una sección de declaración de variables.
FIN_DATOS	Este operador marca el final de una sección de declaración de variables.
DEF_DATO	Este operador sirve para reservar espacio en la memoria para una variable. En el *dag* un nodo con este operador tiene dos hijos: el primer hijo indica el nombre de la variable y el segundo indica el espacio en *bytes* que ocupa tal variable.

[Aho72] Aho,A.V.,J.D. Ullman. "The theory of parsing translation and compiling". Vol I: Parsing. Prentice-Hall, Englewood Cliffs, N.J.

[Aho76] Aho, A.V., S.C. Johnson [1976]. "Optimal code generation for expression trees". Journal of the ACM, 23:3, 488-501.

[Aho77] Aho, A.V., S.C. Johnson, J.D. Ullman [1977]. "Code generation for expressions with common subexpressions". Journal of the ACM, 24:1, 146-160.

[Aho86] Aho, A.V., R. Sethi, J.D. Ullman [1986]. "Compilers principles, techniques and tools". Addison Wesley, Reading, Mass.

[Appe87] Appel,A.W., K.J. Supowit. "Generalizations of the Sethi-Ullman algorithm for register allocation". Software Practice and Experience, 17:6, 417-421.

[Bird82] Bird, P. L. [1982]. "An implementation of a code generator specification language for table driven code generators". Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices, 17:6, 44-55.

[Chri84] Christopher, T.W., P.J. Hatcher, R.C. Kukuk [1984]. "Using dynamic programming to generate optimized code in a Graham-Glanville style code generator". Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices, 19:6, 25-36.

[Dres87] Drechsler, K.H., M.P. Stadel [1987]. "The Pascal-XT code generator". Sigplan Notices, 22:8, 56-78.

[Gana82] Ganapathi, M., C.N. Fischer, J.L. Hennesy [1982]. "Retargetable compiler code generation". Computing Surveys, 14:4, 573-592.

[Grah82] Graham, S.L., R.R. Henry, R.A. Schulman [1982]. "An experiment in table driven code generation". Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices, 17:6, 32-43..

[Grah84] Graham, S.L., P. Algrain, R.R Henry, M.K. McKusick, E. Pelegri-Llopard [1984]. "Experience with a Graham-Glanville style code generator". Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices, 19:6, 13-24.

[Harm85] Harman, T.L., B. Lawson [1985]. "The Motorola MC68000 microprocessor family: assembly language, interface design and system design". Prentice Hall, Englewood Cliffs, New Jersey.

[Hatc86] Hatcher,P.J., T.W. Christopher. "High quality code generation via bottom-up tree pattern matching". Proceedings of the thirteenth annual ACM Symposium on principles of programming languages, 119-130.

[Hoff82] Hoffman,C.M., M.J. O'Donnell. "Pattern matching in trees". Journal of the Association for Computing Machinery, 29:1, 68-95.

[Hors85] Horspool,R.N., A. Scheunemann [1985]. "Automating the selection of code templates". Software-Practice and Experience, 15:5, 503-514.

[John75] Johnson,S.C. 1975 . "YACC: yet another compiler compiler". Computing Services Technical Report No. 32, Bell Laboratories, Murray Hill, N.J.

[Land82] Landwehr, R., H.S. Jansohn [1982]. "Experience with automatic code generator generator". Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices, 17:6, 56-66

[Lesk75] Lesk, M.E., E. Schmidt 1975. "LEX-a lexical analyzer generator". Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J.

[Morg82] Morgan,T.M., L.A. Rowe. "Analyzing exotic instructions for a retargetable code generator". Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices, 17:6, 197-204.

[Ryan85] Ryan, R.R., H. Spiller [1985]. "The C programming language and a C compiler". IBM Systems Journal, 24:1, 37-48.

[Spec87] Spector, D., P.K. Turner [1987]. "Limitations of Graham-Glanville code generation". SIGPLAN Notices, 22:2, 100-107.

[VanB87] Van Biljon, W.R., D.A. Sewry, M.A. Mulders [1987]. "Register allocation in a pattern matching code generator". Software-practice and experience, 17:8, 521-531.

[Yate88] Yates,J.S., R.A. Schwartz. "Dynamic programming and industrial-strength instruction selection: code generation by tiring, but not exhaustive, search". SIGPLAN Notices, 23:10, 131-140.

[Yeun85] Yeung, B.C. [1985]. "8086/8088 assembly language programmming". John Wiley & Sons. Chichester, Great Britain.